Dynamic Inference of Abstract Types

Philip J. Guo Jeff H. Perkins Stephen McCamant Michael D. Ernst
MIT Computer Science and Artificial Intelligence Lab
Cambridge, MA, USA
{pgbovine,jhp,smcc,mernst}@csail.mit.edu

Abstract

An abstract type groups variables that are used for related purposes in a program. We describe a dynamic unification-based analysis for inferring abstract types. Initially, each run-time value gets a unique abstract type. A run-time interaction among values indicates that they have the same abstract type, so their abstract types are unified. Also at run time, abstract types for variables are accumulated from abstract types for values. The notion of interaction may be customized, permitting the analysis to compute finer or coarser abstract types; these different notions of abstract type are useful for different tasks. We have implemented the analysis for compiled x86 binaries and for Java bytecodes. Our experiments indicate that the inferred abstract types are useful for program comprehension, improve both the results and the run time of a follow-on program analysis, and are more precise than the output of a comparable static analysis, without suffering from overfitting.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Languages, Experimentation

Keywords

abstract types, type inference, dynamic analysis, C, C++, Java, interaction, mixed-level analysis, units, values and variables

1. Introduction

Even in explicitly-typed languages, the declared types capture only a portion of the programmer's intent and knowledge about variable values. For example, a programmer may use the `int` type to represent array indices, sensor measurements, the current time, file descriptors, counts, memory addresses, and a host of other unrelated quantities. The type `Pair<int,int>` can represent the coordinates of a point, a Celsius/Fahrenheit conversion, a quotient and remainder returned from a division procedure, etc. Different strings or files can represent distinct concepts. Regular expressions can be applicable to different contents. Variables declared with the same generic type, such as `Object` or `Comparable`, need not hold related values. Figure 1 contains an example.

Use of a single programming language type obscures the differences among conceptually distinct values. This can hinder programmers in understanding, using, and modifying the code, and can hinder tools in performing analyses on code. Therefore, it is desirable to recover finer-grained type information than is expressed in the declared types. We call these finer types *abstract types*; this paper presents a dynamic analysis for inferring abstract types.

Abstract types are useful for program understanding [25].

- They can identify ADTs (abstract data types) by indicating which instances of a declared type are related and which are independent.
- They can reveal abstraction violations when the inferred abstract types unify values that should be separate, as indicated by either the declared types or a programmer's expectations.
- They can indicate where a value may be referenced (only at expressions that are abstract-type-compatible with the value).
- They can be integrated into the program, effectively giving the program a richer type system that can be checked at compile time. For instance, this could be done using `typedef` declarations in C or ADTs in an object-oriented language.

The finer-grained abstract type information can also be supplied to a subsequent analysis to improve the run time or the results of that analysis. Since our abstract type inference is dynamic, its results are most applicable to a follow-on analysis that does not require sound information (for example, using it as optimization hints), that is itself unsound (such as a dynamic analysis, or many machine learning algorithms), that verifies its input, or that produces results for human examination (since people are resilient to minor inaccuracies). Here are a few examples of such analyses.

- Dynamic invariant detection [7, 19, 13] is a machine learning technique that infers relationships among variable values. Abstract types indicate which variables may be sensibly compared to one another. Directing the detection tool to avoid meaningless comparisons eliminates unnecessary computation and avoids overfitting [8].
- Principal components analysis (PCA) approximates a high-dimensional dataset with a lower-dimensional one, by finding correlations among variables. Such correlations permit a variable to be approximated by a combination of other variables; the reduced dataset is generally easier for humans to understand. Abstract types can indicate variables that are *not* related and thus whose correlations would be coincidental. For example, they could be applied to work that uses PCA over program traces to group program constructs [17].
- Dynamic analysis has been used to detect features (or errors) in programs, by finding correlated parts of the program [36, 28, 37, 6, 11, 12]. Abstract types could refine this information, making it even more useful.
- Abstract types can partition the heap, providing useful information to memory hierarchy optimizations. For example, a group of objects that are likely to be connected in the heap can be allocated on the same page or in the same arena. This can reduce paging when accessing the data structure. A related optimization is to allocate related elements to locations

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

```
1.  int totalCost(int miles, int price, int tax) {
2.    int year = 2006;
3.    if ((miles > 1000) && (year > 2000)) {
4.      int shippingFee = 10;
5.      return price + tax + shippingFee;
6.    } else {
7.      return price + tax;
8.    }
9.  }
```

Figure 1: A C procedure that uses `int` as the declared type for variables of three abstract types: distance, money, and time.

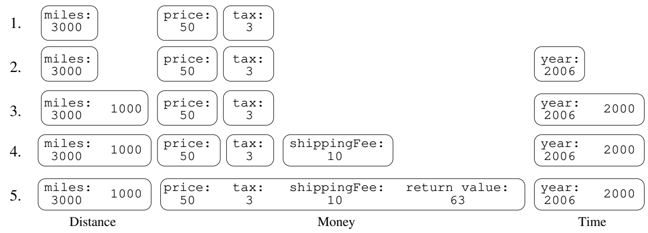

Figure 2: Our analysis inferring abstract types for the procedure shown in Figure 1 during one call: `totalCost(3000, 50, 3)`. Each line represents the values and their abstract types after the execution of the corresponding line in the code. Values in the same box belong to the same abstract type.

that will not contend for cache lines, reducing thrashing for elements likely to be accessed together.

- Abstract types are related to slices [34, 32], so they can be used for many of the same purposes, such as debugging, testing, parallelization, and program comprehension.
- Abstract types chosen to match the operators used in a later analysis can improve efficiency by allowing the analysis to consider only pairs of variables of the same abstract type, or consider all variables of an abstract type together. A general description of this technique is given in Section 2.3.

We have evaluated the results of abstract type inference both on a program understanding task and when fed into a subsequent analysis; see Section 4.

The key idea of our analysis is to recover information that is implicit in the program. The operations in the program encode the programmer's intent: values that interact in the program must be intended to have the same abstract type (or else the interaction indicates a bug), and values that never interact may be unrelated. More concretely, an operation such as `x+y` indicates that the values of `x` and `y` have the same abstract type. The notion of "interaction" is parameterizable and is further described in Section 2.1.1. The analysis ignores the underlying type system, including all casts, and unifies the abstract types of two values when they interact. Abstract types for variables are constructed from the abstract types of values they held throughout execution (see Section 2.2).

Figure 2 shows our analysis inferring abstract types for the procedure of Figure 1 during one particular call. The > and + operators cause their operands to interact and thus unify their abstract types. After line 5 completes, the values (and variables assigned to them) are partitioned into 3 abstract types. Our analysis does not assign the labels "distance", "money", and "time"; it merely partitions the values and variables.

Our abstract type inference operates dynamically on values rather than statically on variables (as in [25, 24]), permitting it to produce precise[1] results. We have not observed overfitting to be a problem in practice. If desired, the information can be checked by a type system or similar static analysis, and the combined dynamic–static system is sound, so its results can be used wherever a static abstract type inference's could be.

We have produced two implementations of dynamic inference of abstract types: one operates on compiled binaries (for Linux/x86) of programs written in languages such as C and C++, and the other works on compiled Java programs (bytecodes, also known as class files). In our experiments, the results of the analysis are accurate (close to the ideal types a developer would specify), aid humans in program understanding, and improve the run time and results of a follow-on client analysis.

