Polymorphic Constraint-Based Type Inference for Objects

TIEJUN WANG and SCOTT F. SMITH
The Johns Hopkins University

Constraint-based type inference infers types with subtyping constraints. Such types can capture detailed data and control flow information about the analyzed program. In the presence of polymorphism, existing constraint-based type inference algorithms sacrifice much precision for efficiency. This paper presents both theoretical and practical results on developing precise and efficient polymorphic constraint-based type inference for object-oriented languages.

We develop a novel theoretical framework for polymorphic constraint-based type inference. A concrete type inference algorithm can be obtained by instantiating the framework with a particular strategy for handling polymorphism. We define well-known algorithms such as Shivers' nCFA and Agesen's Cartesian Product Algorithm (CPA) as instantiations of the framework. We prove the soundness of the framework, which entails the soundness of every algorithm defined as an instantiation of the framework. Using the framework, we develop a novel algorithm, Data Polymorphic CPA (DCPA), which extends Agesen's CPA Algorithm to achieve high precision in the presence of Data Polymorphism.

We further study the construction of practical constraint-based type inference systems for realistic programming languages. We have constructed a type inference system for the full Java language. The system includes implementations of the 0CFA, CPA and DCPA algorithms, and it incorporates a number of novel implementation techniques for achieving scalability. The system is used to statically verify the correctness of Java downcasts. Benchmark results on realistic Java applications show that the DCPA algorithm has good precision and efficiency: it is significantly more accurate than existing algorithms and its efficiency is comparable to CPA.

Categories and Subject Descriptors: H.4.0 [**Information Systems Applications**]: General

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: constraint-based type inference, type systems, polyvariance, program analysis, Java

1. INTRODUCTION

Constraint-based type inference is a method for automatically inferring an expressive form of type consisting of *subtyping constraints*. A subtyping constraint is of form $\tau_1 <: \tau_2$, meaning that type τ_1 is a subtype of τ_2. Intuitively, each type represents a set of data values, and type τ_1 is a subtype of type τ_2 if the value set represented by τ_1 is a subset of the value set represented by τ_2. In particu-

Authors' addresses: Department of Computer Science, The Johns Hopkins University, 3400 N. Charles Street, Baltimore, MD 21218; email: wtj@cs.jhu.edu, scott@cs.jhu.edu. This research was supported by the National Science Foundation grants CCR-9619843 and CCR-9988491.

lar, if there is a data flow from an expression e_1 to expression e_2 in the program, this flow information can be captured with a subtyping constraint $\mathbf{TypeOf}(e_1) <: \mathbf{TypeOf}(e_2)$ since the set of e_1's values is a subset of the set of e_2's values. For example, for a definition $y = (\lambda x.x)$ in a program, a type inference algorithm would generate a subtyping constraint $\mathbf{TypeOf}(\lambda x.x) <: \mathbf{TypeOf}(y)$, which captures the data flow from function $(\lambda x.x)$ to variable y. There is a flow-type duality [Palsberg and O'Keefe 1995], which means constraint-based type inference can statically capture complete and detailed data and control flow information of a program. Such detailed type and flow information is invaluable for analysis and optimization of programs.

It is well-known that polymorphism is an important feature of programming languages, encouraging program reuse and thus programmer productivity by enabling the construction of polymorphic program components that can be reused in different contexts. However, it is difficult for a constraint-based type inference algorithm to be both efficient and highly accurate in the presence of polymorphism.

For example, consider the program $(\lambda f.\, f\, 0;\, \texttt{not}(f\, \texttt{true}))\, (\lambda x.x)$. The identity function $(\lambda x.x)$ is polymorphic: it is used as a function of type $\mathbf{int} \rightarrow \mathbf{int}$ at one call site, and $\mathbf{bool} \rightarrow \mathbf{bool}$ at another. A monomorphic type inference algorithm would fail to typecheck this program because it only assigns a single type to the identity function. In general, to obtain a precise analysis result, the type system often needs to analyze a polymorphic function differently with respect to different calling contexts. This is usually realized via a re-analysis of the function definition for a different context. Following Shivers' terminology [Shivers 1991], we call each analysis of a function definition a *contour*. A naive algorithm would re-analyze a polymorphic function (generate a different contour) for every calling context of the function. This is too expensive to be feasible since each different call-path of the function defines a calling context of the function and the number of such call-paths could be exponential. The challenge of type inference in the presence of polymorphism is to be as precise as possible while minimizing the number of contours generated for polymorphic functions.

Shivers' nCFA family of algorithms [Shivers 1988; Shivers 1991] is an approach for analyzing programs with polymorphism. The basic idea is to associate every function application with a *call-string* of length at most n. The call-string is an analysis-time analogy of the call-path from the entry point of the whole program to this function application, and a call-string of length at most n contains the last n or fewer function call sites on the call-path leading to this application. Two different applications of the same function share the same contour (i.e., analysis of the function) if the two applications have the same call-string. The purpose of using call-strings is for achieving analysis precision by incorporating some context-sensitivity into the analysis, and the parameter n in the nCFA family of algorithms controls the degree of context-sensitivity incorporated.

The length of call-strings needed to precisely analyze calls of a polymorphic function in the program could unfortunately be large. Additionally, there is no efficient way to determine the length of the call-strings needed to precisely analyze a particular program, and so some fixed value of n is chosen which greatly over-analyzes some paths but misses the ones longer than n. So, if n is too small the analysis results are imprecise, but if it is increased to the level needed for significantly better results the

algorithm will not terminate in feasible time. Experiments show that, for some programs, even 1CFA is too expensive to be feasible [Wright and Jagannathan 1998].

Agesen's Cartesian Product Algorithm (CPA) [Agesen 1995; Agesen 1996] is a more feasible approach for analyzing programs with parametric polymorphism. The basic idea is to divide the calling contexts of a function based on the types of the arguments which the function is applied to. For every possible combination of the argument types which the function is applied to, there is one contour generated for the function. If the two applications of a function have the same argument types, they can share a same contour of the function.

Consider the program $(\lambda f.\, f\,0; f\,1; \texttt{not}(f\,\texttt{true}))\,(\lambda x.x)$. There are three applications of the function $(\lambda x.x)$ in the program. CPA would generate two contours for the function corresponding to the two possible argument types of the function. In other words, the function $(\lambda x.x)$ is analyzed twice: one contour is generated for applications with arguments of type **int** and another contour is generated for applications with arguments of type **bool**. Since the function is analyzed differently with respect to the application $f\,\texttt{true}$ and the other two applications, CPA can analyze this program precisely. Furthermore, since the two applications $f\,0$ and $f\,1$ have the same argument type, they would share a same contour of function $(\lambda x.x)$. The 1CFA algorithm can also analyze this program precisely but must generate three contours for $(\lambda x.x)$, so CPA can analyze the program more efficiently than 1CFA.

The above discussion shows how different type inference algorithms use different strategies for handling polymorphism. The first result presented in this paper is a generic theoretical framework for polymorphic constraint-based type inference [Smith and Wang 2000]. A concrete type inference algorithm can be obtained by instantiating the framework with a particular strategy for handling polymorphism. We define nCFA and CPA algorithms as instantiations of the framework. Our purpose in designing a new framework is not primarily to give "yet another framework" for polymorphic analyses, but to develop a framework particularly useful for the development of new polymorphic analyses, for proving the correctness of analyses, and for improving on implementations of existing analyses. We in fact used the framework as an aid in the design of our novel Data Polymorphic CPA (DCPA) algorithm.

The soundness of the framework is established via subject reduction, and thus every algorithm that can be defined as an instantiation of the framework, including nCFA and CPA, is sound. We also present an extension of the framework for analyzing object-oriented languages.

In this paper we also present research results on precise constraint-based type inference in the presence of *data polymorphism*. Data polymorphism refers to the ability of an imperative variable to hold values of different types at runtime [Plevyak and Chien 1994; Agesen 1996]. For example, a variable of type `java.lang.Object` in Java can hold object values of arbitrary types. Data polymorphism arises in programming languages with polymorphic imperative variables, and it is particularly common and important in object-oriented programming. Agesen's CPA can analyze programs with parametric polymorphism effectively, but it loses precision significantly for data-polymorphic programs. Agesen [1996] considers the development of an extension of CPA to handle data polymorphism as an

open problem.

We develop a novel algorithm, Data-Polymorphic CPA (DCPA), which extends CPA with the ability to analyze data-polymorphic programs effectively. The basic idea is to detect those function applications and object creations where CPA might lose precision because of data polymorphism. For those function applications and object creations detected, more contours and object types are generated to improve the precision. Compared to Plevyak and Chien's Iterative Flow Analysis (IFA) [Plevyak and Chien 1994], which analyzes data-polymorphic programs with an iterative multiple-pass approach, the detection of data polymorphism in DCPA is performed online, and DCPA is thus a less complex one-pass algorithm with feasible run-time performance.

This paper lastly presents results on implementing polymorphic constraint-based type inference systems for realistic programming languages. This is a challenging task. First, the basic inference algorithm needs to be extended so that the rich language features can be analyzed effectively. For example, a type inference system for Java should handle features including classes, objects, interfaces, inner classes, exceptions and libraries. And various implementation issues need to be addressed so that a practical system can be obtained from a theoretical algorithm. Such issues include termination, recursion handling, constraint-representation, and scalability.

We have constructed a prototype implementation of a constraint-based type inference system for the full Java language [Wang and Smith 2001]. The system includes implementations of 0CFA, CPA and our novel DCPA algorithm. The system incorporates a series of implementation optimizations that significantly improve the performance of the system. In particular, a novel version of online cycle elimination [Fähndrich et al. 1998] is incorporated and a limited form of unification is also used. The system is used as a tool to statically check whether Java typecasts in a program will always succeed at run-time. Cast-checking is a good test of the accuracy of type inference, since each cast by definition is beyond the Java static type system and so represents a more challenging type inference problem.

We have tested the system with a suite of realistic Java applications. Benchmark results are presented to compare the 0CFA, CPA and DCPA algorithms in terms of both efficiency and analysis precision. DCPA shows good results on benchmark tests: nearly all casts that could be verified statically by a flow-insensitive analysis have been verified by our DCPA implementation; DCPA is substantially more precise than CPA and 0CFA on benchmark programs; and, the efficiency of DCPA is comparable to CPA.

The remainder of the paper is organized as follows. We first give a presentation of our framework for polymorphic constraint-based type inference in Section 2. We define the framework, its instantiation to nCFA and CPA, and prove the soundness of the framework via subject-reduction. In Section 5, we extend the framework to the analysis of object-oriented languages, a more appropriate foundation for our Java implementation. We then study the phenomenon of data polymorphism and present our DCPA algorithm in Section 6. We discuss existing solutions for data polymorphism, explain the key ideas of DCPA, and present a formal definition of DCPA via a stack-based state machine. In Section 7, we present our implementation of the constraint-based type inference system for Java, including an implementation of DCPA. Section 8 presents the experimental results of our system on benchmark

programs, and compares 0CFA, CPA and DCPA in terms of efficiency and precision. We review related work in Section 9. Finally, in Section 10, we summarize the contributions of the paper.

2. A FRAMEWORK FOR POLYMORPHIC TYPE INFERENCE

In this section, we define a framework for polymorphic constraint-based type inference. The framework is a generic one: it takes a concrete strategy for handling polymorphism as a parameter; different algorithms can be defined by instantiating the framework with different polymorphism-handling strategies.

We first describe a language, on which the type inference is defined, and we present a small-step operational semantics for the language. We then give a formal definition of the framework by defining the type language and specifying the two phases of type inference: *constraint generation* and *closure computation*. We then show how to obtain concrete algorithms from the framework by formulating Shivers' nCFA and Agesen's CPA algorithms as instantiations of the framework. Finally, we prove the soundness of the framework via subject-reduction.

2.1 Source Language

The language we study here is a call-by-value lambda calculus. It is similar to the language used in [Palsberg and Pavlopoulou 1998], In Section 5, we will extend the type inference framework defined in this section to analyze object-oriented languages.

Definition 2.1 (The Language).

$$\begin{aligned} e &::= x \mid v \mid \mathtt{succ}\, e \mid \mathtt{if0}\, e\, e\, e \mid e\, e \mid e^l \\ v &::= n \mid (\lambda x.e)^l \end{aligned}$$

where $e \in$ **Expression**, $v \in$ **Value**, $x \in$ **Variable**, $n \in$ **Integer**, and $l \in$ **Label**.

An expression e is a *program* if and only if e is a closed expression. We assume that every subterm of a program is associated with a unique label. Such labels are used for flow analyses: if a flow analysis intends to know what data values would flow to the expression e, a unique label l is attached to e. All functions are labeled, as $(\lambda x.e)^l$. Recursive definitions may be constructed in this language via the Y-combinator.

2.2 Semantics

We now define the semantics of the language.

Definition 2.2 (Reduction Relation $\longrightarrow$). We define a reduction relation among expressions as follows:

$$\begin{array}{lll} \mathtt{succ}\, n & \longrightarrow n' & \text{where } n' = n+1 \\ \mathtt{if0}\, 0\, e_1\, e_2 & \longrightarrow e_1 & \\ \mathtt{if0}\, n\, e_1\, e_2 & \longrightarrow e_2 & \text{where } n \neq 0 \\ (\lambda x.e)^l\, v & \longrightarrow e[v/x] & \\ v^l & \longrightarrow v & \end{array}$$

The reduction relation defines the basic evaluation steps of the language. The notation $e[v/x]$ means the substitution of v for free occurrences of x in e.

Definition 2.3 (Standard Reduction Relation $\longmapsto$). We define a *standard reduction relation* among closed expressions as follows:

$$E[e] \longmapsto E[e'] \text{ iff } e \longrightarrow e'$$

where

$$E ::= [\,] \mid E\,e \mid v\,E \mid \texttt{succ}\,E \mid \texttt{if0}\,E\,e_1\,e_2.$$

In the above definition, an *evaluation context* E is an expression with one sub-expression replaced by a hole, written as $[\,]$. We use the notion $E[e]$ to denote the expression produced by filling the single hole in E with expression e. The standard reduction relation defines the evaluation steps of the language by imposing a deterministic order on evaluating sub-expressions: when evaluating expression $E[e]$, sub-expression e is the next sub-expression to be evaluated.

We define relation $\stackrel{*}{\longmapsto}$ as the reflexive and transitive closure of standard reduction relation $\longmapsto$. The semantics of the language is defined with a partial function *eval* on programs:

$$eval(e) = v \text{ iff } e \stackrel{*}{\longmapsto} v.$$

2.3 The Type System

The type system is based on the Aiken and Wimmers [1993] style constraint system; in particular it is most closely derived from the system in [Eifrig et al. 1995b].

Definition 2.4 (Types). The type grammar is as follows:

$$\begin{array}{llll}
\tau & \in \mathbf{Type} & ::= & t \mid \tau v \mid t_1 \to t_2 \mid \mathbf{int}^{\text{-}} \mid l \\
t & \in \mathbf{TypeVar} & & \\
l & \in \mathbf{Label} & & \\
\overline{t} & \in \mathbf{TypeVarSet} & = & \mathbf{P}_{\text{fin}}(\mathbf{TypeVar}) \\
\tau v & \in \mathbf{ValueType} & ::= & \mathbf{int} \mid (\forall\ \overline{t}.\ t \to \tau \setminus C)^l \\
\tau_1 <: \tau_2 & \in \mathbf{Constraint} & & \\
C & \in \mathbf{ConstraintSet} & = & \mathbf{P}_{\omega}(\mathbf{Constraint})
\end{array}$$

In the above definition, $\tau_1 <: \tau_2$ is a subtyping constraint, with τ_1 as the lower-bound, and τ_2 as the upper-bound, meaning that τ_1 is a subtype of τ_2. The types for the most part are standard, but with a few exceptions we focus on here. Rather than using τ everywhere in types, in some positions we restrict a type to be a type variable t only. These restrictions are designed to make the presentation simpler. Function uses (call sites) are given type $t_1 \to t_2$. **ValueType** are the types for data values. For the purpose of flow analysis, labels l are also used as types l. To ease the construction of the soundness proof, we use two different types for integer-valued expressions: type **int** is only used as a lower bound, and $\mathbf{int}^{\text{-}}$ is only used as an upper bound. Functions are given polymorphic type schemes of the form $(\forall\ \overline{t}.\ t \to \tau \setminus C)^l$, where t is the type variable for the formal argument, τ is the return type, C is the set of constraints bound in this type scheme, $\overline{t}$ is the set of bound type variables, and l is the label for the function definition.

$$\text{(Var)}\quad \frac{A(x) = t}{A \vdash x : t \setminus \{\}}$$

$$\text{(Int)}\quad \frac{}{A \vdash n : \mathbf{int} \setminus \{\}}$$

$$\text{(Succ)}\quad \frac{A \vdash e : \tau \setminus C}{A \vdash \mathtt{succ}\, e : \mathbf{int} \setminus \{\tau <: \mathbf{int}^-\} \cup C}$$

$$\text{(If0)}\quad \frac{A \vdash e_1 : \tau_1 \setminus C_1,\ \ e_2 : \tau_2 \setminus C_2,\ \ A \vdash e_3 : \tau_3 \setminus C_3}{A \vdash \mathtt{if0}\ e_1\, e_2\, e_3 : t \setminus \{\tau_1 <: \mathbf{int}^-,\ \tau_2 <: t,\ \tau_3 <: t\} \cup C_1 \cup C_2 \cup C_3}$$

$$\text{(Abs)}\quad \frac{A, \{x : t\} \vdash e : \tau \setminus C}{A \vdash (\lambda x.e)^l : (\forall\, \overline{t}.\ t \to \tau \setminus C)^l \setminus \{\}}$$
where $\overline{t} = FTV(t \to \tau \setminus C) - FTV(A)$

$$\text{(Appl)}\quad \frac{A \vdash e_1 : \tau_1 \setminus C_1,\ \ e_2 : \tau_2 \setminus C_2}{A \vdash e_1\, e_2 : t_2 \setminus \{\tau_1 <: t_1 \to t_2,\ \tau_2 <: t_1\} \cup C_1 \cup C_2}$$

$$\text{(Label)}\quad \frac{A \vdash e : \tau \setminus C}{A \vdash e^l : \tau \setminus \{\tau <: l\} \cup C}$$

Fig. 1. Type Inference Rules

2.4 Constraint Generation

The first phase of a constraint-based type inference algorithm is to generate an initial set of constraints corresponding to the program under analysis. Constraint generation is defined by the type inference rules in Figure 1.

