Efficient Software Model Checking of Soundness of Type Systems

Michael Roberson Melanie Agnew Paul T. Darga Chandrasekhar Boyapati

Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor, MI 48109
{roberme,melagnew,pdarga,bchandra}@eecs.umich.edu

Abstract

This paper presents novel techniques for checking the soundness of a type system automatically using a software model checker. Our idea is to systematically generate every type correct intermediate program state (within some finite bounds), execute the program one step forward if possible using its small step operational semantics, and then check that the resulting intermediate program state is also type correct—but do so efficiently by detecting similarities in this search space and pruning away large portions of the search space. Thus, given only a specification of type correctness and the small step operational semantics for a language, our system automatically checks type soundness by checking that the progress and preservation theorems hold for the language (albeit for program states of at most some finite size). Our preliminary experimental results on several languages—including a language of integer and boolean expressions, a simple imperative programming language, an object-oriented language which is a subset of Java, and a language with ownership types—indicate that our approach is feasible and that our search space pruning techniques do indeed significantly reduce what is otherwise an extremely large search space. Our paper thus makes contributions both in the area of checking soundness of type systems, and in the area of reducing the state space of a software model checker.

1. Introduction

Type systems provide significant software engineering benefits. Types can enforce a wide variety of program invariants at compile time and catch programming errors early in the software development process. Types serve as documentation that lives with the code and is checked throughout the evolution of code. Types also require little programming overhead and type checking is fast and scalable. For these reasons, type systems are the most successful and widely used formal methods for detecting programming errors. Types are written, read, and checked routinely as part of the software development process. However, the type systems in languages such as Java, ML, or Haskell have limited descriptive power and only perform compliance checking of certain simple program properties. But it is clear that a lot more is possible. There is therefore plenty of recent research interest on type systems for preventing various kinds of programming errors [5, 13, 23, 32, 33, 41].

A formal proof of type soundness lends credibility that a type system does indeed prevent the errors that it claims to prevent, and is a crucial tool in the design of a type system. At present, type soundness proofs are mostly done by hand, if at all. These proofs are usually long, tedious, and consequently error-prone. Small mistakes or overlooked cases in a proof can invalidate large amounts of work. There is therefore a growing interest [1] in machine checkable proofs of type soundness. However, both the above approaches—proofs done by hand (e.g., [14]) or machine checkable proofs (e.g., [34])—require significant manual effort.

This paper presents an alternate approach for checking the soundness of a type system automatically using a software model checker, requiring minimal manual effort. Our idea is to systematically generate every type correct intermediate program state (within some finite bounds), execute the program one step forward if possible using its small step operational semantics, and then check that the resulting intermediate program state is also type correct—but do so efficiently by detecting similarities in this search space and pruning away large portions of the search space. Thus, given only a specification of type correctness and the small step operational semantics for a language, our system automatically checks type soundness by checking that the progress and preservation theorems [37] hold for the language (albeit for program states of at most some finite size).

Our experimental results on several languages—including the language of integer and boolean expressions from [37, Chapters 3 & 8], a typed version of the imperative language IMP from [42, Chapter 2], an object-oriented language which is a subset of Java, and a language with ownership types [10]—indicate that our approach is feasible and that our search space pruning techniques do indeed significantly reduce what is otherwise an extremely large search space. This paper thus offers a promising approach for checking type soundness automatically, thereby enabling the design of novel type systems. In particular, this can enormously help language designers in debugging language specifications.

Note that checking the progress and preservation theorems on all programs states up to a finite size does not *prove* that the type system is sound, because the theorems might not hold on larger unchecked program states. However, in practice, we expect that all type system errors will be revealed by small sized program states. This conjecture, known as the *small scope hypothesis* [28], has been experimentally verified in several domains. Our preliminary experiments using mutation testing [36, 29] suggest that the conjecture also holds for checking type soundness. We also examined all the type soundness errors we came across in literature and found that in each case, there is a small program state that exposes the error. Thus, exhaustively checking type soundness on all programs states up to a finite size does at least generate a high degree of confidence that the type system is sound.

```
t ::=   true   |   false   |  if t then t else t  |   0   |   succ t   |   pred t   |  iszero t
```

constant true	*constant false*	*conditional*	*constant zero*	*successor*	*predecessor*	*zero test*

Figure 1. Abstract syntax of the language of integer and boolean expressions from [37, Chapters 3 & 8].

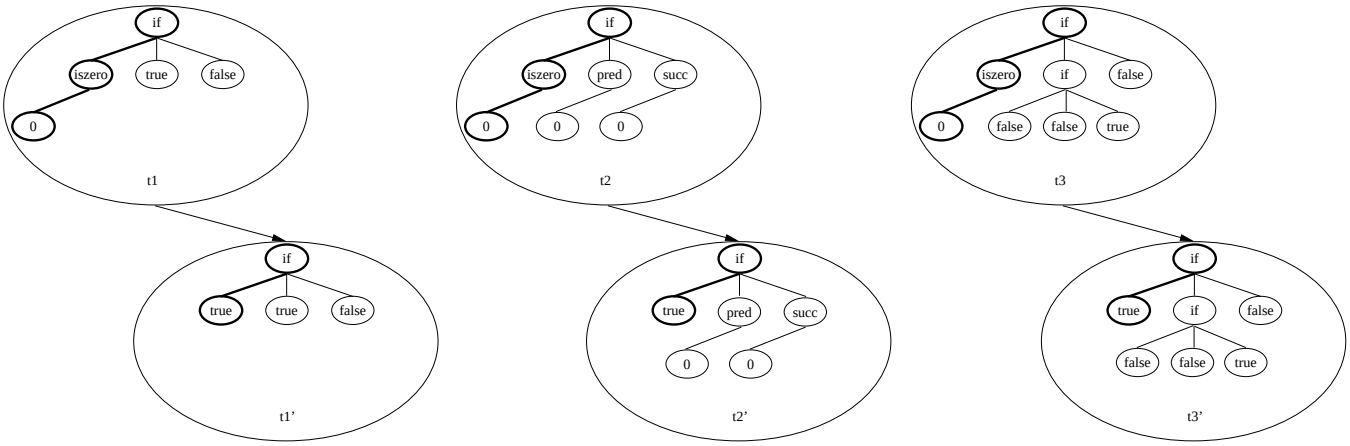

Figure 2. Three abstract syntax trees (ASTs) for the language in Figure 1, before and after a small step evaluation. The tree path touched by the evaluation is highlighted in each case. Note that the tree path is the same in all three cases. Once our system checks the progress and preservation theorems on the AST `t1`, it determines that it is redundant to check the theorems on ASTs `t2` and `t3`.