[1] We say that an analysis is "precise" if it groups in an abstract type only variables that could really interact in execution.

2. Dynamic inference of abstract types

This section presents a unification-based dynamic analysis for partitioning variables into abstract types, based on observing the interactions of the values they held during execution.

Abstract types provide a division of program variables or values into sets of related quantities. For a given program, there are many ways to make this distinction, and different partitions are useful for different purposes. Our analysis is parameterizable to permit computation of different varieties of abstract type. No analysis is guaranteed to produce exactly what the programmer would intend for a given purpose (that is unknowable and inherently subjective, and the program may not perfectly express the programmer's intent), but our goal is to produce information about the program that is sufficiently accurate to be used by people and by tools. Furthermore, the analysis does not give a meaningful name or specification to each abstract type: it just groups values together based on interactions and dataflow, then uses that information to group variables.

A *value* is a concrete instance of an entity that a program operates on, such as a particular dynamic instance of the number `42` or the string `"foobar"`. New values can be created via literals, memory allocation, user or file inputs, or other operations. For example, every time `x+1` is executed, the value stored in `x` is reused, a new value is created for 1, and the addition operator creates a new value. By contrast, `x=y` creates no new values — it merely copies one.

Variables are containers for values, so a variable can hold many distinct values during its lifetime. For example, in "`x=3; y=3; x=y;`" there are two values (both of which represent the same number), and at the end of execution both variables hold one of them. However, `x` has held two distinct values during its lifetime. This is similar to the way objects work — compare "`x=new Object(); y=new Object(); x=y;`" — but we extend the notion to primitives. Unlike objects, for primitives the programming language provides no way to tell by looking at the program state whether two instances of 3 are the same value, but a dynamic analysis can track this information.

A variable can have different abstract types at different static points in a program where it holds different values. Section 4.2 shows that this can be useful in understanding global variables.

Our dynamic abstract type inference works by observing value interactions and dataflow, unifying the abstract types of values that interacted (Section 2.1), and constructing abstract types for variables based on the abstract types of the values they held (Section 2.2).

2.1 Tracking dataflow and value interactions

Our algorithm tracks the flow and interactions of values throughout execution, partitioning values into disjoint sets. To accomplish this, it maintains a tag for each value that represents the value's abstract type. A global union-find data structure called `value_uf` groups the tags into disjoint sets called *interaction sets*; all values in a set belong to the same abstract type. Only values of primitive types (including pointers) get tags; structures, class objects, and arrays are treated as collections of primitives. One tag in each set, called the *leader*, represents the set when performing operations in `value_uf`.

Tags are created and propagated to reflect the dynamic dataflow that occurs during execution. Every new value created during execution receives a unique tag, which is initially placed in a new singleton set within `value_uf` to denote that it represents a unique abstract type.

As a value propagates (is copied) during execution, its tag always accompanies it, thus tracking dataflow. For instance, in an assignment `x=y`, when the value stored in `y` is copied into `x`, the tag associated with the value is copied as well. Procedure arguments and return values are treated in exactly the same way. This propagation of tags is somewhat similar to the propagation of abstract identifiers in a static dataflow analysis, but a key feature is that it occurs only when dataflow actually occurs during execution. In the terminology of static analysis, our technique is completely context-, flow-, and path-sensitive.

2.1.1 Definitions of interaction

In addition to recording dataflow, our analysis classifies values as having the same abstract type (by unifying the sets of their tags in `value_uf`) if they are used together ("interact") in certain operations. The notion of what operations qualify as interactions is parameterizable, and we have implemented 4 definitions of interaction among primitive and reference values. We present them in order from finest (produces many abstract types, each with few members) to coarsest (produces few abstract types, each with many members).

Dataflow - No binary operations are interactions. Thus, every value belongs to a singleton set, which represents a unique abstract type. This tracks dataflow because two variables have the same abstract type if a single value flowed to both of them.

Dataflow and comparisons - Two values that are operands to a comparison operator (e.g., < and ==) interact, so their tags are unified together in one interaction set within `value_uf`. The result of the comparison is a boolean value that is unrelated to the operands, so it receives a fresh tag representing a unique abstract type.

Units - Addition, subtraction, and comparisons are interactions; other operations, including multiplication and division, are not. As a result, variables with the same abstract type could be assigned the same scientific units.

Arithmetic - This is the default mode for our implementations, as our experience suggests that it is easier for users to split up sets that are too large than to combine sets that were not recognized as related.

- Comparisons are interactions between operands.
- All arithmetic (`+`, `-`, `*`, `/`, `%`, etc.) and bitwise (`&`, `|`, etc.) operations are interactions between operands and result, so all 3 values have the same abstract type.
- Shift operations (`<<`, `>>`, etc.) are interactions between the quantity being shifted (left operand) and the result. In practice, the shift amount (right operand) is usually not closely related to the quantity being shifted.

In all modes, logical operators such as `&&`, `||`, and the ternary `?:` operator do not produce interactions, because no close relationship need exist between the operands, especially in C when operands are often numbers or pointers instead of booleans.

2.1.2 Optimizations

Many optimizations to tracking value interactions are possible; we mention a few that we have implemented to date:

Re-use of existing tags for new values: The expression `x+y` conceptually creates a new tag for the sum, then immediately unifies the interaction set of that tag with those of the operands. However, there is no need to create a new tag: the tool can simply reuse one of the operands' tags.

Elimination of operand literal tags: When a literal appears as an operand alongside a variable (e.g., `x+42`), there is no point in creating a new tag for it because that new tag will immediately be unified (merged) into the interaction set of the tag of the other operand.

Garbage collection of tags: In our Java implementation, the use of weak pointers enables tags to be reclaimed when no longer in use. In our binary implementation, a garbage collector periodically scans the tags, permitting unclaimed tags to be reused when new values are created.

Eager canonicalization: Tags can be replaced by their leaders (canonical representatives) whenever tags are manipulated to reduce the total number of non-garbage tags in use.

2.2 Inferring abstract types for variables

Our analysis infers abstract types for variables based on the interaction sets of values: roughly speaking, variables will be in the same abstract type if they held values from the same interaction set. Our approach computes abstract types separately for variables at certain static points in a program; we call each such location a *site* (see Section 3).

This section describes two algorithms for constructing abstract types, which give somewhat different results when the interaction sets themselves change over time: the first algorithm is relatively simple, while the second algorithm is more complex, but corresponds to a more consistent definition of abstract types. Both algorithms can be implemented efficiently, and we have not found the difference between their results to be significant in our experiments.

For the first variable type inference algorithm, each site has an associated union-find data structure representing a partition of the variables at the site. Before execution, each variable is in its own singleton set. Now, suppose that `x` and `y` are two variables at a particular site, and that on a particular execution they have the values $v_{\mathtt{x}}$ and $v_{\mathtt{y}}$. The simple algorithm checks whether $v_{\mathtt{x}}$ and $v_{\mathtt{y}}$ are in the same interaction set at that moment of execution, and if so, merges (unifies) the variable sets containing `x` and `y`. After execution, the abstract types are simply the sets in the union-find structure. A potentially unsatisfying aspect of this algorithm is that while value interactions that occur before the execution of a site affect the variable abstract types, those that occur afterwards may not, if the variable never again has values from that interaction set.