A type environment A is a mapping from program variables to type variables. Given a type environment A, the proof system assigns a type to expression e via the type judgment $A \vdash e : \tau \setminus C$, where τ is the type for e, and C is the set of subtyping constraints that models the flow paths in e. We abbreviate $A \vdash e : \tau \setminus C$ as $\vdash e : \tau \setminus C$ when A is empty. The rules are deterministic except that nondeterminism may arise in the choice of type variables. We restrict type derivations to be of a form where fresh type variables are used whenever it is possible. With this restriction, the first phase of type inference is trivially decidable and is unique modulo choice of type variable names.

Definition 2.5 (Type inference algorithm). Given program e, its inferred type is $\tau \setminus C$ provided $\vdash e : \tau \setminus C$.

The intuition behind the inference rules is that a subtyping constraint $\tau_1 <: \tau_2$ indicates a potential flow from expressions of type τ_1 to expressions of type τ_2. The rules generally follow standard presentations of Aiken-Wimmers constrained type system, except for the (Abs) type. The (Var) rule extracts the type for a variable from the type environment. The (Int) rule assigns type **int** to integer constants. The (Succ) rule generates constraint $\tau <: \mathbf{int}^-$ to ensure the applicability of the `succ` operation. The rule (If0) generates constraint $\tau_1 <: \mathbf{int}^-$ to ensure the applicability of the condition test, and it generates constraints $\tau_2 <: t$ and $\tau_3 <: t$ for flow paths from e_2 and e_3 to the `if0` expression. The (Label) rule generates constraint $\tau <: l$ to associate the possible values of expression e^l with label l.

The (Abs) rule assigns each function a polymorphic type scheme. In this rule, $FTV(\cdot)$ is a function that extracts free type variables; $\overline{t}$ collects all the type variables generated when the inference is applied to the function body; and C collects all the constraints corresponding to the function body. The manner in which type

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

$$\text{(}\forall\text{-Elim)}\quad \frac{(\forall\ \overline{t}.\ t \to \tau \setminus C)^l <: t_1 \to t_2,\quad \tau v <: t_1}{\tau v <: \Theta(t),\quad \Theta(\tau) <: t_2,\quad \Theta(C)}$$
$$\text{where } \Theta = \texttt{Poly}((\forall\ \overline{t}.\ t \to \tau \setminus C)^l <: t_1 \to t_2, \tau v <: t_1)$$

Fig. 2. Constraint Closure Rules

schemes are formed is similar to standard polymorphic constrained type systems, but the significant difference here is that every function is given a type scheme. By contrast, in a system based on let-polymorphism, the `let` construct dictates where type schemes are introduced.

The (Appl) rule generates $t_1 \to t_2$ as type for the application site, with t_1 as argument type, and t_2 as the type for the application result. The constraint $\tau_2 <: t_1$ lets values of e_2 flow in as the actual argument values, and the constraint $\tau_1 <: t_1 \to t_2$ lets the values of e_1 flow to this application site as function values.

We take an intensional view of types: two types are equivalent if and only if they are syntactically identical. In particular, type schemes corresponding to different functions in the program are always different. For example, $(\forall\ \{t_1\}.t_1 \to t_1\ \backslash\{\})^{l_1}$ and $(\forall\ \{t_2\}.t_2 \to t_2\ \backslash\{\})^{l_2}$ are regarded as different if $l_1 \neq l_2$ even though they are equivalent by alpha-conversion. This is because we wish to distinguish different functions in the analysis to obtain precise flow analyses.

We apply the rules on an example studied in [Palsberg and Pavlopoulou 1998]:

$$E_1 \equiv (\lambda f.\texttt{succ}\,((f\ f)\,0))^{l_1}\,(\texttt{if0}\ k\,(\lambda x.x)^{l_2}\,(\lambda y.(\lambda z.z)^{l_3})^{l_4})$$

To ease presentation, each program variable is inferred with a type variable having exactly the same name. The following constraints are generated:
$\vdash\ (\lambda f.\texttt{succ}\,((f\ f)\,0))^{l_1}\ :\ \tau_f\backslash\{\}$, where $\tau_f \equiv (\forall\ \{f, t_1, t_2, t_3, t_4\}.f \to \mathbf{int} \setminus \{f <: t_1 \to t_2, f <: t_1, t_2 <: t_3 \to t_4, \mathbf{int} <: t_3, t_4 <: \mathbf{int^-}\})^{l_1}$,
$\vdash\ (\lambda x.x)^{l_2} : \tau_x\backslash\{\}$, where $\tau_x \equiv (\forall\ \{x\}.x \to x\backslash\{\})^{l_2}$,
$\vdash\ (\lambda y.(\lambda z.z)^{l_3})^{l_4}\ :\ \tau_y\backslash\{\}$, where $\tau_y \equiv (\forall\ \{y\}.y \to (\forall\ \{z\}.z \to z\backslash\{\})^{l_3}\backslash\{\})^{l_4}$.
$\vdash\ E_1\ :\ t_7 \setminus \{\mathbf{int} <: \mathbf{int^-}, \tau_x <: t_5, \tau_y <: t_5, \tau_f <: t_6 \to t_7, t_5 <: t_6\}$

Thus, the initial constraint set generated for this program is: $\{\mathbf{int} <: \mathbf{int^-}, \tau_x <: t_5, \tau_y <: t_5, \tau_f <: t_6 \to t_7, t_5 <: t_6\}$.

2.5 Computation of the Closure

The inference algorithm applied to program e results in a type judgment $\vdash\ e\ :\ \tau \setminus C$ with C as the set of initial constraints generated for e. For a flow analysis, we need to generate all the possible data-flow and control-flow paths and propagate value types along all the data-flow paths. This is achieved by applying the closure rules of Figure 2 to C, propagating information via deduction rules on the subtyping constraints. The (Trans) closure rule is the transitivity rule that models run-time data flow by propagating value types forward along flow paths.

The most important closure rule is (∀-Elim), which performs ∀ elimination. The constraint $(\forall\ \overline{t}.\ t \to \tau \setminus C)^l <: t_1 \to t_2$ indicates a function flowing to a call site, where $(\forall\ \overline{t}.\ t \to \tau \setminus C)^l$ is the type for the function and $t_1 \to t_2$ is the

type representing the call site. The constraint $\tau v <: t_1$ means that a value of type τv flows in as the actual argument. At run-time, upon each function application, all local variables of the function are allocated fresh locations on the stack. In the analysis a corresponding renaming $\Theta \in \mathbf{TypeVar} \xrightarrow{\text{p}} \mathbf{TypeVar}$ is applied to type variables in $\overline{t}$ ($\mathbf{TypeVar} \xrightarrow{\text{p}} \mathbf{TypeVar}$ denotes the partial-function space on **TypeVar**). Unlike the run-time behavior, Θ may not always produce fresh type variables, but may for efficiency elect to reuse existing ones. The partial function Θ is extended to types, constraints, and constraint sets in the usual manner. $\Theta(t)$ for $t \notin \overline{t}$ is defined to be t. We call $\Theta(\tau)$ an *instantiation* of τ. Following the terminology of Shivers' nCFA [Shivers 1991], we call a renaming Θ a *contour*. The $\forall$ is eliminated from $(\forall\, \overline{t}.\ t \to \tau \setminus C)^l$ by applying Θ to C. The ($\forall$-Elim) rule then generates additional constraints to capture the flow from the actual argument τv to the formal argument $\Theta(t)$, and from the return value $\Theta(\tau)$ to the application result t_2.

The ($\forall$-Elim) closure rule is parameterized by function $\texttt{Poly} \in (\mathbf{Constraint} \times \mathbf{Constraint}) \to (\mathbf{TypeVar} \xrightarrow{\text{p}} \mathbf{TypeVar})$, which decides for this particular function, call site, and actual argument type, which contour is to be used (*i.e.*, created or reused, based on whether the renaming is to fresh or already existing variables). Providing a concrete `Poly` instantiates the framework to give a concrete algorithm. For example, the monomorphic analysis 0CFA is defined by letting `Poly` always return the identity renaming. This particular example shows how `Poly` may reuse existing contours. The differing analyses are defined by differing `Poly` which use different strategies for sharing contours. In the next section we will show nCFA and CPA algorithms can be defined by instantiating the framework with particular definitions of the `Poly` function. The parameterization based on *both* the call site and argument is an important feature of our framework; as we will show below, nCFA instantiates contours based on the call site only, and CPA instantiates contours on the argument type only.

Definition 2.6 (Closure). For a constraint set C and renaming function `Poly`, we define $\mathit{Closure}_{\texttt{Poly}}(C)$ as the least superset of C closed under the closure rules of Figure 2.

This closure is well-defined since the rules can be seen to induce a monotone function on constraint sets. More precisely, consider the function F which takes a constraint set C and returns C plus all constraints which directly follow from C by any one of the closure rules. F is monotone: if $C \subseteq C'$, for some C', then the closure rules applied to C' will add strictly more constraints than the closure rules applied to C, and thus F is monotone. By Tarski's Fixed Point Theorem, F then has a least fixed point. By this definition, some `Poly` may produce infinite closures since infinitely many contours may be created. Such analyses are still worthy of study even though they are not implementable.

Definition 2.7 (Analysis). Define $\mathit{Analysis}_{\texttt{Poly}}(e) = \mathit{Closure}_{\texttt{Poly}}(C)$, where the inference algorithm infers $\vdash e : \tau \setminus C$.

The output of an analysis is a set of constraints, which is the closure computed from the initial constraint set generated by the inference rules. The closure contains

complete flow information about the program; various program properties can be deduced from it.

Definition 2.8 (Type-Safe Constraint Set). A constraint set C is type-safe iff C does not contain any *type-contradictory* constraints of the following form:

(1) $\mathbf{int} <: t_1 \rightarrow t_2$;
(2) $(\forall\, \overline{t}.\ t \rightarrow \tau \setminus C)^l <: \mathbf{int}^{\text{-}}$

Definition 2.9 (Type-Checking). A program e is well-typed iff $Analysis_{\texttt{Poly}}(e)$ is type-safe.

For example, analyzing program `succ` $(\lambda x.x)$ would generate a type-contradictory constraint $(\forall\, \{x\}.x \rightarrow x\backslash\{\})^l <: \mathbf{int}^{\text{-}}$, which indicates a type error. A computation state is *wrong* if computation cannot continue due to a run-time type error.

We now show how the results of a conventional control flow analysis can be obtained in our framework. For an expression of interest, a flow analysis aims to statically predict what data values would flow to the expression. Recall that a unique label is attached to each of those expressions of interest. Type inference outputs a constraint closure as result. A flow analysis can be obtained from the closure as follows:

For a program e, if $\vdash\ e\ :\ \tau \setminus C$ and C' is a closure of C, then: if $\mathbf{int} <: l \in C'$, integer values are considered flowing to expression e^l; if $(\forall\, \overline{t}.\ t \rightarrow \tau_1 \setminus C_1)^{l_1} <: l \in C'$, function $(\lambda x.e_1)^{l_1}$ is considered flowing to expression e^l.

The framework enjoys both type soundness and flow soundness. The type soundness ensures that, if a program is typechecked successfully, it cannot go *wrong* during execution; and the flow soundness ensures that any flow analysis produced from the framework correctly approximates the run-time data flow.

The soundness proof is presented in Section 4. Since the proof constructed is independent of the particular strategies for handling polymorphism (specified by the function `Poly`), the soundness of the framework implies that any analysis defined as an instantiation of the framework is also sound.

3. INSTANTIATING THE FRAMEWORK

In this section we present various algorithms as instantiations of our framework.

3.1 nCFA Instantiation

In Shivers' nCFA analysis [Shivers 1991], each function application (call) is associated with a call-string of length at most n. The call-string contains the last n or fewer call sites on the call-path leading to this application. Applications of the same function share the same contour (i.e., analysis of the function) if they have the same call-string. To present nCFA in our framework, type variables are defined with superscripts that denote the call-string:

$$\begin{array}{lll} \alpha & \in & \mathbf{Identifier} \\ s & \in & \mathbf{Superscript} = \mathbf{Identifier\ List} \\ t & \in & \mathbf{TypeVar} \quad ::= \alpha^s \end{array}$$

We use the following list notation: the empty list is $[\,]$, $[\alpha_1, \ldots, \alpha_m]$ is a list of m elements, l_1 @ l_2 appends lists, and $l(1..n)$ is the list consisting of the first

$min(n, length(l))$ elements of list l. Each type variable α^s is tagged with a call-string s. All type variables generated by the inference rules have empty lists as superscripts. By the inference rule (Appl), a call site is inferred with a type $\alpha_1^{[\,]} \to \alpha_2^{[\,]}$, we use α_2 to identify this call site, thus a call-string is a list of such identifiers. All bound type variables of a $\forall$ type have empty list superscripts. When the $\forall$ quantifier is eliminated by the ($\forall$-Elim) closure rule, those bound type variables are renamed by changing the superscripts from empty lists to the appropriate call-strings.

Definition 3.1 (nCFA Algorithm). The nCFA algorithm is an instantiation of the framework with $\texttt{Poly} = \mathbf{CFA}$, where

$$\begin{aligned}&\mathbf{CFA}((\forall\ \overline{t}.\ t \to \tau \setminus C) <: t_1 \to \alpha_2^{s_2}, \tau v <: t_1) = \Theta, \text{ where for each } \alpha^{[\,]} \in \overline{t},\\ &\quad \Theta(\alpha^{[\,]}) = \alpha^{s'}, \text{where } s' = ([\alpha_2] \mathbin{@} s_2)(1..n)\end{aligned}$$

It can be shown by induction that s' is the call-string for application $(\forall\ \overline{t}.\ t \to \tau \setminus C) <: t_1 \to \alpha_2^{s_2}$. The definition of Θ ensures that applications of the same function share the same contour if and only if they have the same call-string.

Consider the program E_1 which we have studied in Section 2.4:

$$E_1 \equiv (\lambda f.\texttt{succ}\,((f\ f)\,0))^{l_1}\ (\texttt{if0}\ k\,(\lambda x.x)^{l_2}\ (\lambda y.(\lambda z.z)^{l_3})^{l_4})$$

Applying nCFA to program E_1, since function $(\lambda f \ldots)^{l_1}$ has only one application, the ($\forall$-Elim) rule generates only one contour Θ for this function, resulting in $\tau_x <: \Theta(f)$ and $\tau_y <: \Theta(f)$. This means that both $(\lambda x.x)^{l_2}$ and $(\lambda y.(\lambda z.z)^{l_3})^{l_4}$ flow to f, and at the application site $f\ f$ there are four applications. One of them, $(\lambda x.x)^{l_2}$ applying to $(\lambda y.(\lambda z.z)^{l_3})^{l_4}$ generates constraints: $(\forall\ \{y\}.y \to (\forall\ \{z\}.z \to z\backslash\{\})^{l_3}\backslash\{\})^{l_4} <: \Theta(t_2)$, $\Theta(t_2) <: \Theta(t_3) \to \Theta(t_4)$, $\mathbf{int} <: \Theta(t_3)$, $\Theta(t_4) <: \mathbf{int}^{\text{-}}$, where t_2 and t_4 are types for expressions $f\ f$ and $((f\ f)\,0)$ respectively. This leads to a type error: $(\forall\ \{z\}.z \to z\backslash\{\})^{l_3} <: \mathbf{int}^{\text{-}}$. Hence nCFA fails to type-check E_1 for arbitrary n. This example shows that nCFA is imprecise even for large n.

3.2 CPA Instantiation

The Cartesian Product Algorithm (CPA) [Agesen 1995; Agesen 1996] is a concrete type inference algorithm for object-oriented languages. For a message sending expression, CPA computes the cartesian product of the types for the actual arguments. For each element of the cartesian product, the method body is analyzed exactly once with one contour generated. The calling-contexts of a method are partitioned by the cartesian product, rather than by call-strings as in nCFA. In our language, each function has only one argument. For each function, CPA generates exactly one contour for each distinct argument type that the function is applied to. To present CPA, type variables are defined with structure:

$$\begin{aligned}\alpha &\in \mathbf{Identifier}\\ t &\in \mathbf{TypeVar} \ ::= \ \alpha \mid \alpha^{\tau v}\end{aligned}$$

The inference rules are constrained to generate type variables without superscripts.

Definition 3.2 (CPA ALGORITHM). The CPA algorithm is the instantiation of the framework with $\texttt{Poly} = \mathbf{CPA}$, where

$$\mathbf{CPA}((\forall\, \overline{t}.\ t \to \tau \setminus C) <: t_1 \to t_2, \tau v <: t_1) = \Theta, \text{ where for each } \alpha \in \overline{t}, \Theta(\alpha) = \alpha^{\tau v}$$

The contours Θ are generated based on the actual argument type τv, independent of the call site $t_1 \to t_2$. This is the opposite of **CFA**, which ignores the value type τv, and only uses the call site $t_1 \to t_2$. Given a particular function and its associated $\forall$ type in a program, this algorithm will generate a unique contour ($\forall$ elimination) for each distinct value type the function is applied to. It however may share contours across call sites. Agesen [1996] presents convincing experimental evidence that the CPA approach is both more efficient and more precise than nCFA.

Again, consider the program E_1 which we have studied in Section 2.4:

$$E_1 \equiv (\lambda f.\texttt{succ}\,((f\ f)\,0))^{l_1}\ (\texttt{if0}\ k\,(\lambda x.x)^{l_2}\ (\lambda y.(\lambda z.z)^{l_3})^{l_4})$$

We sketch how CPA would analyze this program. Even though there is only one application site for the function $(\lambda f \ldots)^{l_1}$, the function is applied to two different actual argument values. So, the ($\forall$-Elim) rule generates two contours Θ_1 and Θ_2 for $(\lambda f \ldots)^{l_1}$ with $\Theta_1(f) = f^{\tau_x}$, $\tau_x <: f^{\tau_x}$, $\Theta_2(f) = f^{\tau_y}$, and $\tau_y <: f^{\tau_y}$. The two contours for function $(\lambda f \ldots)^{l_1}$ split the single static call site $f\ f$ into two call sites: at one call site $(\lambda x.x)^{l_2}$ is applied to itself; at another call site $(\lambda y.(\lambda z.z)^{l_3})^{l_4}$ is applied to itself. Thus the program is type-checked successfully.

4. SOUNDNESS PROOF

In this section, we establish soundness of the type inference framework. The proof is based on the subject-reduction proof technology [Wright and Felleisen 1994]. Compared to other existing soundness proofs for constraint-based type systems [Eifrig et al. 1995b; Eifrig et al. 1995a; Flanagan 1997], our proof has some novel aspects: no model theory or extra type system is employed in the proof. Instead, for two expressions related by the reduction relation, a generalized subtype relation is defined to directly relate the types of the two expressions. A subject reduction property is then established with respect to the generalized subtyping relation.

