Modular Heap Analysis for Higher-Order Programs

Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani

Microsoft Research, India
{t-rakand,grama,kapilv}@microsoft.com

Abstract. We consider the problem of computing summaries for procedures that soundly capture the effect of calling a procedure on program state that includes a mutable heap. Such summaries are the basis for a compositional program analysis and key to scalability. Higher order procedures contain callbacks (indirect calls to procedures specified by callers). The use of such callbacks and higher-order features are becoming increasingly widespread and commonplace even in mainstream imperative languages such as C♯ and Java. Such callbacks complicate compositional analysis and the construction of procedure summaries. We present an abstract-interpretation based approach to computing summaries (of a procedure's effect on a mutable heap) in the presence of callbacks in a simple imperative language. We present an empirical evaluation of our approach.

1 Introduction

In this paper, we present a compositional approach to heap analysis for an imperative language with dynamic memory allocation and higher order functions (or callbacks). Modular/compositional program analysis [1] is a key technique for scaling static analysis to large programs. Our interest is in techniques to compute a *summary* for each procedure that approximates its relational semantics (relating input states to output states). A significant benefit of this approach is that libraries can be analyzed once and the computed library summaries reused for any program that uses the library. This is particularly significant since modern applications rely on large libraries and frameworks.

A typical approach to computing procedure summaries is to first construct a call-graph and then analyze procedures in the call-graph in a bottom-up fashion. Any collection of mutually recursive procedures is iteratively analysed until their summaries reach a fixed point. This approach is feasible when the call-graph can be constructed easily and precisely, e.g., for languages with only direct calls. However, most modern languages permit *indirect* calls (virtual methods, delegates, etc.), which pose several challenges. Determining the targets of indirect calls (which depend on runtime values) is itself a complex analysis, and depends upon the results of heap analysis. One possibility is to integrate heap analysis and the call-graph construction into a single analysis. However, the direct way of doing this gives up on modularity and resorts to a top-down whole-program analysis. An alternative is to use a less precise call-graph construction technique that does not require heap analysis: e.g., type-based techniques such as Class Hierarchy Analysis (CHA). This approach too suffers from several drawbacks.

(a) A library procedure may *call back* a procedure defined by a client of the library. This means that a conservative call-graph cannot be constructed for a library *independent*

A. Miné and D. Schmidt (Eds.): SAS 2012, LNCS 7460, pp. 370–387, 2012.

of its client. Hence, the library cannot be analyzed independent of its client. Thus, we are again forced to resort to a whole-program analysis of each application separately.

(b) A conventional call-graph is necessarily context-insensitive. it identifies the possible targets of an indirect call in a procedure, but the targets actually called may vary with calling contexts. A modular analysis, based on such a call-graph, will compute a procedure summary that is context-insensitive (in terms of precision).

(c) A type-based call-graph can be very imprecise. Assuming that a delegate call, in C♯, can invoke any delegate (essentially a lambda expression) defined in the program can be disastrous. The imprecision of type-based resolution also leads to significant scalability problems, especially with common methods such as `equals` or `hashcode` and common interfaces such as iterators.

Our Approach. Computing summaries describing heap effects is challenging even in the absence of indirect calls, as the summary must capture the effects of a procedure on the heap, without making assumptions about the aliasing in the input heap. Previously, we had presented an abstract-interpretation approach to computing such *first-order heap-effect summaries* (based on prior work by Whaley, Salcianu, and Rinard) [11,8,4]. In this paper, we extend this approach to deal with callbacks and higher-order procedures modularly, by constructing *higher-order heap-effect summaries*. The intuition behind our approach is informally described in Section 2.

The key and first step is to formulate a compositional concrete semantics for a language with higher order procedures in a form suitable for abstraction, as shown in Section 3. We then mimic the same structure to define an abstract semantics for procedures and libraries (Sections 4 and 5), which serves as the basis for our analysis, which can be applied to a library independent of its client(s).

We have implemented our analysis for C♯ and evaluated it on a collection of large applications (Section 6). We find that indirect method calls are a widely used feature of modern languages. We also find that call graphs based on our approach are significantly more precise and compact compared to conventional class hierarchy analysis. As a result, our heap analysis is able to scale to larger applications.

The ideas behind our approach are similar to those used in analyses presented by Vivien *et al.* [10] and Lattner *et al.* [3]. However, neither of these analyses has a theoretical formulation or a correctness proof. We believe that these two seemingly different analyses can be seen as instances of our abstract interpretation formulation. A comparison of our work with these analyses and other related work appears in Section 7.

2 An Informal Overview

We now present an informal overview of how we extend our previous approach [4] to handle higher-order procedures. The previous approach computes a shape-graph like summary for first-order procedures that can be concretized as a transformer of concrete heap graphs. We refer to these representations as transformer graphs. Let τ range over such transformer graphs. We present a formal definition of transformer graphs later, as the intuition behind the extension to higher-order procedures does not depend on this.

Representing Higher-Order Summaries. The basic idea is to extend the summary representation to capture information about callbacks that can occur, delaying instantiating

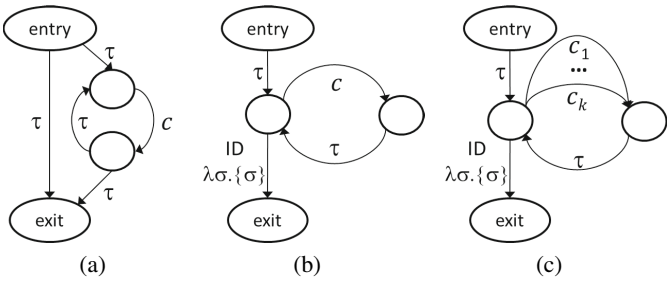

Fig. 1. Informal interpretation of summary

the effects of the callback until sufficient context information is available to determine the actual procedure(s) that are called, using first-order summaries (i.e., transformer graphs) to represent code-fragments that are free of unresolved (indirect) calls.

Consider a procedure whose body is of the form $S_1; S_2; S_3$, where S_1 and S_3 are free of unresolved calls, and S_2 consists of a single unresolved call. In this case, we can compute graph transformers τ_1 and τ_3 that are abstractions of S_1 and S_3 respectively, and utilize a symbolic summary of the form $\tau_1; S_2; \tau_3$ for the procedure. While this is basic idea behind our approach, we refine this idea in several ways.

Exploiting Local Context. Though we don't know the exact side-effects of the indirect call in S_2, we can restrict the scope of these side-effects: the call can affect only the part of the heap reachable via the call parameters and global variables. This observation lets us decompose τ_3 into two parts: τ_3^ℓ, concerning locations that cannot be modified by the call in S_2, and τ_3^g, concerning locations that could be modified by the call. We can then write the summary as $\tau_1; S_2; \tau_3^\ell; \tau_3^g$, which simplifies to $\tau_1 \circ \tau_3^\ell; S_2; \tau_3^g$. Note that the composition "$\tau_1 \circ \tau_3^\ell$" can be computed once, even before the target of the indirect call is determined, simplifying the summary.