This paper also makes contributions in improving the state of art in software model checking [2, 3, 7, 11, 15, 20, 40, 24, 31]. Model checking is a formal verification technique that exhaustively tests a circuit/program on all possible inputs (sometimes up to a given size) to handle *input nondeterminism* and on all possible nondeterministic schedules to handle *scheduling nondeterminism*. For hardware, model checkers have been successfully used to verify fairly complex finite state control circuits with up to a few hundred bits of state information; but not circuits in general that have large data paths or memories. Similarly, for software, model checkers have been primarily used to verify control-oriented programs (with scheduling nondeterminism) with respect to temporal properties; but not much work has been done to verify data-oriented programs (with input nondeterminism) with respect to complex data-dependent properties.

Thus, while there is much research on state space reduction techniques for software model checkers such as partial order reduction [18, 20] and tools based on predicate abstraction [21] such as Slam [2], Blast [24], or Magic [7], none of these techniques seem to be effective in reducing the state space when checking the soundness of a type system—where one must deal with input nondeterminism (to check every input program state) and data-dependent properties (type correctness properties that depend on input program states). In fact, because of input nondeterminism, it is difficult to even formulate the problem of automatically checking type soundness in the context of most software model checkers.

This paper describes techniques for efficiently checking the soundness of a type system automatically using a software model checker by significantly reducing the state space of the model checker. It thus contributes to improving the state of art in software model checking. This paper builds on our recent work on model checking properties of data structures [12]. This paper improves on the techniques in [12] (c.f. Section 6) and applies them to checking type soundness.

Finally, this paper also presents an approach for efficiently model checking the soundness of a type system extension, *assuming* that the base type system is sound. The approach exploits the above assumption to detect and prune significantly more redundant program states. Our experiments with several type qualifier [19] extensions show that checking the soundness of a type system extension using this approach is far more efficient than checking the soundness of the extended type system without assuming that the base type system is sound. We expect this approach to be valuable because researchers often design extensions to existing type systems rather than design a type system from scratch.

The rest of this paper is organized as follows. Section 2 illustrates our approach with an example. Section 3 describes the architecture of our software model checker for checking soundness of type systems. Section 4 describes our approach for checking soundness of type system extensions. Section 5 presents preliminary experimental results. Section 6 discusses related work and Section 7 concludes.

2. Example

This section illustrates our approach with an example. Consider the language of integer and boolean expressions in [37, Chapters 3 & 8]. The syntax of the language is shown in Figure 1. The small step operational semantics and the type

Field	Domain
n0	{true, false, if, 0, succ, pred, iszero, unused}
n1	{true, false, if, 0, succ, pred, iszero, unused}
n2	{true, false, if, 0, succ, pred, iszero, unused}
n3	{true, false, if, 0, succ, pred, iszero, unused}
n4	{true, false, if, 0, succ, pred, iszero, unused}
n5	{true, false, if, 0, succ, pred, iszero, unused}
n6	{true, false, if, 0, succ, pred, iszero, unused}
n7	{true, false, if, 0, succ, pred, iszero, unused}
n8	{true, false, if, 0, succ, pred, iszero, unused}
n9	{true, false, if, 0, succ, pred, iszero, unused}
n10	{true, false, if, 0, succ, pred, iszero, unused}
n11	{true, false, if, 0, succ, pred, iszero, unused}
n12	{true, false, if, 0, succ, pred, iszero, unused}

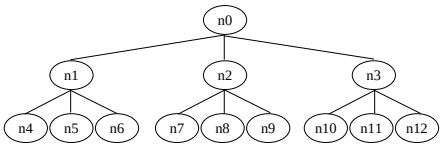

Figure 3. Search space for the language in Figure 1 with ASTs of height at most 3.

checking rules for this language are in [37]. To check type soundness, our system systematically generates and checks the progress and preservation theorems on every type correct program state within some finite bounds.

Figure 2 shows three abstract syntax trees (ASTs) `t1`, `t2`, and `t3`. AST `t1` represents the term '`if (iszero 0) then true else false`'. AST `t2` represents the term '`if (iszero 0) then (pred 0) else (succ 0)`'. AST `t3` represents the term '`if (iszero 0) then (if false then false else true) else false`'. Figure 2 shows the ASTs before and after a small step evaluation according to the small step operational semantics of the language.

A simplified version of our state space reduction technique works as follows. As our system checks the progress and preservation theorems on `t1`, it detects that the small step evaluation of `t1` touches only a small number of AST nodes along a tree path in the AST. These nodes are highlighted in the figure. This means that as long as these nodes remain unchanged, the small step evaluation will behave similarly on all ASTs such as `t2` and `t3`. Our system determines that it is redundant to check the progress and preservation theorems on ASTs such as `t2` and `t3` once it checks the theorems on `t1`. Our system safely prunes those program states from its search space, while still achieving complete test coverage within the bounded domain. The above technique thus ends up checking the progress and preservation theorems on every unique tree path (and some nearby nodes) rather than on every unique AST. Note that the number of unique ASTs of a given maximum height h is exponential in n, where $n = 3^h$, but the number of unique tree paths is only polynomial in n. This leads to significant reduction in the size of the search space and makes our approach feasible.