4.1 Generalized Subtyping Relation

The basic idea of subject reduction is to prove that evaluation does not change the type of the expression. In the presence of subtyping, this is refined to be that if $e_1 \overset{*}{\longmapsto} e_2$, the type of e_2 is a subtype of the type of e_1.

Definition 4.1 (GENERALIZED SUBTYPING RELATION). Given a closed constraint set C, we define a relation among types as the least relation satisfying all the following conditions:

(1) $C \models \tau \preceq \tau$
(2) $C \models \tau_1 \preceq \tau_2$ if $\tau_1 <: \tau_2 \in C$
(3) if $C \models \tau_1 \preceq \tau_2$, and $C \models \tau_2 \preceq \tau_3$, then $C \models \tau_1 \preceq \tau_3$
(4) $C \models (\forall\, \overline{t}.\ t \to \tau_1 \setminus C_1)^l \preceq (\forall\, \overline{t}.\ t \to \tau_2 \setminus C_2)^l$ if $C \models \tau_1 \preceq \tau_2$ and for each $\tau <: \tau' \in C_1$, there exists τ'' s.t. $C \models \tau \preceq \tau''$ and $\tau'' <: \tau' \in C_2$.

This definition is more precisely the least fixed point of a monotone function on relations; the function is monotone because all occurrences of $C \models \tau \preceq \tau'$ are strictly positive. Generalized subtyping places $\forall$ types in a subtyping relationship based on their internal structure. Some interesting properties about generalized subtyping can be established easily:

LEMMA 4.2. *If $C \models \tau v \preceq \tau v'$, then it must be one of the following two cases:*

(1) $C \models \mathbf{int} \preceq \mathbf{int}$*;*

(2) $C \models (\forall\, \overline{t}.\ t \to \tau_1 \setminus C_1)^l \preceq (\forall\, \overline{t}.\ t \to \tau_2 \setminus C_2)^l$*, where* $C \models \tau_1 \preceq \tau_2$ *and for each* $\tau <: \tau' \in C_1$*, there exists* τ'' *s.t.* $C \models \tau \preceq \tau''$ *and* $\tau'' <: \tau' \in C_2$*.*

LEMMA 4.3. *If $C \models t \preceq \tau$, and $t \neq \tau$, then $t <: \tau \in C$.*

The following lemma relates generalized subtyping to the underlying constraint set C:

LEMMA 4.4. *If $C \models \tau v \preceq \tau$ and τ is not a value type, then there exists $\tau v'$ s.t. $C \models \tau v \preceq \tau v'$ and $\tau v' <: \tau \in C$.*

Proof: We use an induction on the structure of the derivation concluding $C \models \tau v \preceq \tau$:

—Case 1: $\tau v <: \tau \in C$. The lemma trivially holds in this case.

—Case 2: there exists τ' s.t. $C \models \tau v \preceq \tau'$, $C \models \tau' \preceq \tau$, and $\tau' \neq \tau$. If τ' is a value type, by induction, there is a value type $\tau v'$ s.t. $C \models \tau' \preceq \tau v'$ and $\tau v' <: \tau \in C$. Thus, $C \models \tau v \preceq \tau v'$ and $\tau v' <: \tau \in C$. If τ' is not a value type, τ' must be a type variable. By induction, there exists $\tau v'$ s.t. $C \models \tau v \preceq \tau v'$ and $\tau v' <: \tau' \in C$. Since τ' is a type variable, $C \models \tau' \preceq \tau$, $\tau' \neq \tau$, by Lemma 4.3, $\tau' <: \tau \in C$. By transitivity, $\tau v' <: \tau \in C$. And we have $C \models \tau v \preceq \tau v'$.

□

Definition 4.5 ($\models$). For a closed constraint set C_1 and constraint set C_2, we define $C_1 \models C_2$, iff, for any $\tau \preceq \tau' \in C_2$, $C_1 \models \tau \preceq \tau'$.

We now prove several lemmas which are important to the subject-reduction proof.

LEMMA 4.6. *If $C \models \tau \preceq \tau'$, and Θ is a renaming s.t. $\Theta(t) = t$ for any $t \in FTV(C)$, then, $C \models \Theta(\tau) \preceq \Theta(\tau')$.*

Proof: We use induction on the structure of the derivation concluding that $C \models \tau \preceq \tau'$, and we do a case analysis on the last step of the derivation:

—Case 1: $\tau = \tau'$. We have $\Theta(\tau) = \Theta(\tau')$.

—Case 2: $\tau <: \tau' \in C$. For any type variable t which occurs freely in τ or τ', $\Theta(t) = t$. Thus, $\Theta(\tau) = \tau$, $\Theta(\tau') = \tau'$, and $C \models \tau \preceq \tau'$.

—Case 3: $C \models \tau \preceq \tau''$, $C \models \tau'' \preceq \tau'$. By induction, $C \models \Theta(\tau) \preceq \Theta(\tau'')$ and $C \models \Theta(\tau'') \preceq \Theta(\tau')$. By transitivity, $C \models \Theta(\tau) \preceq \Theta(\tau')$.

—Case 4: $\tau = (\forall\, \overline{t}.\ t \to \tau_1 \setminus C_1)^l$ and $\tau' = (\forall\, \overline{t}.\ t \to \tau_2 \setminus C_2)^l$. Since $C \models \tau_1 \preceq \tau_2$, by induction, $C \models \Theta(\tau_1) \preceq \Theta(\tau_2)$. Consider a constraint $\Theta(\tau_3) <: \Theta(\tau_3') \in \Theta(C_1)$ with $\tau_3 <: \tau_3' \in C_1$. Since $\tau_3 <: \tau_3' \in C_1$, $C \models \tau_3 \preceq \tau_3''$ and $\tau_3'' <: \tau_3' \in C_2$, thus $\Theta(\tau_3'') <: \Theta(\tau_3') \in \Theta(C_2)$, and by induction, $C \models \Theta(\tau_3) \preceq \Theta(\tau_3'')$. Therefore, we have $C \models (\forall\, \overline{t}.\ t \to \Theta(\tau_1) \setminus \Theta(C_1))^l \preceq (\forall\, \overline{t}.\ t \to \Theta(\tau_2) \setminus \Theta(C_2))^l$.

□

LEMMA 4.7. *If* $A \vdash e : \tau \setminus C$, *and* $C' \models C$, *then there exists a closure* C'' *of* C *s.t.* $C' \models C''$.

Proof: We prove the lemma by inductively constructing a closure C'' of C with only constraints of form $\tau <: \tau'$ where $C' \models \tau \preceq \tau'$. The closure C'' is constructed as follows:

—First, we have $C \subseteq C''$.

—If $\tau v <: t \in C''$ and $t <: \tau \in C''$, by induction, $C' \models \tau v \preceq t$ and $C' \models t \preceq \tau$. Thus by transitivity, $C' \models \tau v \preceq \tau$, and we add $\tau v <: \tau$ into C''.

—If $(\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l <: t_1 \to t_2 \in C''$ and $\tau v <: t_1 \in C''$, by induction, we have $C' \models (\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l \preceq t_1 \to t_2$ and $C' \models \tau v \preceq t_1$. By Lemma 4.2 and Lemma 4.4, we have $C' \models (\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l \preceq (\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^l$, $(\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^l <: t_1 \to t_2 \in C'$, $C' \models \tau v \preceq \tau v'$ and $\tau v' <: t_1 \in C'$. Since $\{(\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^l <: t_1 \to t_2, \tau v' <: t_1\} \subseteq C'$, there is a renaming Θ on $\overline{t}$ s.t. $\{\tau v' <: \Theta(t), \Theta(\tau_2) <: t_2\} \cup \Theta(C_2) \subseteq C'$. We add constraints in set $\{\tau v <: \Theta(t), \Theta(\tau_1) <: t_2\} \cup \Theta(C_1)$ to C_2', and we prove that for any constraint $\tau_3 <: \tau_3'$ in this set, $C' \models \tau_3 \preceq \tau_3'$. By transitivity, $C' \models \tau v \preceq \Theta(t)$. Since $C' \models (\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l \preceq (\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^l$, $C' \models \tau_1 \preceq \tau_2$. By Lemma 4.6, $C' \models \Theta(\tau_1) \preceq \Theta(\tau_2)$. Thus, by transitivity, $C' \models \Theta(\tau_1) \preceq t_2$. Consider constraint $\Theta(\tau_3) <: \Theta(\tau_3') \in \Theta(C_1)$ with $\tau_3 <: \tau_3' \in C_1$. Since $C' \models (\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l \preceq (\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^l$, there exists τ_3'' s.t. $C' \models \tau_3 \preceq \tau_3''$ and $\tau_3'' <: \tau_3' \in C_2$. Since $C' \models \tau_3 \preceq \tau_3''$, by Lemma 4.6, $C' \models \Theta(\tau_3) \preceq \Theta(\tau_3'')$. Since $\tau_3'' <: \tau_3' \in C_2$, $\Theta(\tau_3'') <: \Theta(\tau_3') \in \Theta(C_2)$. Since $\Theta(C_2) \subseteq C'$, we have $C' \models \Theta(\tau_3'') \preceq \Theta(\tau_3')$. Thus, by transitivity, $C' \models \Theta(\tau_3) \preceq \Theta(\tau_3')$.

□

LEMMA 4.8. *If* $C \models C'$, *then, for any* $C' \models \tau \preceq \tau'$, $C \models \tau \preceq \tau'$.

Proof: We use an induction on the structure of the derivation concluding that $C' \models \tau \preceq \tau'$:

—Case 1: $\tau = \tau'$. Thus $C \models \tau \preceq \tau'$.

—Case 2: $\tau <: \tau' \in C$. Since $C \models C'$, $C \models \tau \preceq \tau'$.

—Case 3: $C' \models \tau \preceq \tau''$, $C' \models \tau'' \preceq \tau'$. By induction, $C \models \tau \preceq \tau''$ and $C \models \tau'' \preceq \tau'$. By transitivity, $C \models \tau \preceq \tau'$.

—Case 4: $\tau = (\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l$ and $\tau' = (\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^l$. Since $C' \models \tau_1 \preceq \tau_2$, by induction, $C \models \tau_1 \preceq \tau_2$. Consider a constraint $\tau_3 <: \tau_3' \in C_1$, there exists τ_3'' s.t. $C' \models \tau_3 \preceq \tau_3''$ and $\tau_3'' <: \tau_3' \in C_2$, and, by induction, $C \models \tau_3 \preceq \tau_3''$. Therefore, we have $C \models (\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l \preceq (\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^l$.

□

COROLLARY 4.9. *If* $C_1 \models C_2$ *and* $C_2 \models C_3$, *then* $C_1 \models C_3$.

4.2 Subject Reduction

Since applying inference rules on the same expression twice leads to two derivations which only differ in the choice of type variables, we have:

LEMMA 4.10 (RENAMING). *If* $A \vdash e : \tau \setminus C$, *and there is a mapping* $S \in \mathbf{TypeVar} \xrightarrow{\mathrm{p}} \mathbf{TypeVar}$ *s.t. for any* $t \neq t'$, *we have* $S(t) \neq S(t')$, *then, when* S *is extended naturally as a mapping on types, constraints, and constraint systems and type environments, we have* $S(A) \vdash e : S(\tau) \setminus S(C)$.

Proof: By induction on the size of the derivation concluding with $A \vdash e : \tau \setminus C$. □

LEMMA 4.11 (SUBSTITUTION). *If* $A, \{x : t_x\} \vdash e : \tau \setminus C$, *and* $A \vdash v : \tau v \setminus \{\}$, *then* $A \vdash e[v/x] : \tau' \setminus C'$ *s.t.* $FTV(\tau \setminus C) - \{t_x\} = FTV(\tau' \setminus C')$, $\{\tau v <: t_x\} \models \tau' \preceq \tau$ *and for any constraint* $\tau_1 <: \tau_2 \in C'$, *there exists* τ_3 *s.t.* $\{\tau v <: t_x\} \models \tau_1 \preceq \tau_3$, *and* $\tau_3 <: \tau_2 \in C$.

Proof: We prove by induction on the size of e. If $x \notin \mathit{FreeVar}(e)$, then $e[v/x] = e$, and the lemma trivially holds. We now proceed by assuming that $x \in \mathit{FreeVar}(e)$. We use a case analysis on the last type inference rule used in the last step of the derivation concluding with $A, \{x : t_x\} \vdash e : \tau \setminus C$:

—(Var): Since $x \in \mathit{FreeVar}(e)$, $e = x$, and $e[v/x] = v$. We have $A \vdash e[v/x] : \tau v \setminus \{\}$, By the (Var) rule, we have $A, \{x : t_x\} \vdash e : t_x \setminus \{\}$. We have $\{\tau v <: t_x\} \models \tau v \preceq t_x$, $FTV(t_x \setminus \{\}) = \{t_x\}$, and $FTV(\tau v \setminus \{\}) = \{\}$. Thus the lemma holds.

—(Int): This case cannot happen since $x \in \mathit{FreeVar}(e)$.

—(Succ): $e = \mathtt{succ}\, e_1$, and $e[v/x] = \mathtt{succ}\, e_1[v/x]$. By the (Succ) rule, we have $A, \{x : t_x\} \vdash e_1 : \tau_1 \setminus C_1$ and $A, \{x : t_x\} \vdash e : \mathbf{int} \setminus C$, where $C = \{\tau_1 <: \mathbf{int}^-\} \cup C_1$. By induction: $A \vdash e_1[v/x] : \tau_1' \setminus C_1'$, s.t. $\{\tau v <: t_x\} \models \tau_1' \preceq \tau_1$, and for any constraint $\tau_2 <: \tau_2' \in C_1'$, there exists τ_2'' s.t. $\{\tau v <: t_x\} \models \tau_2 \preceq \tau_2''$, and $\tau_2'' <: \tau_2' \in C_1$. Applying the (Succ) rule on $A \vdash e_1[v/x] : \tau_1' \setminus C_1'$, we get $A \vdash e[v/x] : \mathbf{int} \setminus C'$, where $C' = C_1' \cup \{\tau_1' <: \mathbf{int}^-\}$. First, we have $\{\tau v <: t_x\} \models \mathbf{int} \preceq \mathbf{int}$. Secondly, consider a constraint $\tau_2 <: \tau_2' \in C'$. If the constraint is $\tau_1' <: \mathbf{int}^-$, we have $\{\tau v <: t_x\} \models \tau_1' \preceq \tau_1$ and $\tau_1 <: \mathbf{int}^- \in C$. Otherwise, we have $\tau_2 <: \tau_2' \in C_1'$, there exists τ_2'' s.t. $\{\tau v <: t_x\} \models \tau_2 \preceq \tau_2''$, and $\tau_2'' <: \tau_2' \in C_1 \subseteq C$. By induction, $FTV(\tau_1 \setminus C_1) - \{t_x\} = FTV(\tau_1' \setminus C_1')$. Thus, $FTV(\mathbf{int} \setminus C) - \{t_x\} = FTV(\mathbf{int} \setminus C')$.

—(If0): $e = \mathtt{if0}\, e_1\, e_2\, e_3$, and $e[v/x] = \mathtt{if0}\ e_1[v/x]\ e_2[v/x]\ e_3[v/x]$. By (If0) rule, $A, \{x : t_x\} \vdash e_1 : \tau_1 \setminus C_1$, $A, \{x : t_x\} \vdash e_2 : \tau_2 \setminus C_2$, $A, \{x : t_x\} \vdash e_3 : \tau_3 \setminus C_3$, and $A, \{x : t_x\} \vdash e : \tau \setminus C$, where $\tau = t$, and $C = \{\tau_1 <: \mathbf{int}^-, \tau_2 <: t, \tau_3 <: t\} \cup C_1 \cup C_2 \cup C_3$. By induction, $A \vdash e_1[v/x] : \tau_1' \setminus C_1'$, $A \vdash e_2[v/x] : \tau_2' \setminus C_2'$, and $A \vdash e_3[v/x] : \tau_3' \setminus C_3'$. Thus, by applying the (If0) rule with t as the type for $e[v/x]$, we have $A \vdash e[v/x] : \tau' \setminus C'$, where $\tau' = t$, and $C' = \{\tau_1' <: \mathbf{int}^-, \tau_2' <: t, \tau_3' <: t\} \cup C_1' \cup C_2' \cup C_3'$. First, we have $\{\tau v <: t_x\} \models t \preceq t$. Consider a constraint $\tau_4 <: \tau_4' \in C'$. If $\tau_4 <: \tau_4' \in C_1'$, then, by induction, there exists τ_4'' s.t. $\{\tau v <: t_x\} \models \tau_4 \preceq \tau_4''$ and $\tau_4'' <: \tau_4' \in C_1$. Since $C_1 \subseteq C$, there exists τ_4'' s.t. $\{\tau v <: t_x\} \models \tau_4 \preceq \tau_4''$ and $\tau_4'' <: \tau_4' \in C$. Similarly, if $\tau_4 <: \tau_4'$ is in C_2' or C_3', there exists τ_4'' s.t. $\{\tau v <: t_x\} \models \tau_4 \preceq \tau_4''$ and $\tau_4'' <: \tau_4' \in C$. If $\tau_4 <: \tau_4'$ is $\tau_1' <: \mathbf{int}^-$, by induction, we have $\{\tau v <: t_x\} \models \tau_1' \preceq \tau_1$, and we have $\tau_1 <: \mathbf{int}^- \in C$. If $\tau_4 <: \tau_4'$ is $\tau_2' <: t$ or $\tau_3' <: t$, by induction, we have $\{\tau v <: t_x\} \models \tau_2' \preceq \tau_2$, $\{\tau v <: t_x\} \models \tau_3' \preceq \tau_3$, and we know $\tau_2 <: t \in$

C and $\tau_3 <: t \in C$. By induction, $FTV(\tau_1 \setminus C_1) - \{t_x\} = FTV(\tau_1' \setminus C_1')$, $FTV(\tau_2 \setminus C_2) - \{t_x\} = FTV(\tau_2' \setminus C_2')$, $FTV(\tau_3 \setminus C_3) - \{t_x\} = FTV(\tau_3' \setminus C_3')$. Therefore, $FTV(\tau \setminus C) - \{t_x\} = FTV(\tau' \setminus C')$.

