Precise Constraint-Based Type Inference for Java

Tiejun Wang* and Scott F. Smith

Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218, USA
{wtj,scott}@cs.jhu.edu

Abstract. Precise type information is invaluable for analysis and optimization of object-oriented programs. Some forms of polymorphism found in object-oriented languages pose significant difficulty for type inference, in particular *data polymorphism*. Agesen's Cartesian Product Algorithm (CPA) can analyze programs with parametric polymorphism in a reasonably precise and efficient manner, but CPA loses precision for programs with data polymorphism. This paper presents a precise constraint-based type inference system for Java. It uses Data-Polymorphic CPA (DCPA), a novel constraint-based type inference algorithm which extends CPA with the ability to accurately and efficiently analyze data polymorphic programs. The system is implemented for the full Java language, and is used to statically verify the correctness of Java downcasts. Benchmark results are given which show that DCPA is significantly more accurate than CPA and the efficiency of DCPA is close to CPA.

1 Introduction

A concrete type inference (also known as a concrete class analysis) is an analysis which infers a set of classes for each expression, representing a conservative approximation of the classes of objects the expression may evaluate to at run-time. Such an analysis is vital for many applications, including call-graph construction [GDDC97, TP00], static resolution of virtual method calls [SHR+00], static verification of type casts, application extraction from libraries [Age96, TLSS99], and various whole program optimizations.

The constraint-based type inference [AW93, EST95] is an effective method whereby a concrete class analysis can be implemented. In constraint-based type inference, subtyping constraints are used to capture the flow information of a program. Type constraints are a more algebraic representation of the flow information, and algebraic manipulations allow for optimizations which lead to more efficient and effective analyses [AFFS98, Pot98, FF97, EST95]. This paper focuses on the construction of a precise constraint-based type inference system for Java.

* Partial funding provided by NSF grants CCR-9619843 and CCR-9988491

Polymorphism is widespread in object-oriented programs, and so an accurate analysis must be appropriately polymorphic. For example, a method may apply to arguments of different types, and objects of the same class may have fields assigned with values of different types. A monomorphic type inference algorithm such as 0CFA [Shi91] analyzes each method at the same type across different call sites, and lets all objects created from the same class share the same type, resulting in a significant precision loss on polymorphic programs. A polymorphic analysis is thus needed. But, polymorphic inference is a difficult issue due to subtle trade-offs between expressiveness and efficiency. An expressive algorithm could re-analyze the method for every different method invocation, but this would be very inefficient. The cartesian product algorithm (CPA) [Age95, Age96] addresses this problem, analyzing programs with parametric polymorphism in a manner that makes a reasonable trade-off between expressiveness and efficiency. The basic idea of CPA is to partition the calling context of a method based on the types of the actual arguments passed to the method. If the method is passed arguments of different types at two different invocations, those two different call sites are given different copies of the method type (i.e., different *contours*). And, if the method is invoked on arguments of the same types at two different call sites, those two invocations can efficiently share a contour of the method.

In some cases, an object field can be assigned values of different types, and different objects created from the same class can behave differently. This form of polymorphism is called *data polymorphism*. Data polymorphism occurs quite frequently in object-oriented programs, in particular when generic container classes are used. For example, there might be two instances of `java.util.Vector` created, and the two vectors could contain objects of different types. Since CPA uses a single object type for all objects created from the same class, it loses precision in this case. Here we present a novel type inference algorithm, Data-Polymorphic CPA (DCPA), which extends CPA with the ability to precisely analyze data-polymorphic programs.

We have constructed a prototype implementation of a constraint-based type inference system for the Java language. Several analyses are implemented, including 0CFA, CPA, and DCPA. The system is used as a tool to statically check whether Java type-casts in a program will always succeed at run-time. Cast-checking is a good test of the accuracy of an analysis, since each cast by definition is beyond the Java type system and so represents a more challenging type inference question. DCPA shows good results on benchmark tests: nearly all casts which could be verified statically by a flow-insensitive analysis have been verified by our DCPA implementation. DCPA is shown to be substantially more precise than CPA on Java programs, and to have an efficiency comparable to CPA.

The paper makes several other contributions. We define a generic framework for object-oriented constraint-based type inference that can model parametric polymorphism as well as data polymorphism. This is an object-oriented version of a framework we defined for a functional language with state in [SW00]. We implement 0CFA, CPA and DCPA for the full Java language and get good per-

formance. We also make a series of implementation optimizations to gain a more efficient analysis, including a novel version of the cycle elimination [AFFS98].

2 A Framework for Object-Oriented Constraint Inference

In this section we present a framework for polyvariant constraint-based type inference for objects. It recasts the functional polyvariant framework of [SW00] to handle object-oriented language features. We show how 0CFA and CPA can be expressed as instantiations of the framework. Other frameworks for polyvariant analysis have been proposed, including [PP98, JW95].

2.1 The Types

Our type system is based on Aiken-Wimmers-style set constraints [AW93]. The types are close to those described in [EST95], which gives set constraints for an object-oriented language. The aforecited paper presents constraints for a toy object-oriented language, I-Loop, which differs in some respects from Java. Here we give a type system designed to specify our implementation of a constraint-based type inference system for Java.

Definition 2.1 (Types): The type grammar is as follows.

$$
\begin{array}{lll}
\tau & \in \mathbf{Type} & ::= t \mid \tau v \mid (\forall\, \overline{t}.\ \tau \setminus C) \mid [\, l : \tau \,] \mid \\
 & & \quad (t_1 \times \ldots \times t_n) \to \tau \mid \mathbf{cast}(\delta, t) \mid \\
 & & \quad \mathbf{read}\ \tau \mid \mathbf{write}\ \tau \\
t & \in \mathbf{TypeVar} \supset \mathbf{ImpTypeVar} & \\
u & \in \mathbf{ImpTypeVar} & \\
\tau v & \in \mathbf{ValueType} & ::= \mathbf{int} \mid \mathbf{bool} \mid \ldots \mid \mathbf{obj}(\delta, [\, \overline{l_i : \tau_i} \,]) \\
\overline{t} & \in \mathbf{TypeVarSet} & = \mathbf{P}_{\mathrm{fin}}(\mathbf{TypeVar}) \\
l & \in \mathbf{FieldAndMethodIdentifier} & \\
\delta & \in \mathbf{JavaTypeIdentifier} & \\
\tau_1 <: \tau_2 & \in \mathbf{Constraint} & \\
C & \in \mathbf{ConstraintSet} & = \mathbf{P}_{\omega}(\mathbf{Constraint})
\end{array}
$$

In this definition, **ValueType** are the types for data values, which includes all Java primitive types (**int** and **bool** are shown as examples) and object types. The type for an object value is of the form $\mathbf{obj}(\delta, [\, \overline{l_i : \tau_i} \,])$, where δ is the identifier of its corresponding class, and the notation $[\, \overline{l_i : \tau_i} \,]$ enumerates the type for every instance field and instance method of the object. Every field or instance method has a field/method identifier. Every instance field of an object is mutable and so is given an imperative type variable u as its type; we distinguish imperative type variables for the presentation of data polymorphism. Every method is given a polymorphic type scheme $(\forall\, \overline{t}.\ \tau \setminus C)$, where $\overline{t}$ is the set of bound type variables, C is the set of constraints bound in this type scheme, and τ is an arrow type. Type schemes are also used for analyzing classes and object creation, as will be explained later. An arrow type $(t_1 \times \ldots \times t_n) \to \tau$ represents a method invocation with t_i as arguments and τ as the result.