A Flow-Insensitive Abstraction. In general, when the procedure contains many unresolved calls, this approach will lead to a summary representation that looks like a control-flow-graph where every vertex (other than entry/exit vertices) represents a call-statement, and every edge is annotated with a transformer graph. For efficiency reasons, we utilize a more aggressive, but less precise, flow-insensitive abstraction that uses a single graph transformer τ (instead of one per edge) and a set of call-statements w (thus forgetting the control-flow between call-statements). For our previous example, this produces a summary $(\tau, \{S_2\})$, where τ conservatively approximates both $\tau_1 \circ \tau_3^\ell$ and τ_3^g. Informally, a summary $(\tau, \{c\})$ can be interpreted as the control-flow-graph shown in Fig. 1(a). Since the transformers τ we use are isotonic (i.e., τ is a sound approximation of the identity relation), this interpretation can be simplified to the one shown in Fig. 1(b), which is the basis for our subsequent formalization. A summary $(\tau, \{\ c_1, \cdots, c_k\ \})$ is interpreted as shown in Fig. 1(c).

Computing Higher Order Summaries. We present an algorithm that constructs the desired transformer τ using a Sharir-Pnueli style interprocedural analysis. We present details of this analysis later, but list some of the key components of the analysis here.

Intraprocedural Analysis. We present an abstract semantics for primitive statements that maps an input summary (τ, ω) to an output summary (τ', ω').

Direct and Indirect Calls. The abstract semantics of a direct call statement is defined using a *composition* operator that combines an input summary (τ_r, ω_r) and the called procedure's summary (τ_e, ω_e) into an output summary, after accounting for parameter passing. Initially, an indirect call c is handled in a straightforward fashion, by updating the input summary to include c as an unresolved call. However, as the analysis proceeds, sufficient context information may become available to resolve indirect calls: e.g., when a procedure summary containing an unresolved call is instantiated at a particular call-site. Our analysis identifies indirect calls whose targets can be resolved. If a resolved target's summary is available, then it is instantiated. This is an iterative process, as instantiating a resolved target's summary may create further opportunities for resolving more indirect calls.

Eliminating Indirect Calls. Completely resolving an indirect call can be a multi-stage process. At intermediate stages, we might be able to identify some of the potential targets of an indirect call, but cannot be sure whether all possible targets of the indirect call have been identified. Eventually, sufficient context information may become available to let us determine that all possible targets of the indirect call have been identified. At this point, the indirect call can be dropped from the summary.

3 The Language and Its Concrete Semantics

Syntax. A library (LP,LL) consists of a set of procedures LP and a set of nested libraries LL (denoting libraries it is linked with). A procedure P consists of a name (belonging to the set $Procs$) and a control-flow graph, with an entry vertex $entry(\mathsf{P})$ and an exit vertex $exit(\mathsf{P})$. The entry vertex has no predecessor and the exit vertex has no successor. Every edge of the control-flow graph is labelled by a primitive statement. The set of primitive statements are shown in Fig. 2. We use $u \xrightarrow{S} v$ to indicate an edge in the control-flow graph from vertex u to vertex v labelled by statement S. In the sequel, we abuse notation and do not distinguish between a procedure and its name, e.g. if P is a procedure then P also denotes its name.

We use "function pointers" as the primitive for indirect calls. The statement "x = &P" assigns the address of procedure P to variable x, and the indirect call "$(*x)(a_1, \cdots, a_k)$" calls the procedure pointed to by x. This is sufficient to model common indirect call mechanisms such as virtual functions and delegates. A closure c can be realized as a pair consisting of a function pointer $c.f$ and a data pointer $c.d$, and the call to c modelled as "$(*(c.f))(c.d)$".

Concrete Semantics Domain. Let $Vars$ denote a set of identifiers used as variable names, partitioned into the following disjoint sets: the set of global variables $Globals$, the set of local variables $Locals$ (assumed to be the same for every procedure), and the set of formal parameter variables $Params$ (assumed to be the same for every procedure). Let $Fields$ denote a set of identifiers used as field names. We use a simple language with only two primitive types: pointers to heap objects and function-pointers.

We use a graph-based representation for the concrete state. We use a level of indirection in representing function-pointer variables: the variable stores a procedure-id (identifying a procedure), and a separate table maps the procedure-id to its semantic value (as formalized below). We use procedure-ids, instead of procedure names, to ensure uniqueness of ids, for reasons explained soon.

Let N_c be an unbounded set of heap locations. Let PV_c be an unbounded set of values, disjoint from N_c, used as procedure-ids. A concrete (points-to) graph $g \in \mathbb{G}_c$ is a triple $(\mathsf{V}, \mathsf{E}, \sigma)$, where $\mathsf{V} \subseteq \mathsf{N}_c \cup \mathsf{PV}_c$ represents the set of objects in the heap, $\mathsf{E} \subseteq (\mathsf{V} \cap \mathsf{N}_c) \times \mathit{Fields} \times \mathsf{V}$ represents values of pointer fields in heap objects, and $\sigma \in \Sigma_c = \mathit{Vars} \mapsto \mathsf{V}$ represents the values of program variables. N_c includes a special element null. Variables and fields of new objects are initialized to null.

Let $\mathcal{F}_c = \mathbb{G}_c \mapsto 2^{\mathbb{G}_c}$ be the set of functions that map a concrete graph to a set of concrete graphs. An element of $\mathcal{F}_c$ may also be thought of as a (binary) relation on concrete graphs. The semantics of statements (and procedures), in the absence of indirect calls, can be described using elements of $\mathcal{F}_c$.

We now enrich the domain to support indirect procedure calls. We define two domains $\mathcal{P}_c$ and $\mathcal{T}_c$ recursively as follows: $\mathcal{T}_c = \mathsf{PV}_c \hookrightarrow \mathcal{P}_c$ and $\mathcal{P}_c = \mathbb{G}_c \times \mathcal{T}_c \rightarrow_i 2^{\mathbb{G}_c \times \mathcal{T}_c}$. An element of $\mathcal{T}_c$ is a partial function, binding procedure-ids to their semantics. (We may think of this as a simple "virtual-function table".) The concrete state is enriched by such a table. A procedure uses such a table to dispatch indirect calls (including callbacks). But the procedure may also update the table (e.g., if it returns a procedure-value). However, the procedure can only *add* new entries to the table, but not modify pre-existing entries. The construct $\rightarrow_i$ includes only such functions. A function f in $\mathbb{G}_c \times \mathcal{T}_c \rightarrow 2^{\mathbb{G}_c \times \mathcal{T}_c}$ is defined to be in $\mathbb{G}_c \times \mathcal{T}_c \rightarrow_i 2^{\mathbb{G}_c \times \mathcal{T}_c}$ iff: $(g', t') \in f(g, t)$ implies $\forall n \in \mathrm{dom}(t).t'(n) = t(n)$. The domain $\mathcal{P}_c$ generalizes $\mathcal{F}_c$ and is used to give semantics to higher-order statements and procedures. We define $\mathcal{L}_c$ to be $\mathit{Procs} \hookrightarrow \mathcal{P}_c$.