Our complete state space reduction technique does even better. It detects that in the above example, only the nodes in the redex '`iszero 0`' matter, as long as that is the next redex to be reduced. It therefore prunes all program states where those nodes remain the same and that is the next redex to be reduced. This leads to even greater speedups. In the above example, our system thus ends up checking only $O(n)$ number of program states.

```
1  void search(ConfigurationSpace f, Finitization f) {
2    S = Set of all elements of configuration space c
         within the finite bounds specified by f
3    while (S is not empty) {
4      t = Any configuration in S
5      Check t
6      T = Set of all configurations similar to t including t
7      S  = S - T
8    }
9  }
```

Figure 4. Pseudo-code for the search algorithm.

3. Model Checking Type Soundness

While the basic idea presented in Section 2 is simple, one has to overcome several technical challenges to make it work well in practice. This section describes our approach.

3.1 Specifying Language Semantics

To check type soundness, language designers only need to specify the the small step operational semantics of the language, rules for checking type correctness of intermediate program states, and finite bounds on the size of intermediate program states. The operational semantics can be specified either in an executable language (in our current system, Java) or in a declarative language (in our current system, a language similar to Ott [39]), as long as the declarative specifications can be automatically translated into executable code. The type system, however, must be specified only in the declarative style because that facilitates our static analysis (c.f. Section 3.6). The type system is also automatically translated into executable code to facilitate our dynamic analysis (c.f. Section 3.5). We omit more details due to lack of space, but the model checking techniques we describe in this paper (which are our main contributions) are not tied to our above choice of specification languages and are general enough to work with other languages.

3.2 Search Space

Traditional software model checkers [2, 7, 15, 20, 40, 24, 31] explore a state space by starting from the initial state and systematically generating and checking every successor state. While this approach works well to check software with scheduling nondeterminism, it is not convenient to check software with input nondeterminism. In fact, it is difficult to even formulate the problem of checking type soundness in the context of most software model checkers. Instead, our model checker organizes its search space as follows. Consider the language of integer and boolean expressions in Figure 1. Suppose our system must check the type soundness theorems on all ASTs up to a maximum height h=3. Figure 3 shows the corresponding search space. The search space consists of all possible assignments to the fields, where each field gets a value from its corresponding domain. Every element of this space is an AST. For example, Figure 5 corresponds to an AST representing the term '`if (iszero 0) then (if false false true) else false`'. In Figure 3, there are thirteen fields with eight elements in each of their domains, so the size of this search space is 8^{13}. In general, when checking the integer and boolean language on ASTs of height at

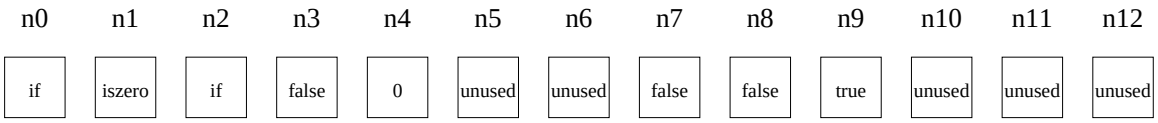

Figure 5. A type correct element of the search space in Figure 3 representing the term '`if (iszero 0) then (if false false true) else false`'.

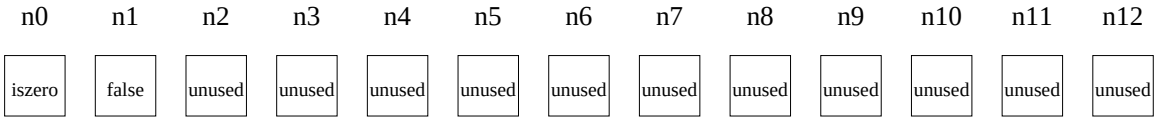

Figure 6. A type incorrect element of the search space in Figure 3 representing the term '`iszero false`'.

most h, the size of the search space is $8^{\frac{3^h-1}{2}}$. Note that many elements of this search space are not type correct or even syntactically correct. For example, the AST in Figure 6 is not type correct because `iszero` cannot be invoked on the boolean constant `false`.

In general, the intermediate state of a program (the configuration space of the operational semantics) can include other components besides an AST, such as a dynamically allocated heap. Our system appropriately constructs a finite search space that includes all such components.

3.3 Search Algorithm

Our search algorithm is simple. Given a language to check for type soundness and some finite bounds on the size of intermediate program states, our system first initializes the search space as we described above. It then systematically explores this space by repeatedly selecting a program state `t` from the space, checking that the progress and preservation theorems hold on `t`, running its analyses to identify other program states similar to `t`, and pruning `t` and the similar (and therefore redundant) program states from the search space. Figure 4 presents the pseudo-code for the search.

3.4 Search Space Representation

Consider the search space of the language in Figure 1, with ASTs of height at most h=8. The size of this search space is about 2^{9841}. Of these, about 2^{2523} ASTs are type correct. However, as our experiments show, our system checks the progress and preservation theorems explicitly only on 41 ASTs. (Our analyses determine that it is redundant to check the theorems on the remaining elements of the search space.) Thus, if we are not careful, then search space management itself could take exponential time and negate the benefits of our search space pruning techniques. We avoid this by using a compact representation of the search space (that is, the set of intermediate program states). We explored two different approaches for representing the search space: (i) using a *reduced ordered binary decision diagram* [6] or BDD (as in our previous work [12]), and (ii) using an incremental SAT solver, MiniSat [16]. In our experiments, we found that the SAT-based approach performs much better, especially on languages with non-tree-based type constraints (that is, on languages whose program states include components other than ASTs). Because of lack of space, we therefore discuss only our SAT-based approach in the rest of this paper.

