Banshee: A Scalable Constraint-Based Analysis Toolkit*

John Kodumal[1] and Alex Aiken[2]

[1] EECS Department, University of California, Berkeley
[2] Computer Science Department, Stanford University

Abstract. We introduce BANSHEE, a toolkit for constructing constraint-based analyses. BANSHEE's novel features include a code generator for creating customized constraint resolution engines, incremental analysis based on backtracking, and fast persistence. These features make BANSHEE useful as a foundation for production program analyses.

1 Introduction

Program analyses that are simultaneously scalable, accurate, and efficient remain expensive to develop. One approach to lowering implementation cost is to express the analysis using *constraints*. Constraints separate analysis *specification* (constraint generation) from analysis *implementation* (constraint resolution). By exploiting this separation, designers can benefit from existing algorithms for constraint resolution. This separation helps, but leaves several problems unaddressed. A generic constraint resolution implementation with no knowledge of the client may pay a large performance penalty for generality. For example, the fastest hand-written version of Andersen's analysis [12] is much faster than the fastest version built using a generic toolkit [2]. Furthermore, real build systems require separate analysis to fit with separate compilation. Small edits to projects are the norm; reanalyzing an entire project for each small change is unrealistic.

We have built BANSHEE, a constraint-based analysis toolkit that addresses these problems [14]. BANSHEE succeeds BANE, our first generation toolkit for constraint-based program analysis [2]. BANSHEE inherits several features from BANE, particularly support for *mixed constraints*, which allow several constraint formalisms to be combined in one application (Section 2). BANSHEE also provides a number of innovations over BANE that make it more useful and easier to use:

– We use a code generator to specialize the constraint back-end for each program analysis (Section 3). The analysis designer describes a set of constructor signatures in a specification file, which BANSHEE compiles into a specialized

* This research was supported in part by NASA Grant No. NNA04CI57A; NSF Grant Nos. CCR-0234689, CCR-0085949, and CCR-0326577. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

constraint resolution engine. Specialization allows checking a host of correctness conditions statically. Software maintenance also improves: specialization allows the designer to modify the specification without wholesale changes to the handwritten part of the analysis. Finally, BANSHEE is truly modular; new constraint formalisms (or *sorts*) can be added without changing existing sorts.

- We have added support for a limited form of incremental analysis via *backtracking*, which allows constraint systems[3] to be rolled back to any previous state (Section 4). Backtracking can be used to analyze large projects incrementally: when a source file is modified, we roll back the constraint system to the state just before the analysis of that file. By choosing the order in which files are analyzed to exploit locality among file edits, we show experimentally that backtracking is very effective in avoiding reanalysis of files.
- We have added support for efficient serialization and deserialization of constraint systems (Section 5). The ability to save and load constraint systems is important for integrating BANSHEE-derived analysis into real build processes as well as for supporting incremental analysis. This feature is nontrivial, especially in conjunction with backtracking; our solution exploits BANSHEE's use of explicit regions for memory management.
- We have written BANSHEE from the ground up, implementing all important optimizations in BANE, while the code generation framework has enabled us to add a host of engineering and algorithmic improvements. In a case study, we show how BANSHEE's specification mechanism allows various points-to analyses to be easily expressed (Section 6) while the performance is nearly 100 times faster than BANE on some standard benchmarks (Section 7).

BANSHEE has reached the point of being a productive tool for developing experimental and production-quality program analyses. As evidence, we cite several BANSHEE applications. A BANSHEE-based polyvariant binding-time analysis for partial evaluation of graphics programs has been used in production at a major effects studio [19, 20]. BANSHEE has been used as part of a software updateability analysis tool [27]. A BANSHEE-based type inference system for Prolog has been developed [23]. Also, for two years a BANSHEE pointer analysis was used as a prototype global alias analysis in a development branch of the `gcc` compiler.

2 Mixed Constraints

BANSHEE is built on *mixed constraints*, which allow multiple constraint *sorts* in one application. A sort s is a tuple $(V_\mathsf{s}, C_\mathsf{s}, O_\mathsf{s}, R_\mathsf{s})$ where V_s is a set of variables, C_s is a set of constructors, O_s is a set of operations, and R_s is a set of constraint relations. Each n-ary constructor $c_\mathsf{s} \in C_\mathsf{s}$ and operation $op_\mathsf{s} \in O_\mathsf{s}$ has a signature $\iota_1 \dots \iota_n \rightarrow \mathsf{s}$ where ι_i is either s_i or $\bar{\mathsf{s}}_i$ for sorts s_i. Overlined arguments in a signature are contravariant; all other arguments are covariant. A n-ary constructor

[3] Note that BANSHEE's solvers are all online, so existing constraints are maintained in a partially solved form as new constraints are added.

