CPA beats oo-CFA

Frédéric Besson

CPA beats ∞-CFA

Frédéric Besson*
Inria, Centre Rennes - Bretagne Atlantique
Campus universitaire de Beaulieu
FR-35042 Rennes cedex
frederic.besson@inria.fr

ABSTRACT

Context-sensitive points-to analysis is the current most scalable technology for constructing a precise control-flow graph for large object-oriented programs. One appealing feature of this framework is that it is parametric thus allowing to trade time for precision. Typical instances of this framework are k-CFAs and Agesen's Cartesian Product Algorithm (CPA). It is common sense that k-CFAs (for increasing ks) form a hierarchy. Yet, what is the relative precision of k-CFA and CPA? Grove and Chambers [2] conjecture that CPA is more precise than ∞-CFA. For a core object-oriented language, we formally compare the precision of ∞-CFA and CPA. We prove that CPA is indeed strictly more precise than ∞-CFA. On a theoretical level, this result confirms the findings of empiric studies concluding the superiority of object-sensitivity with respect to call-string sensitivity.

1. INTRODUCTION

Context-sensitive points-to analysis is a scalable technology for constructing precise control-flow graphs of object-oriented programs. It allows for aggressive inter-procedural optimisations and improves the precision of client analyses [2, 7, 6]. The output of the analysis is a control-flow graph and a points-to graph abstracting the memory heap. For the baseline (context-insensitive) analysis, nodes in the control-flow graph are method names and nodes in the points-to graph are class names. For object-oriented programs such context insensitive graphs are far too imprecise. To improve the accuracy context-sensitive analyses account for the fact that the same method can be run in different *call-contexts* and that the same class can be instantiated in different *allocation-contexts*. For context-sensitive analyses, a node in the control-flow graph is a method name tagged by a call-context and a node in the points-to graph is a class name tagged with an allocation-context.

*This work was partly funded by the IST-FET programme of the European Commission, under the IST-2005-015905 MOBIUS project.

FTfJP '09, July 6 2009, Genova, Italy

As early as 1991, Palsberg and Schwartzbach [10, 11] propose a theoretical parametric framework for typing object-oriented programs. In their setting, context-sensitivity is obtained by explicit code duplication and typing amounts to analysing the expanded code in a context-insensitive manner. The framework accommodates for both call-contexts and allocation-contexts. In particular, it accommodates for the well-known k-CFA hierarchy abstracting the call-context by a call-string made of the last k invocation sites. For Self, a language with multiple dispatch, Agesen [1] proposes the Cartesian Product Algorithm (CPA) which abstracts the call-context by the cartesian product of the abstract objects argument of the call.

To assess the respective merits of different instantiations, scalable implementations are needed. For Cecil and Java programs, Grove *et al.,* [3, 2] have explored the algorithmic design space of contexts for benchmarks of significant size. Latter on, Milanova *et. al.,* [7] have evaluated, for Java programs, a notion of context called *object-sensitivity* which abstracts the call-context by the abstraction of the this pointer[1]. More recently, Lhotak and Hendren [6] have extended the empiric evaluation of object-sensitivity using a BDD implementation allowing to cope with benchmarks otherwise out-of-scope. The findings of the more recent empiric studies are consistent and conclude that *object-sensitivity* performs better than *call-string* sensitivity – for the Java benchmarks considered.

Hind [4] discusses the difficulty of assessing the merits of a pointer analysis: it depends on various factors including the language, benchmarks and client analyses. Yet, from a theoretical standpoint, there are known facts about the respective precision of context-sensitive points-to analyses. Grove and Chambers [2, Figure 19] picture a comprehensive map of context-sensitive analyses ordered by precision of the computed graph. In this map there are analyses for which the relative precision is only conjectured. In particular, CPA is conjectured to be (strictly) more precise than ∞-CFA.

The main contribution of this paper is a formal proof that CPA is strictly more precise than ∞-CFA. Other contributions include the definition of a minimal object-oriented calculus modeling standard dispatch mechanisms in a uniform manner; and a generic soundness proof of context-sensitive points-to analyses. Moreover, the formal development is mechanised inside the Coq proof assistant[2]

[1]At level 1, object-sensitivity is a trimmed down version of the CPA.

[2]The commented Coq source code can be browsed at http://www.irisa.fr/celtique/fbesson/CPABeatsCFA

The rest of the paper is organised as follows. Notations are gathered in Section 2. In Section 3, we introduce the syntax and semantics of our core object-oriented language. In Section 4, we present the specification and generic soundness proof of context-sensitive analyses. In Section 5, we formally show how to relate different analyses and prove that CPA is strictly more precise than ∞-CFA. Section 6 concludes.