Our SAT-based approach works as follows. We represent a set of program states as a boolean formula. For example, for the search space in Figure 3, the formula (`n0=if` $\wedge$ `n1=iszero` $\wedge$ `n4=0`) represents the set of all the terms of the form '`if (iszero 0) then x1 else x2`', where `x1` and `x2` are any two terms. This includes the terms represented by ASTs `t1`, `t2`, and `t3` in Figure 2. Every satisfying assignment of the formula represents a member of the set. If the formula is unsatisfiable, then the set of empty. We use an incremental SAT solver to find satisfying assignments of the formula. With this approach, Line 7 in Figure 4, computing the difference of two sets, takes time linear in the size of the formula, because it simply injects clauses into the incremental SAT solver. But Line 3, checking if a set is empty, and Line 4, choosing an element from a non-empty set, could be expensive operations because they invoke the SAT solver.

3.5 Dynamic Analysis

This section presents our dynamic analyses that are key to making our model checker efficient. It first presents the basic analysis and then describes several additional optimizations.

3.5.1 Monitoring Fields Read

Consider the language in Figure 1. Given an AST, our system first checks if it is type correct. If so, our system checks if the progress and preservation theorems hold for that AST.

Case Where AST is not type correct: Consider checking the program state in Figure 6. As our system type checks the AST, it monitors the fields of the AST that the type checker reads. In this case, the type checker reads `n0` and `n1` and returns false (because `iszero` cannot be invoked on a boolean). That means, regardless of the values of the remaining fields, the type checker will always return false if `n0` and `n1` do not change. Our system therefore prunes all elements of the search space where `n0=iszero` and `n1=false`.

Case Where AST is type correct: Consider checking the program state in Figure 5. The AST is type correct this time. As our system evaluates the AST a small step forward, it once again monitors the fields that small step evaluator reads. In this case, it reads `n0`, `n1`, and `n4`. That means, regardless of the values of the remaining fields, the small step evaluator will still behave similarly if the values of `n0`, `n1`, and `n4` do not change. Our system then determines that regardless of the values of the remaining fields, if the AST is type correct before the small step evaluation, then the

Field	Domain
n0.kind	{true, false, if, getf, putf, ...}
n1.kind	{true, false, if, getf, putf, ...}
n2.kind	{true, false, if, getf, putf, ...}
n0.value	{o0, o1, o2, o3, null}
n1.value	{o0, o1, o2, o3, null}
n2.value	{o0, o1, o2, o3, null}
o0.value	{o0, o1, o2, o3, null}
o1.value	{o0, o1, o2, o3, null}
o2.value	{o0, o1, o2, o3, null}
o3.value	{o0, o1, o2, o3, null}

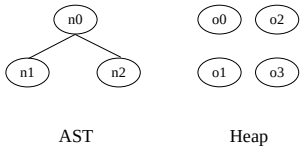

Figure 7. Search space for a language whose intermediate states include an AST and a heap.

AST will be type correct after the small step evaluation. Our system therefore prunes all elements of the search space where `n0=if`, `n1=iszero`, and `n4=0`. This is the basic idea that makes our approach of using exhaustive testing within an extremely large but finite domain feasible. (We discuss correctness issues later in Section 3.6.)

3.5.2 Pruning States With the Same Redex

To optimize further, our system detects that in the above example, only the nodes `n1` and `n4` (corresponding to redex '`iszero 0`') matter, as long as that is the next redex to be reduced. Our system therefore prunes all program states where those nodes remain the same and that is the next redex to be reduced. To make this technique work, our system adds extra bits of information per AST node that indicate whether the node represents a value (and therefore cannot be reduced further), and if not, in what order the node and its children must be reduced. These bits guide the small step evaluator to the appropriate redex, without the evaluator having to read the entire content of the nodes in the path from the root of the AST to the redex. While these bits increase the size of the search space, they end up making the search more efficient by allowing more states to be pruned. Our experimental results indicate that for the above example, our system can scale to check ASTs of up to height 7 in under 3 seconds, and exhaustively cover this space. Note that ASTs of height 7 can contain more than 1000 nodes, and thus can represent expressions of length more than 1000. We do not know of any other model checker that can *exhaustively* cover such a large search space for checking such properties.

3.5.3 Pruning Isomorphic States

The intermediate program states of the language in Figure 1 we have been considering so far includes only an AST. Consider a language whose intermediate program states include an AST as well as a dynamically allocated heap. Figure 7 presents an example of such a search space, where every AST node and every object contains one pointer, the AST has height at most 2, and there are at most 4 heap objects.

Now consider the two elements of the above search space depicted in Figure 8. Clearly, these two elements are isomorphic, because `o0` and `o1` are equivalent memory locations. Therefore, once we check the progress and preservation theorems on the first element, it is redundant to check the theorems on the second element. Our system avoids checking isomorphic structures as follows. Suppose the small step evaluator reads only `n0.kind`, `n0.value`, and `n1.value` when evaluating the first element. Our dynamic analysis concludes that all states with `n0.kind=getf`, `n0.value=o0`, and `n1.value=o1` can be pruned. Our isomorphism analysis then determines that all structures that satisfy the following formula can also be pruned: `n0.kind=getf` $\wedge$ (`n0.value`$\neq$`o0/-null` $\vee$ (`n0.value`$=$`o0` $\wedge$ `n1.value`$\neq$`o0/o1/null`)). In general, to construct the formula, our isomorphism analysis traverses all the relevant fields of a state s. Each time it encounters a fresh object o that a field points to, it includes (in the formula) all other states s' where the fields read by the small step evaluator so far have the same values except that instead of o in s there is another fresh object o' of the same type in s'. Our system then prunes all states denoted by the formula by injecting the negation of the formula into the incremental SAT solver.