$$\begin{array}{rr}
\mathcal{C} \wedge \{\mathcal{X} \subseteq_{\mathsf{Set}} \mathcal{X}\} \rightarrow \mathcal{C} & \mathcal{C} \wedge \{c(\ldots) \subseteq_{\mathsf{Set}} d(\ldots)\} \rightarrow \text{inconsistent} \\
\mathcal{C} \wedge \{e_{\mathsf{Set}} \subseteq_{\mathsf{Set}} 1\} \rightarrow \mathcal{C} & \mathcal{C} \wedge \{c(\ldots) \subseteq_{\mathsf{Set}} 0\} \rightarrow \text{inconsistent} \\
\mathcal{C} \wedge \{0 \subseteq_{\mathsf{Set}} e_{\mathsf{Set}}\} \rightarrow \mathcal{C} & \mathcal{C} \wedge \{1 \subseteq_{\mathsf{Set}} c(\ldots)\} \rightarrow \text{inconsistent} \\
 & \mathcal{C} \wedge \{1 \subseteq_{\mathsf{Set}} 0\} \rightarrow \text{inconsistent} \\
\mathcal{C} \wedge \{c(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) \subseteq_{\mathsf{Set}} c(e'_{\mathsf{s}_1}, \ldots, e'_{\mathsf{s}_n})\} \rightarrow & \\
 & \mathcal{C} \wedge \bigwedge_i \begin{cases} \{e_{\mathsf{s}_i} \subseteq_{s_i} e'_{\mathsf{s}_i}\} & c \text{ covariant in } i \\ \{e'_{\mathsf{s}_i} \subseteq_{s_i} e_{\mathsf{s}_i}\} & c \text{ contravariant in } i \end{cases}
\end{array}$$

Fig. 1. Constraint resolution for the Set sort.

c_{s} is *pure* if the sort of each of its arguments is s. Otherwise, c_{s} is *mixed.* For a sort s, a set of variables, constructors, and operations defines a language of s-expressions e_{s}:

$$\begin{array}{lll}
e_{\mathsf{s}} ::= & & \\
\quad | & v & v \in V_{\mathsf{s}} \\
\quad | & c_{\mathsf{s}}(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) & c_{\mathsf{s}} \text{ with signature } \iota_1 \ldots \iota_n \rightarrow \mathsf{s} \\
 & & \text{and } \iota_i \text{ is } \mathsf{s}_i \text{ or } \bar{\mathsf{s}}_i \\
\quad | & op_{\mathsf{s}}(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) & op_{\mathsf{s}} \text{ with signature } \iota_1 \ldots \iota_n \rightarrow \mathsf{s} \\
 & & \text{and } \iota_i \text{ is } \mathsf{s}_i \text{ or } \bar{\mathsf{s}}_i
\end{array}$$

Constraints between expressions are written $e_{1_{\mathsf{s}}}\ r_{\mathsf{s}}\ e_{2_{\mathsf{s}}}$ where r_{s} is a constraint relation ($r_{\mathsf{s}} \in R_{\mathsf{s}}$). Each sort s has two distinguished constraint relations: an inclusion relation (denoted $\subseteq_{\mathsf{s}}$) and a unification relation (denoted $=_{\mathsf{s}}$). A constraint system $\mathcal{C}$ is a finite conjunction of constraints.

To fix ideas, we introduce two BANSHEE sorts and informally explain their semantics. A formal presentation of the semantics of mixed constraints is given in [8]. We leave the set of constructors C_{s} unspecified in each example, as this set parameterizes the constraint language and is application-specific.

Example 1 *The Set sort is the tuple:* $(V_{\mathit{Set}}, C_{\mathit{Set}}, \{\cup, \cap, 0, 1\}, \{\subseteq, =\})$.

Here V_{Set} is a set of set-valued variables, $\cup, \cap$, $\subseteq$, and $=$ are the standard set operations, 0 is the empty set, and 1 is the universal set. Each pure Set expression denotes a set of *ground terms*: a constant or a constructor $c_{\mathsf{Set}}(t_1, \ldots, t_n)$ where each t_i is a ground term. A subset of the resolution rules for the Set sort is shown in Figure 1; BANSHEE implements these as left-to-right rewrite rules.

Example 2 *The Term sort is the tuple:* $(V_{\mathit{Term}}, C_{\mathit{Term}}, \{0, 1\}, \{\leq, =\})$.

Here V_{Term} is a set of term-valued variables, and $=$ and $\leq$ are unification and conditional unification [25], respectively. The meaning of a pure Term expression is, as expected, a constant or a constructor $c_{\mathsf{Term}}(t_1, \ldots, t_n)$ where t_i are terms. A subset of the resolution rules for the Term sort is shown in Figure 2.[4]

[4] We use "term" to mean both a sort and ground terms (trees) built by constructor application. The intended meaning should be clear from context.

$$
\begin{array}{rll}
\mathcal{C} \wedge \{\mathcal{X} =_{\mathsf{Term}} \mathcal{X}\} & \rightarrow \mathcal{C} & \\
\mathcal{C} \wedge \{c(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) =_{\mathsf{Term}} c(e'_{\mathsf{s}_1}, \ldots, e'_{\mathsf{s}_n})\} & \rightarrow \mathcal{C} \wedge \bigwedge_i \{e_{\mathsf{s}_i} =_{\mathsf{s}_i} e'_{\mathsf{s}_i}\} & \\
\mathcal{C} \wedge \{e_{\mathsf{Term}} \leq e'_{\mathsf{Term}}\} & \rightarrow \mathcal{C} \wedge \{e_{\mathsf{Term}} = e'_{\mathsf{Term}}\} & \text{if } e_{\mathsf{Term}} \text{ is not } 0 \\
& \mathcal{C} & \text{if } e_{\mathsf{Term}} \text{ is } 0 \\
\mathcal{C} \wedge \{c(\ldots) =_{\mathsf{Term}} d(\ldots)\} & \rightarrow \text{inconsistent} &
\end{array}
$$

Fig. 2. Constraint resolution for the Term sort

A system of mixed constraints defines a directed graph where nodes are expressions and edges are *atomic constraints* between expressions. A constraint is atomic if the left- or right-hand side is a variable. To solve the constraints, the constraint graph is closed under the resolution rules for each sort as well as a transitive closure rule.