Read and write operations on instance fields are analyzed with types **read** τ and **write** τ. Type $[\, l : \tau \,]$ is used for analyzing access of an instance field or invocation of an instance method, where l is the identifier of the field/method, and τ is the type specifying the usage of the field/method. A Java downcast expression is represented with type $\mathbf{cast}(\delta, t)$, where t is the result type of the downcast operation and δ is the identifier of the Java type which the expression is cast to. Casts must be an explicit part of the types because they represent a narrowing of a type. The null value is assigned with a special object type $\mathbf{obj}(\delta_0, [\,])$, which is abbreviated as **null**.

2.2 Constraint Generation

The first phase of constraint-based type inference is to generate a set of initial constraints corresponding to the program being analyzed. This set of constraints captures the immediate flow information corresponding to statements and expressions in the program. A closure computation process must follow this to propagate the initial flow information through the program. Every constraint is of the form $\tau_1 <: \tau_2$, meaning that τ_1 is a subtype of τ_2. This constraint intuitively represents the existence of a "flow" from expressions of type τ_1 to expressions of type τ_2.

We now show how constraints are generated for the key object-oriented features of Java. Analysis of more advanced Java features such as exceptions and inner classes will be discussed later. To ease presentation, the type of every expression will always be a type variable, and the type variable for expression `e` will be written as $[\![\texttt{e}]\!]$.

For constant expression `e` of Java primitive type τv, constraint $\tau v <: [\![\texttt{e}]\!]$ is generated. Every static field, local variable or method parameter is assigned a unique type variable. For an assignment expression `e1 = e2`, if `e1` is a static field or a local variable/parameter, $[\![\texttt{e2}]\!] <: [\![\texttt{e1}]\!]$ is generated; if `e1` is the access of instance field l, $[\![\texttt{e1}]\!] <: [\, l : \mathbf{write}\ [\![\texttt{e2}]\!] \,]$ is generated, meaning that `e1` is expected to be an object that has field l, and this field is written with a value of type $[\![\texttt{e2}]\!]$. Similarly, for a read access of an instance field `e.f`, $[\![\texttt{e}]\!] <: [\, l : \mathbf{read}\ [\![\texttt{e.f}]\!] \,]$ is generated, where the field named `f` is identified with abstract field label l.

Every method is given a type of the form $(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C)$, where $t_1, \ldots t_n$ are type variables for the formal arguments, τ is the return type, C collects constraints generated for the method body, and $\overline{t}$ collects type variables locally generated for this method. When the method is an instance method, the last argument t_n is the type of the receiver ("this"). This simple self-passing approach avoids altering the type scheme of an instance method when it is inherited by sub-classes. For method invocation `e.m(e1, ..., eN)`, if a static method is invoked, constraint $(\forall\ \overline{t}.\ \tau \setminus C) <: ([\![\texttt{e1}]\!] \times \ldots \times [\![\texttt{eN}]\!]) \rightarrow [\![\texttt{e.m(e1,...,eN)}]\!]$ is generated, with $(\forall\ \overline{t}.\ \tau \setminus C)$ as the method's type scheme; if an instance method named `m`, with `m` having abstract identifier l, is invoked, $[\![\texttt{e}]\!] <: [\, l : ([\![\texttt{e1}]\!] \times \ldots \times [\![\texttt{eN}]\!] \times [\![\texttt{e}]\!]) \rightarrow [\![\texttt{e.m(e1,...,eN)}]\!] \,]$ is generated, meaning that `e` is an object whose instance method l is invoked with `e1, ..., eN` as arguments, and the return value "flows-to" program point `e.m(e1, ..., eN)`.

Every class is given two type schemes. First there is a *creation-type-scheme* of the form $(\forall\ \overline{t}.\ t \to \mathbf{obj}(\delta, [\, l_i : \tau_i \,]) \setminus \{\})$, which is a type scheme for a function returning an object upon application. In the above object type, every instance field is associated with a fresh imperative type variable u, which is bound in the type scheme. Secondly, there is an *initialization-type-scheme* of the form $(\forall\ \overline{t}.\ t \to \mathbf{null} \setminus C)$, which is a function type scheme taking the object created as argument. The constraint set C collects constraints generated for all the instance initialization code of the class, including a constraint for applying the parent class's initialization-type-scheme. Thus, expression `new C()` is given initial constraints $\tau_1 <: t \to [\![\texttt{new C()}]\!]$, $\mathbf{null} <: t$, and $\tau_2 <: [\![\texttt{new C()}]\!] \to \mathbf{null}$, where τ_1 and τ_2 are the creation-type-scheme and initialization-type-scheme of class `C`, respectively.

A downcast expression `(T)e` is given constraint $[\![\texttt{e}]\!] <: \mathbf{cast}(\delta, [\![\texttt{(T)e}]\!])$, with δ being the identifier associated with the class or interface named `T`. Similar constraints are used to analyze array element assignments, which also incur run-time typecast checks.

2.3 Computation of the Closure

After generating the initial set of constraints corresponding to the static program text, the inference algorithm computes the complete flow information by applying the set of closure rules of Figure 1 to the constraint set. Each rule specifies a condition under which more constraints are generated; the existence of constraints above the line dictates the generation of constraints below the line. The closure computation starts with the initial constraint set, and the closure rules are applied until no more constraints can be generated. This process is an analysis-time analogy of program execution, and the rules conservatively ensure that all potential program execution paths are covered.