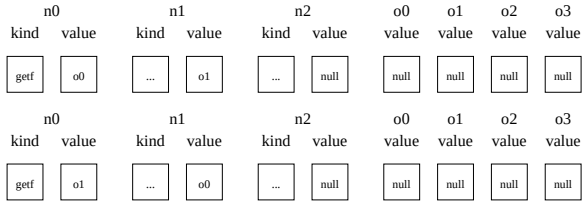

Figure 8. Two isomorphic elements of the search space in Figure 7.

Note that some model checkers also prune isomorphs using heap canonicalization [30]. The difference is, in heap canonicalization, once a checker *visits* a state, it canonicalizes the state and checks if the state has been previously visited. In our isomorphism pruning, once our checker checks a state `s`, it computes a compact formula `F` denoting (often an exponentially large number of) states isomorphic to s, and prunes `F` from the search space. Our checker *never visits* (the often exponentially many) `F`'s states.

3.5.4 Monitoring Information Flow

The basic analysis in Section 3.5.1 conservatively assumes that a function (such as a type checker or a small step evaluator) depends on all the fields it reads. This is not always true. Consider the following example for an illustration.

```
1  class ExpressionLanguage {
2    public boolean typeCheck() {
3      if (n0.kind == IF) {
4        Type t1 = n1.type;
5        Type t2 = n2.type;
6        Type t3 = n3.type;
7        if (n1.type != BOOLEAN) return false;
8        if (n2.type != n3.type) return false;
9        if (n2.type != n0.type) return false;
10       return true;
11     }
12     ...
13   }
14   ...
15 }
```

Suppose the function `typeCheck` returns false on Line 7 because `n1.type!=BOOLEAN`. The above analysis assumes that because `typeCheck` already read `n2.type` and `n3.type`, the return value depends on those fields too—even though it depends only on `n0.kind` and `n1.type`.

To make our analysis more precise, we use dynamic information flow tracking. Consider the search space in Figure 3. There are thirteen input fields. For every value v that a function computes, our system also computes a thirteen-bit shadow value v' that tracks the input fields from which there is an information flow to v. Note that information flow analysis [32] is different from dynamic slicing [43], as the following example shows.

```
1  class InfoFlowDemo {
2    private boolean b;
3    public  boolean typeCheck() {
4      boolean x = false;  if (b) x = true;  return x;
5  }}
```

There is information flow from `b` to `x` above. But if `b` is `false`, then `x` is not control or data dependent on `b` because the branch is not taken. If we use dynamic slicing, then on running `typeCheck` with `b=false` we would incorrectly conclude that `typeCheck` does not depend on `b` and always returns false. To avoid that, our analysis conservatively assumes that after any join point in the control flow graph, all variables depend on the corresponding branch conditional. Thus the return value `x` above depends on `b`.

3.6 Static Analysis

The dynamic analyses described above in effect detect *don't care* fields in a state s, and suggest that all states s' that differ with s only at the don't care fields can potentially be pruned from the search space. The goal of the static analysis is to prove that it is indeed safe to prune those states. To see why the static analysis is necessary, consider the following simple but artificial example, where `typeCheck` returns true iff `a` implies `b`. Suppose we invoke `smallStep` on `a=false` and `b=true`. `typeCheck` returns true before and after the execution of `smallStep`. `smallStep` reads only `a` while `b` is a don't care. Our dynamic analysis suggests that the progress and preservation theorems might verify on all states where `a=false` (and therefore those elements be pruned from the search space). But the suggestion is incorrect because `smallStep` does not verify on `a=false` and `b=false`. `typeCheck` holds before the evaluation of `smallStep` but not after.

```
1  class StaticAnalysisDemo {
2    private boolean a, b;
3    public  boolean typeCheck() { return !a || b; }
4    public  void    smallStep() { a = !a;         }
5  }
```

Our static analysis works as follows. Consider checking the progress and preservation theorems on a state s. Suppose our dynamic analysis identifies fields $f_{1..k}$ as don't cares with respect to the small step evaluation of s. Let v denote the values of the remaining fields in s, and v' in the state s' obtained after the evaluation of a small step. The static analysis partially evaluates the type checking function, say `typeCheck`, with respect to v and v' respectively, to get functions $\texttt{typeCheck}_v(f_{1..k})$ and $\texttt{typeCheck}_{v'}(f_{1..k})$. The analysis then attempts to prove that for all values of $f_{1..k}$ in the bounded domain, $\texttt{typeCheck}_v(f_{1..k})$ implies $\texttt{typeCheck}_{v'}(f_{1..k})$. Note that even though it is the same `typeCheck` function, v and v' are different because the small step evaluator changes the state, so $\texttt{typeCheck}_v$ and $\texttt{typeCheck}_{v'}$ could be different. Our checker invokes the SAT solver to check if the implication holds. If the implication holds, our system prunes the state space as described in Section 3.5. If the implication does not hold, then it means there is a bug in the type system. An instance satisfying the negation of the implication provides a counter example. Our system eliminates quantifiers using skolemization (taking advantage of the fact that the domains are bounded) and then performs the partial evaluation. We currently require the type checker to be specified in a declarative style as described in Section 3.1 to enable us to perform the static analysis. In the future, we also plan to explore partially evaluating executable (Java) code [38].

4. Model Checking Type Extensions

Section 3 presented our approach for efficiently model checking the soundness of a type system. This section presents our approach for efficiently model checking the soundness of certain kinds of type system extensions, *assuming* the base type system (before the extension) was sound. The approach exploits the above assumption to detect and prune significantly more redundant program states. We expect this to be valuable because people often design extensions to existing type systems rather than design a type system from scratch.

Our system supports extensions that add additional type annotations and type checking clauses, but do not change the operational semantics of a language. Several type system extensions follow this discipline, e.g., types for preventing races [17, 5], ownership types [10], and type qualifiers [19].