2. NOTATIONS

Given sets A and B, we write A_e for the set $A \cup \{e\}$ such that $e \notin A$ and $A \to B_\perp$ for the set of partial functions from A to B. Let $f : A \to B$, $f[x \mapsto v]$ is the function identical to f everywhere except for x for which it returns v. For a function $f : A \to B_\perp$, when we write $f(x) = y$, y is (implicitly) an element of B *i.e,* $y \neq \perp$. Tuples with k elements are written $\langle e_1, \dots e_k \rangle$. When it is clear from the context, brackets are dropped. We write A^* for the sets of lists with elements in A. The empty list is written ϵ and $e::l$ is the list whose head is e and tail is l. The null pointer, written 0, is always distinct from any other addresses $Addr$. Therefore, we shall write $Addr_0$ for the set of addresses extended with the null pointer. We write α for an arbitrary address distinct from 0.

3. A CORE O-O LANGUAGE

In this section we define a minimal core language for reasoning about the inter-procedural control-flow of imperative object-oriented languages. Unlike *e.g.,* Featherweight Java [5], our language is not about high-level object-oriented concepts such as classes and inheritance. Even methods are absent and a program is simply given by a list of statements and a dynamic method lookup function.

3.1 Syntax and semantics

We consider (finite) sets of class names $Class$, method names $Meth$, field names $Field$ and variables names Var ($\{v_0, v_1\} \subseteq Var$). A program $p \in Prog$ is a list of statements together with a lookup function.

$$\begin{array}{lcl} S & \triangleq & copy(v, v') \mid new(c) \mid read(f) \mid write(f) \\ & & \mid call(m) \mid ret \\ Lkup & \triangleq & (Var \to Class) \to Meth \to \mathbb{N} \\ Prog & \triangleq & S^* \times Lkup \end{array}$$

where $v, v' \in Var$, $c \in Class$, $f \in Field$, $m \in Meth$.

Our statements are a restricted form of 3 address code. To give them an intuitive meaning, the syntactic translation between these formats is given below.

$$\begin{array}{llll} copy(v, v') & \rightsquigarrow & v := v' & new(c) \rightsquigarrow v_0 := new(c) \\ read(f) & \rightsquigarrow & v_0 := v_0.f & write(f) \rightsquigarrow v_0.f := v_1 \end{array}$$

Because the lookup function has type $(Var \to Class) \to Meth \to \mathbb{N}$, it models uniformly different kinds of dispatch. In general, we get multiple-dispatch for which the target of the call depends on all of the arguments of the caller. Single dispatch is obtained if only the receiver, say v_0, selects the target of the call. Function pointers are degenerated cases of single dispatch for which $Class = \mathbb{N}$ and $Meth = \{\bullet\}$. A static method call is a further specialisation for which only the method name m is used to resolve the call.

The language is equipped with a standard small-steps operational semantics. A program state is a pair $\langle h, \langle pc, e \rangle :: f \rangle \in Heap \times Frame^+$ where 1) $pc \in \mathbb{N}$ is the current program counter *i.e.,* the index in the program of the instruction to be executed next; 2) $e \in Env$ is a local environment mapping variables to addresses; 3) $h \in Heap$ maps allocated addresses to objects; 4) $f \in Frame^*$ is a stack of frames recording caller contexts. Formal definitions of the semantics domains are given below:

$$\begin{array}{lcl} Env & \triangleq & Var \to Addr_0 \\ Frame & \triangleq & \mathbb{N} \times Env \\ Object & \triangleq & Class \times (Field \to Addr_0) \\ Heap & \triangleq & Addr \to Object_\perp \\ State & \triangleq & Heap \times Frame^+ \end{array}$$

We consider an uninterpreted infinite set of addresses $Addr$. An object is a pair $\langle c, o \rangle \in Object$ where c is a class name and o is a mapping from fields to addresses. The heap is a partial mapping from addresses to objects.

The definition of the transition relation $\rhd \subseteq State \times State$ is in Figure 1. The function $nth : \mathbb{N} \to S_\perp$ is such that $nth(pc)$ is either the pc^{th} statement of the program or $\perp$ if the index is out-of-bounds. Therefore, the execution blocks when the program counter pc is indexing a non-existent statement. The statements *copy*, *new*, *read* and *write* are intra-procedural and do not modify the stack of frames. After executing an intra-procedural statement, execution continues in sequence from the next instruction. The $new(c)$ statement picks in the heap an unallocated address α and allocates at that address an object of class c whose fields are all null. The statements *read* and *write* require addresses to be non-null. In our model, the de-reference of null pointers is thus blocking. To resolve a method call $call(m)$, the dispatch dp first computes a (finite) function mapping variables to the class name of the object they point to – the null pointer being mapped to the undefined class udf. Then, the lookup function is called with as argument theses classes and the method name m. It returns the index of the first statement of the callee.