3 Specialization

This section describes the compilation strategy used in Banshee. We omit the low-level details, which are straightforward, and focus on explaining the advantages of our approach.

To use Banshee, the analysis designer writes a specification file defining the constructor signatures for the analysis. Consider a constructor *fun* modeling a function type in a unification-based type inference system with an additional set component to track the function's latent effect, in the style of a type and effect system. The signature is:

$$fun : \mathsf{Term} \ * \ \mathsf{Term} \ * \ \mathsf{Set} \rightarrow \mathsf{Term}$$

which is specified in Banshee as follows (see Section 6 for more explanation):

```
data l_type : term = fun of l_type * l_type * effect
and effect : set
```

In Bane, this signature can be declared at run-time, even during constraint resolution. Bane is an interpreter for a language of constructors and resolution rules, and as such it has the overhead of an interpreter. For example, to apply the constructor *fun*, Bane checks at run-time that there are the right number of arguments of the correct sorts. There is also interpretive overhead in constraint resolution. Consider implementing the rule for constraints between two *fun* expressions:

$$
\begin{array}{l}
\mathcal{C} \wedge \{fun(e_{\mathsf{Term}_1}, e_{\mathsf{Term}_2}, e_{\mathsf{Set}}) =_{\mathsf{Term}} fun(e'_{\mathsf{Term}_1}, e'_{\mathsf{Term}_2}, e'_{\mathsf{Set}})\} \rightarrow \\
\qquad \mathcal{C} \wedge \{e_{\mathsf{Term}_1} =_{\mathsf{Term}} e'_{\mathsf{Term}_1}\} \wedge \{e_{\mathsf{Term}_2} =_{\mathsf{Term}} e'_{\mathsf{Term}_2}\} \wedge \{e_{\mathsf{Set}} =_{\mathsf{Set}} e'_{\mathsf{Set}}\}
\end{array}
$$

To implement this rule, Bane uses the signature to choose the correct constraint relation and the directionality for each component. Because this work

is done dynamically, both require either run-time tests or dynamic dispatch to implement.

From experience we have learned that analyses rely on a small, fixed number of constructors that can be specified statically. BANSHEE uses static signatures to implement customized versions of the constructors and the constraint resolution rules, which allows us to eliminate many kinds of dynamic checks statically. For example, consider again the signature of the *fun* constructor, now declared statically in BANSHEE. From this signature, BANSHEE generates a C function with the following prototype:

```
l_type fun(l_type e0, l_type e1, effect e2)
```

Notice that the dynamic arity and sort checks are no longer necessary—the C type system guarantees that calls to this function have the correct number of arguments (the arity check) and that the types of any actuals match the formal arguments (the sort checks). Similarly, BANSHEE can statically discharge the dynamic checks in resolution rules discussed above.

One of the most important advantages of BANSHEE specifications is that they make program analyses easier to debug and maintain. After writing a BANSHEE specification, the analysis designer's main task is to write C code to traverse abstract syntax, calling functions from the generated interface to build expressions, generate constraints, and extract solutions. This task is typically straightforward, as there should be a tight correspondence between type judgments and the BANSHEE code to implement them. Continuing with the type and effect example, we might wish to implement the following rule for function application:

$$\frac{\Gamma \vdash e_1 : \tau_1 ; \epsilon_1 \quad \Gamma \vdash e_2 : \tau_2 ; \epsilon_2 \quad \tau_1 = \tau_2 \to^{\epsilon} \alpha \quad \alpha, \epsilon \text{ fresh}}{\Gamma \vdash e_1\ e_2 : \alpha ; \epsilon_1 \cup \epsilon_2 \cup \epsilon} \text{ (App)}$$

Assuming a typical set of AST definitions, the corresponding BANSHEE code to implement this rule is[5]:

```
struct type_and_effect analyze(env gamma, ast_node n) {
  if (is_app_node(n)) {
    l_type tau1, tau2, alpha;
    effect epsilon1, epsilon2, epsilon;
    (tau1, epsilon1) = analyze(gamma,n->e1);
    (tau2, epsilon2) = analyze(gamma,n->e2);
    alpha = l_type_fresh();
    epsilon = effect_fresh();
    l_type_unify(tau1, fun(tau2,alpha,epsilon));
    return (alpha, effect_union([epsilon1;epsilon2;epsilon]));
  }
```

[5] We use a little syntactic sugar for pairs and lists in C to avoid showing the extra type declarations.

```
  ...
}
```

This code is representative of the "handwritten" part of a Banshee analysis. Banshee's code generator makes the handwritten part clean: there is a close correspondence between clauses in the type rules and the Banshee code to implement them.

4 Incremental Analysis

Incremental analysis is important in large projects where it is necessary to maintain a global analysis in the face of small edits. In this section we describe Banshee's support for a form of incremental analysis via *backtracking*, a mechanism that is efficient, simple to implement, and applicable to a wide variety of program analysis systems besides Banshee.

We assume *constraint additions*, *constraint deletions*, and *queries* (testing whether the constraints satisfy some fact) are arbitrarily interleaved. Additions and queries are handled by on-line solving; the trick is handling deletions.