(Trans) $$\frac{\tau v <: t, \quad t <: \tau}{\tau v <: \tau}$$

(Read) $$\frac{\mathbf{obj}(\delta, [\, l : u, \ldots \,]) <: [\, l : \mathbf{read}\ \tau \,]}{u <: \tau}$$

(Write) $$\frac{\mathbf{obj}(\delta, [\, l : u, \ldots \,]) <: [\, l : \mathbf{write}\ \tau \,]}{\tau <: u}$$

(Cast) $$\frac{\mathbf{obj}(\delta, [\, \overline{l_i : \tau_i} \,]) <: \mathbf{cast}(\delta', \tau), \quad \delta \text{ Java-subtype-of } \delta'}{\mathbf{obj}(\delta, [\, \overline{l_i : \tau_i} \,]) <: \tau}$$

(Message) $$\frac{\mathbf{obj}(\delta, [\, l : (\forall\ \overline{t}.\ \tau \setminus C), \ldots \,]) <: [\, l : (t'_1 \times \ldots \times t'_n) \to \tau' \,]}{(\forall\ \overline{t}.\ \tau \setminus C) <: (t'_1 \times \ldots \times t'_{n-1} \times t') \to \tau',\ \mathbf{obj}(\delta, [\, l : (\forall\ \overline{t}.\ \tau \setminus C), \ldots \,]) <: t'}$$

(∀-Elim) $$\frac{(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C) <: (t'_1 \times \ldots \times t'_n) \to \tau', \quad \tau v_i <: t'_i}{\tau v_i <: \Theta(t_i), \quad \Theta(\tau) <: \tau', \quad \Theta(C)}$$

Fig. 1. Constraint Closure Rules

The transitivity rule (Trans) models run-time dataflow by propagating value types forward along flow paths. The (Read) rule applies when an object of type $\mathbf{obj}(\delta, [\, l : u, \ldots])$ reaches a read operation on field l, and the result of the read is of type τ. The (Write) rule applies when a write operation on instance field l is applied to an object whose field l is of type u.

The (Cast) rule enforces the Java run-time typecast mechanism. Recall that type $\mathbf{cast}(\delta', \tau)$ is for a downcast operation casting an expression to Java type δ', and τ is type for the result of the cast operation. The rule applies when an object of type $\mathbf{obj}(\delta, [\, \overline{l_i : \tau_i} \,])$ is subject to the cast operation. If δ is a Java-subtype δ', the cast succeeds and the object becomes the result of the cast expression. Otherwise, the cast fails and the downcast is recorded as unsafe by the inference algorithm.

The (Message) rule applies when an object reaches the receiver position of an instance method invocation. Since the last formal argument of the method's type scheme is for the receiver of the method invocation, a new arrow type is created by putting the type of the actual receiver object as the last argument, and a constraint is generated which invokes the method with the actual arguments specified by the new arrow type.

The most important closure rule is ($\forall$-Elim), which models method invocation or object creation. Constraints $\tau v_i <: t'_i$ indicate that values of type τv_i flow in as the actual arguments. At run-time each method invocation allocates fresh locations on the stack for all variables. To model this behavior in the analysis, a renaming $\Theta \in \mathbf{TypeVar} \xrightarrow{\mathrm{p}} \mathbf{TypeVar}$ is applied to type variables in $\overline{t}$. The partial function Θ is extended to types, constraints, and constraint sets in the usual manner. We call a renaming Θ a *contour*, and we call $\Theta(\tau)$ an *instantiation* of τ. The $\forall$ is eliminated from $(\forall\, \overline{t}.\ t \to \tau \setminus C)$ by applying Θ to C.

The ($\forall$-Elim) rule is parameterized by Θ, which decides for this particular function, call site, and actual argument types, which contour is to be used (*i.e.*, if a new contour is to be created or an existing contour reused). Providing a concrete Θ instantiates the framework to give a concrete algorithm. For example, the monomorphic algorithm 0CFA is defined by letting Θ be the identity renaming.

Besides detecting potentially unsafe downcast expressions, the closure computation outputs the *closure*, the set of constraints closed under the closure rules, as the analysis result. The closure contains complete flow information about the program, and various program properties can be deduced from it. For example, a concrete class analysis can be defined as follows.

Definition 2.2 (Concrete Class Analysis): For an expression e in the program, the set of concrete classes for e, $CC(e)$, is the maximal set of classes, such that if the closure contains constraint $\mathbf{obj}(\delta, [\, \overline{l_i : \tau_i} \,]) <: t$, and t is an instantiation of $[\![\mathsf{e}]\!]$, then $\delta \in CC(e)$.

To obtain a CPA instantiation of the framework, Θ in the ($\forall$-Elim) rule is defined as follows:

For each $\alpha \in \overline{t}, \Theta(\alpha) = \alpha^{\tau v_1 \times \ldots \times \tau v_n}$.

The contours Θ are generated based on the actual argument types τv_i, thus two applications of the function share the same contour if and only if the actual argument types are the same.

3 Data Polymorphic Analysis

In this section, data polymorphism and our DCPA closure algorithm are defined.

3.1 Motivation

Data polymorphism is defined in [Age96] as the ability of an imperative program variable to hold values of different types at run-time. For example, a field declared to be of type `Object` in Java can store objects of any class. Hence objects created from the same class can behave differently with their fields assigned with values of different types. Recall that object creation `new C()` is analyzed with constraints $(\forall\, \overline{t}.\ \tau \setminus C) <: t \rightarrow [\![\texttt{new C()}]\!]$ and **null** $<: t$, where $(\forall\, \overline{t}.\ \tau \setminus C)$ is the creation-type-scheme of class `C`. For 0CFA and CPA, the ($\forall$-Elim) rule generates only one contour for the creation-type-scheme of every class, and a single object type is assigned to all objects created from a given class. This may lead to a precision loss in the analysis result.

Consider the program of Figure 2. Two instances of `Hashtable` are created

```
import java.util.Hashtable;
class A {
    public static void main(String args[]) {
        Hashtable ht1=new Hashtable();
        Hashtable ht2=new Hashtable();
        ht1.put("zero", new Integer(0));
        ht2.put("true", new Boolean(true));
        Integer i=(Integer)ht1.get("zero");
    }
}
```

Fig. 2. Java program with data polymorphism

and used differently, yet CPA allows them to share the same object type, and the analysis would imprecisely conclude that the result of `ht1.get("zero")` includes `Boolean` objects. If on the other hand two separate object types were used for the two `Hashtable` instances, this downcast would be statically verified as sound.

In order to more accurately analyze such programs, we have developed a Data-Polymorphic CPA (DCPA) algorithm, which extends CPA to effectively analyze data polymorphic programs. The basic idea is to divide CPA contours into two categories: those unrelated to data polymorphism and can be shared

without losing precision (the *CPA-safe* or *reusable* contours), and those which may contain data polymorphism and thus sharing such contours might cause a loss of precision (the *CPA-unsafe* contours). The DCPA algorithm is the same as CPA except that: after every contour is generated, it judges the contour to be *CPA-safe* or *CPA-unsafe*; when there is a need to reuse an existing contour, yet the contour is already judged to be *CPA-unsafe*, DCPA would generate a fresh contour instead of reusing the existing one. DCPA aims to be precise by detecting as CPA-unsafe those contours which exhibit data polymorphism, and aims to be efficient by declaring as many contours CPA-safe as possible.