The semantics starts from an undefined heap $h_0 = \lambda\alpha.\perp$ and an environment mapping all variables to 0 ($e_0 = \lambda v.0$). The set of reachable states Acc is the set of states accessible from the initial state $s_0 = (h_0, \langle 0, e_0 \rangle :: \epsilon)$ by the reflexive transitive closure of the relation $\rhd$.

$$Acc = \{s \mid s_0 \rhd^* s\}$$

4. POINTS-TO ANALYSIS

An instance of a context-sensitive analysis is obtained by defining a domain CC of call-contexts, a domain AC of allocation contexts – or abstract addresses – an initial call context $\bullet : CC$ and two functions *alloc* modeling abstract object creation and *push* abstracting the push operation on frames at the level of call contexts.

$$\begin{array}{lcl} alloc & : & CC \times \mathbb{N} \times Class \times (Var \to AC_0) \to AC \\ push & : & CC \times \mathbb{N} \times Meth \times (Var \to AC_0) \to CC \end{array}$$

It is worth noticing that the soundness of the analysis does not depend on the actual definitions of *alloc* and *push*. For CPA and ∞-CFA objects are abstracted by their creation site. CPA and ∞-CFA differ in their abstraction of the call context. For CPA, the call-context is either the initial call-context $\bullet$ or a pair made-of the current method being executed and its abstract arguments. For infinite CFA, the

$$\frac{nth(pc) = copy(v, v') \quad e' = e[v \mapsto e(v')]}{h, \langle pc, e\rangle{::}f \rhd h, \langle pc{+}1, e'\rangle{::}f} \qquad \frac{nth(pc) = new(c) \quad e' = e[v_0 \mapsto \alpha] \quad h(\alpha) = \bot \quad h' = h[\alpha \mapsto \langle c, \lambda f.0\rangle]}{h, \langle pc, e\rangle{::}f \rhd h', \langle pc{+}1, e'\rangle{::}f}$$

$$\frac{nth(pc) = read(f) \quad e(v_0) = \alpha \quad h(\alpha) = \langle c, o\rangle \quad e' = e[v_0 \mapsto o(f)]}{h, \langle pc, e\rangle{::}f \rhd h, \langle pc{+}1, e'\rangle{::}f} \qquad \frac{nth(pc) = write(f) \quad e(v_0) = \alpha \quad h(\alpha) = \langle c, o\rangle \quad h' = h[\alpha \mapsto \langle c, o[f \mapsto e(v_1)]\rangle]}{h, \langle pc, e\rangle{::}f \rhd h', \langle pc{+}1, e\rangle{::}f}$$

$$\frac{nth(pc) = call(m) \quad pc' = lk(dp(h, e), m)}{h, \langle pc, e\rangle{::}f \rhd h, \langle pc', e\rangle{::}\langle pc, e\rangle{::}f} \qquad \frac{nth(pc) = ret \quad er = e'[v_0 \mapsto e(v_0)]}{h, \langle pc, e\rangle{::}\langle pc', e'\rangle{::}f \rhd h, \langle pc'{+}1, er\rangle{::}f}$$

$$dp : Heap \times Env \to (Var \to Class)$$
$$dp(h, e)(v) = \begin{cases} c \text{ if } e(v) = \alpha \text{ and } h(\alpha) = (c, o) \\ udf \text{ otherwise} \end{cases}$$

Figure 1: Semantics

call-context is a call-string made of the invocation sites of the callers. As a result, we have:

$$\begin{array}{lcl} AC & = & \mathbb{N} \\ alloc(cc, pc, c, e) & = & pc \\ push_{\bowtie}(cc, pc, m, e) & = & (m, e) \\ push_{\infty}(cc, pc, m, e) & = & pc{::}cc \end{array}$$

4.1 Context-sensitive instrumentation

In order to state and prove correct context-sensitive analyses, the standard approach consists in instrumenting the semantics with an address cache [8, 9]. The address cache is used to dynamically map addresses to their allocation context.

$$Acache \triangleq Addr \to AC_0$$

We also carry along the computation the current abstract call-context. The instrumentation is not intrusive and neither enable new semantic transitions nor disable existing ones. The instrumented semantic relation is defined over an extended state

$$\begin{array}{lcl} IFrame & \triangleq & Frame \times CC \\ IState & \triangleq & Heap \times IFrame^{+} \times Acache \end{array}$$