To avoid the asymmetry in the simple algorithm just described, we developed a more complex algorithm whose definition of abstract type does not depend on the order in which value interactions occur. The key difference in the second algorithm is that rather than creating abstract types grouping variables directly, it constructs abstract types by first grouping value interaction sets, and then using these groups to define sets of variables. The effect of this choice on the algorithm's results is that its abstract types always correspond to unions of whole value interaction sets, rather than parts of interaction sets. In other words, if a variable `x` had a value $v_{\mathtt{x}}$ at

```
# Global union-find structure for value interaction sets
UnionFind<Tag> value_uf

# Per-site union-find structure grouping value interaction set
# leaders for sets of values whose unions form abstract types
UnionFind<Tag> type_uf
# Per-site array picking out a set in type_uf corresponding to all the
# previously-observed values of a variable
Tag[] var_tags

# Update the abstract types for this site, based on an execution when
# the variables had values whose interaction sets are given by new_tags
method site.update_types(Tag[] new_tags):
1. for each variable v:
2.    Tag leader = value_uf.find(var_tags[v])

      # If var_tags[v] is no longer the leader of its interaction set,
      # then its set has been merged with another set in value_uf
3.    if leader != var_tags[v]:
          # Merge corresponding sets in type_uf and
          # maintain that var_tags[v] is the leader
4.        var_tags[v] = type_uf.union(leader, var_tags[v])

      # If needed, create entry for new value in type_uf
5.    Tag new_leader = value_uf.find(new_tags[v])
6.    if new_leader not in type_uf:
7.        type_uf.make_singleton(new_leader)

      # Merge new tag with existing tags in type_uf
8.    var_tags[v] = type_uf.union(var_tags[v], new_leader)
```

Figure 3: Pseudocode for the propagation, occurring at each site execution, that translates from value interaction sets to abstract types for variables. See Section 2.2 for a detailed description.

one execution of a site, the variable `y` had a value v_y at a different execution of the site, and v_x and v_y interacted, then `x` and `y` will be in the same abstract type, even if there was no single execution at which the values of `x` and `y` had interacted or would interact in the future. To implement this approach, the second algorithm does not use a union-find structure representing a partition of variables at each site; instead, it uses a union-find structure representing a partition of value interaction sets.

To be precise, the union-find structure maintains a partition of tag values that at some point were leaders (canonical representatives) of value interaction sets. Figure 3 gives a pseudocode implementation of the algorithm, in which the per-site union-find structure is `type_uf`. Such tags are grouped together either as the value interaction sets grow (if an interaction set merged to create a larger one, the old and new leaders are grouped), or as variables take values from different interaction sets on subsequent site executions (the leaders of the previous and current sets are grouped). To maintain the connection between value interaction sets and variables, the algorithm also keeps track, for each variable, of the leader of the interaction set of the most recently observed value of the variable (in Figure 3, the `var_tags` array).

At each execution of a site, the algorithm first updates the representative tag for each variable value seen on the previous execution, to account for interaction set merges that have occurred between site executions (lines 2–4), and then merges the sets containing the representatives of the previous and current value interaction sets (line 8), creating a new set for the current value if it does not yet exist (lines 5–7). The algorithm's results should also reflect interactions that occur even after the final execution of a site, so it performs a final iteration of propagating merges of interaction sets (lines 2–4) for each site at the very end of execution. At the end of execution, two variables v_1 and v_2 at a site are in the same abstract type if `var_tags[`v_1`]` and `var_tags[`v_2`]` are in the same set in `type_uf`.

2.3 Approximating analyses by abstract types

The abstract types computed using the algorithms above can often be used as approximations of another program analysis. This section describes two general ways in which such approximations are useful; Section 4.3 gives a real example.

The goal of an analysis can often be described as being to compute some relation R that might hold between pairs of variables. Two kinds of abstract type systems can allow such a relation to be computed without considering each pair of variables independently. First, call an abstract type system a *coarsening* of a relation R if whenever a and b are related by R, they have the same abstract type. Second, call an abstract type system a *congruence* for a relation R if whenever a and a' have the same abstract type, and b and b' have the same abstract type, a is related to b by R if and only if a' is related to b' by R. (In the special case when R is itself an equivalence relation, these type systems define respectively coarser and finer partitions of the variables than the one induced by R.) Given a coarsening of a relation R, R can be computed more efficiently by considering each abstract type separately: this is still correct because no members of different types can be related. Given a congruence for a relation R, R can be computed more efficiently by considering all the members of each abstract type together: the results will be the same for all of them.

For instance, as part of correcting Y2K errors, one might be interested in finding pairs of variables that represent the 4- and 2-digit versions of the same year. In a dynamic analysis for this problem, the relation R might hold between the variables `y4` and `y2` after the statement `y4 = y2+c`, if `c` had the value 1900. In this example our algorithm operating in arithmetic mode could be used to compute a coarsening of the relation R, because a pair of values that are computed independently cannot represent the same year. Similarly, our algorithm operating in dataflow mode could be used to compute a congruence for R, since copying a value does not change what year it represents.

3. Implementations

We have implemented two tools for performing dynamic inference of abstract types: DynCompB for binary-compiled executables written in languages such as C and C++, and DynCompJ for JVM-compiled class files produced from languages such as Java.

3.1 Granularity of analysis

Our tools compute abstract types for variables at the entrances and exits of procedures, because these are where humans and tools commonly desire abstract type information; it would be easy to choose other sites instead. At each entrance or exit site, the variables of interest include formal parameters, globals, and (for methods) fields of the current object.

For variables of more complex types, our tools add *derived variables* to refer to the contents of larger data structures, in order to provide richer information. For example, for an array referred to by a pointer `int *foo`, the variable `foo` represents the pointer itself and the derived variable `foo[]` represents the contents of the array. (Although there is only one derived variable for the contents of an array, the interactions of values held by individual elements are tracked separately to improve precision.) Similarly, for a C `struct` or C++/Java object, there is one derived variable per field; variables are derived recursively for arrays, structures, or objects nested inside of structures or objects. For example, for methods in a `Person` class, `this.age` is a derived variable referring to a field of the current object.

The algorithm of Section 2 is not limited to the granularity of analysis described above: it could be used equally well with abstract types computed more or less frequently, or with more or fewer derived variables. We chose the granularity described above as the one most often appropriate for both program understanding tasks and assisting other automated tools.

3.2 Binary implementation

DynCompB uses Valgrind [23] to rewrite a program binary at run time to insert instrumentation code that performs the analysis described in Section 2. Because Valgrind operates on arbitrary Linux/x86 program binaries in the ELF format, DynCompB could be extended to any language that can compile into this form, including all languages that `gcc` supports: C, C++, Objective-C, Ada, Fortran, and Java. DynCompB currently supports C and C++, which are the languages for which demand has been greatest.

The value tracking algorithm (Section 2.1) is performed purely at the binary level, and operates on all code, including libraries. Instrumentation code maintains a 32-bit integer for every byte of memory and every register, to represent the tag of the value currently stored in that location. For every machine instruction that copies values from or to memory or registers, instrumentation code copies the tags of the copied bytes. For every machine instruction that qualifies as an interaction, instrumentation code merges the sets of the operands' tags in `value_uf`. Values that interact with the stack pointer are treated specially: unrelated pointer values (such as local variable addresses) are not counted as related merely because they are calculated as offsets from `ESP` or `EBP`.