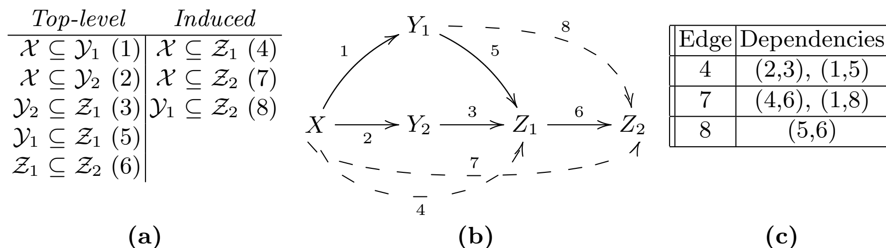
Consider adding constraints (1)-(3) in Figure 3(a) to an initially empty constraint system. Constraints (2)-(3) cause constraint (4) to be added (by transitive closure). We say (1)-(3) are *top-level* constraints (added by the user, solid lines in the constraint graph in Figure 3(b)) and (4) is an *induced* constraint (added by the closure rules, dashed lines in Figure 3(b)).

At this point, if we delete constraint (2), then constraint (4) must be deleted as well, as it would no longer be included in the closed constraint graph. We say constraint(4) *depends on* constraints (2) and (3). The key to incremental analysis is tracking such dependency information.

A straightforward way to track precise dependency information is to explicitly maintain a list of constraints on which each induced constraint depends. For example, adding constraint (5) adds edge (4) again, so the entry (1,5) must be included in (4)'s dependency list. This approach is costly in space, because an induced constraint often is added multiple times [28]. Figures 3(b) and (c) show the closed constraint graph and edge dependencies after constraint (6) and its induced constraints are added to the graph.

Besides the space cost, another major concern is the engineering effort required to support fine-grained incrementality. To our knowledge, there is no general, practical incremental algorithm for maintaining arbitrary data structures [7]. Adding ad-hoc support for incremental updates to each Banshee sort is daunting, as the algorithms are highly optimized. For example, our set constraint solver uses a union-find algorithm to implement partial online cycle elimination [9]. Adding incremental support to union-find alone is not easy—in fact, some published solutions are incorrect [10].

Instead of computing precise dependencies, we use *backtracking*, which is based on an approximation: each induced constraint depends on all constraints introduced earlier. Thus, to delete constraint c, we delete c and all induced



Top-level	*Induced*
$\mathcal{X} \subseteq \mathcal{Y}_1$ (1)	$\mathcal{X} \subseteq \mathcal{Z}_1$ (4)
$\mathcal{X} \subseteq \mathcal{Y}_2$ (2)	$\mathcal{X} \subseteq \mathcal{Z}_2$ (7)
$\mathcal{Y}_2 \subseteq \mathcal{Z}_1$ (3)	$\mathcal{Y}_1 \subseteq \mathcal{Z}_2$ (8)
$\mathcal{Y}_1 \subseteq \mathcal{Z}_1$ (5)	
$\mathcal{Z}_1 \subseteq \mathcal{Z}_2$ (6)	

Edge	Dependencies
4	(2,3), (1,5)
7	(4,6), (1,8)
8	(5,6)

Fig. 3. (a) Constraints. (b) Constraint graph. (c) Edge dependency list.

constraints added after c. Because this notion of dependency is approximate, we must solve the resulting constraint system to rediscover induced constraints that should not have been deleted.

Backtracking is implemented by time stamping constraints. Deleting constraint i is done by scanning all constraints (edges), deleting any induced constraint with a timestamp greater than i, and solving. Consider again the example in Figure 3, and take a constraint's number to be its timestamp. We see that deleting constraint (6) also deletes constraints (7) and (8), but because both (7) and (8) only depend on (6), backtracking is as precise as tracking edge dependencies in this case. Going further and also deleting constraint (3), however, deletes induced constraint (4), which is rediscovered through the transitive path (1,5) when the resulting system is solved. While backtracking can overestimate the set of deleted constraints and incur extra work in rediscovering induced constraints, it has practical advantages over computing edge dependencies. Backtracking uses a linear scan of the constraint graph's edges, while the precise incremental algorithm is linear in the size of edge dependency lists, which may be quadratic in the number of graph edges. The storage overhead of backtracking is just a timestamp per edge, while edge dependency lists raise the worst case storage for an n-node graph from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^3)$.

We have also devised a new data type called a *tracked reference* that adds efficient backtracking support to general data structures. This abstraction simplified the task of incorporating backtracking into Banshee, especially in the presence of optimizations like cycle elimination and projection merging [9, 28]. A tracked reference is a mutable reference that maintains a repository of its old versions. Each tracked reference is tied to a clock; each tick of the clock checkpoints the reference's state. When the clock is rolled back, the previous state is restored. Rolling back is a destructive operation. Implementing tracked references is simple: it suffices to maintain a stack of the references' old contents. A backtracking operation pops entries off the stack until the last clock tick. Appendix A includes a compilable O'Caml implementation of tracked references. We have implemented backtracking by incorporating tracked references systematically into each Banshee data structure. Interestingly, we do not pay any cost for this factoring. For example, applying tracked references to a standard

union-find algorithm yields an algorithm equivalent to a well-known algorithm for union-find with backtracking [29].

The basic approach to adding backtracking to a static analysis is as follows. Given a fully analyzed program and a program edit, we backtrack to the first constraint that changed as a result of the edit[6] and re-analyze the program from that point forward. For projects using standard version control systems, it is natural to use file granularity for changes. An interactive development environment may provide granularity, and Banshee itself can backtrack at constraint level granularity. Thus, we maintain a stack of analyzed files. If a file is modified, we pop the stack to that file, backtrack, and re-analyze all popped files.