We now discuss the idea in detail. We first consider the analysis of object creations. Recall that an object creation expression `new C()` is analyzed with a pair of constraints $(\forall\ \overline{t}.\ \tau\ \backslash\ C) <: t \rightarrow [\![\texttt{new C()}]\!]$ and $\mathbf{null} <: t$, where $(\forall\ \overline{t}.\ \tau\ \backslash\ C)$ is the creation-type-scheme of class `C`. We will refer such a pair of constraints as the *creation points* of class C. If a class contains any polymorphic field (a polymorphic field is a field which can store values of different types), DCPA algorithm always judges contours of the creation-type-scheme of such a class as CPA-unsafe. Thus, for any creation point of a class with polymorphic fields, a fresh object type is generated for the class. On the other hand, if a class (e.g. class `java.lang.Integer`) contains no polymorphic fields, DCPA algorithm would judge the contour of the class's creation type scheme as CPA-safe, thus all objects of the class share a single object type.

We now consider contours generated for method invocations. If CPA loses precision because of data polymorphism, there must be multiple objects from the same class such that those objects are used differently at run-time yet CPA let them share the same object type. The goal of DCPA is to generate more object types so that such imprecision can be avoided. Thus, if a method invocation doesn't cause the creation of any objects, the contour for such a method invocation is CPA-safe. For example, consider the program in Figure 3. DCPA would let the two invocations of the method `id` share a single CPA-safe contour.

Furthermore, if a method invocation creates objects, but the objects created do not escape the method scope via the return value, then the contour for such a method invocation is also CPA-safe. Consider the program in Figure 3. Since the two invocations of method `g` have the same argument type, CPA would let them share a single contour. The two invocations create two `Vector` instances, but the two instances escape the scope of `g` only through static field `a`, not through the return values. If two distinct contours were used for the two invocations of method `g`, there would be two creation points of class `Vector`, and there would be two distinct object types generated for `Vector`. But the two object types would both be lower bounds of the type variable for field `a`, forcing them to be equivalent in any case. Thus, for the two invocations of method `g`, generating two contours are not beneficial, and DCPA would let them share a single CPA-safe contour without any loss of precision.

Even if an object created by a method invocation escapes the method scope through the return value, if the fields of the object are already assigned with values of fixed types, such an invocation is also considered CPA-safe. For exam-

ple, consider method `h` in Figure 3. A `Vector` object is created and returned by the method. But before it is returned, a `Boolean` object is already put in the vector. Thus the two vectors created by the two invocations of `h` would both have contents of `Boolean` type. CPA would let the two invocations of method `h` share a single contour. If two distinct contours were used for the two invocations of method `h`, there would be two creation points of class `Vector`, and there would be two distinct object types generated for `Vector`. But the two object types would both have `Boolean` as content type. Since all fields of `Boolean` objects are also assigned values of fixed types, we consider that the type for the two vectors as already known and consider generating two distinct contours as not beneficial. Therefore, DCPA would let the two invocations of method `h` share a single CPA-safe contour.

```
import java.util.*;
class A {
  static Object a;
  static Object id(Object x) { return x;}
  static Object g(Object x) { a=new Vector(); return x;}
  static Object h() {
    Vector v=new Vector();
    v.addElement(new Boolean(true));
    return v;
  }
  public static void main(String args[]) {
    Object obj=new Hashtable();
    id(obj); id(obj); g(obj); g(obj);
    h(); h();
  }
}
```

Fig. 3. Example Program with CPA-Safe Contours

For some programs, even with the above strategy which aims to identify as many CPA-safe contours as possible, there are still too many contours declared as CPA-unsafe. To make the algorithm feasible, an additional unification heuristic is incorporated into DCPA. The idea is that, whenever two object types of the same class "flow together" (*i.e.*, become lower bounds of a single type variable), the algorithm assumes that the data structures represented by the two object types would be used in the same way and it unifies the two object types. Although this unification mechanism could in theory cause precision loss, in practice we have found that such cases are rare.

3.2 The DCPA Algorithm

We now give a definition of the DCPA closure algorithm. The DCPA algorithm computes a constraint closure from the same initial constraints as defined in the

previous section, using stack-based algorithm. Whenever a contour is generated for a method invocation, the algorithm checks the return type of the contour to judge if the contour is reusable. To do that, it needs to know the "local" set of constraints corresponding to the local computation of the method invocation. Since contours are nested, in general it must keep a stack of constraints which mirrors the call stack, and the closure computation itself is specified by a state-transition machine where the state is a stack of constraints.

First, we define the representation of contours.

Definition 3.1 (Contour): A contour ρ is a triple of form $(c, \overline{\tau v_i}, \Theta)$, recording information about a $\forall$-elimination, where c is the constraint of form $(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C) <: (t'_1 \times \ldots \times t'_n) \to \tau'$, $\overline{\tau v_i}$ is a vector of the actual argument types, and Θ is the renaming on $\overline{t}$ used by the $\forall$-Elimination.

We define the closure computation as a state transition process, where a state is defined as follows:

Definition 3.2 (Closure State): A *closure state* is a pair (S, Ω). S is a stack with frames of form (ρ, C), where ρ is a contour and C is a set of constraints. Ω is the contour cache, which is a set of pairs of form (ρ, b), where ρ is a contour and b is a boolean value.

A contour is an analysis-time analogy of a run-time activation record, and the state is a stack that corresponds to the run-time stack of activation records. A stack frame (ρ, C) represents a contour with C as the set of local constraints for the contour. Every time a new contour is created, it is pushed onto the stack. When the closure computation is finished for constraints generated by this contour, the contour is popped off the stack. The top frame of the stack corresponds to the current contour. The contour cache Ω stores all contours thus far created. Each contour is associated with a boolean value, which is true iff the contour is reusable. We will write $S \wedge [(\rho, C)]$ to indicate a stack with (ρ, C) as the top frame, and $[(\rho_n, C_n), \ldots, (\rho_1, C_1)]$ as the stack with n frames enumerated from bottom to top. For constraint c and state (S, Ω), we write $c \in S$ if $c \in C_i$ and (ρ_i, C_i) is a frame in S.

Rules for closure state transition are shown in Figure 3.2. Every rule specifies a transition from the closure state above line to the state below the line. The notation (S, Ω) **with** C specifies a closure state where every constraint in C occurs somewhere in S, and there exists at least one constraint in C which occurs in the top frame of S.

The (S-Trans) transitivity rule applies only when at least one constraint amongst $\tau v <: t$ and $t <: \tau$ is in set C_1 (the constraint set in the top frame of the state stack), and the constraint $\tau v <: \tau$ generated by the transitivity is added to C_1. This serves to maintain the invariant that any constraints generated from the constraint set in the stack top will also become part of the constraint set in the stack top. The rules (S-Read), (S-Write), (S-Cast), (S-Message) can be understood similarly. In (S-Message) and subsequent rules, notation $\times_n^{i=1}\ t_i$ abbreviates $t_1 \times \ldots \times t_n$.