—(Abs): We have $e = (\lambda x_1.e_1)^l$. Since $x \in FreeVar(e)$, we have $x \neq x_1$ and $e[v/x] = (\lambda x_1.e_1[v/x])^l$. By the (Abs) rule: $A, \{x : t_x, x_1 : t_1\} \vdash e_1 : \tau_1 \setminus C_1$, and $A, \{x : t_x\} \vdash e : \tau \setminus C$, where $\tau = (\forall\ \overline{t}.\ t_1 \to \tau_1 \setminus C_1)^l$, $C = \{\}$ and $\overline{t} = FTV(\tau_1 \setminus C_1) \cup \{t_1\} - FTV(A) - \{t_x\}$. By induction: $A, \{x_1 : t_1\} \vdash e_1[v/x] : \tau_1' \setminus C_1'$ and $FTV(\tau_1 \setminus C_1) - \{t_x\} = FTV(\tau_1' \setminus C_1')$. Thus, $\overline{t} = FTV(\tau_1' \setminus C_1') \cup \{t_1\} - FTV(A)$. Applying (Abs) rule, we have: $A \vdash e[v/x] : \tau' \setminus C'$, where $\tau' = (\forall\ \overline{t}.\ t_1 \to \tau_1' \setminus C_1')^l$ and $C' = \{\}$. By induction, for any constraint $\tau_2 <: \tau_2' \in C_1'$, there exits τ_2'' s.t. $\{\tau v <: t_x\} \models \tau_2 \preceq \tau_2''$, and $\tau_2'' <: \tau_2' \in C_1$. Also by induction, $\{\tau v <: t_x\} \models \tau_1' \preceq \tau_1$. Thus, $\{\tau v <: t_x\} \models (\forall\ \overline{t}.\ t_1 \to \tau_1' \setminus C_1')^l \preceq (\forall\ \overline{t}.\ t_1 \to \tau_1 \setminus C_1)^l$. Since $FTV(\tau \setminus C) = FTV(\tau_1 \setminus C_1) \cup \{t_1\} - \overline{t}$, $FTV(\tau' \setminus C') = FTV(\tau_1' \setminus C_1') \cup \{t_1\} - \overline{t}$, and $FTV(\tau_1 \setminus C_1) - \{t_x\} = FTV(\tau_1' \setminus C_1')$, we have $FTV(\tau \setminus C) - \{t_x\} = FTV(\tau' \setminus C')$.

—(Appl): $e = e_1\, e_2$, and $e[v/x] = e_1[v/x]\ e_2[v/x]$. By the (Appl) rule: $A, \{x : t_x\} \vdash e_1 : \tau_1 \setminus C_1$, $A, \{x : t_x\} \vdash e_2 : \tau_2 \setminus C_2$, and $A, \{x : t_x\} \vdash e : \tau \setminus C$, where $\tau = t_2$, and $C = \{\tau_1 <: t_1 \to t_2, \tau_2 <: t_1\} \cup C_1 \cup C_2$. By induction: $A \vdash e_1[v/x] : \tau_1' \setminus C_1'$, and $A \vdash e_2[v/x] : \tau_2' \setminus C_2'$. Thus, by applying the (Appl) rule with $t_1 \to t_2$ as the arrow type for the application, we have: $A \vdash e[v/x] : \tau' \setminus C'$, where $\tau' = t_2$, and $C' = \{\tau_1' <: t_1 \to t_2, \tau_2' <: t_1\} \cup C_1' \cup C_2'$. First, we have $\{\tau v <: t_x\} \models t_2 \preceq t_2$. Consider a constraint $\tau_3 <: \tau_3' \in C'$. If $\tau_3 <: \tau_3' \in C_1'$, then, by induction, there exists τ_3'' s.t. $\{\tau v <: t_x\} \models \tau_3 \preceq \tau_3''$ and $\tau_3'' <: \tau_3' \in C_1$. Since $C_1 \subseteq C$, there exists τ_3'' s.t. $\{\tau v <: t_x\} \models \tau_3 \preceq \tau_3''$ and $\tau_3'' <: \tau_3' \in C$. If $\tau_3 <: \tau_3' \in C_2'$, similarly, there exists τ_3'' s.t. $\{\tau v <: t_x\} \models \tau_3 \preceq \tau_3''$ and $\tau_3'' <: \tau_3' \in C$. If $\tau_3 <: \tau_3'$ is $\tau_1' <: t_1 \to t_2$, by induction, $\{\tau v <: t_x\} \models \tau_1' \preceq \tau_1$, and we know $\tau_1 <: t_1 \to t_2 \in C$. If $\tau_3 <: \tau_3'$ is $\tau_2' <: t_1$, by induction, $\{\tau v <: t_x\} \models \tau_2' \preceq \tau_2$, and we know $\tau_2 <: t_1 \in C$. By induction, $FTV(\tau_1 \setminus C_1) - \{t_x\} = FTV(\tau_1' \setminus C_1')$ and $FTV(\tau_2 \setminus C_2) - \{t_x\} = FTV(\tau_2' \setminus C_2')$. Thus, $FTV(\tau \setminus C) - \{t_x\} = FTV(\tau' \setminus C')$.

—(Label): $e = e_1^l$, and $e[v/x] = e_1[v/x]^l$. By the (Label) rule: $A, \{x : t_x\} \vdash e_1 : \tau_1 \setminus C_1$, and $A, \{x : t_x\} \vdash e : \tau \setminus C$, where $\tau = \tau_1$, and $C = \{\tau_1 <: l\} \cup C_1$. By induction: $A \vdash e_1[v/x] : \tau_1' \setminus C_1'$. Applying the (Label) rule, we get $A \vdash e[v/x] : \tau' \setminus C'$ where $\tau' = \tau_1'$ and $C' = \{\tau_1' <: l\} \cup C_1'$. First, by induction, $\{\tau v <: t_x\} \models \tau_1' \preceq \tau_1$, thus $\{\tau v <: t_x\} \models \tau' \preceq \tau$. Secondly, consider a constraint $\tau_2 <: \tau_2' \in C'$. If $\tau_2 <: \tau_2'$ is $\tau_1' <: l$, by induction, $\{\tau v <: t_x\} \models \tau_1' \preceq \tau_1$. And we know $\tau_1 <: l \in C$. Otherwise, $\tau_2 <: \tau_2' \in C_1'$. By induction, there exists τ_2'' s.t. $\{\tau v <: t_x\} \models \tau_2 \preceq \tau_2''$ and $\tau_2'' <: \tau_2' \in C_1$. Since $C_1 \subseteq C$, there exists τ_2'' s.t. $\{\tau v <: t_x\} \models \tau_2 \preceq \tau_2''$ and $\tau_2'' <: \tau_2' \in C$. By induction, $FTV(\tau_1 \setminus C_1) - \{t_x\} = FTV(\tau_1' \setminus C_1')$. Thus, $FTV(\tau \setminus C) - \{t_x\} = FTV(\tau' \setminus C')$.

□

We now establish a subject-reduction property on reduction relation $\longrightarrow$.

LEMMA 4.12 (SUBJECT REDUCTION ON $\longrightarrow$). *if $e \longrightarrow e'$, $A \vdash e : \tau \setminus C$, C'' is a closure of C, then: $A \vdash e' : \tau' \setminus C'$, $C'' \models \tau' \preceq \tau$, and $C'' \models C'$.*

Proof: We proceed with a case analysis on the relation $e \longrightarrow e'$.

—Case 1: $e = \mathtt{succ}\, n$, $e' = n'$. By (Int) and (Succ) inference rules, we have $A \vdash \mathtt{succ}\, n : \mathbf{int} \setminus \{\mathbf{int} <: \mathbf{int}^{-}\}$. By (Int) rule, we have $A \vdash n' : \mathbf{int} \setminus \{\}$. Since $\{\mathbf{int} <: \mathbf{int}^{-}\} \models \mathbf{int} \preceq \mathbf{int}^{-}$, and $\{\mathbf{int} <: \mathbf{int}^{-}\} \models \{\}$, the lemma holds in this case.

—Case 2: $e = \mathtt{if0}\, 0\, e_1\, e_2$, $e' = e_1$. By (if0) rule, we have: $A \vdash e' : \tau' \setminus C'$ and $A \vdash e : \tau \setminus C$, where $\tau = t$, and $C' \cup \{\tau' <: t\} \subseteq C$. Since $\tau' <: t \in C$ and $C \subseteq C''$, we have $C'' \models \tau' \preceq \tau$. Since $C' \subseteq C$ and $C \subseteq C''$, we have $C'' \models C'$.

—Case 3: $e = \mathtt{if0}\, n\, e_1\, e_2$, $e' = e_2$. This case can be proved in a similar way as the case 2.

—Case 4: $e = (\lambda x.e_1)^l\, v$, $e' = e_1[v/x]$. By rule (Abs), $A, \{x : t\} \vdash e_1 : \tau_1 \setminus C_1$, and $A \vdash (\lambda x.e_1)^l : (\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l \setminus \{\}$. By the (Int) or (Abs) rule, $A \vdash v : \tau v \setminus \{\}$. And, by rule (Appl), $A \vdash e : \tau \setminus C$, where $\tau = t_2, C = \{(\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l <: t_1 \to t_2,\ \tau v <: t_1\}$. Since C'' is a closure of C, constraints generated by applying the (∀-Elim) closure rule on C are in C''. Thus, there exists a renaming Θ s.t. $\Theta(C_1) \cup \{\tau v <: \Theta(t), \Theta(\tau_1) <: t_2\} \subseteq C''$. By Renaming Lemma (Lemma 4.10), we have $A, \{x : \Theta(t)\} \vdash e_1 : \Theta(\tau_1) \setminus \Theta(C_1)$. Applying Substitution Lemma (Lemma 4.11) on this constraint, we have: $A \vdash e_1[v/x] : \tau' \setminus C'$ s.t. $\{\tau v <: \Theta(t)\} \models \tau' \preceq \Theta(\tau_1)$, and for any constraint $\tau_2 <: \tau_2' \in C'$, there exists τ_2'' s.t. $\{\tau v <: \Theta(t)\} \models \tau_2 \preceq \tau_2''$, and $\tau_2'' <: \tau_2' \in \Theta(C_1)$. Since $\Theta(\tau_1) <: t_2 \in C''$, $C'' \models \Theta(\tau_1) \preceq t_2$. Since $\{\tau v <: \Theta(t)\} \models \tau' \preceq \Theta(\tau_1)$, and $\tau v <: \Theta(t) \in C''$, $C'' \models \tau' \preceq \Theta(\tau_1)$. By transitivity, $C'' \models \tau' \preceq t_2$. To show $C'' \models C'$, we take an arbitrary constraint $\tau_2 <: \tau_2' \in C'$ and show that $C'' \models \tau_2 \preceq \tau_2'$. There exists τ_2'' s.t. $\{\tau v <: \Theta(t)\} \models \tau_2 \preceq \tau_2''$, and $\tau_2'' <: \tau_2' \in \Theta(C_1)$. Since $\tau v <: \Theta(t) \in C''$ and $\Theta(C_1) \subseteq C''$, we have $C'' \models \tau_2 \preceq \tau_2''$ and $C'' \models \tau_2'' \preceq \tau_2'$. By transitivity, $C'' \models \tau_2 \preceq \tau_2'$. Hence $C'' \models C'$.

—Case 5: $e = v^l$, $e' = v$. By the (Label) rule, since we have $A \vdash v^l : \tau \setminus C$, we have $A \vdash v : \tau \setminus C_1$ and $C = C_1 \cup \{\tau <: l\}$. By the (Int) or (Abs) rule, we have $\tau = \tau v$, $C_1 = \{\}$, and $C = \{\tau v <: l\}$. Since $C'' \models \tau v \preceq \tau v$ and $C'' \models \{\}$, the lemma holds in this case.

□

We now establish the subject-reduction property on the standard reduction relation.

LEMMA 4.13 (SUBJECT REDUCTION ON $\longmapsto$). *If $e_1 \longmapsto e_2$, $A \vdash e_1 : \tau_1 \setminus C_1$, and C is a closure of C_1, then there exist τ_2 and C_2 s.t. $A \vdash e_2 : \tau_2 \setminus C_2$, $C \models \tau_2 \preceq \tau_1$ and $C \models C_2$.*

Proof: Since $e_1 \longmapsto e_2$, there exists an evaluation context E s.t. $e_1 = E[e_1']$, $e_2 = E[e_2']$, and $e_1' \longrightarrow e_2'$. We prove the lemma with an induction on the size of E.

—Case 1: $E = [\,]$. The lemma trivially follows from the Subject Reduction Lemma on $\longrightarrow$.

—Case 2: $E = E_1\, e$. Since $A \vdash E_1[e_1']\, e : \tau_1 \setminus C_1$, by (Appl) rule, we have: $A \vdash E_1[e_1'] : \tau_1' \setminus C_1'$, $A \vdash e : \tau_e \setminus C_e$, $\tau_1 = t_2$, and $C_1 = \{\tau_1' <: t_1 \to t_2, \tau_e <: t_1\} \cup C_1' \cup C_e$. Since C is a closure of C_1, C is superset of a closure of C_1'. Thus, by induction, $A \vdash E_1[e_2'] : \tau_2' \setminus C_2'$ s.t. $C \models \tau_2' \preceq \tau_1'$ and $C \models C_2'$. Apply (Appl) rule with $t_1 \to t_2$ as the arrow type for the application, we have: $A \vdash$

$E_1[e_2]\, e \ : \ \tau_2 \setminus C_2$, where $\tau_2 = t_2$ and $C_2 = \{\tau_2' <: t_1 \to t_2, \tau_e <: t_1\} \cup C_2' \cup C_e$. First, $C \models t_2 \preceq t_2$. Secondly, consider constraint $\tau_3 <: \tau_3' \in C_2$. If $\tau_3 <: \tau_3' \in C_2'$, since $C \models C_2'$, $C \models \tau_3 \preceq \tau_3'$; If $\tau_3 <: \tau_3' \in C_e$, since $C_e \subseteq C_1 \subseteq C$, $C \models \tau_3 \preceq \tau_3'$; If $\tau_3 <: \tau_3'$ is $\tau_2' <: t_1 \to t_2$, since $\tau_1' <: t_1 \to t_2 \in C_1 \subseteq C$ and $C \models \tau_2' \preceq \tau_1'$, $C \models \tau_2' \preceq t_1 \to t_2$; If $\tau_3 <: \tau_3'$ is $\tau <: t_1$, since $\tau <: t_1 \in C_1 \subseteq C$, $C \models \tau \preceq t_1$. Hence, $C \models C_2$.

—Case 3: $E = v\, E_1$. This case can be proved in a similar way as the case 2.

—Case 4: $E = \texttt{succ}\, E_1$. Since $A \vdash \texttt{succ}\, E_1[e_1'] \ : \ \tau_1 \setminus C_1$, by (Succ) rule, we have: $A \vdash E_1[e_1'] \ : \ \tau_1' \setminus C_1'$, $\tau_1 = \mathbf{int}$, and $C_1 = \{\tau_1' <: \mathbf{int^-}\} \cup C_1'$. Since C is superset of a closure of C_1', by induction, $A \vdash E_1[e_2'] \ : \ \tau_2' \setminus C_2'$ s.t. $C \models \tau_2' \preceq \tau_1'$ and $C \models C_2'$. Apply (Succ) rule, we have: $A \vdash \texttt{succ}\, E_1[e_2'] \ : \ \tau_2 \setminus C_2$, where $\tau_2 = \mathbf{int}$ and $C_2 = \{\tau_2' <: \mathbf{int^-}\} \cup C_2'$. First, $C \models \mathbf{int} \preceq \mathbf{int}$. Secondly, consider constraint $\tau_3 <: \tau_3' \in C_2$. If $\tau_3 <: \tau_3' \in C_2'$, since $C \models C_2'$, $C \models \tau_3 \preceq \tau_3'$; if $\tau_3 <: \tau_3'$ is $\tau_2' <: \mathbf{int^-}$, since $\tau_1' <: \mathbf{int^-} \in C_1 \subseteq C$, and $C \models \tau_2' \preceq \tau_1'$, thus $C \models \tau_2' \preceq \mathbf{int}$. Hence, $C_1 \models C_2$.

—Case 5: $E = \texttt{if0}\, E_1\, e_3\, e_4$. Since $A \vdash \texttt{if0}\, E_1[e_1']\, e_3\, e_4 \ : \ \tau_1 \setminus C_1$, by (If0) rule, we have: $A \vdash E_1[e_1'] \ : \ \tau_1' \setminus C_1'$, $A \vdash e_3 \ : \ \tau_3 \setminus C_3$, $A \vdash e_4 \ : \ \tau_4 \setminus C_4$, $\tau_1 = t$, and $C_1 = \{\tau_1' <: \mathbf{int^-}, \tau_3 <: t, \tau_4 <: t\} \cup C_1' \cup C_3 \cup C_4$. Since C is also a closure of C_1', by induction, $A \vdash E_1[e_2'] \ : \ \tau_2' \setminus C_2'$ s.t. $C \models \tau_2' \preceq \tau_1'$ and $C \models C_2'$. Apply (If0) rule with t as the type for $\texttt{if0}\, E_1[e_2]\, e_3\, e_4$, we have: $A \vdash \texttt{if0}\, E_1[e_2]\, e_3\, e_4 \ : \ \tau_2 \setminus C_2$, where $\tau_2 = t$ and $C_2 = \{\tau_2' <: \mathbf{int^-}, \tau_3 <: t, \tau_4 <: t\} \cup C_2' \cup C_3 \cup C_4$. First, $C \models t \preceq t$. Secondly, consider constraint $\tau_5 <: \tau_5' \in C_2$. If $\tau_5 <: \tau_5' \in C_2'$, since $C \models C_2'$, $C \models \tau_5 \preceq \tau_5'$; If $\tau_5 <: \tau_5' \in C_3$, since $C_3 \subseteq C_1 \subseteq C$, $C \models \tau_5 \preceq \tau_5'$; If $\tau_5 <: \tau_5' \in C_4$, since $C_4 \subseteq C_1 \subseteq C$, $C \models \tau_5 \preceq \tau_5'$; If $\tau_5 <: \tau_5'$ is $\tau_2' <: \mathbf{int^-}$, since $\tau_1' <: t \in C_1 \subseteq C$, and $C \models \tau_2' \preceq \tau_1'$, thus $C \models \tau_2' \preceq \mathbf{int}$; If $\tau_5 <: \tau_5'$ is $\tau_3 <: t$, since $\tau_3 <: t \in C_1 \subseteq C$, $C \models \tau_3 \preceq t$; If $\tau_5 <: \tau_5'$ is $\tau_4 <: t$, since $\tau_4 <: t \in C_1 \subseteq C$, $C \models \tau_4 \preceq t$. Hence, $C \models C_2$.