Files are pushed back on to the stack in the order they are (re-)analyzed, but we have the flexibility to choose this order. We believe there is locality in program modifications: developers work on one or a few files at a time, and new code is more likely to be modified than old, stable code. When reanalyzing files, the order of files on the stack is preserved except that the modified file is analyzed last, thus placing it at the top of the stack, reflecting the belief that it is most likely to be the next file modified. As long as there is locality among edits, edited files will on average be close to the top of the stack under this strategy.

5 Persistence

We briefly explain our approach to making Banshee's constraint systems persistent. Persistence is useful when incorporating incremental analyses into standard build processes. We require persistence (rather than a feature to save and load in some simpler format) as we must reconstruct the representation of our data structures to support online constraint solving and backtracking.

Persistence is achieved by adding serialization and deserialization to the region-based memory management library used by Banshee [11]. Constraint systems are saved by serializing a collection of regions, and loaded by deserializing regions and updating pointer values stored in those regions. Initially, we implemented serialization using a standard pointer tracing approach, but found this strategy to be very slow because pointer tracing has poor spatial locality. Region-based serialization writes sequential pages of memory to disk, which is orders of magnitude faster. To handle deserialization, we associate an update function with each region, which is called on each object in the region to update any pointer-valued fields. With region-based serialization, we are able to serialize a 170 MB constraint graph in 2.4 seconds vs. 30 seconds to serialize the same graph by tracing pointers.

6 Case Study: Points-to Analysis

We continue with realistic examples derived from points-to analyses formulated in Banshee, showing how Banshee can be used to explore different design

[6] Here we also remove top-level constraints that may have changed due to the edit.

$$\frac{\Gamma(x) = ref(\ell_x, \mathcal{X}_{\ell_x}, \overline{\mathcal{X}}_{\ell_x})}{\Gamma \vdash x : ref(\ell_x, \mathcal{X}_{\ell_x}, \overline{\mathcal{X}}_{\ell_x})} \text{ (Var)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \&e : ref(0, \tau, \overline{1})} \text{ (Addr)}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau \subseteq ref(1, \mathcal{T}, \overline{0})}{\Gamma \vdash *e : \mathcal{T}} \text{ (Deref)} \qquad \frac{\begin{array}{cc} \Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \tau_2 \\ \tau_1 \subseteq ref(1, 1, \overline{\mathcal{T}}_1) & \tau_2 \subseteq ref(1, \mathcal{T}_2, \overline{0}) \\ \multicolumn{2}{c}{\mathcal{T}_2 \subseteq \mathcal{T}_1} \end{array}}{\Gamma \vdash e_1 \ = \ e_2 : \tau_2} \text{ (Assign)}$$

Fig. 4. Constraint generation for Andersen's analysis

points and prototype variations of a given program analysis. In the first three examples, we refine the degree of subtyping used in the points-to analysis; much research on points-to analysis has focused on this issue [5, 24, 25]. In the fourth example, we extend points-to analysis to receiver class analysis in an object-oriented language with explicit pointer operations (e.g. C++). This analysis computes the function call graph on the fly instead of using a pre-computed call graph obtained from a coarser analysis (e.g., class hierarchy analysis).

6.1 Andersen's Analysis

Andersen's points-to analysis constructs a *points-to graph* from a set of abstract memory locations $\{\ell_1, \ldots, \ell_n\}$ and set variables $\mathcal{X}_{\ell_1}, \ldots, \mathcal{X}_{\ell_n}$. Intuitively, a reference is an object with an abstract location and methods $get : void \to \mathcal{X}_{\ell_x}$ and $set : \mathcal{X}_{\ell_x} \to void$, where $\mathcal{X}_{\ell_x}$ represents the points-to set of the location. Updating the location corresponds to applying the *set* function to the new value. Dereferencing a location corresponds to applying the *get* function. References are modeled by a constructor *ref* with three fields: a constant ℓ_x representing the abstract location, a covariant field $\mathcal{X}_{\ell_x}$ representing the *get* function, and a contravariant field $\overline{\mathcal{X}}_{\ell_x}$ representing the *set* function.

Figure 4 shows a subset of the inference rules for Andersen's analysis for C programs. The type judgments assign a set expression to each program expression, possibly generating some side constraints. To avoid having separate rules for l-values and r-values, each type judgment infers a set denoting an l-value. Hence, the set expression in the conclusion of (Var) denotes the location of program variable x, rather than its contents.

In Banshee, Andersen's analysis is specified as follows:

```
specification andersen : ANDERSEN =
  spec
    data location : set
    data T : set = ref of +location * +T * -T
  end
```

We outline Banshee's specification syntax, which is inspired by ML recursive data type declarations. Each `data` declaration defines a disjoint alphabet of constructors. For example, the declaration `data location : set` defines `location`

to be a collection of constructors of sort Set. The `location` alphabet serves only as a source of fresh constants, modeling the statically unknown set of abstract locations. While static constructor signatures are an important idea in BANSHEE, dynamic sets of constants are useful in many analyses. But all constants have a fixed, known signature, so generating them dynamically does not interfere with any of our static optimizations.