At the intra-procedural level, all statements except object allocation keep the instrumentation unchanged. The $new(c)$ statement maps the newly allocated address to its allocation context computed by the *alloc* function. At the inter-procedural level, the address cache is threaded along the execution while call contexts follow the stack discipline. The modified transitions are given in Figure 2. For an address cache $ac : Addr \to AC_0$, we write $ac_0 : Addr_0 \to AC_0$ an extended address cache mapping null to null and $e^{ac} = \lambda v.ac_0(e(v))$ the abstraction of the environment with respect to the address cache. *IAcc* is the set of instrumented reachable states from $is_0 = \langle \lambda\alpha.\bot, \langle 0, \lambda v.0, \bullet\rangle{::}\epsilon\rangle$ by the reflexive transitive closure of the instrumented semantics relation.

4.2 Context-sensitive points-to analysis

The principle of a context-sensitive points-to analysis is to abstract addresses by their allocation context and the stack of frame by a call-context. Moreover, environments are abstracted in a set-based manner by a mapping from variables

$$\frac{nth(pc) = new(c) \quad e' = e[v_0 \mapsto \alpha] \quad h(\alpha) = \bot \quad h' = h[\alpha \mapsto \langle c, \lambda f.0\rangle] \quad ac' = ac[\alpha \to alloc(cc, pc, c, e^{ac})]}{h, \langle pc, e, cc\rangle{::}f, ac \rhd h', \langle pc{+}1, e', cc\rangle{::}f, ac'}$$

$$\frac{nth(pc) = call(m) \quad cc' = push(cc, pc, m, e^{ac}) \quad pc' = lk(dp(h, e), m)}{h, \langle pc, e, cc\rangle{::}f, ac \rhd h, \langle pc', e, cc'\rangle{::}\langle pc, e, cc\rangle{::}f, ac}$$

$$\frac{nth(pc) = ret \quad er = e'[v_0 \mapsto e(v_0)]}{h, \langle pc, e, cc\rangle{::}\langle pc', e', cc'\rangle{::}f), ac \rhd h, \langle pc'{+}1, er, cc'\rangle{::}f, ac}$$

Figure 2: Instrumented semantics

to sets of allocation contexts. The abstract domains used by the analysis are summarised below.

$$\begin{array}{lcl} Env^{\sharp} & \triangleq & Var \to \mathcal{P}(AC_0) \\ CCEnv & \triangleq & CC \times \mathbb{N} \to Env^{\sharp} \\ HClass & \triangleq & \mathcal{P}(AC \times Class) \\ HObject & \triangleq & \mathcal{P}(AC \times Field \times AC_0) \\ IState^{\sharp} & \triangleq & HClass \times HObject \times CCEnv \end{array}$$

The result of the analysis is a triple $(hc^{\sharp}, ho^{\sharp}, e^{\sharp}) \in IState^{\sharp}$ defined as the least solution to the constraints defined in Figure 3. The analysis is demand-driven and only propagates information about reachable code. For this purpose, we use a predicate *isdef* to ensure that an abstract environment $pe \in Env^{\sharp}$ binds all the variables to at least one abstract value. For $e \in (Var \to AC_0)$ and $pe \in Env^{\sharp}$, we write $e \in pe$ if $\forall v, e(v) \in pe(v)$ and define $isdef(pe)$ by $\exists e, e \in pe$. To resolve virtual calls in the abstract, we define the abstract dispatch relation.

$$dp^{\sharp} \subseteq (Var \to AC_0) \times (Var \to Class)$$
$$(e, d) \in dp^{\sharp} \text{ iff } \forall v, \begin{cases} e(v) = 0 \Rightarrow d(v) = udf \\ e(v) = \alpha^{\sharp} \Rightarrow (\alpha^{\sharp}, d(v)) \in hc^{\sharp} \end{cases}$$

We write $\sqsubseteq$ the point-wise ordering *i.e.,* $e_1 \sqsubseteq e_2$ if and only if $\forall v, e_1(v) \subseteq e_2(v)$. As a result, a constraint of the form

$e[v_i \mapsto s] \sqsubseteq e'$ denotes a set of subset constraints of the form $e(v_0) \subseteq e'(v_0), \ldots, e(v_{i-1}) \subseteq e'(v_{i-1}), s \subseteq e'(v_i), e(v_{i+1}) \subseteq e'(v_{i+1}) \ldots e(v_n) \subseteq e'(v_n)$.