—Case 6: $E = E_1^l$. Since $A \vdash E_1[e_1']^l \ : \ \tau_1 \setminus C_1$, by (Label) rule, we have: $A \vdash E_1[e_1'] \ : \ \tau_1 \setminus C_1'$, and $C_1 = \{\tau_1 <: l\} \cup C_1'$. Since C is superset of a closure of C_1', by induction, $A \vdash E_1[e_2'] \ : \ \tau_2' \setminus C_2'$ s.t. $C \models \tau_2' \preceq \tau_1$ and $C \models C_2'$. Apply (Label) rule, we have: $A \vdash E_1[e_2']^l \ : \ \tau_2 \setminus C_2$, where $\tau_2 = \tau_2'$ and $C_2 = \{\tau_2' <: l\} \cup C_2'$. First, since $C \models \tau_2' \preceq \tau_1$ and $\tau_2 = \tau_2'$, $C \models \tau_2 \preceq \tau_1$. Secondly, consider constraint $\tau_3 <: \tau_3' \in C_2$. If $\tau_3 <: \tau_3' \in C_2'$, since $C \models C_2'$, $C \models \tau_3 \preceq \tau_3'$; If $\tau_3 <: \tau_3'$ is $\tau_2' <: l$, since $\tau_1 <: l \in C_1 \subseteq C$ and $C \models \tau_2' \preceq \tau_1$, $C \models \tau_2' \preceq l$. Hence, $C \models C_2$.

□

We are now ready to prove the Subject Reduction Theorem:

THEOREM 4.14 (SUBJECT REDUCTION). *If $e \overset{*}{\longmapsto} e'$, $\vdash e \ : \ \tau \setminus C$, and C'' is a closure of C, then there exist τ' and C' s.t. $\vdash e' \ : \ \tau' \setminus C'$, $C'' \models \tau' \preceq \tau$, and $C'' \models C'$.*

Proof: We use an induction on the length of the reduction sequence $e \overset{*}{\longmapsto} e'$. Suppose $e \longmapsto e_1$ and $e_1 \overset{*}{\longmapsto} e'$. By Lemma 4.13, $\vdash e_1 \ : \ \tau_1 \setminus C_1$, $C'' \models \tau_1 \preceq \tau$ and $C'' \models C_1$. By Lemma 4.7, there exists a closure C_1' of C_1 s.t. $C'' \models C_1'$. Since $e_1 \overset{*}{\longmapsto} e'$, by induction, there exist τ' and C' s.t. $\vdash e' \ : \ \tau' \setminus C'$, $C_1' \models \tau' \preceq \tau_1$, and $C_1' \models C'$. Since $C'' \models C_1'$, by Corollary 4.9, we have $C'' \models \tau' \preceq \tau$, and $C'' \models C'$. □

4.3 Type Soundness and Flow Soundness

We now prove the type soundness of the type inference framework. For expression e, we say the evaluation of e does not go *wrong*, if it is not the case that: $e \stackrel{*}{\longmapsto} e'$ and e' contains a subexpression of any of the following forms: $\texttt{succ}\,(\lambda x.e)^l$; $\texttt{if0}\,(\lambda x.e)^l\,e_1\,e_2$; $n\,e$.

THEOREM 4.15 (TYPE SOUNDNESS). *If* $\vdash\ e\ :\ \tau \setminus C$, C' *is a closure of* C, *and* C' *is type-safe, then the evaluation of* e *cannot go* wrong.

Proof: Suppose the evaluation of e does go *wrong*, then there exists e' s.t. $e \stackrel{*}{\longmapsto} e'$ and e' contains a subexpression of any of the following forms: $\texttt{succ}\,(\lambda x.e)^l$; $\texttt{if0}\,(\lambda x.e)^l\,e_1\,e_2$; $n\,e$. Thus, for any derivation $\vdash\ e'\ :\ \tau' \setminus C''$, C'' is not type-safe. Furthermore, since $\vdash\ e\ :\ \tau \setminus C$, C' is a closure of C, and $e \stackrel{*}{\longmapsto} e'$, by Theorem 4.14, we have $\vdash\ e'\ :\ \tau' \setminus C''$ and $C' \models C''$. Since C'' is not type-safe, there exists $\mathbf{int} <: t_1 \to t_2 \in C''$ or $(\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l <: \mathbf{int}^{\text{-}} \in C''$. If $\mathbf{int} <: t_1 \to t_2 \in C''$, $C' \models \mathbf{int} \preceq t_1 \to t_2$. By Lemma 4.2 and Lemma 4.4, $\mathbf{int} <: t_1 \to t_2 \in C'$. If $(\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l <: \mathbf{int}^{\text{-}} \in C''$, $C' \models (\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^l \preceq \mathbf{int}^{\text{-}} \in C''$. By Lemma 4.2 and Lemma 4.4, there exists $(\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^l <: \mathbf{int}^{\text{-}} \in C'$. Thus, C' is not type-safe, and this is a contradiction. □

THEOREM 4.16 (FLOW SOUNDNESS). *For program* e, *if* $\vdash\ e\ :\ \tau \setminus C$, *and* C' *is a closure of* C, *then: if* $e \stackrel{*}{\longmapsto} E[n^l]$, *then* $\mathbf{int} <: l \in C'$; *if* $e \stackrel{*}{\longmapsto} E[v^l]$ *where* $v = (\lambda x.e_1)^{l_1}$, *then* $(\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^{l_1} <: l \in C'$.

Proof: Suppose $e \stackrel{*}{\longmapsto} E[v^l]$. By Theorem 4.14, we have $\vdash\ E[v^l]\ :\ \tau' \setminus C''$ and $C' \models C''$. An induction on the structure of evaluation context E can prove that there must be derivations $\vdash\ v\ :\ \tau v \setminus \{\}$ and $\vdash\ v^l\ :\ \tau v \setminus \{\tau v <: l\}$ which are sub-derivations of $\vdash\ E[v^l]\ :\ \tau' \setminus C''$, and $\tau v <: l \in C''$. Since $C' \models C''$, we have $C' \models \tau v \preceq l$. By Lemma 4.2 and Lemma 4.4, we have: if $v = n$, then $\tau v = \mathbf{int}$ and $\mathbf{int} <: l \in C'$; if $v = (\lambda x.e_1)^{l_1}$, then $\tau v = (\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^{l_1}$, $C' \models (\forall\ \overline{t}.\ t \to \tau_2 \setminus C_2)^{l_1} \preceq (\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^{l_1}$, and $(\forall\ \overline{t}.\ t \to \tau_1 \setminus C_1)^{l_1} <: l \in C'$. □

The flow soundness theorem ensures the correctness of any flow analysis obtained from our type inference framework: if an expression evaluates to a data value at run-time, then, the flow analysis statically predicts that the data value would flow to the expression.

5. ANALYZING OBJECT-ORIENTED LANGUAGES

In this section, we present a polymorphic constraint-based type inference framework for object-oriented languages. It extends the basic type inference framework studied in section 2 with the ability to analyze object-oriented languages.

Definition 5.1 (THE OBJECT LANGUAGE).

$$\begin{array}{l} e ::= \ldots \mid \mathbf{new}\ \delta \mid e.f \mid e.f = e \mid e.m(\overline{e}) \mid \mathbf{this} \mid (\delta)e \mid e; e \\ \mid \mathbf{class}\ \texttt{Object}\ \{\ \} \mid \mathbf{class}\ \delta\ \mathbf{extends}\ \delta'\ \{\ \overline{f};\ \overline{m(\overline{x})e}\ \} \end{array}$$

where $e \in \mathbf{Expression}$, $\overline{e}$ is an expression list, $\{x, \mathbf{this}\} \in \mathbf{Variable}$, $\overline{x}$ is an variable list, $\{l, f, m\} \subseteq \mathbf{Label}$, $\overline{f}$ is an label list, and $\{\delta, \delta', \texttt{Object}\} \subseteq \mathbf{ClassName} \subseteq \mathbf{Variable}$.

The language defined above extends the language in Section 2.1 with key object-oriented language features. The object-oriented part of the language is basically the Featherweight Java [Igarashi et al. 2001] extended with field assignments ($e.f = e$). For simplicity, we omit the class constructors, which can be analyzed as global functions with an extra parameter "this". A class definition **class** δ **extends** δ' $\{$ $\overline{f}$; $\overline{m(\overline{x})e}$ $\}$ defines class δ with δ' as the parent class, $\overline{f}$ as the field variable list, and $\overline{m(\overline{x})e}$ as the instance method list. `Object` is a special class without fields and methods. As in Featherweight Java, if the receiver for a field access or method invocation is "this", the receiver needs to be explicitly written out as **this**. To analyze the object-oriented language, additional types need to be defined.

Definition 5.2 (Types). The type grammar is as follows.

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

In the above definition, **ValueType** denotes the types for data values, which include primitive type **int**, object types and bottom type $\bot$. The type $\bot$ (also written as **null**) is the type for null value.

The type for an object value is of form $\mathbf{obj}(\delta, [\,\overline{l_i : \tau_i}\,])$, where δ is the name of the corresponding class, and the notation $[\,\overline{l_i : \tau_i}\,]$ enumerates the type for every instance field and instance method of the object. 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. The type scheme we define here is the same as the type scheme defined in the framework of Section 2 except that the type scheme here can have multiple formal arguments.

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 downcast operation $(\delta)e$ is analyzed with type $\mathbf{cast}(\delta, t)$, where t is the type for the result of the downcast operation.

5.1 Constraint Generation

We now define the constraint generation process for the object-oriented language. Analysis of more advanced features in Java such as exceptions and inner classes will be discussed later.

In addition to the rules in Figure 1, more type inference rules are defined in Figure 3.

Every class is analyzed with a constraint $(\forall\, \overline{t}.\ t_0 \to \mathbf{obj}(\delta, [\, l_i : \tau_i\,]) \setminus \{\}) <: t_\delta$, where t_δ is a special type variable generated for the class, and the type scheme is the *creation-type-scheme* for the class. For analyzing forward or recursive references

$$\text{(Seq)}\quad \frac{A \vdash e_1 : \tau_1 \setminus C_1,\ \{\overline{\tau <: t_\delta}\} \subseteq C_1,\ A, \{\overline{\delta : \tau}\} \vdash e_2 : \tau_2 \setminus C_2}{A \vdash e_1; e_2 : \tau_2 \setminus C_1 \cup C_2}$$

$$\text{(Read)}\quad \frac{A \vdash e : \tau \setminus C}{A \vdash e.f : t \setminus \{\tau <: [\, f : \mathbf{read}\ t\,]\} \cup C}$$

$$\text{(Write)}\quad \frac{A \vdash e_1 : \tau_1 \setminus C_1,\ \ e_2 : \tau_2 \setminus C_2}{A \vdash e_1.f = e_2 : \tau_2 \setminus \{\tau_1 <: [\, f : \mathbf{write}\ \tau_2\,]\} \cup C_1 \cup C_2}$$

$$\text{(Cast)}\quad \frac{A \vdash e : \tau \setminus C}{A \vdash (\delta)e : t \setminus \{\tau <: \mathbf{cast}(\delta, t)\} \cup C}$$

$$\text{(Meth)}\quad \frac{A, \{\overline{x_i} : \overline{t_i}, \mathbf{this} : t'\} \vdash e : \tau \setminus C}{A \vdash m(\overline{x_i})e : (\forall\, \overline{t}.\ (\overline{t_i} \times t') \to \tau \setminus C) \setminus \{\}}$$
where $\overline{t} = FTV((\overline{t_i} \times t') \to \tau \setminus C) - FTV(A) - T_\delta$

$$\text{(Msg)}\quad \frac{A \vdash e : \tau \setminus C,\ A \vdash \overline{e_i : \tau_i \setminus C_i}}{A \vdash e.m(\overline{e_i}) : t \setminus \{\tau <: [\, m : (\overline{t_i} \times t') \to t\,],\ \overline{\tau_i <: t_i}\} \cup C \cup \overline{C_i}}$$

$$\text{(Obj)}\quad \frac{}{A \vdash \mathbf{class}\ \texttt{Object}\ \{\ \} : t_{Object} \setminus \{(\forall\, \{t\}.\ t \to \mathbf{obj}(\texttt{Object}, [\,]) \setminus \{\}) <: t_{Object}\}}$$

$$\text{(New)}\quad \frac{}{A \vdash \mathbf{new}\ \delta : t \setminus \{t_\delta <: t' \to t, \mathbf{null} <: t'\}}$$

$$\text{(Class)}\quad \frac{A(\delta') = (\forall\, \overline{t}'.\ t'_0 \to \mathbf{obj}(\delta', [\overline{f' : u'}, \overline{m'_i : \tau'_i}]) \setminus \{\}),\ A \vdash \overline{m_j(\overline{x})e_j : \tau_j \setminus \{\}}}{A \vdash \mathbf{class}\ \delta\ \mathbf{extends}\ \delta'\ \{\ \overline{f};\ \overline{m_j(\overline{x})e_j}\ \} : t_\delta \setminus \{\tau'' <: t_\delta\}}$$
where $\tau'' = (\forall\, \overline{t}.\ t_0 \to \mathbf{obj}(\delta, [\overline{f' : u''}, \overline{f : u}, \overline{m_j : \tau_j}, \overline{m_k : \tau_k}]) \setminus \{\})$,
$\{\overline{m_k}\} = \{\overline{m'_i}\} - \{\overline{m_j}\},\ \tau_k = \tau'_i$ for each $m_k = m'_i$,
and $\overline{t} = FTV(t_0 \to \mathbf{obj}(\delta, [\overline{f' : u''}, \overline{f : u}, \overline{m_j : \tau_j}, \overline{m_k : \tau_k}]) \setminus C) - FTV(A) - T_\delta$

Fig. 3. Type Inference Rules for Objects

of class definitions, a special type variable t_δ is generated for every class δ in the program. The set of all such type variables is written as T_δ in Figure 3. The *creation-type-scheme* for class δ is of the form $(\forall\ \overline{t}.\ t_0 \to \mathbf{obj}(\delta, [\, l_i : \tau_i\,]) \setminus \{\})$, which is a type scheme for a function returning an object upon application. In the object type $\mathbf{obj}(\delta, [\, l_i : \tau_i\,]) \setminus \{\})$, every instance field is associated with a fresh imperative type variable u, which is bound in the type scheme. The (New) rule analyzes **new** δ with constraints $\{t_\delta <: t' \to t, \mathbf{null} <: t'\}$. Thus, object creation is analyzed as a special form of function application. This allows the possibility of generating different object types by instantiating the creation-type-scheme differently. As we will discuss in Section 6, this is vital for precise analysis of data-polymorphic programs.

The (Obj) rule assigns class `Object` with *creation-type-scheme* $(\forall\ \{t\}.\ t \to \mathbf{obj}(\texttt{Object}, [\,]) \setminus \{\})$. The (Class) rule assigns class δ with *creation-type-scheme* $(\forall\ \overline{t}.\ t_0 \to \mathbf{obj}(\delta, [\overline{f' : u''}, \overline{f : u}, \overline{m_j : \tau_j}, \overline{m_k : \tau_k}]) \setminus \{\})$, where the type for each field is a fresh imperative type variable, the type for each method m_j defined in δ is τ_j, and the types for all other methods $\overline{m_k}$ are inherited from the parent class. The (Seq) rule analyzes a sequential composition $e_1; e_2$. By adding $\{\overline{\delta : \tau}\}$ to the typing environment, it makes the types for the classes $\overline{\delta}$ defined in e_1 available for analyzing e_2.

The (Meth) rule assigns an instance method $m(\overline{x_i})e$ with type $(\forall\ \overline{t}.\ (\overline{t_i} \times t') \to \tau \setminus C)$, where $\overline{t_i}$ are type variables for the formal arguments $\overline{x_i}$, t' is the type for "this", τ is the return type, C collects constraints generated for the method body e, and $\overline{t}$ collects type variables locally generated when analyzing this method.

By adding an extra argument for "this", we avoid altering the type scheme of an instance method when it is inherited by subclasses. A method invocation is analyzed by the (Msg) rule, in which t' is the extra "this" argument, and constraints $\overline{\tau_i <: t_i}$ model the flow from actual arguments to formal arguments.

The (Write) rule analyzes $e_1.f = e_2$ with constraint $\tau_1 <: [\, f : \mathbf{write}\ \tau_2\,]$, meaning that e_1 is expected to be an object that has field f, and this field is written with a value of type τ_2. Similarly, the (Read) rule analyzes $e.f$ with constraint $\tau <: [\, f : \mathbf{read}\ t\,]$, meaning that e is expected to be an object that has field f, and the type of reading this field is t. The (Cast) rule analyzes $(\delta)e$ with constraint $\tau <: \mathbf{cast}(\delta, t)$, meaning that the type of e is subject to a downcast operation to δ and t is the type for the result of the cast operation.

5.2 Computation of the Closure

After generating the initial set of constraints corresponding to the program text, the inference algorithm computes the complete flow information by applying the set of closure rules in Figure 4 to the constraint set. Each rule here specifies a condition under which more constraints are generated, following the format of Section 2. The closure computation starts with the initial constraint set, and the closure rules are applied until no more constraints can be generated.

The (Trans) rule is unchanged from the functional framework. 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 $\mathbf{cast}(\delta', \tau)$ is the type for a downcast operation casting an expression to Java type δ', with τ as the 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 of 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. Here the "Java type" means Java interface or class, and we say δ is a Java subtype of δ' if only if δ is a subtype of δ' according to the Java static type system. This rule is more general than what is needed for the core object language we defined in which δ' can only be a class.

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 for invoking the method with the actual arguments specified by the new arrow type. To avoid mixing applications of different object types reaching the receiver position of the same method invocation, a fresh new type variable t' replaces t'_n in the new arrow type.

The most important closure rule is ($\forall$-Elim), which models method invocation and object creation. It is the ($\forall$-Elim) rule in Figure 2 extended to allow multiple arguments. 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 local variables and parameters of the method. To model this behavior in the analysis, a renaming $\Theta \in \mathbf{TypeVar} \xrightarrow{\text{p}} \mathbf{TypeVar}$ is applied to

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

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

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

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

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

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

Fig. 4. Constraint Closure Rules for Objects

type variables in $\overline{t}$, similarly to the functional framework. The nCFA instantiation of the framework follows the functional one given in Section 3. To obtain a CPA instantiation here, Θ in the ($\forall$-Elim) rule is defined as follows:

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

The contours Θ are generated based on all actual argument types τv_i, thus two invocations of the method share the same contour if and only if all actual argument types are the same. The original CPA algorithm [Agesen 1996] uses the cartesian product of class names of argument values for contour selection. Our CPA definition differs in that we use object types rather than class names. Because CPA only generates one object type per class, our formulation of CPA behaves the same as the original CPA.