New tags are created in response to two events: dynamic occurrences of literals, and program input. For example, each time the instructions corresponding to the C code `y=42` are executed, a new tag is assigned to the memory location where `y` is located. Initialization of memory, such as the zero-filling performed by `calloc`, is another example of new tags created from literals. When a system call such as `read` stores new data in the program's address space, these bytes also receive fresh tags.

The conversion between value interactions and variable abstract types (Section 2.2) is implemented by instrumenting the binary to, at each site, pause normal execution, use debugging information to locate variables, read the tags of their values from memory into `new_tags`, and execute the algorithm of Figure 3. Because most variables hold values that span more than one byte, when those values are read, the interaction sets of tags for all the bytes are merged in `value_uf` to denote that those bytes all represent a single value.

3.3 Java implementation

DynCompJ, our implementation of dynamic abstract type inference for Java, transforms class files to track value interactions. It supports any class file that can run on a version 1.5 JVM.

Java values are objects or primitives. A map (`tag-map`) associates each object with its entry in the global union-find data structure `value_uf`. Whenever an operation (`==`, `!=`) is performed on two objects, instrumentation code merges their tags in `value_uf`.

By contrast, primitives cannot be uniquely identified by their value (e.g., it is not possible to tell whether two instances of 3 represent the same value). A primitive value thus requires a unique external tag to represent it in `value_uf`. Each time a value is manipulated (stored in a variable, pushed on the stack, etc.), its tag must be carried with it. The basic approach is to provide a tag storage location for each place that a value can be stored.

Java stack - A global *tag stack* parallels the normal Java stack. Each instruction that manipulates primitive values on the Java stack also updates the tag stack.

Fields - Each object with primitive member fields has an associated tag array that contains the tag for each primitive in the object. A map (`field-map`) from each object to its tag array is maintained. A single global tag array handles all static primitive fields in a similar manner.

Locals - Similarly to objects, each active stack frame has a *tag frame*, an array of tags for primitive local variables. Instrumentation code creates the tag frame when the method is entered. Parameter tags are stored in the tag frame in the same manner as locals.

Classes in the JDK need to be instrumented so that variables that interact in the JDK are properly tracked. Our technique statically instruments the JDK to create a second (instrumented) copy of each method [29]. The instrumented methods are used by the user code, and the original methods are used by the instrumentation code. The fields in some core classes (String, Class, Object) cannot be modified, added, or deleted since their layout is known to the JVM. Our analysis does not require any changes to the fields of a class.

4. Experiments

We have evaluated our abstract type inference tools in several ways. First, we carefully analyzed a small program by hand, to verify that the results were accurate (Section 4.1). Second, we performed a case study of two programmers who were trying to understand and modify unfamiliar programs; we observed whether the analysis results assisted the programmers (Section 4.2). Third, we measured how much the inferred types improved the results and efficiency of a follow-on analysis (Section 4.3). Fourth, we compared the results of static abstract type inference to our dynamic abstract type inference (Section 4.4). Fifth, we measured the effect of test suites on the dynamic abstract type inference results (Section 4.5).

Our experiments use the following subject programs. All line counts are non-comment, non-blank. We ran each program once, on a single input that is also noted below.

- `wordplay` (C, 740 LOC): anagram generator, using a 38KB dictionary
- RNAfold (C, 1804 LOC, of which 706 LOC are in `fold.c`): secondary structure predictor, folding a length-100 RNA sequence, only inferring types for variables within `fold.c`
- SVM-Light (C, 5834 LOC): support vector machine learning tool, training an SVM on a 474-line input
- `bzip2` (C, 5128 LOC): file compressor, running on a 50KB text file of `gcc` source code
- `flex` (C, 11,977 LOC): lexical analyzer generator, running on the lexical grammar for C
- `perl` (C, 104,528 LOC, of which 16,976 are the implementations of non-hot opcodes): scripting language implementation, interpreting a 664-line sorting program from its test suite
- `bzip2` (Java, 1275 LOC): file compressor, running on a 4KB source file
- `javac` (Java, 39,594 LOC, of which 12,506 are in the `comp` package): Java compiler, compiling its `comp` package

4.1 Accuracy

In order to assess the accuracy of our tools, we performed a careful manual analysis of all 21 global variables of the `wordplay` anagram generator program, then ran DynCompB to compare the results. An exhaustive manual examination is feasible for `wordplay` because it is small (740 lines), but would not be for larger programs. Indeed, the difficulty of such an analysis is a motivation

	Variables	Declarations and comments from source code

1	keymem
	largestlet
	words2mem
	*words2
	*words2ptrs
	*wordss
		`char *keymem; /* Memory block for keys */`
		`char largestlet;`
		`char *words2mem; /* Memory block for candidate words */`
		`char **words2; /* Candidate word index (pointers to the words) */`
		`char **words2ptrs; /* For copying the word indexes */`
		`char **wordss; /* Keys */`

2	ncount
	*lindx1
	*lindx2
	*findx1
	*findx2
		`int ncount; /* Num. of candidate words */`
		`int *lindx1;`
		`int *lindx2;`
		`int findx1[26];`
		`int findx2[26];`

3	longestlength
	max_depth
	*wordsn
		`int longestlength; /* Length of longest word in words2 array */`
		`int max_depth;`
		`int *wordsn; /* Lengths of each word in words2 */`

4	*wordmasks	`int *wordmasks; /* Mask of which letters are contained in each word */`
5	rec_anag_count	`int rec_anag_count; /* For recursive alg, keeps track of num. of anagrams found */`
6	adjacentdups	`int adjacentdups;`
7	specfirstword	`int specfirstword;`
8	maxdepthspec	`int maxdepthspec;`
9	silent	`int silent;`
10	vowelcheck	`int vowelcheck;`

Figure 4: The 10 abstract types for the 21 global variables in `wordplay`. Each abstract type contains one or more variables. A pointer variable stands for any element of an array, not just the first element. DynCompB computed 11 abstract types (it erroneously separated `*wordsn` from the other variables in type 3), and Lackwit computed 7 abstract types (it erroneously grouped the 15 variables of types 1–4 into a single abstract type).

for our tools. The manual analysis applied human understanding to every use of the global variables, all variables with which they interacted, all possible aliasing relationships, source code comments, etc. Figure 4 shows the results.

Type 1 represents the abstract type of "words" in the program. Variable `largestlet` represents the largest letter found in some word, and although it is of type `char` instead of `char*`, code inspection confirms that it has the same abstract type as the other variables in the set.

Type 2 contains variables related to indices into arrays of words. These code comments reveal how the programmer intended to use these variables:

```
/* Create indexes into words2 array by word length.
   Words of length i will be in elements lindx1[i]
   through lindx2[i] of array words2.
...
/* Create indexes into wordss array by first letter.
   Words with first letter "A" will be will be in
   elements findx1[i] through findx2[i] of array wordss.
```

`*lindx1` and `*lindx2` are indices into the array `words2`, and `*findx1` and `*findx2` are indices into the array `wordss`. Thus, all four indices belong to the same abstract type since the contents of the `words2` and `wordss` arrays both belong to the same abstract type. `ncount` interacts with `*lindx2` to produce these indices.