We define a partial order $\sqsubseteq_c$ on $\mathcal{F}_c$ as: $f_a \sqsubseteq_c f_b$ iff $\forall g \in \mathbb{G}_c.f_a(g) \subseteq f_b(g)$. Let $\sqcup_c$ denote the corresponding least upper bound (join) operation defined by: $f_a \sqcup_c f_b = \lambda g.f_a(g) \cup f_b(g)$. For any $f \in \mathcal{F}_c$, we define $\hat{f} : 2^{\mathbb{G}_c} \mapsto 2^{\mathbb{G}_c}$ by: $\hat{f}(G) = \cup_{g \in G} f(g)$. We define the relational composition of two elements in $\mathcal{F}_c$ as: $f_a \circ f_b = \lambda g.\hat{f_b}(f_a(g))$. We extend these operators to the domain $\mathcal{P}_c$, $\mathcal{T}_c$ and $\mathcal{L}_c$ following the structure of their recursive definitions. E.g, $\sqsubseteq_c$ is extended as follows: for any $p_1, p_2 \in \mathcal{P}_c$, $p_1 \sqsubseteq_c p_2$ iff $(g', t') \in p_1(g, t) \Rightarrow \exists (g'', t'') \in p_2(g, t)$ s.t. $g' = g''$, $t' \sqsubseteq_c t''$ where $t_1 \sqsubseteq_c t_2$ iff $\forall n \in dom(t_1), t_1(n) \sqsubseteq_c t_2(n)$. For any $l_1, l_2 \in \mathcal{L}_c$, $l_1 \sqsubseteq_c l_2$ iff $\forall \mathsf{P} \in dom(l_1), l_1(\mathsf{P}) \sqsubseteq_c l_2(\mathsf{P})$.

Concrete Semantics. A primitive statement S has a semantics $[\![S]\!]_c \in \mathcal{P}_c$, as shown in Fig. 2. The semantics of call statements and the semantics of the procedures and libraries that contain them are mutually interdependent. Hence, we parameterize the semantics of call statements with a parameter AP, defined as follows. Let $(\mathsf{LP}, \mathsf{LL})$ be a library consisting of a set of procedures LP and a set of nested libraries LL. Let LLP denote the set of all procedures in LL. A direct call in the library can only reference procedures defined in LP or in LL. The semantics of $(\mathsf{LP}, \mathsf{LL})$ is defined as the least fixed point of a collection of equations (defined below) which contains a variable φ_{P} for every procedure P in LP. Define the partial function AP as follows: AP maps every $\mathsf{P} \in \mathsf{LP}$ to variable φ_{P}, and it maps every $\mathsf{P} \in \mathsf{LLP}$ to its semantics $[\![\mathsf{P}]\!]_c$. For simplicity, we assume

Statement S	**Concrete semantics** $[\![S]\!]_c\ ((\mathsf{V},\mathsf{E},\sigma),t)$
$v_1 = v_2$	$\{\ ((\mathsf{V},\mathsf{E},\sigma[v_1 \mapsto \sigma(v_2)]),t)\ \}$
$v = new\ C$	$\{\ ((\mathsf{V} \cup \{n\},\mathsf{E} \cup \{n\} \times \mathit{Fields} \times \{\mathit{null}\},\sigma[v \mapsto n]),t) \mid n \in N_c \setminus \mathsf{V}\ \}$
$v_1.f = v_2$	$\{\ ((\mathsf{V},\{\langle u,l,v\rangle \in \mathsf{E} \mid u \neq \sigma(v_1) \vee l \neq f\} \cup \{\langle\sigma(v_1),f,\sigma(v_2)\rangle\},\sigma),t)\ \}$
$v_1 = v_2.f$	$\{\ ((\mathsf{V},\mathsf{E},\sigma[v_1 \mapsto n]),t) \mid \langle\sigma(v_2),f,n\rangle \in \mathsf{E}\ \}$
$v = \&\mathsf{P}$	**if** $\mathsf{P} \in \mathrm{dom}(\mathsf{AP})$ **then** $\{\ ((\mathsf{V},\mathsf{E},\sigma[v \mapsto n]),t[n \mapsto \mathsf{AP}(\mathsf{P})]) \mid n \notin \mathrm{dom}(t)\ \}$ **else** $\{\}$
$\mathsf{P}(v_1,\cdots,v_k)$	**if** $\mathsf{P} \in \mathrm{dom}(\mathsf{AP})$ **then** $\mathit{Call}_S(\mathsf{AP}(\mathsf{P}))$ **else** $\{\}$
$(*v)(v_1,\cdots,v_k)$	**if** $\sigma(v) \in \mathrm{dom}(t)$ **then** $\mathit{Call}_S(t(\sigma(v)))$ **else** $\{\}$

Fig. 2. Statements in our language and their concrete semantics

that procedure names are unique across LP and LL. We can eliminate this assumption by using unique qualified names for procedures. However, the semantics presented is valid for all clients, including those that may reuse procedure names used in LP or LL. To avoid name-capture when control flows back and forth between a client and the library via callbacks, we identify procedures using unique ids generated at runtime, as illustrated by the semantics of the statement "$v = \&\mathsf{P}$". These unique ids are used as indices into the "virtual function table" t.

A direct call to procedure P or taking the address of procedure P is valid only if $\mathsf{P} \in \mathrm{dom}(\mathsf{AP})$. In this case, the semantics of the statement is defined in terms of $\mathsf{AP}(\mathsf{P})$. An indirect call, however, may reference other procedures, e.g., such as those defined by clients of the libraries. The run-time parameter t is used to resolve indirect calls. Given $f \in \mathcal{P}_c$, $\mathit{Call}_S(f)$ is essentially the same as f, but accounts for parameter passing and pushing/popping activation records and is defined in the Appendix.