A concrete class analysis is a standard object-oriented program analysis in which a conservative approximation to the set of classes a given expression could take on at run time is made. It is defined similarly to how a flow analysis was defined for the functional language, replacing function labels l with class names δ.

Definition 5.3 (Concrete Class Analysis). For an expression e in the program, if the type variable for e is t by the inference rules, the set of concrete classes for e, $CC(e)$, is the 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 t, then $\delta \in CC(e)$.

For example, analyzing the program of Figure 5 with the 0CFA or CPA algorithm produces $CC(\texttt{ht1.get("zero")}) = \{\texttt{Boolean}, \texttt{Integer}\}$.

We do not present a soundness proof of the object-oriented type inference framework due to space considerations. We take an approach based on an encoding of objects into an extended functional language. The functional language defined in Section 2 can be extended with records and references, and the type inference framework and the soundness proof can be extended accordingly. Eifrig et al. [1995b] presents such a language with a sound constraint-based type inference system. This aforecited system has let-polymorphism only, and so a combination of this

```
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. 5. Java program with data polymorphism

paper with the results of Section 2 will provide soundness. The existing type rules are compatible with the record and reference extensions, the main drawback with adding these features is the overall increase in complexity.

Once this framework with records and references has been defined, the soundness of the object-oriented framework can be established by translating it into this extended language using a standard object-encoding mechanism. Eifrig et al. [1995a] presents one example of how the translation approach can be used to establish soundness of object-based type systems.

6. DATA POLYMORPHIC ANALYSIS

In this section we study data polymorphism. We first give a detailed introduction to the concept of data polymorphism, illustrate the imprecision of 0CFA and CPA for data-polymorphic programs, and discuss the difficulty which data polymorphism brings to type inference. We then introduce Data-Polymorphic CPA (DCPA), a novel algorithm which extends CPA with the ability to effectively analyze data-polymorphic programs.

6.1 Motivation

Data polymorphism is defined in [Agesen 1996] 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 $(\forall\, \overline{t}.\ \tau \setminus C) <: t \rightarrow [\![\texttt{new C()}]\!]$[1] and **null** $<: t$, where $(\forall\, \overline{t}.\ \tau \setminus C)$ is the creation-type-scheme of class `C`. For 0CFA and CPA, the (∀-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 5. Two instances of `Hashtable` are created 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.

[1]To ease presentation, we write the type variable for expression e as $[\![\texttt{e}]\!]$

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 which can be shared without losing precision (the *CPA-safe* or *reusable* contours), and those related to data polymorphism and thus sharing such contours might cause a loss of precision (the *CPA-unsafe* contours). The DCPA algorithm generally follows CPA, but 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*, the DCPA algorithm would generate a fresh contour instead of reusing the existing one. The DCPA algorithm 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 $(\forall\ \overline{t}.\ \tau \setminus C) <: t \rightarrow [\![\texttt{new C()}]\!]$ and $\mathbf{null} <: t$, where $(\forall\ \overline{t}.\ \tau \setminus C)$ is the creation-type-scheme of class `C`. We will call such a pair of constraints a *creation point* of class C. If a class contains any polymorphic field (field which may store values of different types according to Java's static type system), the 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, the 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 lets them share the same object type. The goal of the DCPA algorithm is to generate more contours so that such imprecision can be avoided. However, the DCPA algorithm should also be efficient and only generate more contours when necessary. Thus, if a method invocation doesn't cause the creation of any objects, the contour for such a method invocation is marked CPA-safe and can be reused. For example, consider the program in Figure 6. The DCPA algorithm 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 directly, then the contour for such a method invocation is also CPA-safe. Consider the program in Figure 6. 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`. Since the type variable for field `a` would be their only upper bound, and our analysis is flow-insensitive, the two object types would be equivalent in terms of closure computation. Thus, for the

```
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 f(Hashtable x) { x.put("DCPA", new Vector()); return x;}
  static Object h() {
    Vector v=new Vector(); v.addElement(new Boolean(true)); return v;
  }
  static Object k() {
    Vector v=new Vector(); v.addElement(new Hashtable()); return v;
  }
  public static void main(String args[]) {
    Hashtable obj=new Hashtable();
    id(obj); id(obj); g(obj); g(obj); f(obj); f(obj);
    h(); h(); k(); k();
  }
}
```

Fig. 6. Example Program with CPA-Safe and CPA-unsafe Contours

two invocations of method `g`, generating two contours would not be beneficial, and DCPA would let them share a single CPA-safe contour without any loss of precision. Method `f` in Figure 6 is another example. Even though a `Vector` instance is created locally by `f`, since the return value (the `Hashtable` instance) is shared by the two invocations, DCPA would let the two invocations share the same CPA-safe contour without any loss of precision. Similarly, for any Java method whose return type is `void` or `int`, all contours are CPA-safe.

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 values of fixed types, such an invocation is also considered CPA-safe. For example, consider method `h` in Figure 6. 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. The DCPA algorithm would let the two invocations of method `h` share a single contour. In contrast, the DCPA algorithm would generate two separate CPA-unsafe contours for the two invocations of method `k` in Figure 6.

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 generated. To make the algorithm feasible, an additional unification heuristic is incorporated into the DCPA algorithm. The idea is 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.

6.2 The DCPA Algorithm

In this section we define the DCPA algorithm. DCPA is only a closure algorithm. The initial set of constraints are those produced by the inference rules in the frame-

Algorithm DCPA(C_0, Ω, S)

1 **if** $C_1 \subseteq C_0$ **and** applying a non-(∀-Elim) rule on C_1 results in C_2 **and** $C_2 \not\subseteq C_0$
2 **then return** DCPA$(C_0 \cup C_2, \Omega, S)$
3 **if** $\{(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C) <: (t'_1 \times \ldots \times t'_n) \to \tau', \tau v_i <: t'_i\} \subseteq C_0$
4 **and** no Θ s.t. $\{\tau v_i <: \Theta(t_i), \Theta(\tau) <: \tau'\} \cup \Theta(C) \subseteq C_0$
5 **then if** $(C'_0, (\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C), \overline{\tau v_i}', \Theta', i') \in \Omega$ **and** $(\overline{\tau v_i} = \overline{\tau v_i}'$ **or**
6 $(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C) \in S$ **and** $JavaType(\overline{\tau v_i}) = JavaType(\overline{\tau v_i}'))$
7 **then return** DCPA$(C_0 \cup \{\tau v_i <: \Theta(t_i),\ \Theta'(\tau) <: \tau'\} \cup \Theta'(C), \Omega, S)$
8 **else** $\Theta \leftarrow$ a fresh renaming on type variables in $\overline{t}$
9 $\rho \leftarrow (C_0, (\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C), \overline{\tau v_i}, \Theta, i)$
10 $C_1 \leftarrow C_0 \cup \{\tau v_i <: \Theta(t_i)\} \cup \Theta(C)$
11 $(C_2, \Omega') \leftarrow$ DCPA$(C_1, \Omega \cup \{\rho\}, S \cup \{(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C)\})$
12 **if** $\Theta(\tau)$is incomplete for ρ in C_2
13 **then** $\Omega' \leftarrow \Omega' - \{\rho\}$
14 **return** DCPA$(C_2 \cup \{\Theta(\tau) <: \tau'\}, \Omega', S)$
15 **return** (C_0, Ω)

Fig. 7. DCPA Algorithm without unification

work defined in Section 5. The closure computation starts with the initial constraint set. Applying a closure rule on the constraint set results in more constraints added to the constraint set. Such a process continues until no more constraints can be added to the constraint set. Thus there exists a *current constraint set* before any rule application. Recall that the (∀-Elim) rule in the framework is defined as follows:

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

First, we define the representation of *contour*, which records information about a (∀-Elim) rule application in the closure computation.

Definition 6.1 (Contour). A contour ρ is a tuple of form $(C_0, (\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C), \overline{\tau v_i}, \Theta, i)$, where C_0 is the current constraint set before the ∀-elimination, $(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C)$ is the ∀-type, $\overline{\tau v_i}$ is the vector of the actual argument types, Θ is the renaming on $\overline{t}$ used by the ∀-elimination, and i is the unique identifier for this contour.

We define the state of a closure computation as follows:

Definition 6.2 (Closure State). A *closure state* is a tuple (C, Ω, S), where C is the current constraint set, Ω is the *contour cache*, which is the set of reusable contours, and S is a set of ∀-types.

We first define a DCPA algorithm without unification in Figure 7. The algorithm is defined as a recursive function which takes a *closure state* as input and returns a pair (C, Ω) as output. The closure computation starts with an initial closure state (C_0, Ω_0, S_0), where C_0 is the initial constraint set produced by inference rules, and Ω_0 and S_0 are empty sets. The closure computation finishes when DCPA(C_0, Ω_0, S_0) returns (C_1, Ω_1), where constraint set C_1 is the result closure.

In Figure 7, lines 1 and 2 describe the non-(∀-Elim) rule application case: if C_1 in the current constraint set is subject to a non-(∀-Elim) rule application, and the

rule has not been applied to C_1 yet, the algorithm applies the rule and adds the resulting constraints C_2 to the current constraint set.

Lines 3 to 14 describe the case of (∀-Elim) rule applications. The condition in line 3 means that the constraints $\{(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C) <: (t'_1 \times \ldots \times t'_n) \to \tau', \tau v_i <: t'_i\}$ in current constraint set C_0 are subject to (∀-Elim) rule application, and the condition in line 4 means that the (∀-Elim) rule has not been applied to those constraints yet. Under such conditions, the algorithm applies the (∀-Elim) rule by either reusing an existing contour or creating a new one. Lines 5 to 7 describe the reusing contour case. When a new contour is generated for a ∀-elimination, the algorithm adds the ∀-type to S when computing the closure for all constraints generated from the ∀-elimination (see the recursive call on DCPA in line 11). Thus, at line 6, the condition $(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C) \in S$ means that the method invocation corresponding to the ∀-elimination is considered recursive by the algorithm, and S is the set of ∀-types approximating the call-path leading to the method invocation. The condition in lines 5 to 6 means that there is already a contour in the resuable contour cache for the same ∀-type, and either $\overline{\tau v_i} = \overline{\tau v_i}'$, which means the contour and the current ∀-elimination share the same argument types, or the method invocation is considered recursive and the argument types for the contour and current ∀-elimination are of the same Java types. In such a case, the algorithm reuses the contour by using the same renaming Θ' of the contour for this ∀-elimination. The recursive detection is to ensure the termination of DCPA. For recursive method invocations, DCPA selects contour in the same way as CPA: different object types of the same class (Java type) are considered equivalent.

Lines 8 to 14 describe the case of new contour creation. In this case, since there is no contour in Ω to reuse, the algorithm creates a new contour ρ, which is defined in lines 8 and 9. The contour corresponds to a function application (method invocation or object creation). The algorithm needs to judge if the contour is reusable (CPA-safe) or not. Thus, as discussed in section 6.1, the algorithm needs to know the following properties about the function application: whether an object is created locally by the function application; whether an object is reachable from the return value; what the type for an object is when the function returns. To do that, the algorithm needs to determine the set of local constraints corresponding to the local computation of the function application. According to the (∀-Elim) closure rule, this ∀-elimination would generate such a set of constraints: $\{\tau v_i <: \Theta(t_i), \Theta(\tau) <: \tau'\} \cup \Theta(C)$, in which constraints $\tau v_i <: \Theta(t_i)$ correspond to the flow from actual arguments to formal arguments for this function application, constraints $\Theta(C)$ correspond to the function body, and $\Theta(\tau) <: \tau'$ correspond to the flow from the return value of the function to the application result. All those constraints except for $\Theta(\tau) <: \tau'$ are considered local constraints of the closure. The local constraint set is initially defined as C_1 in line 10. In line 11, the complete local constraint set C_2 for contour ρ is computed via the recursive call. The condition in line 12 means that the return type $\Theta(\tau)$ is *incomplete* for ρ in local constraint set C_2. In such a case, ρ is judged as unreusable (CPA-unsafe). The condition refers to the following definition:

Definition 6.3 (Incomplete Type). Type τ is defined as *incomplete* for contour $\rho = (C_0, (\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \to \tau \setminus C), \overline{\tau v_i}, \Theta)$ in constraint set C_1 iff either of

the following two cases holds:

(1) $\tau = \mathbf{obj}(\delta, [\overline{l_i : \tau_i}])$, τ does not appear in C_0, and there exists τ_i which is *incomplete* for ρ in C_1;
(2) $\tau = t$, for all $\tau v <: t \in C_1$, τv is *incomplete* for ρ in C_1, and for all $\{\tau v_1 <: t, \tau v_2 <: t\} \subseteq C_1$, $JavaType(\tau v_1) = JavaType(\tau v_2)$.

Consider a contour ρ and its local constraint set C_1. The condition "τ does not appear in C_0" in the first case means that the object type τ is created by generating contour ρ. Thus, the first case judges an object type τ as *incomplete* if τ is created locally by ρ, and some type variables for the fields of τ are judged as *incomplete*. The second case judges a type variable t as *incomplete* if either there is no value type lowerbound for t in C_1 or all value type lowerbounds for t are *incomplete* and from the same class. The DCPA algorithm judges ρ as CPA-unsafe if and only if the return type of ρ is *incomplete*. If ρ is for an object creation and the $\forall$-type is a *creation-type-scheme* for a polymorphic class, the return type of ρ is an object type with all its field type variables unbounded, thus the return object type is *incomplete* and ρ is CPA-unsafe. If ρ is for a method invocation and the return type of ρ is *incomplete*, then the return value of the method invocation is an object created locally within the method scope and, within the method scope, at least one field of the object is assigned with no values or assigned only with object values of *incomplete* types. In such a case, there are objects created locally by the method invocation which escapes the method scope via the return value directly, and those objects are not assigned with values of fixed-types (Java primitive types and object types which are either not *incomplete* or from multiple different classes). In such a case, the contour is judged as CPA-unsafe. Thus the definition of *incomplete* types implements the ideas discussed in Section 6.1. Consider the example program in Figure 6. The invocations of methods `id`, `g` and `f` all return the `Hashtable` instance created outside the scope of those methods. Thus, for the contours corresponding to those method invocations, the object type for the `Hashtable` instance is not *incomplete*, the return types for those contours are not *incomplete*, and those contours are judged as CPA-safe.

If the contour ρ is unreusable (CPA-unsafe), it is removed from the reusable contour cache Ω' in line 13. At line 14, the algorithm continues the closure computation with the updated constraint set and contour cache. Finally, according to line 15, when no more closure rules can be applied in current constraint set, the closure computation finishes.

LEMMA 6.4. *The DCPA algorithm in Figure 7 is an instantiation of the framework in Section 5.*

Proof: Consider the algorithm in Figure 7. In lines 1 and 2, all non-($\forall$-Elim) closure rules are applied as the framework stipulates. Consider the ($\forall$-Elim) rule. First, the conditions in lines 3 to 4 ensure that for any combination of the constraints on which the ($\forall$-Elim) rule is applicable, the rule is applied once and only once, and there is one and only one renaming Θ used for the rule application. Thus DCPA gives a well-defined Θ for every ($\forall$-Elim) rule application. When conditions in lines 3 to 4 are satisfied, the ($\forall$-Elim) rule requires the generation of such a constraint set: $C_3 = \{\tau v_i <: \Theta(t_i), \Theta(\tau) <: \tau'\} \cup \Theta(C)$. For the contour reuse case, exactly

Algorithm DCPA(C_0, Ω, S)
1 **if** $\{\mathbf{obj}(\delta, [\overline{l_i : \tau_i}]) <: t, \mathbf{obj}(\delta, [\overline{l_i' : \tau_i'}]) <: t\} \subseteq C_0$
2 **then** $\Theta \leftarrow U(\mathbf{obj}(\delta, [\overline{l_i : \tau_i}]), \mathbf{obj}(\delta, [\overline{l_i : \tau_i'}]))$
3 **if** $\Theta(C_0) \neq C_0$
4 **then return** DCPA$(\Theta(C_0), \Theta(\Omega), S)$
5 **if** $C_1 \subseteq C_0$ **and** applying a non-(∀-Elim) rule on C_1 results in C_2 **and** $C_2 \not\subseteq C_0$
6 **then return** DCPA$(C_0 \cup C_2, \Omega, S)$
7 **if** $\{(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C) <: (t_1' \times \ldots \times t_n') \rightarrow \tau', \tau v_i <: t_i'\} \subseteq C_0$
8 **and** no Θ exists s.t. $\{\tau v_i <: \Theta(t_i), \Theta(\tau) <: \tau'\} \cup \Theta(C) \subseteq C_0$
9 **then if** $(C_0', (\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C), \overline{\tau v_i}', \Theta', i') \in \Omega$ **and** $(\overline{\tau v_i} = \overline{\tau v_i}'$ **or**
10 $(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C) \in S$ **and** $JavaType(\overline{\tau v_i}) = JavaType(\overline{\tau v_i}'))$
11 **then return** DCPA$(C_0 \cup \{\tau v_i <: \Theta(t_i),\ \Theta'(\tau) <: \tau'\} \cup \Theta'(C), \Omega, S)$
12 **else** $\Theta \leftarrow$ a fresh renaming on type variables in $\overline{t}$
13 $\rho \leftarrow (C_0, (\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C), \overline{\tau v_i}, \Theta, i)$
14 $S' \leftarrow S \cup \{(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C)\}$
15 $(C_2, \Omega') \leftarrow$ DCPA$(C_0 \cup \{\tau v_i <: \Theta(t_i)\} \cup \Theta(C), \Omega \cup \{\rho\}, S')$
16 $\rho' \leftarrow (C_0', (\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C), \overline{\tau v_i}', \Theta', i)$ s.t. $\rho' \in \Omega'$
17 **if** $\Theta'(\tau)$ is incomplete for ρ' in C_2
18 **then** $\Omega' \leftarrow \Omega' - \{\rho'\}$
19 **return** DCPA$(C_2 \cup \{\Theta'(\tau) <: \tau''\}, \Omega', S)$
20 **return** (C_0, Ω)

Fig. 8. DCPA Algorithm

constraints in C_3 are added to the closure in line 7. For the contour creation case in lines 8 to 14, constraints in $C_4 = C_2 \cup \{\Theta(\tau) <: \tau'\}$ are added to the closure. Since a constraint is never removed from the closure after the constraint is added, we have $C_1 \subseteq C_2$ in line 11. Thus, $C_3 \subseteq C_4$, and DCPA follows the framework when applying the (∀-Elim) rule. Finally, notice that every constraint added to the closure is generated by following a closure rule in the framework. □

We now complete the DCPA algorithm definition by adding unification. The DCPA algorithm unifies two object types when they are from the same class and they are lower bounds of the same type variable. The unification algorithm is defined as follows.

Definition 6.5 (Unification of Object Types). The unifier of the two object types of the same class δ, $U(\mathbf{obj}(\delta, [\overline{l_i : \tau_i}]), \mathbf{obj}(\delta, [\overline{l_i : \tau_i'}]))$, is a composition of substitutions $\lfloor \tau_i'/\tau_i \rfloor$ for each i s.t. τ_i is an imperative type variable.

We give a complete DCPA definition with unification in Figure 8. In Figure 8, lines 1 to 4 describe the unification application. The condition in line 1 means that the current constraint set C_0 is subject to a unification application, and the condition in line 3 means that the unification has not been applied to C_0 yet. When the two conditions are satisfied, the algorithm applies the unification to the current closure state in line 4. The remaining part of the algorithm presentation (lines 5 to 20) simply follows the algorithm in Figure 7 with the effect of unification incorporated.

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

Definition 6.6 (DCPA Algorithm). For a program e with initial constraint set C, if DCPA$(C, \{\}, \{\})$ returns (C', Ω), then C' is a DCPA closure for e.

We now discuss the soundness of DCPA. Given a program, if applying the DCPA algorithm without unification results in closure C_1, and applying the DCPA algorithm with unification results in closure C_2, then C_2 can be obtained from C_1 by unifying some type variables in C_1 and adding more constraints accordingly. Thus, if there are no type-contradictory constraints in C_2, then there are no type-contradictory constraints in C_1 as well. So, if the DCPA algorithm without unification is sound, then the DCPA algorithm with unification is sound as well. By Lemma 6.4, the DCPA algorithm without unification is an instantiation of the framework in Section 5. Thus the soundness of the DCPA algorithm follows from the soundness of the framework.

In the presence of nested type schemes, CPA may not terminate for some programs. In [Smith and Wang 2000] we have formalized a provably terminating CPA algorithm. The basic idea is to share more contours to prevent a flow-dependency relation in the program that may cause the generation of an infinite number of contours. To avoid the cost of instantiating nested type schemes, no nested type scheme is generated in our system. Thus, in our system, CPA would always terminate without any special mechanism to enforce termination.

We now discuss the termination issue of the DCPA algorithm. Without the recursion detection mechanism, the DCPA may not terminate for some recursive programs. For example, consider a simple Java program:

```
class C {
    Object x;
    static void f(C y) { f(new C());}
    public static void main(String args[]) { f(new C()); }
}
```

Applying the DCPA without recursion detection incurs infinite loop: every contour generated for method `f` causes a new object type generated for class `C`, which in turns causes a new contour generated for `f`. With recursion detection, the DCPA generates only one contour for `f`. Since two object types are always unified whenever they flow together, the possible number of argument types for any $\forall$-elimination is finite. With an induction on the length of the shortest call-path from the main function to any method invocation or object creation, it can be proved that the number of $\forall$-eliminations in the DCPA closure computation for any method invocation or object creation is finite. Thus, the DCPA algorithm terminates for arbitrary programs.

7. IMPLEMENTATION

In this section, we discuss our implementation of constraint-based type inference for Java. The algorithms we have implemented include 0CFA, CPA and DCPA. The system is itself written in Java. It takes Java source code as input and statically checks the validity of all downcasts 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 for the object values 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.

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.

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.

7.1 Java Language Features

In Section 5, we have discussed how type inference can be performed on the key features of object-oriented languages. In this section, we will discuss in more detail how the analysis of various Java language features is implemented in our type inference system for Java.

We first discuss the concrete implementation of object types. Recall that an object type is of the form $\mathbf{obj}(\delta, [\overline{l_i : \tau_i}])$, where δ is the identifier of the corresponding class, and the notation $[\overline{l_i : \tau_i}]$ enumerates the type for every instance field and instance method of the object. A naive implementation would store in every object type all type schemes for the instance methods. This is inefficient since different object types of the same class have the same set of type schemes for the instance methods of the class. In our system, the class identifier δ for 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 all instance methods of this class, type schemes for constructors and static methods, and the class' creation-type-scheme. The method-lookup-table is an analysis-time analogy of the *virtual-method-table* used by object-oriented languages to perform dynamic method dispatch at run-time. In our system, every method-lookup-table is represented as an array of type schemes, and every instance method is identified by an integer that serves as an index to the array. Thus fetching a type scheme for an instance method during closure computation is efficiently implemented. Besides the class identifier δ, every object type also contains an array of types, and every instance field of the class is identified by an integer which servers as an index to the array. Thus fetching the type for an instance field during closure computation is also efficiently implemented. Due to the nature of inheritance in Java, the integer identifier for an instance method or an instance field needs not to be changed upon inheritance.

Object creations are analyzed as follows. As we have discussed before, an expression `new C()` is analyzed with constraints $\tau_1 <: t \rightarrow [\![\texttt{new C()}]\!]$, and $\mathbf{null} <: t$, where τ_1 is the creation-type-scheme of class `C`. When the object creation invokes a constructor, constraints for invoking the constructor are also generated. Each constructor is analyzed as if it is a static method with an extra argument for "this", thus the type scheme for every constructor is also independent of the meaning of "this". This is important since a constructor of a parent class can also be called by a constructor of a subclass with "this" refer to an instance of the subclass.

Java interface features are analyzed as follows. Just like every Java class, every Java interface corresponds to an interface identifier, which is implemented as an interface object. An interface object contains references to the interface objects for

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

Arrays are special objects in Java. Thus, 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 downcast 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 (for methods, constructors and object creations) 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 generated. Thus, during the closure computation, 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 becoming a lower bound of **exception**, if τ corresponds to an exception class, 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`. With this approach, exception objects are not analyzed as accurately as other objects. But, exception objects are usually used very simply, and from the benchmark programs studied we have not seen a need for a more complex handling of exception types.

Since our type inference is a whole program analysis, reachable library code from the program must also be included in program analysis. We use the Sun JDK library source code. Native library methods were manually replaced with type-compatible Java code. For example, the `clone` method in `Object` is analyzed as: `static Object clone(Object x) { return x;}`. When analyzing a Java program, the system loads the reachable Java source code lazily.

The reflection features of Java pose a significant difficulty to any static analysis. For example, it is impossible for a static analysis to determine precisely which class is dynamically loaded by an expression `Class.forName(x)`. Our *ad hoc* solution for analyzing code with reflection is to manually replace code using reflection with type-

compatible code without reflection. For example, `Class.forName(x).newInstance()` can be replaced with an expression `true ? (Object)new A() : (Object)new B()` if it is certain that either class `A` or class `B` is loaded. Such a replacement is always possible, since we require that the source code for all reachable classes is available and there are only finitely many classes.

7.2 Constraint Representation

Constraint representation is important for the efficiency and scalability of the analysis. In our system, constraints are represented in a form derived from the *constraint map* representation defined by Trifonov and Smith [1996]. Our constraint representation applies a well-known idea that a closed constraint set may contain much redundant information. For example, after applying the (Read) closure rule on constraint $\mathbf{obj}(\delta, [\, l : u, \ldots \,]) <: [\, l : \mathbf{read}\ \tau \,]$, the constraint $u <: \tau$ is generated, and the original constraint $\mathbf{obj}(\delta, [\, l : u, \ldots \,]) <: [\, l : \mathbf{read}\ \tau \,]$ can be removed since it contains no more useful information than the constraint $u <: \tau$.

In general, a constraint $\tau_1 <: \tau_2$ ($\tau_1 \notin \mathbf{TypeVar}$ and $\tau_2 \notin \mathbf{TypeVar}$) is removed once the applicable closure rules have been applied on it. The only exception is application constraints of the form $(\forall\ \overline{t}.\ (t_1 \times \ldots \times t_n) \rightarrow \tau \setminus C) <: (t'_1 \times \ldots \times t'_n) \rightarrow \tau'$. Even if the (∀-Elim) closure rule has been applied on the application constraint, there could be constraints of form $\tau v_i <: t'_i$ appearing later in the closure computation and those constraints would cause the (∀-Elim) rule to be applied on the application constraint again. Thus the application constraint is still needed as long as type variables t'_i are needed, and we store the application constraint as a reachable component of type variables t'_i. Besides the application constraints and constraints of form $\tau_1 <: \tau_2$ ($\tau_1 \notin \mathbf{TypeVar}$ and $\tau_2 \notin \mathbf{TypeVar}$), there are only two forms of constraints: $\tau_1 <: t$ and $t <: \tau_2$. Thus, during the closure computation, it suffices to only store application constraints and constraints of form: $\tau_1 <: t$ and $t <: \tau_2$, All other constraints are not needed for future closure computation and there is no need to store them.

Based on the above ideas, our system represents the constraint closure set via a set of type variables. Type variables themselves are implemented as follows:

```
class TypeVar extends Type {
  HashMap lb;
  HashMap ub;
  ReachableApplication ra;
  ...
}
```

The above code fragment shows the key components of the representation of a type variable: the field `lb` is the set (implemented with a hashtable) of value type lower-bounds of the type variable; the field `ub` is the set (implemented with a hashtable) of type upper-bounds of the type variable; and the field `ra` is a linked list of all reachable application constraints using this type variable as a formal argument.

Such a constraint representation enables a simple form of automatic garbage collection over unreachable constraints: many constraints which are not needed for future closure computation become unreachable in the constraint representation,

and such constraints will be garbage-collected automatically by the Java run-time garbage collector.

The system also incorporates other optimizations on constraint representation and closure computation: the closure rule (Trans) only propagates value type lower bounds in the forward direction and transitivity is only through type variables; for any constraint of form $t_1 <: t_2$, we only store the constraint once by storing t_2 in the upper-bound set `ub` of t_1, and t_1 is not stored in the lower-bound set `lb` of t_2. Those optimizations reduce the memory consumption of the analysis and simplify the closure computation.

Our implementation of constraints is not as sophisticated as existing optimization techniques for constraint systems [Eifrig et al. 1995a; Pottier 1996; Pottier 1998; Flanagan and Felleisen 1997; Flanagan 1997; Fähndrich 1999; Su et al. 2000]. Those techniques can be applied to our system to further improve the performance.

7.3 Optimizations

We now discuss other optimizations implemented in our system to improve the system performance and ensure the feasibility of the system for analyzing realistic Java applications.

7.3.1 *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 that do not have subclasses and only have fields capable of storing values of monomorphic types. 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. Since we do not generate a type variable as type for such a variable or expression, there is no type propagation and other closure computation effort for analyzing the variable or expression.

7.3.2 *Additional Contour Sharing.* There are several additional strategies incorporated in the system for sharing more contours than the CPA and DCPA algorithms we have defined so far.

One issue is to prevent the algorithm from creating too many contours on certain pathological cases. Agesen [1996] defines the notion of *megamorphism*, which means that too many different value types flow to a single call site as arguments. Consider an example studied by Agesen [1996]. If a program contains a message invocation with a receiver and 18 formal arguments, and there are three value types flow to both the receiver and formal argument positions, then CPA would generate $3^{19} \approx 10^9$ contours for this method. This is infeasible. To prevent CPA from blowing up, the number of contours generated for a megamorphic call site is reduced as follows. For any call site, the system counts the number of different value types reaching an argument position of a method invocation; if the number exceeds a pre-determined threshold (a small number in the range of 2 to 4), then the call site is considered megamorphic and the number of contours generated for this call site is reduced. This idea is also employed in our system.

For some programs, DCPA may also blow up when too many contours are judged as CPA-unsafe. The Java program in Figure 9 is an example. At run-time the program will generate $2^{10} - 1 \approx 10^3$ instances of class `Pair`. When DCPA is applied

```
class Pair {
  Object a, b;
  Pair(Object obj1, Object obj2) {
    a=obj1; b=obj2;
  }
  public static void main(String args[]) {
    Object o=f1();
  }

  static Object f1() {
    return new Pair(f2(), f2());
  }

  static Object f2() {
    return new Pair(f3(), f3());
  }
  ...
  static Object f10() {
    return new Pair(new java.util.Vector(), new java.util.Vector());
  }
}
```

Fig. 9. Java Program with Exponential Behavior

to this program, each method of `f1, f2, ..., f10` creates a pair and returns it, thus every contour of those methods are CPA-unsafe. This means that there will be about 10^3 creation points, and about 10^3 distinct object types generated for class `Pair`. Furthermore, it is rare for a program to have 10^3 different kinds of type behaviors for objects from the same class. Thus, generating such a huge number of contours is unnecessary. To prevent DCPA from generating too many contours in this case, the system always regards a contour as CPA-safe (reusable) when too many object types are created locally by the contour. With such a mechanism, the contour for method `f2` would be judged as CPA-safe and shared by the two invocations of `f2`, and so is the contour for method `f3, f4, ....` Thus the exponential behavior of DCPA on this program is avoided.

The system has incorporated another optimization when generating CPA and DCPA contours: If an argument is never used in the method body, the argument is ignored when CPA and DCPA make contour selections.

7.3.3 *Online Cycle Elimination.* Another optimization incorporated in our system is the partial online cycle elimination [Fähndrich et al. 1998]. 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 [Fähndrich et al. 1998], 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 that 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.

7.4 Flow Sensitivity

A constraint-based type inference is a flow-insensitive analysis. It ignores the order of statements and expressions in the program and computes type information that is valid for the whole program rather than particular program points. This reduces the complexity of the analysis. Yet a more precise analysis could be obtained if the analysis takes the order of statements and expressions into account.

We have incorporated a limited form of flow-sensitivity into our system for accurately analyzing a common Java programming idiom:

```
if (x instanceof C) {  C a = (C) x;  ...  }
```

In the above program fragment, at the entry point of the true branch, it is certain that values of variable `x` are instances of class `C`. Thus the downcast `(C)x` always succeeds at run-time. Our system uses a simple algorithm to analyze the condition expression of every `if` statement specially. It conservatively estimates whether conditions of form `x instanceof C` would definitely hold or definitely not hold at the entry point of every true or false branch of an `if` statement, and if so the system would use that fact in the analysis of the true or false branch to achieve a better analysis result. This special processing is integrated into the type inference pass, and it is uniformly applied in our implementations of 0CFA, CPA and DCPA algorithms.

8. EVALUATION

In this section, we present the experiment results of our type inference system on several benchmark programs. The goal of the experiment is to test the feasibility of the system for analyzing realistic Java applications, and to compare the DCPA algorithm with algorithms 0CFA and CPA in terms of precision and efficiency.

8.1 Benchmark Programs

We have used several realistic Java applications in our experiment. Information about the benchmark programs is presented in Table I. The following benchmark programs have been used: *jlex*[2] is a lexical analyzer generator; *toba*[3] is a Java-to-C code translator; *javacup*[4] is a Java parser generator; *jtar*[5] is an archive utility; *bloat*[6] is a Java bytecode optimizer; *self* is our system itself used as a benchmark; *sablecc*[7] is a compiler generator; *javac* and *javadoc* are standard tools in Sun's Java SDK.

In Table I, the column "lines" shows the number of lines of source code in the benchmark program only. The numbers for *javac* and *javadoc* are not available since there is some code reachable from both the two applications and the Java standard library implementations, and so it is difficult to have an accurate division between user code and libraray code. Since our system is a whole-program analysis,

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

Program	lines	methods	casts
jlex	7835	398	65
toba	6417	777	63
javacup	10592	532	459
jtar	11904	1446	10
bloat	18841	1053	205
self	23122	1304	130
javadoc	–	2314	310
sablecc	23111	2811	519
javac	–	2933	606

Table I. Benchmark Programs

	0CFA		CPA			DCPA		
Program	safe	time	safe	time	Θ	safe	time	Θ
jlex	10.8%	3.3	16.9%	3.5	1.5	100%	3.8	2.3
toba	4.8%	4.8	4.8%	5.1	1.6	22.2%	6.4	2.7
javacup	8.1%	3.6	8.1%	3.9	1.4	89.3%	4.5	3.6
jtar	0%	6.7	0%	7.1	1.5	100%	11.0	2.4
bloat	7.8%	5.3	8.8%	5.9	1.5	30.2%	7.1	4.3
self	36.9%	4.5	51.5%	5.9	1.9	93.8%	10.7	3.4
javadoc	25.3%	10.1	40.8%	13.0	1.9	77.7%	23.6	4.3
sablecc	34.5%	10.9	35.2%	10.9	2.5	61.7%	21.3	4.1
javac	17.1%	12.5	30.5%	30.0	2.5	50.1%	74.8	7.1

Table II. Experiment Results

every benchmark program is analyzed along with the reachable library code. The column "methods" shows the number of reachable methods in the whole program including libraries. The number of reachable methods is the one detected by the 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.

8.2 Experimental Results

All programs are tested with three algorithms: 0CFA, CPA and DCPA. Table II presents the experimental results.

The columns labeled "safe" indicate the percentage of 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 866MHZ Pentium processor and 512M of memory. All benchmarks except *javac* were analyzed with 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

```
import java.io.*;
class Dynamic {
    int kind;
    Object value;
    Dynamic(int k, Object v) { kind=k; value=v;}
    public static void main(String args[]) throws Exception {
        FileOutputStream out = new FileOutputStream("Hopkins");
        ObjectOutputStream s1 = new ObjectOutputStream(out);
        s1.writeObject("Type Inference");
        s1.writeObject(new Boolean(true));
        s1.flush(); out.close();

        FileInputStream in = new FileInputStream("Hopkins");
        ObjectInputStream s2 = new ObjectInputStream(in);
        String str = (String)s2.readObject(); in.close();

        p(new Dynamic(1, new Integer(0)));
        p(new Dynamic(2, new Boolean(true)));
    }
    static void p(Dynamic d) {
        switch(d.kind) {
        case 1: Integer i=(Integer)d.value; break;
        case 2: Boolean b=(Boolean)d.value; break;
        }
    }
}
```

Fig. 10. Java program with dynamic down-casts

downcasts that 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.

The program in Figure 10 shows typical examples of the fundamentally dynamic downcasts appearing in Java programs. The first case is related to input/output operations. In the program, though it is the programmer's intention that the first object read from the file named `Hopkins` is a `String` object, it is usually impossible for a static analysis to discover that. Thus the downcast `(String)s2.readObject()` is fundamentally dynamic. Another case is about a common programming idiom: there is a class with a polymorphic field which can store values of different types, and the class has another integer-valued field which records the type of the value stored in the polymorphic field. The class `Dynamic` is such a class: the field `value`

```
import java.util.*;
class Flow {
    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));