Each `data` declaration may be followed by an optional list of `|`-separated constructor declarations defining the (statically fixed) set of n-ary constructors. In this example, we define a single ternary constructor `ref`, which uses *variance annotations*. A signature element prefixed with `+` (resp. `-`) denotes a covariant (resp. contravariant) field. By default, fields are nonvariant.

6.2 Steensgaard's Analysis and One Level Flow

Andersen's analysis has cubic time complexity. Steensgaard's coarser, near-linear time analysis is implemented using the Term sort. The Andersen's specification is modified by eliminating the duplicate `T` field in the ref constructor and removing variance annotations:

```
specification steensgaard : STEENSGAARD =
  spec
    data location : set
    data T : term = ref of location * T
  end
```

Experience shows the lack of subtyping in Steensgaard's analysis leads to many spurious points-to relations. Another proposal is to use one level of subtyping. Restricting subtyping to one level is nearly as accurate as full subtyping [5]. Altering the specification to support the new analysis is again simple:

```
specification olf : OLF
  spec
    data location : set
    data T : set = ref of +location * T
  end
```

Recall the `location` field models a set of abstract locations. Making this field covariant allows subtyping at the top level. However, the `T` field is nonvariant, which restricts subtyping to the top level: at lower levels, the engine performs unification. An alternative explanation is that this signature implements the following sound rule for subtyping with updateable references [1]:

$$\frac{\ell_x \subseteq \ell_y \qquad \mathcal{X}_{\ell_x} = \mathcal{X}_{\ell_y}}{ref(\ell_x, \mathcal{X}_{\ell_x}) \leq ref(\ell_y, \mathcal{X}_{\ell_y})} \text{ (sub-ref)}$$

6.3 Receiver Class Analysis

Now that we have explored subtyping in points-to analysis, we focus on adding new capabilities to the analysis. We use the points-to information as the basis of a receiver class analysis (RCA) for an object-oriented language with explicit pointer operations. RCA approximates the set of classes to which each expression in the program can evaluate. In a language like C++, the analysis must also use points-to information to track the effects of pointer operations.

In addition to modeling object initialization and pointer operations, our analysis must accurately simulate the effects of method dispatch. To accomplish these tasks, new constructors representing class definitions and dispatch tables are added to our points-to analysis specification. To simplify the example, we assume that methods have a single formal argument in addition to the implicit `this` parameter. Here is the Banshee specification for this example, using Andersen's analysis as our base points-to analysis:

```
specification rca : RCA =
  spec
    data location : set
    data T : set = ref of +location * +T * -T
                 | class of +location * +dispatch
    and dispatch : row(method)
    and method : set = fun of -T * -T * +T
  end
```

The `class` constructor contains a location field containing the name of the class and a `dispatch` field representing the dispatch table for that class' objects. Notice that `dispatch` uses a new sort, Row [8]. We first explain how this abstraction is intended to work before describing the Row sort.

An object's dispatch table is modeled as a collection of methods (each in turn modeled by the `method` constructor) indexed by name. Given a dispatch expression like `e.foo()`, our analysis should compute the set of classes that `e` may evaluate to, search each class's dispatch table for a method named `foo`, and execute it (abstractly) if it exists. Methods are modeled by the `fun` constructor. Methods model the implicit `this` parameter with the first `T` field, the single formal parameter by the second `T` field, and a single return value by the third `T` field. Recall that the function constructor must be contravariant in its domain and covariant in its range, as reflected in the specification.

For this approach to work, our dispatch table abstraction must map between method names and `method` terms, which we do using the Row sort. A Row of base sort `s` (written Row(s)) denotes a partial function from an infinite set of names to terms of sort `s`. Row expressions, which we do not further explain here, are used to model record types with width and depth subtyping.

Figure 5 shows new rules for object initialization and method dispatch. These rules in conjunction with the rules in Figure 4 comprise our receiver class analysis. For a class `C`, rule (New) returns a `class` expression with label ℓ_C. The `dispatch` component of this expression is a row mapping labels ℓ_{m_i} to `methods`

$$\frac{}{\Gamma \vdash \texttt{new C} : ref(0, class(\ell_C, < \ell_{m_i} : fun(\mathcal{X}_{this}, \mathcal{X}_{arg}, \mathcal{X}_{ret}) \ldots >), \overline{1})} \text{ (New)}$$

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \tau_2 \\ \tau_1 \subseteq ref(1, \mathcal{T}_1, \overline{0}) & \tau_2 \subseteq ref(1, \mathcal{T}_2, \overline{0}) \\ \multicolumn{2}{c}{\mathcal{T}_1 \subseteq class(1, < \ell_m : fun(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_{ret}) \ldots >)} \end{array}}{\Gamma \vdash e_1 \texttt{.m}(e_2) : ref(0, \mathcal{T}_{ret}, \overline{1})} \text{ (Dispatch)}$$

Fig. 5. Rules for receiver class analysis (add to the rules from Figure 4).

Benchmark	Description	Preproc LOC	Andersen(s)		
			BANE(+gc)	BANE	BANSHEE
gs	Ghostscript	437211	35.5	27.0	6.9
spice	Circuit simulation program	849258	14.0	11.3	3.0
pgsql	PostgreSQL	1322420	44.8	34.9	6.0
gimp	GIMP v1.1.14	7472553	1688.8	962.9	20.2
linux	Linux v2.4 (default config)	3784959	—	—	54.5

Table 1. Benchmark data for Andersen's analysis

for each method $\texttt{m}_i$ defined in `C`. To remain consistent with the type judgments for Andersen's analysis (where the result of each type judgment is an l-value) we wrap the resulting class in a `ref` constructor. Note that since our analysis is context-insensitive, each instance of `new C` occurring in the program yields the same `class` expression, which is created when the definition of `C` is analyzed. In (Dispatch), e_1 is analyzed and assumed to contain a set of `class`es. For each class defining a method `m` (i.e. the associated `dispatch` row contains a mapping for label ℓ_m), the corresponding method body is selected and constrained so actual parameters flow into the formal parameters and the return value of the function (lifted to an l-value) is the result of the entire expression.