Semantics of Procedures. We now define the concrete summary semantics $[\![\mathsf{P}]\!]_c \in \mathcal{P}_c$ for every procedure P in LP using the following equations, in the style of Sharir-Pnueli. For every procedure P in LP, we introduce a new variable φ_u for every vertex in the control-flow graph (of P) and a new variable $\varphi_{u,v}$ for every edge $u \to v$ in the control-flow graph. We also introduce a variable φ_{P}. The semantics is defined as the least fixed point of the following set of equations. The value of φ_u in the least fixed point is a function that maps any concrete state (g,t) to the set of concrete states that arise at program point u when the procedure containing u is executed with an initial state (g,t). Similarly, $\varphi_{u,v}$ captures the states after the execution of the statement labelling edge $u \to v$.

$$\varphi_v = \lambda(g,t).\ \{(g,t)\} \qquad v \text{ is an entry vertex} \tag{1}$$

$$\varphi_v = \bigsqcup_c \{\varphi_{u,v} \mid u \to v\} \qquad v \text{ is not an entry vertex} \tag{2}$$

$$\varphi_{u,v} = \varphi_u \circ [\![S]\!]_c \qquad \text{where } u \xrightarrow{S} v \tag{3}$$

$$\varphi_{\mathsf{P}} = \varphi_{exit(\mathsf{P})} \tag{4}$$

We define $[\![\mathsf{P}]\!]_c$ to be the value of φ_{P} in the least fixed point of equations (1)-(4).

Semantics of Libraries. (Note that an application or program is just a special case of a library.) The semantics of a library is captured by an element of $\mathcal{L}_c$, that maps (the name of) every procedure in the library to its semantics: $[\![(\mathsf{LP}, \mathsf{LL})]\!]_c = \{\ (\mathsf{P}, [\![\mathsf{P}]\!]_c)\ |\ \mathsf{P} \in \mathsf{LP}\ \}$.

4 Abstract Domains and Concretization

We now formally present an abstract interpretation that analyzes a library $(\mathsf{LP}, \mathsf{LL})$ and computes a sound approximation of its concrete semantics presented earlier. Our algorithm first analyzes all the libraries in LL, uses these results to analyze and compute a summary for every procedure in LP. The algorithm can also be used to analyze an application (whole program) or a single method in isolation, which are just special cases of a library.

The Abstract Graph Domain. We utilize an abstract (points-to) graph to represent a set of concrete graphs. Our formulation is parameterized by a set N_a, the universal set of all abstract graph nodes, and a set PV_a, the set of abstract procedure-ids. An abstract graph $g \in \mathbb{G}_a$ is a triple $(\mathsf{V}, \mathsf{E}, \sigma)$, where $\mathsf{V} \subseteq \mathsf{N}_a \cup \mathsf{PV}_a$ represents the set of abstract heap objects, $\mathsf{E} \subseteq (\mathsf{V} \cap \mathsf{N}_a) \times \mathit{Fields} \times \mathsf{V}$ represents possible values of pointer fields in the abstract heap objects, and $\sigma \in \mathit{Vars} \mapsto 2^{\mathsf{V}}$ is a map representing the possible values of program variables. Given a concrete graph $g_1 = \langle \mathsf{V}_1, \mathsf{E}_1, \sigma_1 \rangle$ and an abstract graph $g_2 = \langle \mathsf{V}_2, \mathsf{E}_2, \sigma_2 \rangle$ we say that a function $h : \mathsf{V}_1 \mapsto \mathsf{V}_2$ is an embedding of g_1 into g_2, denoted $g_1 \preceq_h g_2$, iff:

$$\langle x, f, y \rangle \in \mathsf{E}_1 \Rightarrow \langle h(x), f, h(y) \rangle \in \mathsf{E}_2, \qquad \forall v \in \mathit{Vars}.\ \{\ h(\sigma_1(v))\ \} \subseteq \sigma_2(v)$$

The concretization $\gamma_G(g_a)$ of an abstract graph g_a is defined to be the set of all concrete graphs that can be embedded into g_a: $\gamma_G(g_a) = \{g_c \in \mathbb{G}_c \mid \exists h. g_c \preceq_h g_a\}$

The Transformer Graph Domain. A transformer graph τ [4] is a graph-based representation that can be used to abstract the relational semantics of a first-order procedure or code fragment. It is based on weak-updates to the heap. Hence, given any input graph, a transformer graph identifies a set of heap objects that may be added to the input graph, and a set of points-to edges that may be added to the input graph.

The set $\mathcal{F}_a$ of transformer graphs is defined as follows. An element $\tau \in \mathcal{F}_a$ is a tuple $(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$ where, $\mathsf{EV} \subseteq \mathsf{N}_a$ is the set of external vertices, $\mathsf{IV} \subseteq \mathsf{N}_a \cup \mathsf{PV}_a$ is the set of internal vertices, $\mathsf{EE} \subseteq \mathsf{V} \times \mathit{Fields} \times \mathsf{EV}$ is the set of external edges, where $\mathsf{V} = \mathsf{EV} \cup \mathsf{IV}$, $\mathsf{IE} \subseteq \mathsf{V} \times \mathit{Fields} \times \mathsf{V}$ is the set of internal edges, $\pi \in \mathit{Vars} \mapsto 2^{\mathsf{V}}$ is a map representing the possible values of program variables in the initial state and $\sigma \in \mathit{Vars} \mapsto 2^{\mathsf{V}}$ is a map representing the possible values of program variables in the final state. Internal nodes and edges are used to represent new nodes and points-to edges to be added to the input graph. External nodes and external edges are used to create symbolic access-paths evaluated against an input graph to determine the sources and targets of edges to be added. More generally, an external node in the transformer graph acts as a proxy for a set of *vertices in the final output graph*, which may include nodes that exist in the input graph as well as new nodes added to the input graph.

Formally, let τ be $(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$ and $g_c \in \mathbb{G}_c$ be $(\mathsf{V}_c, \mathsf{E}_c, \sigma_c)$. To apply τ to g_c, we first compute a mapping $\eta : \mathsf{EV} \cup \mathsf{IV} \mapsto (\mathsf{IV} \cup \mathsf{V}_c)$, as illustrated in the Appendix. (See [4] for more details.) We define the resulting output graph $\tau\langle g_c \rangle$ to be $(\mathsf{V}', \mathsf{E}', \sigma')$ where $\mathsf{V}' = \mathsf{V}_c \cup \mathsf{IV}$, $\mathsf{E}' = \mathsf{E}_c \cup \{\langle v_1, f, v_2\rangle \mid \langle u, f, v\rangle \in \mathsf{IE}, v_1 \in \eta(u), v_2 \in \eta(v)\}$, and $\sigma' = \lambda x.\hat{\eta}(\sigma(x))$. (Note that the output graph contains concrete and abstract vertices, but can be considered an abstract graph for suitably defined N_a and PV_a.) We define the concretization function $\gamma_T : \mathcal{F}_a \to \mathcal{F}_c$ as follows: $\gamma_T(\tau) = \lambda g_c.\ \gamma_G(\tau\langle g_c\rangle)$.

Define a partial order $\sqsubseteq_{co}$ on $\mathcal{F}_a$ as follows: $\tau_1 \sqsubseteq_{co} \tau_2$ iff $\mathsf{EV}_1 \subseteq \mathsf{EV}_2$, $\mathsf{EE}_1 \subseteq \mathsf{EE}_2$, $\mathsf{IV}_1 \subseteq \mathsf{IV}_2$, $\pi_1 \sqsubseteq \pi_2$, $\mathsf{IE}_1 \subseteq \mathsf{IE}_2$ and $\sigma_1 \sqsubseteq \sigma_2$, where $\sqsubseteq$ for π and σ is defined as pointwise inclusion: $\sigma_1 \sqsubseteq \sigma_2$ iff $\forall x.\sigma_1(x) \subseteq \sigma_2(x)$.