        Enumeration e;
        for (e=ht1.elements(); e.hasMoreElements();) {
            Integer i=(Integer)e.nextElement();
        }
        for (e=ht2.elements(); e.hasMoreElements();) {
            Boolean b=(Boolean)e.nextElement();
        }
    }
}
```

Fig. 11. Java program for flow-sensitivity

stores an `Integer` object if and only if the value of field `kind` equals to 1. In the example, since it is usually impossible for a static analysis to statically determine the exact value of expression `d.kind`, the two downcasts `(Integer)d.value` and `(Boolean)d.value` are fundamentally dynamic. Type annotations may be used to guide the analysis of those dynamic cases.

Another reason that some downcasts cannot be statically verified is because our analysis is flow-insensitive. Consider the program in Figure 11. The variable `e` is used as an iterator over the values of the hashtable `ht1` and hashtable `ht2` at the different points of the program. Since our analysis is flow-insensitive, it cannot determine that, when the control flow reaches the downcast `(Integer)e.nextElement()`, the variable `e` stores only the iterator for `ht1`; and, when the control flow reaches the downcast `(Boolean)e.nextElement()`, the variable `e` stores only the iterator for `ht2`. Thus, the two downcasts cannot be verified by our system.

As we have discussed, some special form of flow-sensitivity is incorporated into our system for accurately analyzing a common Java programming idiom:

```
if (x instanceof C) {  C a = (C) x;  ... }
```

But, sometimes the simple algorithm for adding such a special form of flow-sensitivity is not powerful enough. For example, consider following java program fragment:

```
if (! x instanceof C) return false;
C a = (C) x; ...
```

The downcast always succeeds at run-time, yet our system fails to verify it.

One possible approach for adding flow-sensitivity to a constraint-based type inference is to use the Static Single Assignment (SSA) transformation [Cytron et al. 1991; McConnell and Johnson 1992]. SSA transformation puts the program into a form where every variable is assigned at only one program point. With the program transformed into a SSA form, some flow-insensitivity is incorporated to the constraint-based type inference and the downcasts in Figure 11 could be verified.

In summary, DCPA appears to produce nearly optimal results as a flow-insensitive static analysis for downcast checking on the benchmark programs we have tested. There are downcasts in the benchmark programs which DCPA fails to verify, but nearly all of them cannot even be verified by an analysis which would use a fresh contour for every function application. DCPA is also implemented efficiently: it successfully analyzes all of the benchmark programs that are all realistic Java applications; comparing the time and the average number of contours of CPA and DCPA, we can see that the efficiency of DCPA is comparable to CPA.

9. RELATED WORK

There have been several frameworks developed for polyvariant flow analyses, in terms of union and intersection types [Palsberg and Pavlopoulou 1998], abstract interpretation [Nielson and Nielson 1997], flow graphs [Jagannathan and Weeks 1995], and more implementation-centric [Grove et al. 1997]. In particular, Palsberg and Pavlopoulou [1998] develop an elegant framework for polyvariant flow analyses in a type system with union/intersection types and subtyping.

Let-polymorphism is the classic form of polymorphism used in subtype-free type inference, and it has been adapted to constrained types in [Aiken and Wimmers 1993; Eifrig et al. 1995b], as well as directly in the flow analysis setting by Wright and Jagannathan [1998]. Another representation of polymorphism found in subtype-free languages is via rank-2 intersection types [Jim 1996], which has also been applied to polyvariant flow analysis [Bannerjee 1997]. The Church group has developed type systems of union and intersection types decorated with flow labels to indicate the flow information [Wells et al. 1997].

The framework presented in this paper is the first proposal to use polymorphic constrained type schemes to model polyvariant flow analyses. There are two reasonable approaches to model polyvariant analyses with constraint-based type systems: a monomorphic constrained type theory may be extended with union and intersection types, or universally quantified constrained type schemes may be used. We take the latter approach due to its closeness to the existing implementations of polyvariant analyses: re-analysis of a function definition is closely analogous to a new instantiation of a type scheme. There also are implementation advantages obtained by basing analyses on polymorphic constrained types. Compared to the flow graph based approach used in other implementations of flow analyses [Agesen 1996; Grove et al. 1997; Plevyak and Chien 1994], our framework has several advantages: re-analysis of a function in a different polyvariant context is also realized by instantiation of the function's constrained type scheme, and does not require reanalysis of the function body; existing optimization techniques for constraint systems [Eifrig et al. 1995a; Pottier 1996; Pottier 1998; Flanagan and Felleisen 1997; Flanagan 1997; Fähndrich 1999; Su et al. 2000] can be applied, leading to more efficient implementation of the analyses. For above reasons, previous frameworks for polyvariant flow analyses are not suitable for our purpose; and we develop a new framework to model CFA, CPA and the new DCPA algorithm.

Type systems with constraints have been extensively studied [Mitchell 1984; Reynolds 1985; Fuh and Mishra 1988; Kozen et al. 1992]. Aiken and Wimmers [1993] develop a type system with type inclusion constraints and type inference.

Heintze constructs a set-based analysis of ML programs [Heintze 1994]. Pottier [1996; 1998] develops a type inference framework with subtyping constraints.

Aiken et al. [1998] develop a toolkit for constraint-based program analysis. They develop various effective techniques for optimizing the implementation of constraint systems, including partial online cycle elimination [Fähndrich et al. 1998] and projection merging [Su et al. 2000].

Flanagan and Felleisen [1997;1997] develop a componential set-based analysis. The analysis is used in MrSpidey [Flanagan et al. 1996], a static debugger for Scheme programs. Componential set-based analysis uses two effective techniques: a constraint simplification algorithm that can effectively reduce the size of constraint system; and a methodology that allows analyzing the whole program by combining constraints obtained for multiple program components. Such techniques can also be applied in our system for further improvement of analysis efficiency.

Compared to those constraint-based analyses, which are either monomorphic or let-polymorphic, our system takes a different and flow-based approach to handle polymorphism, and can effectively analyze data-polymorphic programs.

Mossin [Mossin 1997] develops a flow analysis with a polymorphic constraint system for typed programs. Similar to our system, instantiating constrained type schemes is realized by constraint copying. Rehof and Fähndrich [2001] develop a type-based flow analysis with polymorphic subtyping. Using instantiation constraints, the analysis computes context-sensitive flow information by applying context-free language reachability techniques, and it obviates the need of constraint copying. In comparison, our system uses constraint-copying method and focuses on reducing the number of instantiations (contours) generated for type schemes.

Type inference for object-oriented programming has been extensively studied [Suzuki 1981; Borning and Ingalls 1982; Johnson 1986; Graver and Johnson 1990; Vitek et al. 1992]. Palsberg, Schwartzbach and Oxhøj [1991;1992;1994] have pioneered the use of constraint-based approach for analyzing object-oriented languages.

Agesen [1996] develops the CPA algorithm and has implemented it as a type inference system for object-oriented language Self. The type inference system is also used for constructing an application extractor that extracts the reachable code of a Self application from the Self programming environment. The development of our polymorphic type inference framework and the Data-Polymorphic CPA (DCPA) algorithm is inspired by Agesen's work on CPA algorithm. Our DCPA algorithm extends CPA with the ability to effectively analyze data-polymorphic programs.

Plevyak and Chien's iterative flow analysis (IFA) [1994; 1996] is a precise constraint-based analysis of object-oriented programs. Like our DCPA algorithm, IFA aims to achieve precise type inference result in the presence of parametric polymorphism and data polymorphism. To our knowledge, IFA is the only existing system in the literature capable of analyzing data-polymorphic programs precisely. IFA analyzes programs with an iterative approach. Each iteration pass is a whole program analysis, which refines the analysis result obtained from the last pass. The iteration continues until no more refinement can be achieved. Compared to IFA, the DCPA algorithm detects data polymorphism online, and does not need generational iteration. Another advantage of the DCPA algorithm is that it is significantly less complex than IFA.

Grove et al. [1997; 1998] have implemented a framework for analyzing object-

oriented languages, and have implemented a family of analyses, including nCFA and CPA. The analyses have been used for call-graph construction and interprocedural optimizations. They report that substantial speed-ups have been achieved with the interprocedural analyses and optimizations.

Duggan [1999] has proposed a system to automatically detect polymorphic Java classes. It is currently not implemented or tested with benchmarks and so its feasibility and performance are unclear. Recently Donovan et al. [2004] have developed a system for automatically converting Java programs to use generic libraries. The system is based on a constraint-based analysis using our polymorphic constraint framework presented in this paper.

Rountev et al. [2001] develop a points-to analysis for Java using annotated constraints. Whaley and Lam [2002] develop an inclusion-based points-to analysis for Java. Those systems have achieved good efficiency and scalability on large sets of Java programs. While those systems handle the polymorphism in the similar way as the 1CFA algorithm, our system achieves type inference precision beyond 1CFA and CPA.

O'Callahan [2001] 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.

As with the CPA and IFA algorithms, the DCPA algorithm aims to improve the analysis precision as much as possible in the presence of polymorphism. There is another family of analyses for object-oriented languages, which aim to be as efficient as possible. Such algorithms emphasize the efficiency of the analysis over the analysis precision.

Dean et al. [1995] develop the *Class Hierarchy Analysis*, which is a simple and fast analysis based on the class hierarchy of the program. Bacon and Sweeney [1996; 1997] develop *Rapid Type Analysis*, which is a refinement of the Class Hierarchy Analysis. The analysis has been used for supporting various compiler optimizations of C++ programs, including static resolution of virtual function calls. Diwan et al. [1996; 1996; 1998] develop several simple and effective analyses for statically-typed object-oriented languages. The analyses have been used by a whole-program optimizer for Modula-3 programs. Tip and Palsberg [2000] develop a spectrum of constraint-based analyses for Java. Those analyses are less precise yet more efficient than 0CFA. The analyses are used for call-graph construction. Sundaresan et al. [2000] develop several analyses for Java. Those analyses are also less precise yet more efficient than 0CFA. The analyses are used for resolving virtual method calls in Java programs.

Those fast analyses are useful because for many purposes it has become clear that a more precise 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.

10. CONCLUSION

This paper presents following results:

—A generic framework for polymorphic constraint-based type inference is developed, and the soundness of the framework is proved. The framework can be used for developing new type inference algorithms and for establishing the soundness of constraint-based program analyses. The framework is also extended with the ability to analyze object-oriented languages.

—We define Shivers' nCFA and Agesen's CPA as instantiations of the framework, and hence establish the soundness of those algorithms.

—We develop a novel algorithm, Data-Polymorphic DCPA, which extends the CPA with the ability of analyzing data-polymorphic programs effectively. Experiments show that DCPA is significantly more precise than 0CFA and CPA in terms of downcast verification, and the efficiency of DCPA is comparable to CPA.

—We implement a constraint-based type inference system for Java. The system includes implementations of algorithms 0CFA, CPA and DCPA, and it uses a series of novel implementation optimizations that are essential to the performance of the system. The system is essentially a concrete class analysis tool, and it can be used for various important applications. A simple demonstration of our system is available at: `http://www.cs.jhu.edu/~wtj/precise`.

REFERENCES

Agesen, O. 1995. The cartesian product algorithm. In *Proceedings ECOOP'95*. Lecture notes in Computer Science, vol. 952. Springer-Verlag.

Agesen, O. 1996. Concrete type inference: Delivering object-oriented applications. Ph.D. thesis, Stanford University. Available as Sun Labs Technical Report SMLI TR-96-52.

Aiken, A., Fähndrich, M., Foster, J. S., and Su, Z. 1998. A toolkit for constructing type- and constraint-based program analyses. In *Second International Workshop on Types in Compilation (TIC'98)*. Kyoto, Japan.

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

Bacon, D. F. 1997. Fast and effective optimization of statically typed object-oriented languages. Ph.D. thesis, University of California at Berkeley.

Bacon, D. F. and Sweeney, P. F. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*. San Jose, California, 324–341.

Bannerjee, A. 1997. A modular, polyvariant, and type-based closure analysis. In *International Conference on Functional Programming*.

Borning, A. H. and Ingalls, D. H. H. 1982. A type declaration and inference system for smalltalk. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*. Albuquerque, New Mexico.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (October), 451–490.

Dean, J., Grove, D., and Chambers, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*. 77–101.

Diwan, A. 1996. Understanding and improving the performance of modern programming languages. Ph.D. thesis, University of Massachusetts.

Diwan, A., McKinley, K. S., and Moss, J. E. B. 1998. Type-based alias analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. ACM SIGPLAN Notices 33(5). 106–117.

Diwan, A., Moss, J. E. B., and McKinley, K. S. 1996. Simple and effective analysis of statically-typed object-oriented programs. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*. San Jose, California, 292–305.

Donovan, A., Kie. zun, A., Tschantz, M. S., and Ernst, M. D. 2004. Converting Java programs to use generic libraries. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*. Vancouver, BC, Canada, 15–34.

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

Eifrig, J., Smith, S., and Trifonov, V. 1995a. Sound polymorphic type inference for objects. In *OOPSLA '95*. 169–184.

Eifrig, J., Smith, S., and Trifonov, V. 1995b. Type inference for recursively constrained types and its application to OOP. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*. Electronic Notes in Theoretical Computer Science, vol. 1. Elsevier. http://www.elsevier.nl/locate/entcs/volume1.html.

Fähndrich, M. 1999. Bane: A library for scalable constraint-based program analysis. Ph.D. thesis, University of California at Berkeley.

Fähndrich, M., Foster, J. S., Su, Z., and Aiken, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Flanagan, C. 1997. Effective static debugging via componential set-based analysis. Ph.D. thesis, Rice University.

Flanagan, C. and Felleisen, M. 1997. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*. ACM SIGPLAN Notices, vol. 32, 5. ACM Press, New York, 235–248.

Flanagan, C., Flatt, M., Krishnamurthi, S., Weirich, S., and Felleisen, M. 1996. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 23–32.

Fuh, Y.-C. and Mishra, P. 1988. Type inference with subtypes. In *European Symposium on Programming*.

Graver, J. O. and Johnson, R. E. 1990. A type system for Smalltalk. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*. 136–150.

Grove, D. 1998. Effective interprocedural optimization of object-oriented languages. Ph.D. thesis, University of Washington.

Grove, D., DeFouw, G., Dean, J., and Chambers, C. 1997. Call graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

Heintze, N. 1994. Set based analysis of ML programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming*. 306–317.

Igarashi, A., Pierce, B., and Wadler, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems 23,* 3 (May).

Jagannathan, S. and Weeks, S. 1995. A unified treatment of flow analysis in higher-order language s. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*. 393–408.

Jim, T. 1996. What are principal typings and what are they good for? In *Conference Record of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*.

Johnson, R. E. 1986. Type-checking smalltalk. In *OOPSLA'86 Object-Oriented Programming, Systems, Languages, and Applications*. Portland, Oregon, 315–321.

Kozen, D., Palsberg, J., and Schwartzbach, M. I. 1992. Efficient inference of partial types. In *Foundations of Computer Science*.

McConnell, C. and Johnson, R. E. 1992. Using static single assignment form in a code optimizer. *ACM Letters on Programming Languages and Systems 1,* 2 (June), 152–160.

Mitchell, J. C. 1984. Coercion and type inference (summary). In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*.

Mossin, C. 1997. Flow analysis of typed higher-order programs. Ph.D. thesis, DIKU, University of Copenhagen.

Nielson, F. and Nielson, H. R. 1997. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Conference Record of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*. 332–345.

O'Callahan, R. 2001. Generalized aliasing as a basis for program analysis tools. Ph.D. thesis, Carnegie-Mellon University.

Oxhøj, N., Palsberg, J., and Schwartzbach, M. I. 1992. Type inference with subtypes. In *ECOOP'92 European Conference on Object-Oriented Programming*. Lecture notes in Computer Science, vol. 615. Springer-Verlag, 329–349.

Palsberg, J. and O'Keefe, P. 1995. A type system equivalent to flow analysis. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*. 367–378.

Palsberg, J. and Pavlopoulou, C. 1998. From polyvariant flow information to intersection and union types. In *Conference Record of POPL'98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. San Diego, California, 197–208.

Palsberg, J. and Schwartzbach, M. 1994. *Object-Oriented Type Systems*. Wiley.

Palsberg, J. and Schwartzbach, M. I. 1991. Object-oriented type inference. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 146–161.

Plevyak, J. 1996. Optimization of object-oriented and concurrent programs. Ph.D. thesis, University of Illinois at Urbana-Champaign.

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

Pottier, F. 1996. Simplifying subtyping constraints. In *First International Conference on Functional Programming*. 122–133.

Pottier, F. 1998. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. Baltimore, Maryland, 228–238.

Rehof, J. and Fähndrich, M. 2001. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *Conference Record of POPL'01: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. London, United Kingdom, 54–66.

Reynolds, J. C. 1985. Three approaches to type structure. In *TAPSOFT proceedings*. Lecture notes in Computer Science, vol. 185. 97–138.

Rountev, A., Milanova, A., and Ryder, B. G. 2001. Points-to analysis for java using annotated constraints. In *ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA 2001)*.

Shivers, O. 1988. Control flow analysis in Scheme. In *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*.

Shivers, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie-Mellon University. Available as CMU Technical Report CMU-CS-91-145.

Smith, S. and Wang, T. 2000. Polyvariant flow analysis with constrained types. In *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, G. Smolka, Ed. Lecture Notes in Computer Science, vol. 1782. Springer Verlag, 382–396.

Su, Z., Fähndrich, M., and Aiken, A. 2000. Projection merging: Reducing redundancies in inclusion constraint graphs. In *the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*. Boston, MA.

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

Suzuki, N. 1981. Inferring types in smalltalk. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*. Williamsburg, Virginia, 187–199.

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

Trifonov, V. and Smith, S. 1996. Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*. Lecture notes in Computer Science, vol. 1145. Springer-Verlag, 349–365.

Vitek, J., Horspool, R. N., and Uhl, J. S. 1992. Compile-time analysis of object-oriented programs. In *Proceedings of CC'92, 4th International Conference on Compiler Construction*. Lecture notes in Computer Science, vol. 641. Springer-Verlag, 236–250.

Wang, T. and Smith, S. F. 2001. Precise constraint-based type inference for Java. In *European Conference on Object-Oriented Programming(ECOOP'01)*. Budapest, Hungary.

Wells, J. B., Dimock, A., Muller, R., and Turbak, F. 1997. A typed intermediate language for flow-directed compilation. In *Theory and Practice of Software Development (TAPSOFT)*. Number 1214 in Lecture notes in Computer Science. Springer-Verlag.

Whaley, J. and Lam, M. S. 2002. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*. 180–195.

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Information and Computation 115(1)*, 38–94.

Wright, A. K. and Jagannathan, S. 1998. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems 20,* 1 (Jan.), 166–207.

Received November 2002;