We conclude the case study by noting that BANSHEE can be used to explore many analysis issues that we have not illustrated here. For example, field-sensitive analyses can be implemented using BANSHEE's Row sort to model structures. Polymorphic recursive analyses can be implemented using a BANSHEE-based library for context-free language reachability [15]. BANSHEE also has a modular design that allows new sorts to be added to the system in case an analysis demands a customized set of resolution rules.

7 Experiments

To demonstrate the scalability and performance of BANSHEE, we implemented field- and context-insensitive Andersen's analysis for C and tested it on sev-

eral large benchmarks.[7] We also ran the same analysis implemented with the Bane toolkit. While Bane is written in SML and Banshee in C, among other low-level differences, the comparison does demonstrate Banshee's engineering improvements. Table 1 shows wall clock execution times in seconds for the benchmarks. Benchmark size is measured in preprocessed lines of code (the two largest benchmarks, gimp and Linux, are approximately 430,000 and 2.2 million source lines of code, respectively). We compiled the Linux benchmark using a "default" configuration that answers "yes" to all interactive configuration options. All reported times for this experiment include the time to calculate the transitive lower bounds of the constraint graph, which simulates points-to queries on all program variables [9]. Parse times are not included. Interestingly, a significant fraction of the analysis time for the Bane implementation is spent in garbage collection, which may be because almost all of the objects allocated during constraint resolution (nodes and edges in the constraint graph) are live for the entire analysis. We also report (in the column labeled Bane) the wall clock execution time for Andersen's analysis exclusive of garbage collection. The C front-end used in the Bane implementation cannot parse the Linux source, so no number is reported for that benchmark. Although it is difficult to compare to other implementations of Andersen's analysis using wall-clock execution time, we note that our performance appears to be competitive with the fastest hand-optimized Andersen's implementation for answering all points-to queries [12].

We also evaluated the strategy described for backtracking-based incremental analysis (Section 4) by running Andersen's analysis on CQual, a type qualifier inference tool. CQual contains approximately 45,000 source lines of C code (250,000 lines preprocessed). We chose CQual because of our familiarity with its build process: without manual guidance, it is difficult to compile and analyze multiple versions of a code base spanning several years. We looked at each of the 13 commits made to CQual's CVS repository from November 2003 to May 2004 that modified at least one source file and compiled successfully. For each commit we report three different numbers (Figure 6(a))[8]:

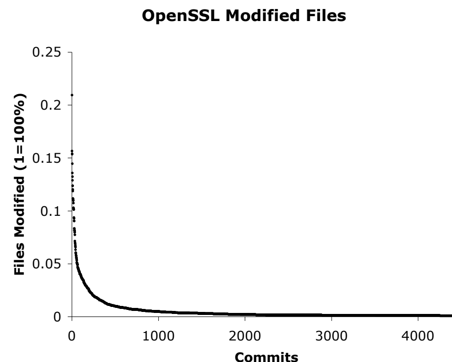
- Column 3: Andersen's analysis run from scratch; this is the analysis time assuming no backtracking is available.
- Column 4: Incremental Andersen's analysis, assuming that analysis of the previous commit is available. To compute this number, we take the previous analysis, backtrack (pop the analysis stack) to the earliest modified file, and re-analyze all files popped off of the stack, placing the modified files on top (as described in Section 4). The initial stack (for the full analysis of the first commit) contained the files in alphabetical order.
- Column 5: Incremental Andersen's analysis, assuming that the files modified during this commit are already on the top of the analysis stack. To compute this number, we pop and reanalyze just the modified files.

[7] All experiments were run on a 2.80 GHz Intel Xeon machine with 4 GB of memory running Red Hat Linux.

[8] These times do not include the time to serialize or deserialize the constraints.

Commit Date	Files Modified	All Files(s)	Cross Commit	Modified Only(s)
11-16	0/58	2.0	-	-
11-17	1/58	2.0	0.9	0.05
12-03	1/58	2.0	0.9	0.15
12-10	1/58	2.0	1.0	0.04
12-11	1/58	2.3	2.0	0.09
12-12	5/58	2.3	1.8	0.51
2-29	1/58	2.0	0.5	0.08
3-05	1/58	2.0	0.9	0.06
3-05	25/59	2.0	2.0	1.0
3-05	5/60	2.0	2.4	0.27
3-11	4/60	2.4	1.0	0.5
3-22	3/61	2.0	0.14	0.14
5-03	2/61	2.0	1.2	0.13

(a) Data for CQual experiment.



(b) OpenSSL files modified per commit.

Fig. 6. Backtracking experiments.