Higher Order Summaries. As explained earlier, we represent abstract higher order summaries as pairs (τ, ω) consisting of a transformer graph τ and a set of (indirect) call-statements ω. Formally, let $\mathit{CallStmt} = \mathit{Vars} \times \mathit{Vars}^*$ denote the set of all indirect call-statements. We define the abstract summary domain $\mathcal{P}_a = \mathcal{F}_a \times 2^{\mathit{CallStmt}}$. We extend $\sqsubseteq_{co}$ to $\mathcal{P}_a$ as follows: $(\tau_1, \omega_1) \sqsubseteq_{co} (\tau_2, \omega_2)$ iff $\tau_1 \sqsubseteq_{co} \tau_2$ and $\omega_1 \subseteq \omega_2$. We define $\mathcal{L}_a$ to be the set $\mathit{Procs} \hookrightarrow \mathcal{P}_a$ (of partial functions from Procs to $\mathcal{P}_a$).

As explained above, a first-order procedure can be summarized using a transformer graph. Now consider a procedure that has no indirect calls, but has statements of the form "$x = \&\mathsf{P}$". A transformer graph τ is sufficient, in this case, to capture the procedure's effect on the graph component of state. However, the procedure's effect also includes updates to the function-table component of the state. In our approach, this effect is captured by the entire library summary (an element of $\mathcal{L}_a$), which also summarizes the effects of procedures (such as "P" above). Thus, the complete meaning of τ (in this case) can be captured only in the context of a library summary $\mathsf{L}_a \in \mathcal{L}_a$. The function $\gamma_{\mathsf{M}} : \mathcal{F}_a \times \mathcal{L}_a \to \mathcal{P}_c$ formalizes this below. The semantics of a higher-order summary $(\tau, \omega) \in \mathcal{P}_a$ is, in turn, formalized by $\gamma_{\mathsf{H}} : \mathcal{P}_a \times \mathcal{L}_a \mapsto \mathcal{P}_c$, as this too is dependent on the entire library summary. Finally, the semantics of a library summary $\mathsf{L}_a \in \mathcal{L}_a$ is formalized by a function $\gamma : \mathcal{L}_a \mapsto \mathcal{L}_c$. These functions are mutually recursive.

Given $f \in \mathcal{P}_c$, we define f^i inductively as $f^0 = \lambda t.\lambda g.\{g\}$ and $f^{i+1} = f^i \circ f$. We define f^* to be $\sqcup_c\{f^i \mid i \geq 0\}$. We define $\gamma_{\mathsf{M}}, \gamma_{\mathsf{H}}, \gamma$ as below.

$$\gamma_{\mathsf{M}}\ (\tau, \mathsf{L}_a) = \lambda(g_c, t_c).\{\ (g'_c, t'_c) \mid \exists h.g_c \preceq_h \tau\langle g_c\rangle \wedge t'_c = t_c \uplus \{\ (n, \gamma(\mathsf{L}_a)(h(n))) \mid h(n) \in \mathrm{dom}(\mathsf{L}_a)\ \}\}$$

$$\gamma_{\mathsf{H}}\ ((\tau, \omega), \mathsf{L}_a) = \gamma_{\mathsf{M}}(\tau, \mathsf{L}_a) \circ \Big(\bigsqcup_c (\{[\![S]\!]_c \circ \gamma_{\mathsf{M}}(\tau, \mathsf{L}_a) \mid S \in \omega\})\Big)^*$$

$$\gamma\ (\mathsf{L}_a) = \{\ (\mathsf{P}, \gamma_{\mathsf{H}}(\mathsf{L}_a(\mathsf{P}), \mathsf{L}_a)) \mid \mathsf{P} \in \mathrm{dom}(\mathsf{L}_a)\ \}$$

The definition of $\gamma(\mathsf{L}_a)$ is straight forward: it maps every procedure P in the library L_a to the concretization of its abstract summary given by $\mathsf{L}_a(\mathsf{P})$. The function $\gamma_{\mathsf{H}}((\tau, \omega), \mathsf{L}_a)$ interprets (τ, ω) as a control flow graph, as shown in Figure 1, and computes the concrete state transformer in $\mathcal{P}_c$ at the exit point of the control flow graph (via a fix-point computation) where, the transfer functions for the edges labelled by τ are given by $\gamma_{\mathsf{M}}(\tau, \mathsf{L}_a)$ and the transfer functions for the edges labelled by the call statements are given by their concrete semantics defined in Figure 2. $\gamma_{\mathsf{M}}(\tau, \mathsf{L}_a)$ is defined as the function in $\mathcal{P}_c$ that maps a concrete state (g_c, t_c) to the set of all concrete states that are

compatible with the abstract graph $\tau\langle g_c\rangle$ and the abstract library $\llcorner_a$. A concrete state (g'_c, t'_c) is compatible with an abstract graph, abstract library pair $(g_a, \llcorner_a)$ iff $g'_c \preceq_h g_a$ and every entry (n, f) in the virtual function table t'_c either belongs to the input table t_c or f is the concrete image of the abstract summary of the procedure $h(n)$. ($\gamma_{\mathsf{M}}(\tau, \llcorner_a)$ assumes that the abstract procedure ids PV_a are procedure names $Procs$.)

5 Abstract Semantics

Let $(\mathsf{LP}, \mathsf{LL})$ be a library, consisting of a set of procedures LP and a set of other libraries LL it links to. Let LLP denote the set of all procedures in LL. Assume that we have analyzed LL and computed summaries for every procedure in LLP. The abstract semantics of $(\mathsf{LP}, \mathsf{LL})$ is captured by an element of $\mathcal{L}_a$ as follows: $[\![(\mathsf{LP}, \mathsf{LL})]\!]_a = \{\ (\mathsf{P}, [\![\mathsf{P}]\!]_a) \mid \mathsf{P} \in \mathsf{LP}\ \}$, where, $[\![\mathsf{P}]\!]_a$ is the value of the variable ϑ_{P} in the least fix point of the collection of abstract semantic equations defined shortly. Define function $\llcorner_s \in \mathcal{L}_a$ with domain $\mathsf{LP} \cup \mathsf{LLP}$ as follows: $\llcorner_s$ maps every $\mathsf{P} \in \mathsf{LP}$ to variable ϑ_{P}, and it maps every $\mathsf{P} \in \mathsf{LLP}$ to its pre-computed summary.