For intra-procedural statements, constraints have the form $f^\sharp(e^\sharp(cc, pc)) \sqsubseteq e^\sharp(cc, pc{+}1)$ where $f^\sharp$ denotes the abstract semantics of the statement. If before statement $copy(v, v')$ v' is referencing α then v is referencing α after the statement. The abstraction of the other variables is unchanged. If before statement $read(f)$, v_0 is referencing α and if there is an edge (α, f, β) in the points-to graph, then after $read(f)$, v_0 is referencing β. After $new(c)$, the abstraction of v_0 is a singleton α computed by the *alloc* function with respect to the current context cc, program counter pc and class c. The $write(f)$ rule does not modify abstract environments but updates the points-to graph with new edges. As this is a weak update, edges are never removed.

Inter-procedural rules are dealing with call-contexts. The rule $call(m)$ is splitting the current environment with respect to the dispatch function and the abstract *push* function. The target of the call identified by a call-context cc' and a computed program counter $lk(d, m)$ collects (pointwise) the environments resolved there. The rule modelling a return is the one departing the most from the concrete semantics. Its role is to match a call and a return by making sure that the call-context of the callee cc is obtained by a *push* from the call-context of the caller cc'. The return *i.e.,* variable v_0 is then copied from callee to caller.

4.3 Correctness theorem

The correctness theorem states that the result of a context-sensitive points-to analysis *i.e.,* the triple $(hc^\sharp, ho^\sharp, e^\sharp)$ over-approximates the reachable states $IAcc$ of the concrete (instrumented) semantics. To formalise the over-approximation, we define a concretisation function $\gamma : IState^\sharp \to \mathcal{P}(IState)$. A key component of the concretisation is the address cache ac which establishes the correspondence between concrete addresses and allocation contexts. Concretisations of environments γ_P, heaps γ_H and stack of frames γ_F are therefore indexed by an address cache ac. Figure 4 summarises the formal definition of all those concretisation functions.

THEOREM 1. *Let $\langle hc^\sharp, ho^\sharp, e^\sharp\rangle$ be a solution to the constraints, we have $IAcc \subseteq \gamma(hc^\sharp, ho^\sharp, e^\sharp)$.*

The correctness proof relies on key invariants of the instrumented semantics.

- **Absence of dangling pointers.** All the addresses in the stack frame are bound in the heap and all the fields of the objects in the heap refer themselves to addresses allocated in the heap (or possibly the null pointer).
- **Monotonic and faithful address cache.** In the address cache, allocated addresses are mapped to their allocation context. Hence, the address cache is only growing and allocation contexts of allocated addresses are invariant.
- **Instrumented stack discipline.** When a return point is reached with state

  $$h, \langle pc, e, cc\rangle{::}\langle pc', e', cc'\rangle{::}f, ac$$

  then there exists a state $h', \langle pc', e', cc'\rangle{::}f, ac'$ such that $nth(pc') = call(m)$ and $push(cc', pc', m, e'^{ac} = cc$ and if $ac'(\alpha) = \alpha^\sharp$ then $ac(\alpha) = \alpha^\sharp$.

Given these invariants, the proof is by induction over the length of the derivation.

5. CPA BEATS ∞-CFA

In this section, we shall prove that CPA is more precise than ∞-CFA. In the sequel, CPA is identified by a $\bowtie$ subscript and ∞-CFA by a ∞ subscript. In particular, we write $A_\bowtie = \langle hc_\bowtie, ho_\bowtie, e_\bowtie\rangle$ for the result of CPA and $A_\infty = \langle hc_\infty, ho_\infty, e_\infty\rangle$ for the result of ∞-CFA. CPA is more precise than ∞-CFA if it captures less concrete states. As the soundness of our points-to analyses is stated over instrumented states, to compare their relative precision, we introduce an erasure function $\mathcal{E} : \mathcal{P}(IState) \to \mathcal{P}(State)$ which given a set of instrumented states removes their instrumentation *i.e.,* address caches and call-contexts of the stack of frames. Hence, proving that CPA is more precise than ∞-CFA amounts to proving the following theorem.