(S-Trans) $\dfrac{(S \wedge [(\rho_1, C_1)], \Omega)\ \textbf{with}\ \{\tau v <: t, t <: \tau\}}{(S \wedge [(\rho_1, \{\tau v <: \tau\} \cup C_1)], \Omega)}$

(S-Read) $\dfrac{(S \wedge [(\rho_1, C_1)], \Omega)\ \textbf{with}\ \{\textbf{obj}(\delta, [\, l : u, \ldots\,]) <: [\, l : \textbf{read}\ \tau\,]\}}{(S \wedge [(\rho_1, \{u <: \tau\} \cup C_1)], \Omega)}$

(S-Write) $\dfrac{(S \wedge [(\rho_1, C_1)], \Omega)\ \textbf{with}\ \{\textbf{obj}(\delta, [\, l : u, \ldots\,]) <: [\, l : \textbf{write}\ \tau\,]\}}{(S \wedge [(\rho_1, \{\tau <: u\} \cup C_1)], \Omega)}$

(S-Cast) $\dfrac{(S \wedge [(\rho_1, C_1)], \Omega)\ \textbf{with}\ \{\textbf{obj}(\delta, [\overline{l_i : \tau_i}]) <: \textbf{cast}(\delta', \tau)\},\ \delta \text{ Java-subtype-of } \delta'}{(S \wedge [(\rho_1, \{\textbf{obj}(\delta, [\overline{l_i : \tau_i}]) <: \tau\} \cup C_1)], \Omega)}$

(S-Message) $\dfrac{(S \wedge [(\rho_1, C_1)], \Omega)\ \textbf{with}\ \{\textbf{obj}(\delta, [\, l : (\forall\ \overline{t}.\ \tau \setminus C), \ldots\,]) <: [\, l : (\times_n^{i=1}\ t_i') \to \tau'\,]\}}{\begin{array}{l}(S \wedge [(\rho_1, \{(\forall\ \overline{t}.\ \tau \setminus C) <: ((\times_{n-1}^{i=1}\ t_i') \times t') \to \tau',\\ \textbf{obj}(\delta, [\, l : (\forall\ \overline{t}.\ \tau \setminus C), \ldots\,]) <: t'\} \cup C_1)], \Omega)\end{array}}$

(S-∀-Reuse) $\dfrac{\begin{array}{l}(S \wedge [(\rho_1, C_1)], \Omega)\ \textbf{with}\ \{(\forall\ \overline{t}.\ (\times_n^{i=1}\ t_i) \to \tau \setminus C) <: (\times_n^{i=1}\ t_i') \to \tau', \tau v_i <: t_i'\},\\ (((\forall\ \overline{t}.\ (\times_n^{i=1}\ t_i) \to \tau \setminus C) <: (\times_n^{i=1}\ t_i'') \to \tau'', \overline{\tau v_i}, \Theta), \texttt{true}) \in \Omega\end{array}}{(S \wedge [(\rho_1, C_1 \cup \{\Theta(\tau) <: \tau'\})], \Omega)}$

(S-∀-Begin) $\dfrac{\begin{array}{l}(S \wedge [(\rho_1, C_1)], \Omega)\ \textbf{with}\ \{(\forall\ \overline{t}.\ (\times_n^{i=1}\ t_i) \to \tau \setminus C) <: (\times_n^{i=1}\ t_i') \to \tau', \tau v_i <: t_i'\},\\ (((\forall\ \overline{t}.\ (\times_n^{i=1}\ t_i) \to \tau \setminus C) <: \times_n^{i=1}\ t_i'' \to \tau'', \overline{\tau v_i}, \Theta''), \texttt{true}) \notin \Omega,\\ (((\forall\ \overline{t}.\ (\times_n^{i=1}\ t_i) \to \tau \setminus C) <: \times_n^{i=1}\ t_i' \to \tau', \overline{\tau v_i}, \Theta'), b) \notin \Omega\end{array}}{\begin{array}{l}(S \wedge [(\rho_1, C_1), (\rho, \{\tau v_i <: \Theta(t_i)\} \cup \Theta(C))], \Omega)\\ \text{where } \rho = ((\forall\ \overline{t}.\ (\times_n^{i=1}\ t_i) \to \tau \setminus C) <: (\times_n^{i=1}\ t_i') \to \tau', \overline{\tau v_i}, \Theta),\\ \text{and } \Theta \text{ is a fresh renaming on type variables in } \overline{t}\end{array}}$

(S-∀-End) $\dfrac{(S \wedge [(\rho_2, C_2), (\rho_1, C_1)], \Omega)}{\begin{array}{l}(S \wedge [(\rho_2, C_2 \cup C_1 \cup \{\Theta(\tau) <: \tau'\})], \Omega \cup \{(\rho_1, b)\})\\ \text{where for } \rho_1 = ((\forall\ \overline{t}.\ (\times_n^{i=1}\ t_i) \to \tau \setminus C) <: (\times_n^{i=1}\ t_i') \to \tau', \overline{\tau v_i}, \Theta),\\ b \text{ is } \texttt{true} \text{ iff } \rho_1 \text{ is reusable in } (S \wedge [(\rho_2, C_2), (\rho_1, C_1)], \Omega)\end{array}}$

(S-Unify) $\dfrac{(S, \Omega)\ \textbf{with}\ \{\textbf{obj}(\delta, [\overline{l_i : \tau_i}]) <: t, \textbf{obj}(\delta, [\overline{l_i' : \tau_i'}]) <: t\}}{\Theta((S, \Omega)),\quad \text{where } \Theta = U(\textbf{obj}(\delta, [\overline{l_i : \tau_i}]), \textbf{obj}(\delta, [\overline{l_i : \tau_i'}]))}$

Fig. 4. DCPA Constraint Closure Rules

The (∀-Elim) rule of the framework corresponds to three DCPA closure rules. (S-∀-Reuse) is used in the case that the ∀-elimination reuses an existing contour. A contour ρ is reusable for this application if $(\rho, \texttt{true}) \in \Omega$ and ρ has the same argument types as this application. According to the (∀-Elim) closure rule, this ∀-elimination would generate such a set of constraints: $\{\tau v_i <: \Theta(t_i), \Theta(\tau) <: \tau'\} \cup \Theta(C)$. Since this is a reuse of an existing contour, all those constraints except for the constraint $\Theta(\tau) <: \tau'$ have already been generated and those constraints are already in the constraint sets of the stack. Thus, only constraint $\Theta(\tau) <: \tau'$ is added to the constraint set in the top frame of the stack.