Node Abstraction. First, we fix the set N_a and PV_a. Recall that the domain $\mathcal{F}_a$ defined earlier is parameterized by these sets. We utilize an allocation-site based merging strategy for bounding the size of the transformer graphs. We utilize the labels attached to statements as allocation-site identifiers. Let $Labels$ denote the set of statement labels in the given program. We define N_a to be $\{n_x \mid x \in Labels \cup Params \cup Globals\}$. We define PV_a to be the set $Procs$ of procedure names.

The Sharir-Pnueli Equations. For every procedure $\mathsf{P} \in \mathsf{LP}$, we define the following set of equations, approximating the concrete semantics equations 1-3. We introduce a variable ϑ_u for every vertex u in the control-flow graph of P, and a variable $\vartheta_{u,v}$ for every edge $u \to v$ in the control-flow graph.

$$\vartheta_v = (\mathsf{ID}, \emptyset) \qquad v \text{ is an entry vertex} \tag{5}$$

$$\vartheta_v = \sqcup_{co}\{\vartheta_{u,v} \mid u \xrightarrow{S} v\} \qquad v \text{ is not an entry vertex} \tag{6}$$

$$\vartheta_{u,v} = [\![S]\!]_a(\vartheta_u) \qquad \text{where } u \xrightarrow{S} v \tag{7}$$

$$\vartheta_{\mathsf{P}} = \mathit{simplify}\ \llcorner_s\ \vartheta_{\mathit{exit}(\mathsf{P})} \tag{8}$$

Here, ID is a transformer graph consisting of a external vertex for each global variable and each parameter (representing the identity function). Formally, $\mathsf{ID} = (\mathsf{EV}, \emptyset, \pi, \emptyset, \emptyset, \pi)$, where $\mathsf{EV} = \{n_x \mid x \in Params \cup Globals\}$ and $\pi = \lambda v.\ v \in Params \cup Globals \to \{n_v\} \mid v \in Locals \to \{null\}$.
These equations are straightforward, as they leave the abstraction work to the abstract semantics of statements, explained below. The summary for the procedure, ϑ_{P}, is obtained by simplifying the abstract value $\vartheta_{\mathit{exit}(\mathsf{P})}$ associated with the exit vertex of the procedure as explained later.

Primitive Statements. The abstract semantics $[\![S]\!]_a$ of primitive statements other than call-statements is shown in Fig. 3. Given a set-valued function $f : A \mapsto B$ and $a \in A, b \in B$, we use $f[a \hookrightarrow b]$ to denote a weak update of a i.e, $f[a \hookrightarrow b] = f[a \mapsto$

$$
\begin{array}{|ll|}
\hline
[\![v_1 = v_2]\!]_a(\tau,\omega) & = (\tau[\sigma \mapsto \sigma[v_1 \hookrightarrow \sigma(v_2)]]), \omega) \\
[\![l : v = new\ C]\!]_a(\tau,\omega) & = ((\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV} \cup \{n_l\}, \mathsf{IE} \cup \{n_l \times \mathit{Fields} \times \{\mathit{null}\}\}, \sigma[v \hookrightarrow n_l]), \omega) \\
[\![v_1.f = v_2]\!]_a(\tau,\omega) & = (\tau[\mathsf{IE} \mapsto \mathsf{IE} \cup \{(\sigma(v_1) \setminus \mathsf{PV}_a) \times \{f\} \times \sigma(v_2)\}], \omega) \\
[\![l : v_1 = v_2.f]\!]_a(\tau,\omega) & = \textbf{let}\ A = \{n \mid \exists n_1 \in \sigma(v_2), \langle n_1, f, n\rangle \in \mathsf{IE}\}\ \textbf{in} \\
 & \quad \textbf{let}\ X = (\sigma(v_2) \setminus \mathsf{PV}_a)\ \textbf{in} \\
 & \quad \textbf{let}\ B = X \cap \mathit{Escaping}(\tau)\ \textbf{in} \\
 & \quad \textbf{if}\ (B = \emptyset)\ \textbf{then}\ (\tau[\sigma \mapsto \sigma[v_1 \hookrightarrow A]], \omega) \\
 & \quad \textbf{else} \\
 & \quad ((\mathsf{EV} \cup \{n_l\}, \mathsf{EE} \cup B \times \{f\} \times \{n_l\}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma[v_1 \hookrightarrow A \cup \{n_l\}]), \omega) \\
[\![v = \&\mathsf{P}]\!]_a(\tau,\omega) & = (\tau[\sigma \mapsto \sigma[v_1 \hookrightarrow \{\mathsf{P}\}]], \omega) \\
\hline
\end{array}
$$

Fig. 3. Abstract semantics of primitive statements, where $\tau = (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$

$$
\begin{array}{|ll|}
\hline
\mathit{Call}^\sharp_S((\tau_r,\omega_r),(\tau_e,\omega_e)) & = (\mathit{pop}^\sharp_S(\tau_e \langle\!\langle \mathit{push}^\sharp_S(\tau_r) \rangle\!\rangle_a, \tau_r), \omega_r \cup \omega_e) \\
[\![\mathsf{P}(v_1, \cdots, v_k)]\!]_a(\tau,\omega) & = \mathit{Call}^\sharp_S((\tau,\omega), \mathsf{L}_s(\mathsf{P})) \\
\mathit{MarkParam}(\tau, X) & = \tau[\pi \mapsto \lambda x.\textbf{if}\ x \in X\ \textbf{then}\ \pi(x) \cup \sigma(x)\ \textbf{else}\ \pi(x)] \\
[\![(*v)(v_1,..,v_k)]\!]_a(\tau,\omega) & = (\mathit{MarkParam}(\tau, \{\ v_1,..,v_k\ \} \cup \mathit{Globals}), \omega \cup \{(*v)(v_1,..,v_k)\}) \\
\hline
\end{array}
$$

Fig. 4. Abstract semantics of calls

$f(a) \cup b]$. Given $\tau = (\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma_1)$, let $\tau[\sigma \mapsto \sigma_2]$ denote $(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma_2)$ and we use a similar notation for updating other components of τ. The set $\mathit{Escaping}(\tau)$ used in the semantics of $l : v_1 = v_2.f$ is defined as $\{x \mid \exists w \in \mathit{range}(\pi).\ x$ is reachable from w via $\mathsf{EE} \cup \mathsf{IE}$ edges $\}$. Our abstract semantics closely resembles the one used in [4], with one difference. Unlike in the earlier analysis, we perform weak updates on all the variables (to conservatively over-approximate the transformers of all the code segments between the indirect calls). In our implementation, we minimize the precision loss due to weak updates via variable renaming.