Type 3 contains variables that test whether the base case of a recursive function has been reached, in this line of code:

```
if ((max_depth - *level) * longestlength < strlen(s))
```

Type 4 contains the contents of the `wordmasks` array, which holds "mask[s] of which letters are contained in each word".

The remaining variables belong in singleton types (5–10); their types are conceptually distinct from all other global variables.

We ran `wordplay` with DynCompB on an 18-letter string to anagram and a 38KB dictionary, achieving 76% coverage of the executable lines. DynCompB places the variables in 11 abstract types, compared to the 10 abstract types of the manual analysis. The only difference is that DynCompB splits type 3 of Figure 4 into two abstract types, one containing `longestlength` and `max_depth`, and the other containing `*wordsn`.

Comments in the code indicate that the variables `*wordsn` and `longestlength` should belong to the same abstract type representing "length of word in `words2` array". DynCompB fails to recognize this relationship because their values never actually interact. `longestlength` is initialized with return values of independent calls to `strlen()`, not from cached elements of `wordsn`. No analysis — static or dynamic — that infers abstract types via value interactions could notice this relationship.

We performed a similar hand inspection of the `bzip2` and `flex` programs; these analyses were not exhaustive, due to the size and complexity of the programs. However, for each pair of variables that we examined, we were able to reasonably determine that DynCompB's decision (whether putting the variables in the same type or in different types) was the right one.

On these larger programs, we did notice the effect of the limited coverage of our test cases. In `bzip2` and `flex`, several error-handling procedures were rarely or never executed. We did not notice ill results of overfitting for procedures that were executed more than ten times.

4.2 User studies

Two MIT researchers (who are not members of our research group) were struggling with reverse engineering problems. They volunteered to try using DynCompB to assist them with their C programming.

4.2.1 RNAfold

The first researcher is a computational biologist who had recently refactored RNAfold, an 1804-line RNA folding program [14]. The refactoring converted 55 `int` variables of the abstract type "energy" into type `double`. The program has hundreds of non-energy-related `int` variables. His hand analysis had statically emulated the operation of DynCompB, building up sets of related variables by tracing assignments, function calls, and binary operators in the source code. It took him 16 hours of work, done over a week, to find all the energy variables. He described the process as tedious and error-prone; two iterations were required before he found everything.

We ran DynCompB on RNAfold with a test case of a single 100 base pair RNA sequence extracted from a public database (achieving 73% statement coverage of `fold.c`, where the algorithm resides). We showed the inferred sets of abstract types to the researcher and observed his reactions and comments. One 60-element set contained all of the energy variables the researcher had found. The set also contained 5 index variables, which had interacted with energy variables during (unnecessarily) complex initialization code. Although the researcher's notion of abstract types did not perfectly match the tool's definition, this minor mismatch was no hindrance: the researcher easily and quickly recognized and filtered out the few non-energy variables.

The DynCompB results gave the researcher increased confidence in his refactoring. He estimated that instead of spending 16 hours of manual analysis, he could have set up the test, run the tool, observed the results, and filtered out inaccuracies in 1 or 2 hours.

Types	C programs						Java progs		Aver-
Types	word	rna	svm	bzip2	flex	perl	bzip2	javac	age
Represent.	23	42	46	37	184	263	56	21.4	84
Declared	6.5	20	14	8.8	88	50	15	21.0	28
Abstract	3.6	15	6.5	1.9	23	5.3	12	1.4	8.6

Figure 5: Average number of variables in a type for each site. Section 4.3.1 describes the varieties of type. Abstract types are computed by DynCompB and DynCompJ.

4.2.2 *SVM-Light*

The second researcher was trying to understand SVM-Light, a 5834-line support vector machine implementation [15]. His goal was to create a hardware implementation. He was familiar with SVM algorithms but unsure of how SVM-Light implemented a particular algorithm.

We ran SVM-Light once, on a 474-line file from LIBSVM [4] (achieving 37% statement coverage). The DynCompB output, the variable names, and the source code confirmed his intuitions about how the mathematical variables in the SVM algorithm mapped into program variables in code. For example, at one particular site, there was a perfect correspondence between his notion of abstract types and what the tool inferred, for variables that dealt with calculating error bounds to determine whether to shift the SVM up to higher dimensions.

The researcher noted that the variable `buffer` appeared in large sets at many sites, and he initially suspected imprecision in our tool. After double-checking the code, he saw that `buffer` was used pervasively to hold temporary calculation results for many different operations. He had thought that `buffer` was confined to a few core operations, so he learned a new fact about the program in addition to verifying what he already knew.

4.3 Dynamic invariant detection

Section 4.2 evaluated whether abstract types are useful to programmers. This section evaluates whether abstract types can improve the results of a follow-on analysis, making it run faster or produce better output.

As described in Section 1, abstract types are applicable to many analyses; for concreteness, our evaluation uses the Daikon dynamic invariant detection system [7]. Given a trace of the run-time values computed by a program, Daikon performs machine learning, seeking relationships among these values.

Finer types than those that appear in the program source can aid Daikon in two ways. First, they can improve run-time performance, because there is no need to hypothesize or check properties over variables of different abstract types. Second, and more importantly, they can improve output quality by reducing irrelevant output (false positives). Daikon has been applied to dozens of problems in software engineering and other fields [26], so improving it is a practical and realistic problem.

In the terminology of Section 2.3, DynComp (that is, DynCompB or DynCompJ) computes a coarsening of Daikon's invariants. This does not just optimize Daikon's performance; it also changes its behavior to remove undesirable invariants. We did not use DynComp to compute a congruence, because Daikon already implements an optimization [26] based on the same principle.

4.3.1 *Methodology*

Figure 5 shows the number of variables that share a type in our subject programs, averaged over all sites (procedure entrances and exits). These averages do not include C pointer variables, because abstract types of pointers (as opposed to their contents) are rarely interesting.

Representation types group all variables into four types based on their machine representation: integer, floating-point, string, and Java object references.

Declared types group variables by their declared types. Amongst primitives, this distinguishes each primitive representation (such as float and double or signed and unsigned) as well as any type aliases defined using `typedef`. Similarly, all classes are distinguished. This grouping may not be correct. For example, relationships between a `float` and a `double` may be interesting. Also, superclasses and their subclasses (such as `Number` and `Integer`) and interfaces and their implementors (such as `List` and `ArrayList`) may have interesting relationships. Daikon can use this to improve run-time performance in the absence of more accurate information.

Abstract types use the output of our dynamic analysis. Figure 5 indicates that the abstract types are significantly finer-grained than the declared types.

We measure the effect of the type declarations on the size of Daikon's output (the number of hypothesized invariants), and also on Daikon's run time and maximum memory size. For implementation simplicity, Daikon assumes that the type information that it is given is transitive; that is, if variables `a` and `b` have the same abstract type, and so do `b` and `c`, then variables `a` and `c` have the same abstract type. This is not necessarily true in our dynamic context, but our tools performed this merging (which reduces precision) before presenting the information to Daikon. DynComp's run time was approximately the same as Daikon's, so it is a reasonable preprocessing step.