THEOREM 2.

$$\mathcal{E} \circ \gamma_\bowtie(A_\bowtie) \subseteq \mathcal{E} \circ \gamma_\infty(A_\infty)$$

To structure the proof we introduce another points-to analysis Ω-CFA that keeps more context information than both ∞-CFA and CPA. As far as allocation context is concerned, Ω-CFA is like CPA and ∞-CFA and abstracts addresses by their allocation site ($alloc(cc, pc, c, e) = pc$). However, it keeps more information about call-contexts than CPA and ∞-CFA: $push_\Omega(cc, pc, m, e) = \langle m, e, pc\rangle{::}cc$. Hence, by construction, Ω-CFA is more precise than ∞-CFA and CPA. Therefore, to prove our theorem, it remains to prove that CPA is also at least as precise as Ω-CFA. To do so, we exhibit a *precise* translation function $\mathcal{T} : IState^\sharp_\Omega \to IState^\sharp_\bowtie$ mapping the result of Ω-CPA to the result of CPA such that:

$$\begin{array}{lll} P1 & : & A_\bowtie \sqsubseteq \mathcal{T}(A_\Omega) \\ P2 & : & \mathcal{E} \circ \gamma_\bowtie \circ \mathcal{T}(A_\Omega) \subseteq \mathcal{E} \circ \gamma_\Omega(A_\Omega) \end{array}$$

Property $P1$ means that the result of A_Ω can be plunged (by applying $\mathcal{T}$) into the CPA domain in such a way that $A_\bowtie$ is more precise. There are many *imprecise* $\mathcal{T}$ functions with this property. Property $P2$ ensures the precision of the translation by requiring that $\mathcal{T}$ is compatible with the concretisations $\gamma_\bowtie$ and γ_Ω.

LEMMA 1. *Given $P1$ and $P2$, we have $\mathcal{E} \circ \gamma_\bowtie(A_\bowtie) \subseteq \mathcal{E} \circ \gamma_\infty(A_\infty)$*

PROOF. By applying the monotony of $\mathcal{E}$ and $\gamma_\bowtie$ over $P1$, we get $\mathcal{E} \circ \gamma_\bowtie(A_\bowtie) \subseteq \mathcal{E} \circ \gamma_\bowtie \circ \mathcal{T}(A_\Omega)$. By $P2$ and transitivity of $\subseteq$ we get $\mathcal{E} \circ \gamma_\bowtie(A_\bowtie) \subseteq \mathcal{E} \circ \gamma_\Omega(A_\Omega)$. Because A_Ω is more precise than A_∞, this concludes the proof. □

The formal definition of $\mathcal{T}$ is given below: Because the two analyses are sharing the same heap abstraction, no translation occurs for $hc^\sharp$ and $ho^\sharp$. For the environments, the function $\mathcal{T}_E$ merges all the abstract environments for which the top-element of the call-context stack matches the context of CPA. Lemma 2 states that the merging of Ω-contexts is benign and does not introduce spurious environments.

LEMMA 2. *If $e \in \mathcal{T}_E(e^\sharp)((m, o), pc)$ then there exists l and pco such that $nth(pco) = call(m) \wedge e \in e^\sharp((m, o, pco){::}l, pc)$.*

PROOF. If $e \in \mathcal{T}_E(e^\sharp)((m, o), pc)$ then by definition of $\mathcal{T}_E$, we have that forall v, there exists l and pco such that

$$nth(pco) = call(m), e(v) \in e^\sharp_\Omega((m, o, pco) :: l, pc)(v)$$

$$\frac{nth(pc) = copy(v, v') \quad isdef(e^\sharp(cc, pc))}{e^\sharp(cc, pc)[v \mapsto e^\sharp(cc, pc)(v')] \sqsubseteq e^\sharp(cc, pc{+}1)} \qquad \frac{nth(pc) = new(c) \quad e \in e^\sharp(cc, pc) \quad \alpha = alloc(cc, pc, c, e)}{e^\sharp(cc, pc)[v_0 \mapsto \{\alpha\}] \sqsubseteq e^\sharp(cc, pc{+}1)}$$

$$\frac{nth(pc) = read(f) \quad isdef(e^\sharp(cc, pc)) \quad s = \{\beta \mid \alpha \in e^\sharp(cc, pc)(v_0), (\alpha, f, \beta) \in ho^\sharp\}}{e^\sharp(cc, pc)[v_0 \mapsto s] \sqsubseteq e^\sharp(cc, pc{+}1)} \qquad \frac{nthp(pc) = write(f) \quad isdef(e^\sharp(cc, pc))}{e^\sharp(cc, pc) \sqsubseteq e^\sharp(cc, pc{+}1)}$$

$$\frac{nth(pc) = new(c) \quad e \in e^\sharp(cc, pc) \quad \alpha = alloc(cc, pc, c, e)}{\alpha, c \in hc^\sharp \quad \alpha, f, 0 \in ho^\sharp} \qquad \frac{nthp(pc) = write(f) \quad isdef(e^\sharp(cc, pc)) \quad \alpha \in e^\sharp(cc, pc)(v_0) \quad \beta \in e^\sharp(cc, pc)(v_1)}{\alpha, f, \beta \in ho^\sharp}$$