Call Statements. Fig. 4 presents the abstract semantics of call statements. The abstract semantics of a direct call statement utilizes the function L_s defined earlier, which maps every $\mathsf{P} \in \mathsf{LP}$ to variable ϑ_P, and every $\mathsf{P} \in \mathsf{LLP}$ to its pre-computed summary. The operation $\mathit{Call}^\sharp(,)$ composes the graph transformer τ_r before the call-site with the graph transformer τ_e of the callee's summary, to find the resultant graph transformer. The ω component is obtained simply by taking the union of the set of indirect calls in the caller and callee. We use operations $\mathit{push}^\sharp_S$ and $\mathit{pop}^\sharp_S$ as abstractions of the parameter passing mechanism and pushing/popping an activation record. These operations, defined in the Appendix, are straightforward, except for one point: $\mathit{pop}^\sharp_S$ updates local variables of the caller *weakly*, defining their value as the join of their original value (in the caller) and their final value (in the callee). This is done so that variables referred to in indirect call-sites from the callee's summary that are added to the caller can be interpreted soundly.

The definition of $\langle\!\langle\rangle\!\rangle$ is analogous to the definition of the $\langle\rangle$ operator used to define the concretization function. While $\tau\langle g\rangle$ models *relation application* (it returns a representation of all graphs related to g by τ), $\tau_1 \langle\!\langle \tau_2 \rangle\!\rangle_a$ models *relation composition*. A formal definition of this operation appears in the appendix. When a callee's summary is instantiated at a call-site as above, we may be able to *resolve* some of the indirect calls

in the callee (i.e., determine the actual targets of these calls). The procedure *simplify* is used to perform such resolution and simplify the result, as explained later.

The semantics of indirect call statements is mostly straightforward: the statement is simply added to the list of unresolved calls. However, the transformer graph is updated as indicated by function $MarkParam$. Recall that we wish to construct a transformer graph that simultaneously approximates multiple code fragments, each starting/ending at an indirect-call, entry vertex, or exit vertex. The code fragment starting after the given indirect call may be thought of as having parameters $\{ v_1, \cdots, v_k \} \cup Globals$: these are the roots of part of the caller's heap that is accessible to and may be modified by the callee. Hence, nodes pointed to by these variables are marked as parameter nodes.

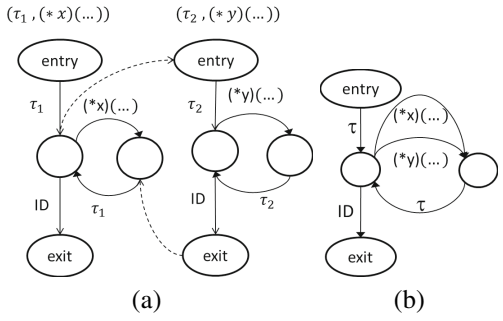

Fig. 5. Resolving an indirect call

Resolving Indirect Calls. We now describe how a summary is simplified when an indirect call is resolved. Consider the scenario shown in Fig. 5(a). Let $(\tau_1, \{(*x)(\cdots)\})$ be a summary computed during the analysis. Suppose the possible values of x includes a procedure P (i.e., P $\in \sigma_{\tau_1}(x)$). Then, the indirect call in the summary may invoke P. Let $(\tau_2, \{(*y)(\cdots)\})$ be the summary of P (either a precomputed summary or a partially-computed summary if P is part of the analysis scope). Fig. 5(a) shows the combined control-flow graph we get from the two summaries. The goal of the resolution process is to simplify, via abstraction, this control-flow graph to one in normal form, as shown in Fig. 5(b). A couple of points are worth noting about the summary shown in Fig. 5(b). Firstly, the original indirect call instruction $(*x)(\cdots)$ is still present in the summary. This cannot be dropped until all possible targets of x have been determined and instantiated (as detailed later). Secondly, the indirect calls $(*x)(\cdots)$ and $(*y)(\cdots)$ are treated as if they are indirect calls of the summarized method. In reality, the second is an indirect call in a target of the first.

The above procedure can be generalized to the case of summaries with multiple indirect calls. In general, instantiating a summary can trigger further summary instantiations: e.g., sufficient context information may become available to resolve other indirect calls. Hence, the resolution process is an iterative one of identifying indirect calls that can be resolved and then instantiating summaries of identified targets. The operation $inline : \mathcal{L}_a \mapsto (\mathcal{P}_a \mapsto \mathcal{P}_a)$ that realizes this iterative procedure is defined in Fig. 6.

The function $inlineCall$ (in Fig. 6) performs the inlining operation for a single indirect call as illustrated in Fig. 5, which involves a fix-point computation. Notice the cycle in the control flow graph in Fig. 5(a) passing though the edges labelled τ_1 and τ_2, the transformer graph τ shown in Fig. 5(b) is the fixed point of this cycle; $inlineCall$ computes this fixed point. The functions $inlineOnce$ (and $inlineCalls$) extend the $inlineCall$ operation to a set of indirect calls by applying it sequentially on every resolvable call in the input summary. (An indirect call is resolvable if its

$$
\begin{array}{l}
\mathit{lfp}\ f\ v = \text{if } (f\ v) = v \text{ then } v \text{ else } \mathit{lfp}\ f\ (f\ v) \\
\mathit{inlineCall}\ ((\tau_e, \omega_e), S)\ (\tau_r, \omega_r) = \\
\quad \text{let } \tau_1 = \mathit{Call}^\sharp_S((\tau_r, \omega_r), (\tau_e, \omega_e)) \text{ in} \\
\quad \text{let } \tau_2 = \tau_r \langle\!\langle \tau_1 \rangle\!\rangle_a \text{ in} \\
\quad \text{let } \tau_3 = \mathit{lfp}\ (\lambda f. f \langle\!\langle f \rangle\!\rangle_a)\ \tau_2 \text{ in} \\
\quad (\tau_3 \langle\!\langle \tau_r \rangle\!\rangle_a, \omega_r \cup \omega_e) \\
\mathit{inlineCalls}\ \Sigma\ \psi = \begin{cases} \psi & \text{if } \Sigma = \{\} \\ \mathit{inlineCalls}\ \Sigma'\ (\mathit{inlineCall}\ x\ \psi) & \text{if } \Sigma = \{x\} \uplus \Sigma' \end{cases} \\
\mathit{inlineOnce}\ L_a\ (\tau, \omega) = \\
\quad \text{let } \Sigma = \{(L_a(\mathsf{P}), c) \mid c \in \omega \wedge c = (*v)(v_1, \cdots, v_k) \wedge \mathsf{P} \in \sigma(v) \cap \mathrm{dom}(L_a)\} \text{ in} \\
\quad \mathit{inlineCalls}\ \Sigma\ (\tau, \omega) \\
\mathit{inline}\ L_a\ \psi = \mathit{lfp}\ (\lambda \psi'.\ \mathit{inlineOnce}\ L_a\ \psi')\ \psi
\end{array}
$$

Fig. 6. Definition of *inline*

target variable points to (abstract) procedure ids.) However, applying *inlineOnce* function may result in more resolvable indirect calls. Moreover, the summaries inlined by *inlineOnce* could be mutually inter-dependent (if the procedures they correspond to are mutually recursive in the context of the input summary). Both these cases are uniformly handled by the *inline* function which repeatedly applies the *inlineOnce* operation until a fixed point.