The rule (S-∀-Begin) creates a new contour. The condition $(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C) <: t_1'' \times \ldots \times t_n'' \to \tau'', \overline{\tau v_i}, \Theta''), \texttt{true}) \notin \Omega$ means that there is no reusable (CPA-safe) contour for the same function applied to the same argument type τv_i in the contour cache. The condition $(((\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C) <: t_1' \times \ldots \times t_n' \to \tau', \overline{\tau v_i}, \Theta'), b) \notin \Omega$ means that it is not the case that there is already a contour generated for this application. When the two conditions are satisfied, a fresh contour is generated for the application. The local constraint set of the new contour is initialized to include constraints $\tau v_i <: \Theta(t_i)$ and the fresh copy of all bound constraints, $\Theta(C)$. The constraint $\Theta(\tau) <: \tau'$ corresponding

to the flow from the return value of the function to the application result is not considered a local constraint of the contour. Finally, the new contour along with its local constraint set is pushed on to the stack.

The rule (S-∀-End) ends the creation of a contour. It pops the current contour off the stack, merges the local constraints of this contour with those in the new top frame of the stack, generates constraint $\Theta(\tau) <: \tau'$ corresponding to the flow from the return value to the application result, and checks if the contour is reusable and records it in the contour cache. The definition of *reusable* is now defined, via the notions of *local type* and *complete type*.

Definition 3.3 (Local Types): Object type $\mathbf{obj}(\delta, [\overline{l_i : \tau_i}])$ is *local* to state $([(\rho_n, C_n), \ldots, (\rho_1, C_1)], \Omega)$ if $\mathbf{obj}(\delta, [\overline{l_i : \tau_i}])$ does not appear as a subterm of some constraint in C_i for any $i \neq 1$.

Type $\mathbf{obj}(\delta, [\overline{l_i : \tau_i}])$ is declared local if and only if it is created locally by the current contour.

Definition 3.4 (Complete Types): Type τ is *complete* in state (S, Ω) iff any of the following cases holds:

1. τ is a primitive Java type, e.g. **int** or **bool**;
2. $\tau = t$, $\tau v_1 <: t \in S$, $\tau v_2 <: t \in S$, and $\tau v_1 \neq \tau v_2$;
3. $\tau = t$, $\tau' <: t \in S$, and τ' is complete in (S, Ω);
4. $\tau = \mathbf{obj}(\delta, [\overline{l_i : \tau_i}])$, and for any τ_i which is imperative type variable u_i, u_i is complete in (S, Ω);
5. $\tau = \mathbf{obj}(\delta, [\overline{l_i : \tau_i}])$, and $\mathbf{obj}(\delta, [\overline{l_i : \tau_i}])$ is not local to (S, Ω).

In the first case above, **int** and **bool** and all other Java primitive types are judged complete. In cases 2. and 3., a type variable is considered complete if it has a complete type or at least two different value types as lower bounds. In cases 4. and 5., object type $\mathbf{obj}(\delta, [\overline{l_i : \tau_i}])$ is complete if the type variables for all its fields are complete, or it is not created locally by the current contour. The above definition is subtle, and the details are critical in obtaining an accurate and efficient analysis.

Definition 3.5 (Reusable Contour): A contour $((\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C) <: (t'_1 \times \ldots \times t'_n) \rightarrow \tau', \overline{\tau v_i}, \Theta)$ is *reusable* in state (S, Ω) iff $\Theta(\tau)$ is complete in (S, Ω) and there does not exist $(((\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C) <: t''_1 \times \ldots \times t''_n, \overline{\tau v_i}, \Theta'), \texttt{false}) \in \Omega$.

The above definition judges a contour ρ as not reusable if the return type of the contour is not complete.

The rule (S-Unify) unifies two object types when they are from the same class, making them lower bounds of the same type variable. The unification algorithm is defined as follows.

Definition 3.6 (Unification of Object Types): The unifier of the two object types of same class l, $U(\mathbf{obj}(\delta, [\, l_i : \tau_i \,]), \mathbf{obj}(\delta, [\, l_i : \tau'_i \,]))$, is a composition of substitutions $\lfloor \tau_i / \tau'_i \rfloor$ for each i s.t. τ_i is an imperative type variable.

We now define the closure process itself, via a state transition relation which is prioritized to make sure a contour is not popped until its analysis has finished.

Definition 3.7 (DCPA State Transition Relation $\mapsto$): $s_1 \mapsto s_2$ iff closure state s_1 transits to s_2 by a rule in Figure 3.2, and one of the following three cases hold: the rule is neither (S-∀-Begin) nor (S-∀-End); the rule is (S-∀-Begin), and the only other applicable rule on s_1 is (S-∀-End); or, the rule is (S-∀-End), and no other rules are applicable on s_1. $\overset{*}{\mapsto}$ is the transitive, reflexive closure of $\mapsto$.

The closure computation is a state-transition process via relation $\mapsto$ until a fixed point is reached.

Definition 3.8 (DCPA Closure Computation): Given a constraint set C, if $([(\rho_0, C)], \{\}) \overset{*}{\mapsto} ([(\rho_0, C')], \Omega)$, (where ρ_0 is a special dummy contour), and for any state s such that $([(\rho_0, C')], \Omega) \mapsto s$, we have $s = ([(\rho_0, C')], \Omega)$, then we say $([(\rho_0, C)], \{\}) \overset{\texttt{DCPA}}{\longmapsto} ([(\rho_0, C')], \Omega)$, and also will write $C \overset{\texttt{DCPA}}{\longmapsto} C'$, meaning C' is the DCPA closure of C.

Now the type inference with DCPA Algorithm can be defined as follows:

Definition 3.9 (DCPA Algorithm): For a program e with initial constraint set C, if $C \overset{\texttt{DCPA}}{\longmapsto} C'$, then C' is a DCPA closure for e.

4 Implementation

In this section, we discuss our implementation of 0CFA, CPA and DCPA for Java. The system is itself written in Java. It takes Java source code as input and statically checks the validity of all down-casts in the program. For each Java downcast of the form `(T)e`, casting expression `e` to Java class or interface `T`, the system computes a set of Java classes which are conservative approximations of the classes which `e` can take on at run-time. If the algorithm discovers that `e` might evaluate to an object of class `C`, and `C` is not a Java subtype of `T`, then the downcast is reported as *unsafe*; otherwise the cast is *safe*. If a downcast is judged as safe by the system, it is guaranteed to succeed at run-time. The system currently can handle all standard Java language features, including objects, classes, interfaces, inner classes, and exceptions. The only feature the system cannot handle automatically is the reflection mechanism of Java.

Though our system is built as a downcast checker, it is essentially a concrete class analysis tool for Java. So, it could also be used as an analysis tool for static resolution of virtual method calls and other compiler optimizations.