4.3.2 *Results*

Figure 6 shows the results. Daikon ran faster, used less memory, and produced fewer invariants when using abstract types, compared to its default of using representation types.

Compared to the declared type heuristic, abstract types produced substantially fewer invariants (even though, as explained above, declared types caused Daikon to miss important properties), and Daikon generally ran faster and in less memory. The key exception is Perl, where one of Daikon's optimizations, which is currently applied to one type at a time, was more effective when applied to more variables at once. We plan to generalize this optimization, which should make using abstract types no slower than using declared types, while still producing substantially better output.

For programs where very few invariants were computed (such as wordplay, RNAfold, and others), memory size is dominated by the JVM, the Daikon program itself, and the data being processed; thus, the choice of types has little end-to-end impact on run time and overall memory usage. In any event, optimizations are most important for large runs of Daikon.

In some cases, the use of abstract types improves run time and memory use less than it improves the number of invariants. This is the result of optimizations in Daikon, which is able to symbolically represent very large numbers of properties in a small amount of memory.

Representation types were sometimes nearly as good as declared types. When most of a program's variables are declared as `int`, there is little difference between the two, except that declared types treat pointers differently, and Daikon's optimizations already work well with pointers.

Treatment	Daikon		
	time	memory	# invariants
wordplay			
Representation types	10	27	1,081,934
Declared types	9.5	26	830,676
Lackwit	8.7	25	48,877
Abstract types	8.5	25	26,385
RNAfold			
Representation types	170	140	292,603
Declared types	170	140	285,031
Lackwit	170	140	99,732
Abstract types	150	140	81,322
SVM-Light			
Representation types	150	80	1,984,402
Declared types	150	80	1,970,681
Lackwit	140	70	544,386
Abstract types	140	70	565,215
bzip2			
Representation types	130	90	28,608,435
Declared types	120	80	13,316,420
Lackwit	110	70	600,834
Abstract types	110	70	392,180
flex			
Representation types	930	380	1,410,893,630
Declared types	860	340	1,401,545,090
Lackwit	520	150	37,720,499
Abstract types	460	110	1,801,367
Perl non-hot opcode implementations			
Representation types	1600	1800	4,157,673,644
Declared types	770	920	585,066,116
Abstract types	1000	950	8,849,326
bzip2 (Java)			
Representation types	8200	1800	1,659,782,660
Declared types	8200	1800	1,283,375,117
Abstract types	970	480	34,102,566
javac `comp` package			
Representation types	850	1100	23,751,142
Declared types	460	490	12,726,250
Abstract types	360	440	5,529,947

Figure 6: Effect of types on a follow-on analysis, dynamic invariant detection. All tests were run on a P4 3.6GHz PC with 3GB RAM. Run time is in seconds, and memory size is in MB. The "# invariants" column includes redundant properties, most of which the Daikon tool suppresses upon output.

4.3.3 *Hand examination of differences*

In order to verify that the invariants eliminated by the abstract type information were in fact spurious, we exhaustively examined all the differences between Daikon's output using declared types and abstract types for all of SVM-Light and RNAfold, and for a portion of `javac`. Less detailed examinations of other programs yielded similar results.

For SVM-Light, we performed the hand evaluation together with the researcher in the user study of Section 4.2.2. He confirmed that all but one eliminated invariant were spurious, because they all falsely related variables of different abstract types. There were several instances of Daikon falsely relating two integer variables when run with abstract types from DynCompB, one of which was used as an enumerated value but declared as an `int` and assigned small constant symbolic values defined in `#define` statements. For the one invariant that he said should not have been eliminated, the two variables had a control-flow-based (rather than a dataflow-based) relationship, so our tool was not able to place them into the same abstract type.

For RNAfold, almost all eliminated invariants were spurious.

We also carefully examined the 38 object invariants that were eliminated by abstract type information in the `Annotate` class of `javac`. 20 of these indicated that two boolean variables (each of which represented command line options) were equal. With one exception, the command-line options were unrelated and the invariants were spurious. The other 18 invariants indicated that two object references of the same type were not equal. In 15, this information was clearly uninteresting. For example, two of the variables represented different lexical elements that it would make no sense to compare. The remaining 3 invariants were over variables that keep track of whether warnings were found in various classes. These variables do not hold values that interact, so it is unlikely that these invariants would be interesting.

Abstract types	C programs					
	wordplay	rna	svm	bzip2	flex	perl
Static	9.1	30	14	2.2	46	n/a
Dynamic	3.6	15	6.5	1.9	23	5.3

Figure 7: Average number of variables in an abstract type, as computed by the static tool Lackwit and the dynamic tool DynCompB (those numbers are replicated from Figure 5). Lackwit was unable to process `perl`.

4.4 Comparison to static analysis

Abstract type inference can be performed statically as well as dynamically, and the two approaches have different tradeoffs. This section repeats the evaluation of Sections 4.1, 4.2, and 4.3, using a static abstract type inference tool, Lackwit (see Section 5.1), whose goals and output format are the same as those of DynCompB. (A similar static tool for Java, named Ajax [24], exists but does not scale to any of our subject programs.)

4.4.1 *Accuracy*

Figure 7 shows the average size of an abstract type as computed by the static tool Lackwit, as compared to those computed by our dynamic tool DynCompB. Sometimes Lackwit groups more variables into an abstract type on average than the average number of variables according to declared types (compare with Figure 5). This is possible because neither Lackwit nor DynCompB take declared types into account, so they might put an `int` variable and a `short` variable into the same abstract type if they held values that interact. These numbers alone cannot indicate correctness, so we performed a source code inspection on several programs to determine whether variables that the tools assigned to the same type actually correspond to the same programmer-intended abstract type.

Our hand examination of `bzip2` and `flex` focused on the differences in the Lackwit and DynCompB output, which we hoped would indicate the strengths and weaknesses of the static and dynamic approaches. Typically the DynCompB results were finer and were correct, so far as we could tell.

As in Section 4.1, we carefully examined the `wordplay` results. Lackwit assigns the last 6 variables of Figure 4 to singleton types (types 5–10), just like DynCompB and our hand analysis. However, Lackwit mistakenly assigns the first 15 global variables to a single abstract type. Lackwit is unable to make any distinction among them because of its conservative assumptions about runtime behavior. As one example, consider why it merges types 1 and 3. `wordplay` is invoked from the command line via:

```
wordplay -d<depth> <word> -f <word list file>
```

In C, command-line arguments are passed into the program within the `argv` string array. The static analysis does not distinguish the elements of an array, but gives them all the same abstract type. `wordplay` assigns the numeric value of the `<depth>` argument to `max_depth` (Type 3), and the `<word>` argument interacts with the word variables in Type 1. Thus, Lackwit merges types 1 and 3. Other conservative approximations cause Lackwit to spuriously group other variables together into one large set.

By comparison, the hand analysis assigns these 15 global variables to four distinct abstract types (types 1–4), and DynCompB assigns them to five distinct abstract types.

4.4.2 User studies

By inspecting the abstract types produced by DynCompB for RNAfold, the researcher saw that one particular set contained all the energy variables plus 5 variables of other abstract types (see Section 4.2.1). Lackwit grouped 10 extraneous variables into that same type in addition to the variables that DynCompB placed there, thus making the results strictly worse.