Column 4 gives the expected benefit of backtracking if the analysis is run only once per commit. However, if the developer runs the analysis multiple times before committing (each time the code is compiled, or each time the code is edited in an interactive environment) then Column 5 gives a lower bound on the eventual expected benefit. To see this, assume that a single source file is modified during an editing task. The first time the analysis is run, that source file may be placed on the bottom of the stack, so after a program edit, a complete reanalysis might be required. Subsequently, however, that file is on top of the stack and we only pay the cost of reanalyzing a single file. In general, if n files are modified in an editing task, at worst we analyze the entire code base n times (to move each file one by one from the bottom to the top of the stack) and subsequently only pay (at most) the cost to reanalyze the n modified files.

Backtracking, then, can be an effective incremental analysis technique as long as only a small fraction of the files in a code base is modified per editing task. Figure 6(a) shows this property holds for CQual. To test this hypothesis on a larger, more active code base with more than a few developers, we looked at the CVS history for OpenSSL, which contains over 4000 commits that modify source code. For each commit, we recorded the percentage of source files modified. Figure 6(b) shows a plot of the sorted data. The percentage of files modified obeys a power law: very infrequently, between 5 and 25 percent of the files are modified, but in the common case, less than .1 percent of the files are modified. We have confirmed similar distributions hold for other code bases as well.

8 Related Work

Many related frameworks have been used to specify static analyses. In [26], modal logic is used as a specification language to compile specialized implementations

of dataflow analyses. Datalog [4] is a database query language based on logic programming that has recently received attention as a specification language for static analyses. The subset of pure set constraints implemented in BANSHEE is equivalent to chain datalog [31] and also context-free language reachability [18]. There are also obvious connections to bottom-up logic [17]. Implementations of these frameworks have been applied to solve static analysis problems. The bddbddb system is a deductive database that uses a binary-decision diagram library as its back-end [30]. Other toolkits that use BDD back-ends include CrocoPat [3] and Jedd [16]. An efficient algorithm for Dyck context-free language reachability has been shown to be useful for solving various flow analysis problems [21]. A demand-driven version of the algorithm also exists [13], though we have not so far seen a fully incremental algorithm described. Our description of a precise incremental algorithm, as well as our backtracking algorithm, can be applied to Dyck-CFLR problems via a reduction in [15].

We are not aware of previous work on incrementalizing set constraints, though work on incrementalizing transitive closure is abundant and addresses related issues [6, 22]. The CLA (compile, link, analyze) [12] approach to analyzing large code bases supports a form of file-granularity incrementality similar to traditional compilers: modified files can be recompiled and linked to any unchanged object files. This approach has some advantages. For example, since CLA doesn't save any analysis results, object file formats are simpler, and there is no need for persistence. However, CLA defers all analysis work until after the link phase, so the only savings is the cost of parsing and producing object files.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
2. A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, 1998.
3. D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 216. IEEE Computer Society, 2003.
4. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
5. M. Das. Unification-based pointer analysis with directional assignments. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2000.
6. C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 381. IEEE Computer Society, 2000.
7. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109–121, 1986.

8. M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 114–126, London, United Kingdom, 1997. Springer-Verlag.
9. M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998.
10. Z. Galil and G. F. Italiano. A note on set union with arbitrary deunions. *Information Processing Letters*, 37(6):331–335, 1991.
11. D. Gay and A. Aiken. Language support for regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.
12. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.
13. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 104–115. ACM Press, 1995.
14. J. Kodumal. Banshee: A toolkit for constructing constraint-based analyses. http://banshee.sourceforge.net, 2005.
15. J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 207–218. ACM Press, 2004.
16. O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM Press, 2004.
17. D. McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.
18. D. Melski and T. Reps. Interconvertbility of set constraints and context-free language reachability. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 74–89. ACM Press, 1997.
19. J. Ragan-Kelley. Personal communication, November 2004.
20. J. Ragan-Kelley. *Practical Interactive Lighting Design for RenderMan Scenes*. Undergraduate thesis, Stanford University, Department of Computer Science, 2004.
21. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, January 1995.
22. L. Roditty. A faster and simpler fully dynamic transitive closure. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 404–412. Society for Industrial and Applied Mathematics, 2003.
23. A. Sayeed. Proshee. http://proshee.sourceforge.net, 2005.
24. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1997.
25. B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, Jan. 1996.
26. B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21(2):115–139, 1993.

27. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 183–194, 2005.
28. Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95. ACM Press, 2000.
29. J. Westbrook and R. E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM Journal on Computing*, 18(1):1–11, 1989.
30. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, June 2004.
31. M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 230–242. ACM Press, 1990.

A Tracked Reference Implementation

We include a complete listing of the tracked reference datatype in OCaml.

```
module S = Stack
exception Tick

type clock = {
  mutable time : int;
  repository : (unit -> unit) S.t
}
type 'a tref = clock * 'a ref

let tref (clk: clock) (v :'a) : 'a tref =
  (clk, ref v)
let read (clk,r: 'a tref) : 'a =
  !r
let write (clk,r: 'a tref) (v : 'a) : unit =
  let old_v = !r in
  let closure = fun () -> r := old_v in
    begin
      S.push closure clk.repository;
      r := v
    end
let clock () : clock =
  {
    time = 0;
    repository = S.create ()
  }
let time (clk : clock) : int =
  clk.time
let tick (clk : clock) : unit =
  let closure =
    fun () -> raise Tick in
    begin
      S.push closure clk.repository;
      clk.time <- clk.time + 1;
    end
let rollback (clk : clock) : unit =
  try
    while (not (S.is_empty clk.repository)) do
      let closure = (S.pop clk.repository) in
        closure()
    done
  with Tick -> (clk.time <- clk.time - 1)
```