Our system works as follows. Consider checking the progress and preservation theorems on a program state p. Given p and a function (such as a type checker or a small step evaluator), the dynamic analyses described in Section 3.5 separate the fields of p into *relevant* fields and fields that are *don't cares*. Let F_s be the set of fields that are relevant w.r.t. the small step evaluation of p. Of these, let $F_s^c \subseteq F_s$ be the set of fields that affect the control flow of the small step evaluator. Let F_t be the set of fields that are relevant w.r.t. the type checking that happens after the small step evaluation (to check the preservation theorem). Of these, let $F_t^e \subseteq F_t$ be the set of fields that are relevant w.r.t. the type checking clauses in the type system extension. Our approach then treats all the fields not in $F_s \cap (F_s^c \cup F_t^e)$ as don't care fields, runs the static analysis described in Section 3.6, and if that is successful, prunes all states p' that differ from p only at the don't care fields. This is in contrast to the approach in Section 3 that treats only the fields not in F_s as don't cares. This can lead to significant speedups as our results indicate.

The Clarity [9] tool also proves soundness of certain type qualifier extensions using a theorem prover. However, the tool seems to be limited in kinds of qualifier extensions it can handle, because of the limitations of an automated theorem prover in discharging complex proof obligations. Our model checking based tool, however, is more general because it uses exhaustive testing, albeit in a finite domain.

5. Experimental Results

This section presents our preliminary experimental results. We implemented a rudimentary software model checker as described in this paper. We extended the Polyglot [35] compiler framework to automatically instrument the operational semantics and the type systems of languages to perform our dynamic analyses (described in Section 3.5). We used MiniSat [16] as our incremental SAT solver. We performed all our experiments on a Linux Fedora Core 4 machine with a Pentium 4 3.2 GHz processor and 1 GB memory using Sun's Java 1.5.0_08.

5.1 Model Checking Soundness of Type Systems

We present results for the following four languages, each with increasing complexity:

1. The language of integer and boolean expressions from [37, Chapters 3 & 8].

2. A typed version of the imperative language IMP from [42, Chapter 2].

   This language has variables, so its type checking rules include an environment context. This language also has `while` loops.

3. An object-oriented language Featherweight Java [25].

   This language has classes and objects and methods. The semantics of method calls require term-level substitution (of the formal parameters of a method with their actual values).

4. An extension to Featherweight Java to support ownership types [10], that we call Ownership Java.

   This language has classes parameterized by owner parameters. Therefore the semantics of a method call require both term-level and type-level substitution. To make this language meaningful, we also added a `null` value to Featherweight Java and extended Featherweight Java to support mutations to objects in the heap. For this, we had to model the heap explicitly in the configuration space unlike in Featherweight Java.

Our system works best when each small step of evaluation touches only a small part of the program state. That way, the rest of the state can be treated as a don't care, and a large portion of the state space can be pruned away after each small step of evaluation. But when running experiments, we noticed that for a `while` expression in IMP, the small step evaluator clones the entire body of the `while`, as shown below. σ contains the values of variables.

$\langle$`while c do b`, $\sigma\rangle \rightarrow \langle$`if c then (b; while c do b)`, $\sigma\rangle$

However, we realized that the preservation theorem always holds for the above small step of evaluation, because the same expressions are type checked under the same type environment before and after the small step. We therefore introduced a special construct in our specification language to express such cases. Our tool recognizes that the validity of the preservation theorem is independent of the term being cloned, and thus treats the term as a don't care.

Benchmark	Max Expression Length	**States Checked**	Time (s)
`Arithmatic Expressions`	1	**1**	0.082
	2	**3**	0.105
	3	**3**	0.080
	4	**5**	0.107
	...	...	...
	13	**11**	0.102
	40	**17**	0.139
	121	**23**	0.307
	364	**29**	0.756
	1093	**35**	2.784
	3280	**41**	22.176
`IMP`	1	**1**	0.025
	2	**16**	0.034
	3	**20**	0.042
	4	**42**	0.065
	5	**69**	0.086
	6	**68**	0.093
	7	**68**	0.097
	...	...	...
	15	**139**	0.237
	31	**250**	0.646
	63	**485**	2.387
	127	**913**	9.905
	255	**1797**	53.580
	511	**3653**	315.719
`Featherweight Java`	1	**4**	0.725
	2	**11**	0.549
	3	**14**	0.312
	4	**21**	0.272
	5	**36**	0.243
	6	**43**	0.261
	7	**50**	0.317
	...	...	...
	15	**198**	0.738
	31	**550**	2.282
	63	**1254**	9.048
	127	**2662**	42.019
	255	**5478**	202.009
`Ownership Java`	1	**32**	0.946
	2	**1795**	12.606
	3	**1978**	14.592
	4	**8513**	72.824
	5	**8648**	78.920
	6	**15487**	155.789
	7	**18370**	214.947
	...	...	...
	15	**200611**	5815.375

Figure 9. Checking soundness of type systems. Our system achieves significant state space reduction. E.g., there are over 2^{786} well typed IMP programs of length upto 511, but we check only 3653 states to cover this space.

A similar situation occurs in Featherweight Java, where method calls have a method inlining semantics. Method calls have an additional problem, which is that one small step substitutes all the formals with actuals in the method body, and thus touches the entire method body. To address this, we redefined the semantics of method calls by performing the substitution incrementally. That is, each small step of evaluation performs substitution on at most one AST node. We also changed the type checking rules to properly handle partially substituted program states.