Hand-inspection of the abstract types produced by Lackwit for SVM-Light revealed no significant differences from the results of DynCompB.

4.4.3 Dynamic invariant detection

We compared Daikon's output using the types produced by Lackwit with the types produced by DynCompB for several examples. We exhaustively reviewed the differences for RNAfold and SVM-Light and noted mixed results: there were some cases where DynCompB produced abstract types that more closely mimicked the researcher's notion, and others where Lackwit did. We believe that a more exhaustive or longer test would have been able to improve the results of DynCompB.

4.5 Effect of test suites on results

The results of a dynamic analysis are inherently dependent on the executions of the program being analyzed. Section 4.4 indicated that even for very modest test suites, dynamic abstract type inference produces results that are as good as, or better than, a static analysis. Supplying additional executions to the dynamic analysis could make the results even better, though a conservative static analysis would be a bound for the dynamic results. This section briefly considers the effect of program executions on our dynamic analysis.

Figure 8 plots the average size of abstract types computed by DynCompJ, analyzing `javac` compiling files in its `comp` package. For example, when `javac` compiled a 12-line file, DynCompJ produced abstract types containing 1.33 elements on average. The largest file in the `comp` package is 3049 lines long (yielding an average set size of 1.43). The last two points on the graph represent compiling the entire `comp` package at once (12,506 lines of code, set size: 1.44), and compiling all of the javac source code (39,594 lines, set size: 1.46). Even quite a modest test achieves close to the results of compiling all of javac.

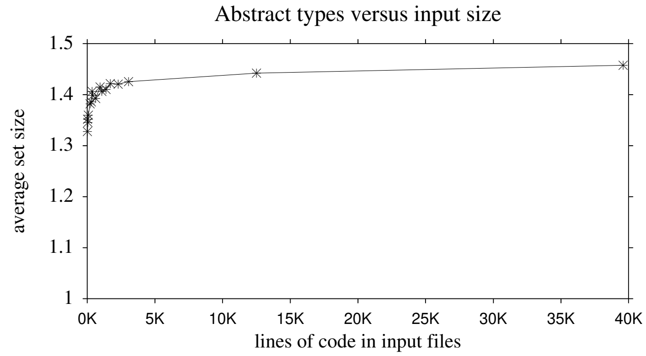

Figure 8: Size of abstract types computed for the `javac` program, plotted against the size of the file(s) being compiled.

5. Related work

This section gives additional details about the previous static approaches to abstract type inference, and also compares the present work to more distantly-related research in type inference (including of unit types), points-to analysis, and slicing.

5.1 Static abstract type inference

The most closely-related projects are Lackwit [25] and Ajax [24], which perform a static analysis with a similar purpose as our dynamic one. Though they can be effective for small programs, and their algorithms are theoretically scalable to quite large programs, these tools suffer from some limitations of precision that are almost impossible to avoid in a static analysis.

Ajax is a more flexible framework than Lackwit, but we consider both because Lackwit works only with C programs, and Ajax only with Java. Their key implementation technique is to summarize each procedure with a polymorphic type. A type system with parametric polymorphism allows types to contain type variables (represented by Greek letters) that can be consistently replaced with another type. For instance, a function that returns the first element of a list might have a type "α list $\rightarrow \alpha$", representing the fact that it can be applied to a list of integers, returning an integer (when α is replaced by `int`), or to a list of booleans to return a boolean, and so on. Lackwit and Ajax effectively devise new type systems for C and Java that are distinct from the usual ones: they reflect some of the same data structure, but they allow general polymorphism as in ML, and type constructors like `int` can be subscripted by tag variables ($\mathtt{int}_\alpha$). Inferring abstract types then reduces to giving a type to each function; because the type is polymorphic, different uses of the function need not use values of the same abstract type.

For instance, if a function uses two integer arguments together in an operation, the parameters would both get the type $\mathtt{int}_\alpha$ (where α is a variable):

```
void f(int a, int b, int c) { a + b; }
// Type: (intα, intα, intβ) -> void
....
f(x, y, 5); // x, y both of type intμ
....
f(z, w, 7); // z, w both of type intψ
```

At each site where the function is called, its argument types must have the same tags (both `x` and `y` are of type $\mathtt{int}_\mu$), but there is no constraint between different calls ($\mathtt{int}_\mu$ and $\mathtt{int}_\psi$ can be distinct).

Lackwit and Ajax construct these types using the Hindley-Milner type inference algorithm or a generalization [20, 24]. Lackwit's algorithm starts by assigning unique type variables to program variables, and then merges them by collecting equality constraints from the program and matching the structure of types in a process called "unification". Though the theoretical worst-case running time of the algorithm is superpolynomial, this worst case does not occur in practice: the algorithm's running time is close to linear in the program size. Unfortunately, the algorithmic scalability of Lackwit and Ajax does not suffice to make the real tools applicable to large programs. In our experience, Lackwit itself runs fairly quickly, but querying its database to extract results is very slow when sets are large. Ajax fails on many moderate-sized programs, and its operation is significantly slowed by the need to re-analyze large parts of the Java standard libraries. Therefore, we were unable to fairly compare the performance of Lackwit and Ajax to that of DynCompB and DynCompJ.

Lackwit and Ajax's polymorphic types improve their precision by providing a form of context sensitivity: the arguments to a procedure need not have the same abstract type at every call site. Another effect of this approach is that the abstract types that Lackwit and Ajax compute for a procedure are based on its *implementation*, but are independent of the way in which the procedure is *used* by its callers.

By contrast, the abstract types computed by our algorithm reflect all of the values passed to a procedure. Lackwit and Ajax are flow-insensitive, presuming that a variable has a single abstract type throughout its scope; our algorithm avoids this limitation by tracking values individually. One might imagine using a flow-sensitive static analysis, or making other changes, but many of the limitations of Lackwit observed above would be shared by just about any static analysis. For instance, in the `wordplay` example discussed in Section 4.4.1, Lackwit unifies global variables that should have different abstract types into the same type because they are initialized using elements of the argument array `argv`. It is rare for a static analysis to even give different treatment to elements of an array based on their index; we know of no static analysis that would distinguish between, say, "array elements for which the preceding element was the string `"-d"`" and "array elements for which the preceding element was the string `"-f"`".

5.2 Other type inference

Other kinds of type inference can also be performed either statically or dynamically. Lackwit and Ajax are based on techniques such as Hindley-Milner type-inference [20] that have been extensively studied in connection with statically-typed functional languages such as ML [21]. Types can also be statically inferred for languages that have only dynamic types, such as Scheme [10] and Smalltalk [1, 30]; the problem is practically much more difficult in this case. Dynamic approaches to type inference have been tried less frequently, but are relatively lightweight, and can be effective in contexts like reverse engineering when the results are further modified by a developer [27].