4.1 Java Language Issues

We have up to now ignored several features of Java; here we provide a sketch of how they are handled by the implementation. Recall that an extra "this" argument is added for the type scheme of every instance method. Type schemes for constructors also have an extra "this" argument. Because of this, the processing of inheritance is simple: no classes, methods or constructors are re-analyzed upon inheritance. Every class is represented as a class object, which has the

following components: a reference to the class object for the parent class, a method-lookup-table containing type schemes for instance methods defined in this class, type schemes for constructors and static methods, and the class's creation-type-scheme and initialization-type-scheme. Such a class object implements the "class identifier" concept in our type system, and is shared by all object types created from the class.

For each abstract method of a Java interface, a hashtable is built, which associates every concrete class implementing this interface with the an instance method of the class, corresponding to the abstract method. Such hashtables are used during closure computation to determine the type scheme of the target method for a virtual method call through an interface.

Array objects are analyzed as special object types with a single field representing the array contents. Array store expressions in Java require a run-time check to ensure the type safety. Thus those expressions are analyzed in a similar way as down-cast expressions. And, the system also statically checks the type safety of every array store expression.

Inner Classes are analyzed as follows. In Java, from an instance of the inner class, an instance of every enclosing class is accessible. Thus, for every enclosing class, an additional field is added to the object types of the inner class, representing the enclosing instance. Similarly, the creating-type-scheme of an inner class contains an extra "this" argument for every enclosing class. An inner class may also access local variables in the surrounding lexical context. For every local variable accessed, we add a special field in the inner class for it, and thus convert the variable access to a field access. In this way, all type schemes generated in our system enjoy the following property: bound type variables of one type scheme never appear in the scope of another type scheme. Namely, no nested type schemes are. Thus, when instantiating a type scheme by renaming all types and constraints in its scope, no type scheme needs to be renamed.

Java exception-handling features are analyzed in a simple manner. Each exception class (*i.e.*, subclass of `java.lang.Throwable`) is represented by a unique type variable. If exception class `A` is a subclass of exception class `B`, constraint $t_1 <: t_2$ is generated, where t_1 and t_2 are type variables for classes `A` and `B` respectively. A statement `throw e` produces a special constraint $[\![\texttt{e}]\!] <:$ **exception**, where **exception** is a special type such that for any object type τ of an exception class becoming a lower-bound of **exception**, a constraint $\tau <: t$ is generated with t as the type variable corresponding to the exception class of τ. A statement of form `try {...} catch(T e) {...}` produces constraint $t <: [\![\texttt{e}]\!]$, where t is the type variable for the exception class `T`.

We also make special effort to accurately analyze a common Java programming idiom:

```
if (x instanceof A) {
   A c = (A) x;   ...
}
```

At the entry point of the true branch, it is certain that values of variable `x` are instances of class `A`. The analysis uses a simple algorithm to conservatively

estimate whether the boolean condition of an `if` statement further constrains the class of any object by presence of `instanceof`, and if so uses that fact in the analysis.

Since our analysis is a whole program analysis, libraries must be included in program analysis. We use the Sun JDK library source. Native library methods were manually replaced with type-compatible Java code. The reflection features of Java pose a significant difficulty for any static analysis. For example, it is impossible for a static analysis to determine precisely which class is dynamically loaded. Our solution is to manually replace library code using reflection with type-compatible code without reflection. For example, the code fragment `Class.forName(x).newInstance()` can be replaced with `new A()` if it is certain that class `A` is loaded.

4.2 Optimizations

We now discuss some optimizations implemented in our system to improve the performance for analyzing realistic Java applications.

Optimization with Monomorphic Types Monomorphic types include Java primitive types (*e.g.*, `int`, `boolean`) and object types of monomorphic classes. Monomorphic classes (*e.g.*, `String`, `Integer`) are those classes which do not have subclasses and only have fields capable of storing values of monomorphic values. If according to the Java static type declaration, a variable or expression is of a monomorphic type, we use the monomorphic type directly as the type for such a variable or expression without any type inference effort.

Additional Contour Sharing DCPA as defined will in fact not always terminate on recursive programs. We previously formalized a provably terminating CPA algorithm [SW00]. Since no nested type schemes are generated in our current system, CPA would always terminate without any special termination mechanism. To ensure the termination of DCPA, our system performs online recursion detection, and always treats contours for recursive methods as CPA-safe.

Another issue is to prevent the algorithm from creating too many contours on certain pathological cases. Agesen [Age96] defines the notion of *megamorphism*, which means too many different value types flow to a single call site as arguments. To prevent CPA from blowing up, the number of contours generated for a megamorphic call site is reduced. This idea is also employed in our system. In theory, DCPA may also blow up when too many contours are judged as CPA-unsafe. To prevent DCPA from blowing up in this case, a contour is regarded as CPA-safe when too many object types are created locally by the contour.

Online Cycle Elimination Partial online cycle elimination [AFFS98] is another optimization incorporated in our system. The basic idea is to detect cycles of the form $t_1 <: t_2 \ldots <: t_n <: t_1$, and collapse all type variables on such cycles

into a single variable. We have implemented the cycle elimination mechanism using a novel approach. Instead of performing cycle detection as a separate operation at every update of the constraint system, as in [AFFS98], we piggyback the cycle detection operation on the process of propagating value types along flow paths. Whenever the (Trans) closure rule is applied on constraint $\tau v <: t$, τv also needs to be propagated to type variables which are upper bounds of t. Our system performs cycle detection on t while propagating τv forward. It keeps track of type variables visited on the current flow path so an already-visited variable will be discovered. In this way, the overhead of cycle detection is reduced.

Automatic Constraint Garbage Collection Constraints which will induce no future closure computation have served their purpose and can be garbage collected. It is possible to precisely detect which constraints are garbage [EST95, Pot98], but these algorithms are nontrivial and the act of detecting the garbage itself slows down the analysis. We use a simpler form of garbage collection which is automatic: constraints are represented in the implementation in a manner such that many unreachable constraints will be automatically collected by the Java run-time garbage collector.

There are additional constraint-based optimizations which could be included in our system to further improve its performance, including constraint graph minimization [FF97] and precise garbage collection.

5 Experimental Results

Benchmark results are presented in Table 5. The following benchmark programs were used: *jlex*[1] is a lexical analyzer generator; *toba*[2] is a Java-to-C code translator; *javacup*[3] is a Java parser generator; *jtar*[4] is an archive utility; *bloat*[5] is a Java bytecode optimizer; *self* is our system itself used as a benchmark; *sablecc*[6] is a compiler generator; *javac* and *javadoc* are standard tools in Sun's Java SDK.