We checked each benchmark *exhaustively* on states up to a maximum size. For the language of arithmetic expressions, we checked all states up to a given maximum expression length l. For the imperative language IMP, we checked all states up to a given maximum expression length l and l variables and l integer literals. For Featherweight Java, we checked all program states with at most four classes, where each class can have at most two fields and two methods (in

Benchmark	Max Expression Length	**States Checked**	Time (s)
MiniJava	1	**5**	0.543
	2	**46**	0.485
	3	**94**	0.576
	4	**142**	0.766
	5	**188**	0.966
	6	**246**	1.163
	7	**1153**	4.366
	...	...	...
	15	**3293**	16.598
	31	**7576**	65.752
	63	**16269**	261.394
MiniJava+NonNull	1	**3**	0.201
	2	**8**	0.243
	3	**35**	0.358
	4	**40**	0.390
	5	**67**	0.489
	6	**72**	0.523
	7	**149**	0.840
	...	...	...
	15	**377**	2.336
	31	**833**	7.759
	63	**1745**	31.180
	127	**3569**	143.401
	255	**7217**	748.063
MiniJava+Tainted	1	**3**	0.492
	2	**8**	0.262
	3	**46**	0.475
	4	**51**	0.541
	5	**89**	0.643
	6	**94**	0.866
	7	**199**	1.188
	...	...	...
	15	**505**	2.841
	31	**1117**	9.718
	63	**2341**	38.655
	127	**4789**	171.884
	255	**9685**	843.275

Figure 10. Checking soundness of type system extensions with the technique in Section 4. The results show that checking a type system extension *assuming* the base type system is sound is more efficient than checking the base type system (or therefore the extended type system).

addition to inherited fields and methods), and where every method and the main expression have up to a given maximum length l. For the Ownership Java, we checked all states with at most four heap objects and at most four classes where each class can have at most two owner parameters, two methods, and two fields, and every method and the main expression have up to a given maximum length l.

We report both the number of states explicitly checked by our checker and the time taken by our checker. Note that we did not yet optimize the execution time of our checker, but we report it here nonetheless to provide a rough idea. The results indicate that our approach is feasible and that our model checker achieves significant state space reduction. For example, the number of well typed IMP programs of maximum length 511 is over 2^{786}, but our checker explicitly checks only 3653 states to exhaustively cover this space.

5.2 Model Checking Soundness of Type System Extensions

To test our technique for checking soundness of type system extensions, we first extended Featherweight Java with several features including an explicit heap representation, mutations to objects, null pointers, integer and boolean primitive types, arithmetic and boolean operations on the primitive types, `for` and `while` loops, and an `if` construct. We call the resulting language, which is a subset of Java, MiniJava.

Max AST Height	**Percentage of Errors Caught**
1	**0**
2	**36**
3	**92**
4	**100**
5	**100**
6	**100**
7	**100**

Figure 11. Evaluating the small scope hypothesis. An AST height of 4 was sufficient to catch all the type system errors we introduced into Ownership Java.

We extended MiniJava with two type qualifier extensions: `nonNull` and `tainted`. `nonNull` variables may not have null values assigned to them. The `tainted` qualifier describes the reliability of data—variables that have been declared as untainted may not have tainted data assigned to them and objects which are untainted may not have tainted fields. For each language, we checked all states with at most four heap objects and at most four classes where each class can have at most two methods and two fields, and every method and the main expression have up to a given maximum length l.

The results in Figure 10 show that checking the soundness of a type system extension *assuming* the base type system is sound is far more efficient than checking the soundness of the base type system itself. Also, the number of states checked for each extended type system (not shown in the figure) is the same as the the number of states checked for the base type system because the operational semantics of the languages are identical. Thus, checking the soundness of a type system extension is significantly more efficient than checking the soundness of the extended type system.

5.3 Validating the Small Scope Hypothesis

Finally, Figure 11 presents the experimental results that suggest that exhaustive testing within a small finite domain does indeed catch all type system errors in practice, a conjecture also known as the *small scope hypothesis* [28, 29, 36]. We introduced twenty different errors into the type system of Ownership Java (one at a time) and five different errors into the operational semantics. Some were small mistakes such as forgetting to include a type checking clause. Some were deeper errors as the following examples illustrate.

The Java compiler rejects as ill typed a term containing a type cast of a value of declared type `T1` into a type `T2` if `T1` is neither a subtype nor supertype of `T2`. The Ownership Java (as also the Featherweight Java) compiler, however, accepts such a term as well typed. We changed Ownership Java to reject such casts as ill typed. Our model checker then correctly detected that the preservation theorem does not hold for the changed language. The term `(T2) (Object<world>) new T1()` provides a counter example. It is well typed initially. But after a small step of evaluation, the term simplifies to `(T2) new T1()` which is ill typed in the changed language. The preservation theorem therefore does not hold.

We also introduced a subtle bug (c.f. [4, Figure 24]) into Ownership Java such that the *owners as dominators* property does not hold. Our checker correctly detected the bug.

The results in Figure 11, while preliminary, do indicate that exhaustive testing within a small finite domain is an effective approach for checking soundness of type systems. Moreover, we examined all the type soundness errors we came across in literature and found that in each case, there is a small program state that exposes the error. This lends credibility to the validity of the small scope hypothesis in practice.

6. Related Work

This section presents related work on software model checking. Model checking is a formal verification technique that exhaustively tests a circuit/program on all possible inputs (sometimes up to a given size) to handle *input nondeterminism* and all possible nondeterministic schedules to handle *scheduling nondeterminism*. There has been much research on model checking of software. Verisoft [20] is a stateless model checker for C programs. Java PathFinder (JPF) [40] is a stateful model checker for Java programs. XRT [22] checks Microsoft CIL programs. Bogor [15] is an extensible framework for building software model checkers. CMC [31] is a stateful model checker for C programs that has been used to test large software including the Linux implementation of TCP/IP and the ext3 file system.

For hardware, model checkers have been successfully used to verify fairly complex finite state control circuits with up to a few hundred bits of state information; but not circuits that have large data paths or memories. Similarly, for software, model checkers have been primarily used to verify control-oriented programs (with scheduling nondeterminism) with respect to temporal properties; but not much work has been done to verify data-oriented programs (with input nondeterminism) with respect to complex data-dependent properties.