$$\frac{}{\lambda v.\{0\} \sqsubseteq e^\sharp(\bullet, 0)}$$

$$\frac{nth(pc) = call(m) \quad e \in e^\sharp(cc, pc) \quad (e, d) \in dp^\sharp \quad cc' = push(cc, pc, m, e)}{\lambda v.\{e(v)\} \sqsubseteq e^\sharp(cc', lk(d, m))} \qquad \frac{nth(pc) = ret \quad isdef(e^\sharp(cc, pc)) \quad nth(pc') = call(m) \quad e' \in e^\sharp(cc', pc') \quad cc = push(cc', pc', m, e') \quad r = e^\sharp(cc, pc)(v_0)}{e^\sharp(cc', pc')[v_0 \mapsto r] \sqsubseteq e^\sharp(cc', pc'{+}1)}$$

Figure 3: Points-to analysis

$$\begin{array}{lcl}
\gamma_P & : & Acache \to Env^\sharp \to \mathcal{P}(Env) \\
\gamma_P^{ac}(pe) & = & \{e \mid e^{ac} \in pe\}
\end{array}$$

$$\begin{array}{lcl}
\gamma_H & : & Acache \to (HClass \times HObject) \to \mathcal{P}(Heap) \\
\gamma_H^{ac}(hc^\sharp, ho^\sharp) & = & \left\{ h \;\middle|\; \begin{array}{ll} h(\alpha) = \bot \Rightarrow & ac(\alpha) = \bot \\ h(\alpha) = \langle c, o\rangle \Rightarrow & \exists \alpha^\sharp. ac(\alpha) = \alpha^\sharp, (\alpha^\sharp, c) \in hc \\ & \wedge \forall f, \langle \alpha^\sharp, f, ac_0(o(f))\rangle \in ho \end{array} \right\}
\end{array}$$

$$\begin{array}{lcl}
\gamma_F & : & Acache \to CCEnv \to \mathcal{P}(IFrame^+) \\
\gamma_F^{ac}(e^\sharp) & = & \left\{ \langle pc_n, e_n, cc_n\rangle :: \ldots :: \langle pc_0, e_0, \bullet\rangle :: \epsilon \;\middle|\; \begin{array}{l} nth(pc_i) = m_i \quad e_i \in \gamma_P^{ac}(e^\sharp)(cc_i, pc_i) \\ cc_{i+1} = push(cc_i, pc_i, m_i, e_i^{ac}) \end{array} \right\}
\end{array}$$

$$\begin{array}{lcl}
\gamma & : & IState^\sharp \to \mathcal{P}(IState) \\
\gamma(hc^\sharp, ho^\sharp, e^\sharp) & = & \left\{ h, f, ac \;\middle|\; h \in \gamma_H^{ac}(hc^\sharp, ho^\sharp) \wedge f \in \gamma_F^{ac}(e^\sharp) \right\}
\end{array}$$

Figure 4: Concretisation functions

$$\begin{array}{l}
\mathcal{T}_E : CCEnv_\Omega \to CCEnv_\bowtie \\
\quad \mathcal{T}_E(e^\sharp)(\bullet_\bowtie, pc) = e^\sharp(\bullet_\Omega, pc) \\
\quad \mathcal{T}_E(e^\sharp)((m, o), pc) = \bigsqcup \left\{ e^\sharp((m, o, pco)::l, pc) \;\middle|\; \begin{array}{l} l \in CC_\Omega \\ nth(pco) = call(m) \end{array} \right\} \\
\mathcal{T} : IState_\Omega^\sharp \to IState_\bowtie^\sharp \\
\mathcal{T}(hc^\sharp, ho^\sharp, e^\sharp) = (hc^\sharp, ho^\sharp, \mathcal{T}_E(e^\sharp))
\end{array}$$

where $\bigsqcup S = \lambda v. \bigcup \{e(v) \mid e \in S\}$.

Figure 5: Translation function

We prove that it is possible to swap quantifiers and exhibit a single stack l and pco such that $nth(pco) = call(m) \wedge e \in e^\sharp((m, o, pco)::l, pc)$. The intuition is that the prefix stack l has no impact on abstract environments in the scope of the callees of m – much as in the concrete semantics. For instance, if pc is the target of the call to m, the abstract environment is o and is thus independent from l. By induction over the depth of subsequent calls, we can conclude that all prefix stacks are equivalent and cannot be distinguished. As a result, we can pick any stack l and pco such that for some v, $nth(pco) = call(m) \wedge e(v) \in e^\sharp_\Omega((m, o, pco) :: l, pc)(v)$. For any other variable v', we will also have $e(v') \in e^\sharp_\Omega((m, o, pco) :: l, pc)(v')$. □