5.3 Units analysis

Abstract types are intuitively similar to units of measurement. A unit, like an abstract type, is an additional identity that can be associated with a number, and which gives information about what other values can sensibly be operated on together with a value. Also like abstract types, units are poorly supported by existing languages, and can be inferred from a program's operations. "Unit types" [16, 3] might be considered a variant of abstract type, but they have an additional algebraic structure not present in the abstract type systems considered so far. For instance, the unit type of the product of two quantities does not have the unit type of either factor; instead, it has a product unit type. Unit types can be inferred by extending abstract type inference with algebraic constraint solving; this can be done either dynamically or statically. The "units" mode of DynComp computes an approximation to physical unit types: if DynComp in this mode puts two variables in the same abstract type, they must have the same units, but several different abstract types might represent variables with one set of units, if those variables do not interact directly.

Erwig and Burnett [9] perform an analysis on spreadsheets that they refer to as "unit inference," but like our analysis they do not model the algebraic properties of units. Instead, their nonstandard type system represents a multidimensional hierarchical classification of entities, and requires that values represent complete levels of the hierarchy: for instance, adding apples and oranges is illegal if bananas also exist. They also give an inference algorithm, but it operates primarily from spreadsheet layout and column headings rather than operations.

5.4 Points-to analysis

Abstract type inference is particularly important for variables of primitive types such as integers, whose types in the original program rarely give information about their meaning. Though abstract type inference also groups pointer (or reference) variables according to the dataflow between them, and could easily be extended to group pointers with their referents (treating the dereference operation as another kind of interaction), it is usually more useful to distinguish pointers according to what they point to. This extension gives the problem of points-to or aliasing analysis, which answers questions of the form "what could `p` point to?" or "could `p` and `q` ever point to the same location?". Points-to analysis can be performed dynamically [22], and this is in some ways easier than dynamic abstract type inference because no tags are necessary: the numeric value of a pointer uniquely identifies what it is pointing at.

However, points-to analysis has been studied more extensively as a problem for static analysis. Abstract type inference is one of many problems that are hard to solve statically without accurate information about pointers, since some abstract type must be given to the value produced by a pointer dereference. Lackwit and Ajax's assignment of polymorphic types to references effectively represents a kind of pointer analysis. Pointer analysis has been studied extensively, but finding the best trade-off between precision and performance for a particular problem is still an area of active research [5, 35]. Many points-to analyses could be converted into abstract type inference algorithms with similar performance and scalability characteristics by adding special-case treatment of operators on primitive types.

The well-known almost-linear-time points-to analysis of Steensgaard [31] has an additional connection to our algorithm in its use of an efficient union-find data structure. Like our analysis, Steensgaard's algorithm uses a union-find structure to represent a partition of program variables, but beyond that their uses are rather different. In our algorithm, the goal is to compute an undirected (symmetric) and transitive relation, and the union-find structure represents the equivalence classes of the relation. In Steensgaard's algorithm, the goal is to compute a directed relation, points-to, that is not transitive, and the union-find structure is used to partition the domain of the relation so that an over-approximation to it can be represented in linear space. Steensgaard's algorithm is more closely related to the analysis that Lackwit and Ajax perform: like them it was inspired by unification-based type inference.

5.5 Slicing

Our dynamic abstract type inference relates variables that are connected by dataflow and by co-occurrence in primitive operations. Those portions of a program that are connected by dataflow and control dependence can be queried using the technique of slicing; a backward slice of a particular statement or expression indicates all the other statements or expressions whose computation can affect the given one. Static slicing approximates this relation, and dynamic slicing [2, 18, 33] computes it exactly for a given computation. In the general case, dynamic slicing amounts to maintaining a full execution trace of a program, and much dynamic slicing research focuses on how to collect and maintain this trace information efficiently. Our analysis considers similar issues with respect to collection, but pre-computes the result for any abstract type query in the compact form of disjoint variable sets. From a program slice, one can construct an abstract type consisting of the variables mentioned in the slice, and conversely the statements that use variables of a given abstract type form a slice. Under this correspondence, our abstract types correspond to slices that ignore control dependencies.

6. Conclusion

Abstract types are an automatically-derived representation of the interactions among a program's variables and values. Previous work has shown that they can be approximated by static analysis; we have introduced the first dynamic analysis that computes them precisely for a given set of program tests. Our implementations scale to sizeable programs, and only limited testing is required to construct usefully-accurate abstract types. In our experiments, the abstract types helped programmers to better understand their code, and they also improved the performance and results of a follow-on analysis tool.

Acknowledgments

Robert O'Callahan implemented Lackwit and Ajax, and discussed dynamic abstract type inference with us. Charles O'Donnell and Rodric Rabbah evaluated our tools. This research was supported by DARPA, EDG, NASA, and NSF.

References

[1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, Aug. 1996.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, pages 246–256, White Plains, NY, June 20–22, 1990.

[3] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, and G. L. Steele Jr. Object-oriented units of measurement. In *OOPSLA*, pages 384–403, Oct. 2004.

[4] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[5] M. Das, B. Liblit, M. Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS*, pages 260–278, July 2001.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Computer*, 29(3):210–224, Mar. 2003.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

[8] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458, June 2000.

[9] M. Erwig and M. M. Burnett. Adding apples and oranges. In *PADL*, pages 171–191, Jan. 2002.

[10] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *PLDI*, pages 23–32, May 1996.

[11] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CMSR*, pages 314–323, Mar. 2005.

[12] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *ICSM*, pages 347–356, Sept. 2005.

[13] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *ECOOP*, pages 431–456, July 2003.

[14] I. Hofacker. Vienna RNA package. `http://www.tbi.univie.ac.at/~ivo/RNA/`.

[15] T. Joachims. Making large-scale support vector machine learning practical. In *Advances in kernel methods: support vector learning*, pages 169–184. MIT Press, Cambridge, MA, USA, 1999.

[16] A. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, April 1996.

[17] A. Kuhn, O. Greevy, and T. Gîrba. Applying semantic analysis to feature execution traces. In *PCODA*, pages 48–53, Nov. 2005.

[18] J. R. Larus and S. Chandra. Using tracing and dynamic slicing to tune compilers. Technical Report 1174, University of Wisconsin – Madison, Madison, WI, Aug. 26, 1993.

[19] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, June 2003.

[20] R. Milner. A theory of type polymorphism in programming. *J. Comp. Syst. Sci.*, 17(3):348–375, 1978.

[21] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[22] M. Mock, M. Das, C. Chambers, and S. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *PASTE*, pages 66–72, June 2001.

[23] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *RV*, July 2003.

[24] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 2001.

[25] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, May 1997.

[26] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *FSE*, pages 23–32, Nov. 2004.

[27] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery. Dynamic type inference to support object-oriented reengineering in Smalltalk. In *ECOOP '98 Workshop on OO Reengineering*, pages 76–77, July 1998.

[28] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC/FSE*, pages 432–449, Sept. 1997.

[29] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE*, pages 114–123, Nov. 2005.

[30] S. A. Spoon and O. Shivers. Demand-driven type inference with subgoal pruning: Trading precision for scalability. In *ECOOP*, pages 51–74, June 2004.

[31] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41, Jan. 1996.

[32] F. Tip. A survey of program slicing techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994.

[33] G. A. Venkatesh. Experimental results from dynamic slicing of C programs. *ACM TOPLAS*, 17(2):197–216, Mar. 1995.

[34] M. Weiser. Program slicing. *IEEE TSE*, SE-10(4):352–357, July 1984.

[35] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, June 2005.

[36] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[37] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *J. Syst. Softw.*, 54(2):87–98, Oct. 2000.