Thus, most of the research on reducing the state space of a software model checker has focused on checking programs with scheduling nondeterminism. Tools such a Slam [2], Blast [24], and Magic [7] use heuristics to construct and check an abstraction of a program (usually predicate abstraction [21]). Abstractions that are too coarse generate false positives, which are then used to refine the abstraction and redo the checking. This technique is known as Counter Example Guided Abstraction and Refinement or *CEGAR*. There are also many static [20] and dynamic [18] partial order reduction systems for concurrent programs. There are many other symmetry-based reduction techniques as well (e.g., [26]). However, none of the above techniques seem to be effective in reducing the state space of a model checker when checking the soundness of a type system—where one must deal with input nondeterminism (to check every input program state) and data-dependent properties (type correctness properties that depend on input program states). In fact, because of input nondeterminism, it is difficult to even formulate the problem of checking type soundness automatically in the context of most software model checkers.

Tools such as Alloy [27] and Korat [3] systematically generate all test inputs that satisfy a given precondition. However, these tools generate and test every valid state and so do not achieve as much state space reduction as our system.

This paper builds on our recent previous work on model checking properties of data structures [12]. While our previous paper focused on checking properties of tree-based data structures, this paper focuses on checking soundness of type systems. It improves on the techniques in [12] by using an incremental SAT solver instead of a BDD to better handle non-tree-based constraints and by using a static analysis to be sound even with non-tree-based constraints. It also includes several domain specific optimizations, such as efficiently checking the soundness of type system extensions and special support to clone terms in an operational semantics.

A recent paper [8] describes a technique for checking properties of programming languages specified in αProlog, using a bounded backtracking search in an αProlog interpreter. However, [8] does not use our state space reduction techniques and does not scale nearly as well as our model checker.

7. Conclusions

This paper presents a software model checker that *automatically* checks the soundness of a type system, given only a specification of type correctness of intermediate program states and the small step operational semantics. Other proofs of type soundness are done either by hand or are machine checked, but require significant manual assistance in both cases. This paper also presents an approach for checking the soundness of a type system extension *assuming* that the base type system is sound. This approach is significantly more efficient than checking the soundness of the extended type system without making the above assumption.

The paper presents techniques that significantly reduce the state space of a model checker for checking type soundness. This paper thus makes contributions both in the area of checking soundness of type systems, and in the area of reducing the state space of a software model checker.

We have tested our system on several small to medium sized languages that include several features such as term and type level substitution, explicit heap, objects, etc., and found our approach to be feasible. We expect our system to be particularly useful to researchers who design novel type systems, but formalize only a core subset of their type systems, as is the standard practice in the research community.

We have not yet tested our system on large realistic languages, partly because of effort it takes to formalize such languages. We plan to explore that in our future work. We have some evidence that our system might scale reasonably well because the number of states explicitly checked by our tool seems to increase only linearly with the number of syntactic constructs in a language. Our evidence also suggests that our tool might work well for checking small extensions to large languages, assuming the large language is sound.

References

[1] B. E. Aydemir et al. Mechanized metatheory for the masses: The POPLMARK challenge, May 2005. http://www.cis.upenn.edu/ plclub/wiki-static/poplmark.-pdf.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In

International Symposium on Software Testing and Analysis (ISSTA), July 2002. Winner of an ACM SIGSOFT distinguished paper award.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[5] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[6] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys 24(3)*, 1992.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *International Conference on Software Engineering (ICSE)*, June 2003.

[8] J. Cheney and A. Momigliano. Mechanized metatheory model-checking. In *Principle and Practice of Declarative Programming (PPDP)*, July 2007.

[9] B. Chin, S. Markstrum, and T. D. Millstein. Semantic type qualifiers. In *Programming Language Design and Implementation (PLDI)*, June 2005.

[10] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[12] P. Darga and C. Boyapati. Efficient software model checking of data structure properties. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2006.

[13] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[14] S. Drossopoulou and S. Eisenbach. Java is type safe—probably. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1997.

[15] M. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *Computer Aided Verification (CAV)*, January 2005.

[16] N. Een and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, June 2005.

[17] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)*, June 2000.

[18] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages (POPL)*, January 2005.

[19] J. S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *Programming Language Design and Implementation (PLDI)*, May 1999.

[20] P. Godefroid. Model checking for programming languages using VeriSoft. In *Principles of Programming Languages (POPL)*, January 1997.

[21] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, June 1997.

[22] W. Grieskamp, N. Tillmann, and W. Shulte. XRT—Exploring runtime for .NET: Architecture and applications. In *Workshop on Software Model Checking (SoftMC)*, July 2005.

[23] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[24] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Principles of Programming Languages (POPL)*, January 2002.

[25] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1999.

[26] C. N. Ip and D. Dill. Better verification through symmetry. In *Computer Hardware Description Languages*, April 1993.

[27] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[28] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering (TSE) 22(7)*, July 1996.

[29] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report TR-921, MIT Laboratory for Computer Science, September 2003.

[30] M. Musuvathi and D. Dill. An incremental heap canonicalization algorithm. In *SPIN workshop on Model Checking of Software (SPIN)*, August 2005.

[31] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. Dill. CMC: A pragmatic approach to model checking real code. In *Operating System Design and Implementation (OSDI)*, December 2002.

[32] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages (POPL)*, January 1999.

[33] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)*, January 2002.

[34] T. Nipkow and D. von Oheimb. Java light is type-safe—definitely. In *Principles of Programming Languages (POPL)*, January 1998.

[35] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction (CC)*, April 2003.

[36] J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, October 2000.

[37] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[38] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for Java. *Transactions on Programming Languages and Systems (TOPLAS) 25(4)*, July 2003.

[39] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. In *International Conference on Functional Programming (ICFP)*, October 2007.

[40] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated Software Engineering (ASE)*, September 2000.

[41] D. Walker. A type system for expressive security policies. In *Principles of Programming Languages (POPL)*, January 2000.

[42] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[43] X. Zhang and R. Gupta. Cost effective dynamic program slicing. In *Programming Language Design and Implementation (PLDI)*, June 2004.