Lemma 3 states that for a given abstract heap, CPA is more precise than Ω-CFA

LEMMA 3. *Given an abstract heap $(hc^\sharp, ho^\sharp)$, we have $e^\sharp_\bowtie \sqsubseteq \mathcal{T}(e^\sharp_\Omega)$.*

PROOF. The proof is by induction over the definition of $e^\sharp_\bowtie$. Suppose that $e \in e^\sharp_\bowtie((m, o), pc)$, by induction hypothesis we have that $e \in \mathcal{T}_E(e^\sharp)((m, o), pc)$. By Lemma 2 we can exhibit a stack l and pco such that $nth(pco) = call(m) \wedge e(v) \in e^\sharp_\Omega((m, o, pco) :: l, pc)(v)$. The proof follows by case analysis over the rules defining $e^\sharp_\bowtie$ and $e^\sharp_\Omega$. □

LEMMA 4. $A_\bowtie \sqsubseteq \mathcal{T}(A_\Omega)$

PROOF. The proof is straightforward because we already proved that $e^\sharp_\bowtie \sqsubseteq \mathcal{T}(e^\sharp_\Omega)$ and because both analyses are using the same *alloc* function. □

It remains to prove that $\mathcal{E} \circ \gamma_\bowtie \circ \mathcal{T}(A_\Omega) \subseteq \mathcal{E} \circ \gamma_\Omega(A_\Omega)$. As A_Ω and $A_\bowtie$ are using the same heap abstraction, they compute the same address cache. Once again, the key argument is Lemma 2 allowing to show that $\mathcal{T}(A_\Omega)$ retains enough information to rebuild the call-stack of A_Ω.

5.1 The origin of ϖ-CFA imprecision

An interesting point not elucidated by the proof is the reason why CPA is *strictly* more precise than ∞-CFA. A reason already discussed in [2] is that the abstraction of environments is set-based thus responsible for a loss of precision. This precision loss is not specific to ∞-CFA but shared by all context-sensitive points-to analyses. Yet, an effect of CPA context-sensitivity is that environments often map variables to singletons (a property identified by Agesen [1]) thus minimising the effect of the non-relational abstraction of environments.

EXAMPLE 1. *Suppose a function* `f(x,y)` *dispatched as follows:* $(A, A) \mapsto f_1; (B, B) \mapsto f_1; (A, B) \mapsto f_2; (B, A) \mapsto f_2$.
After running the following pseudo-code,

```
x:=(?:new A();new B());
y:=(?:new A();new B());
f(x,y)
```

f_1 is called with arguments $\{(A, A), (B, B)\}$.

CPA obtains a precise result concluding that f_1 is called in context (A, A) or (B, B). ∞-CFA incurs a loss pf precision because it considers a single context per call. It thus concludes that f_1 is called with arguments $(A, A) \sqcup (B, B) = (\{A, B\}, \{A, B\})$ and thus looses precision.

6. CONCLUSION

A few years back, Grove and Chambers [2] conjectured that CPA was more precise than ∞-CFA. To our knowledge, our proof is the first to firmly establish this result. As CPA is a form of *object-sensitivity* and ∞-CFA is the *ultimate call-string* sensitivity, this theoretical result backs up recent empirical studies [7, 6] concluding than *object-sensitivity* performs better than *call-string sensitivity*.

Compared to the object-oriented community, researchers in the functional community are better equipped to carry-out formal proofs: various flavors of the lambda-calculus, many alternative semantics and a variety of abstract machines. For our formalisation, we introduce a core object-oriented language well-suited for reasoning about control-flow analyses. The language is minimal allowing, with little overhead, a mechanised reasoning into a proof-assistant.

As further work we shall investigate how to effectively compute context-sensitive analyses with an *infinite* number of contexts. At the top of the list is ∞-CFA but other instances such as *infinite object-sensitivity* of allocation contexts are of interest.

7. REFERENCES

[1] O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *ECOOP'95*, pages 2–26, 1995.

[2] D. Grove and C. Chambers. A framework for call graph construction algorithms. *Toplas*, 23(6):685–746, 2001.

[3] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, 1997.

[4] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *PASTE'01*, pages 54 – 61. ACM, 2001.

[5] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *Toplas*, 23(3):396–450, 2001.

[6] O. Lhoták and L. J. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), 2008.

[7] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[8] F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL'97*, pages 332–345. ACM, 1997.

[9] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2004.

[10] J. Palsberg and M.I. Schwartzbach. Object-oriented type inference. In *OOPSLA'91*, pages 146–161, 1991.

[11] J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.