The column "lines" shows the number of lines of source code in the benchmark program only. Since our system is a whole-program analysis, every benchmark was analyzed along with the reachable library code. The column "methods" shows the number of reachable methods in the whole program including libraries detected by DCPA algorithm; The column "casts" shows the number of downcasts in the benchmark program only. Downcasts reachable in the library code are also checked, but checking downcasts in user code is the goal of the system and only those casts are reported. Each benchmark is analyzed with 0CFA, CPA and DCPA. The columns labeled "safe" indicate the percentage of

[1] see www.cs.princeton.edu/~appel/modern/java/JLex/
[2] see www.cs.arizona.edu/sumatra/toba/
[3] see www.cs.princeton.edu/~appel/modern/java/CUP/
[4] see www.angelfire.com/on/vkjava/
[5] see www.cs.purdue.edu/homes/hosking/bloat
[6] see www.sable.mcgill.ca/sablecc/

				0CFA		CPA			DCPA		
Program	lines	methods	casts	safe	time	safe	time	Θ	safe	time	Θ
jlex	7835	398	65	10.8%	3.3	16.9%	3.5	1.5	100%	3.8	2.3
toba	6417	777	63	4.8%	4.8	4.8%	5.1	1.6	22.2%	6.4	2.7
javacup	10592	532	459	8.1%	3.6	8.1%	3.9	1.4	89.3%	4.5	3.6
jtar	11904	1446	10	0%	6.7	0%	7.1	1.5	100%	11.0	2.4
bloat	18841	1053	205	7.8%	5.3	8.8%	5.9	1.5	30.2%	7.1	4.3
self	23122	1304	130	36.9%	4.5	51.5%	5.9	1.9	93.8%	10.7	3.4
javadoc	–	2314	310	25.3%	10.1	40.8%	13.0	1.9	77.7%	23.6	4.3
sablecc	23111	2811	519	34.5%	10.9	35.2%	10.9	2.5	61.7%	21.3	4.1
javac	–	2933	606	17.1%	12.5	30.5%	30.0	2.5	50.1%	74.8	7.1

Table 1. Benchmark Data

total user downcasts which have been statically verified. The columns labeled "time" report system execution time in seconds, including time for parsing, type inference and closure computation. For CPA and DCPA, columns labeled "Θ" report the average number of contours generated for each type scheme; this is always 1 for 0CFA.

The benchmark results were obtained using the Sun JDK 1.3 on a PC with a 866MHZ Pentinum processor and 512M of memory. All benchmarks except *javac* were analyzed with a 80M maximum heap size. *javac* has a very complex inheritance hierarchy, and its analysis is significantly more complex than the other benchmarks. The results for *javac* with 0CFA, CPA, DCPA were obtained with 80M, 96M and 160M maximum heap size, respectively.

As can be seen in the benchmarks, DCPA can verify significantly more downcasts than either CPA or 0CFA. For example, all downcasts in user code of *jlex* and *jtar* have been statically verified. This shows that CPA and 0CFA are not precise enough for downcast checking, and in general, DCPA is a much more precise type inference algorithm for object-oriented languages. We have manually studied the downcasts which cannot be verified by DCPA for some benchmark programs. Nearly all of them cannot be verified even with an analysis that would generate a fresh contour for every function application. Some of the remaining downcasts could be verified by a flow-sensitive analysis, but most are fundamentally "dynamic", with safety that depends on the state of execution, and thus not verifiable by any static analysis of this variety. For example, DCPA can only verify 22.2% of the downcasts in toba, but a manual inspection shows that nearly all of the remaining downcasts are fundamentally dynamic. In summary, DCPA appears to produce nearly optimal results as a flow-insensitive static analysis for downcast checking on the programs we have tested. DCPA also has good efficiency: comparing the time and the average number of contours of CPA and DCPA, we can see that DCPA's efficiency is comparable to CPA; furthermore, realistic Java applications can be analyzed.

6 Related Work

Plevyak and Chien's iterative flow analysis (IFA) [PC94] is a precise constraint-based analysis of object-oriented programs. To the best of our knowledge, IFA is the only system in the literature capable of analyzing data polymorphic programs precisely. IFA uses an iterative approach in which the whole analysis must be iterated multiple times. Compared to IFA, our system detects data polymorphism online, and does not need generational iteration.

O'Callahan [Cal99] has built a system for the analysis of Java bytecode. The system is used for static verification of Java downcasts. His type schemes are much more compact yet less precise than the constraint-based type schemes used in our system. While our system aims to reuse contours across different call sites and only produces a few contours on average for each type scheme, his system is fully context-sensitive and always instantiates a type scheme differently in every different context. Duggan has also proposed a system to automatically detect polymorphic Java classes [Dug99]. His proposal does not appear to be as precise as ours. It is currently not implemented or tested with benchmarks and so its feasibility and performance are unclear.

The analysis we produce here is perhaps the most precise constraint-based analysis for object-oriented programs thus far developed. We are going opposite the common trend today, which is toward less expressive (but more efficient) analyses (*e.g.,* [SHR$^+$00, TP00]). These fast analyses are getting popular because for many purposes it has become clear that a fine-grained analysis is not needed. But, this paper shows that there still are purposes, including cast-checking, where it is critical to have a very fine-grained analysis.

References

[AFFS98] Alexander Aiken, Manuel Fhndrich, Jeffrey S. Foster, and Zhendong Su. Partial online cycle elimination in inclusion constraint graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

[Age95] Ole Agesen. The cartesian product algorithm. In *Proceedings ECOOP'95*, volume 952 of *Lecture notes in Computer Science*, 1995.

[Age96] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD thesis, Stanford University, 1996. Available as Sun Labs Technical Report SMLI TR-96-52.

[AW93] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[Cal99] Robert O' Callahan. Optimizing a solver of polymorphism constraints: SEMI. Technical Report CMU-CS-99-136, CMU, June 1999.

[Dug99] Dominic Duggan. Modular type-based reverse engineering of parameterized types in java code. In *ACM SIGPLAN Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, 1999.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95*, pages 169–184, 1995.

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 235–248, New York, June 15–18 1997. ACM Press.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1997.

[JW95] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 393–408, 1995.

[PC94] John Plevyak and Andrew Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, 1994.

[Pot98] François Pottier. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, September 1998.

[PP98] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *POPL*, 1998.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. Available as CMU Technical Report CMU-CS-91-145.

[SHR+00] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Valee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.

[SW00] Scott Smith and Tiejun Wang. Polyvariant flow analysis with constrained types. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, volume 1782 of *Lecture Notes in Computer Science*, pages 382–396. Springer Verlag, March 2000.

[TLSS99] F. Tip, C. Laffra, P. Sweeney, and D. Streeter. Practical experience with an application extractor for java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1999.

[TP00] F. Tip and J. Palsberg. Scalable propagation-based call graph construction. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2000.