Eliminating Calls. Once all targets of an indirect call $(*x)(\cdots)$ have been identified and their summaries instantiated, the call can be omitted from the summary. Transformer graphs use external nodes to represent *unresolved values*: e.g., input parameters. If x does not point to any external node, then all possible values of x are known. However, the converse is not true: even if x points to an external node, all possible values of x may already have been determined, as illustrated by the example in Fig. 7.

```
R () {
r = new T(); r.f = &P;
x = new T(); x.f = &Q;
while(*) {
 t1 = r.f; (*t1)(x);
 t2 = x.f; (*t2)(r);
} }
```

Fig. 7. Example program

The indirect calls in lines 4 and 5 can potentially call procedures P and Q. However, these indirect calls could also potentially update the values of $x.f$ or $r.f$, thus changing the procedures that are called in subsequent executions of these statements. The transformer graphs correctly account for this possibility by creating external nodes (that represent the *updated* values of $x.f$ and $r.f$ after these indirect calls). However, assume that procedures P and Q do *not* update the values of $x.f$ and $r.f$. Once the summaries of P and Q are instantiated in the summary of R, we can determine that no new values are possible for $t1$ and $t2$, even though they point to external nodes, and that all possible targets of these indirect calls have been instantiated. We can, hence, eliminate these indirect calls from the summary. The algorithm in Fig. 8 iteratively identifies potentially unresolved calls and eliminates the other calls. This elimination creates opportunities to identify and eliminate useless external nodes. Due to space constraints, we do not describe how this is done.

$$
\begin{array}{l}
\mathit{dropResolvedCalls}\ ((\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma), \omega)\ = \\
\quad \text{let } \mathit{reach}(X) = \{y \mid \exists x \in X.\ y \text{ is reachable from } x \text{ via. } \mathsf{IE} \cup \mathsf{EE} \text{ edges } \} \\
\quad \text{let } e_{init} = \mathit{reach}(\cup\{\ \pi(x) \mid x \in \mathit{Params} \cup \mathit{Globals}\ \}) \\
\quad \text{let } e_m((*x)(a_1, \ldots, a_k)) = \mathit{reach}(\pi(a_1)) \cup \ldots \cup \mathit{reach}(\pi(a_k)) \\
\quad \text{let } \mathit{unresolved} = \mathit{lfp}\ (\lambda X.\{(*x)(a_1, \ldots, a_k) \in \omega \mid (\sigma(x) \cap \mathsf{EV}) \in \widehat{e_m}(X) \cup e_{init}\})\ \emptyset \\
\quad ((\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma), \mathit{unresolved}) \\
\mathit{simplify}\ L_a\ \psi\ = \mathit{dropResolvedCalls}(\mathit{inline}\ L_a\ \psi)
\end{array}
$$

Fig. 8. The Simplification Procedure

Other Optimizations and Details. Our analysis computes the fixed point of a (large) collection of equations generated from a given library. Similar to a conventional modular analysis, we analyze each procedure one at a time. Typically, in a modular analysis, the *dependences* between the equations can be identified statically and guide the order in which equations are processed for fixed point computation (which generally is a bottom-up or reverse topological order of the call-graph). Indirect calls, however, mean that some of the dependences can only be identified during the course of the analysis making it impossible to devise an optimal order of processing. We use a combination of an initial approximate call-graph constructed using class hierarchy analysis and call-graph edges identified dynamically during our analysis to guide the order in which procedures are iteratively analyzed. We also exploit an optimization to identify and merge *equivalent* call statements: $(*a_0)(a_1, \cdots, a_k)$ and $(*b_0)(b_1, \cdots, b_k)$ are equivalent if the abstract values of a_i and b_i are the same for every i. Finally, at the exit point of each method (after the *simplify* operation) we remove the internal/external vertices not reachable from $\mathit{Params}, \mathit{Globals}$ and the arguments of unresolved indirect calls from the method summary (analogous to *garbage collection*). We omit details of several other optimizations due to space constraints.

Correctness. We say that a concrete value $\llcorner_c \in \mathcal{L}_c$ is *correctly represented* by an abstract value $\llcorner_a \in \mathcal{L}_a$, denoted $\llcorner_c \sim \llcorner_a$, iff $\llcorner_c \sqsubseteq_c \gamma(\llcorner_a)$, and similarly for the other domains as well.

Theorem 1. $[\![(\mathsf{LP}, \mathsf{LL})]\!]_c \sim [\![(\mathsf{LP}, \mathsf{LL})]\!]_a$.

6 Experimental Evaluation

We have implemented a flow-insensitive version of our analysis for C♯ using the *Microsoft Phoenix framework*. Our implementation, referred to as *SEAL* (for Side-Effects Analysis), is available at *http://www.rise4fun.com/seal*. SEAL is reasonably well tested, with over 50 testcases, many using higher-order features of C♯ such as delegates and LINQ. However, SEAL does not currently handle reflection and concurrency.

Fig. 9 shows the benchmarks used in our empirical evaluation along with their source code sizes. All benchmarks except *System.Core* which is a part of the .NET framework, are popular open source libraries from *http://www.codeplex.com*. We analysed each benchmark using the pre-computed summaries for parts of the .NET framework, namely,

Benchmark	LOC	*Methods*	*Pure*	*Cond. Pure*	*Impure*	*Impure & incomp*	*Time*
DocX *(dx)*	10K	612	285	89	61	177	1m17s
Facebook APIs *(fb)*	21K	4112	1886	91	1336	799	1m59s
Dynamic Data Display *(ddd)*	25K	2266	1285	334	258	389	3m58s
TestApis *(test)*	25K	1080	503	205	189	183	2m50s
Newtonsoft Json *(json)*	27K	1867	675	532	234	426	27m34s
Quickgraph *(qg)*	34K	3380	1703	653	628	396	1m50s
NRefactory *(nr)*	43K	3004	998	1036	262	708	21m49s
CUL *(cul)*	56K	3963	1519	1275	855	314	5m13s
PdfSharp *(pdf)*	96K	3883	1405	344	1031	1103	9m53s
DotSpatial *(ds)*	250K	11579	4656	2718	1737	2468	1h51m2s
System.core *(sys)*	unknown	3092	1190	752	445	705	11m28s

Fig. 9. Results of running SEAL on 11 benchmark programs. On all the benchmarks, SEAL used at most 4GB of memory.

mscorlib, *system* and *system.core* DLLs. Unlike a typical whole-program analysis that would (re) analyse the .NET DLLs while analysing every benchmark, SEAL analyses the .NET DLLs once and reuses their summaries during the subsequent analyses. Furthermore, *DotSpatial* consists of 7 inter-dependent DLLs which were analysed one at a time in a modular fashion (the numbers presented are the aggregate of all the DLLs).

Except in the case of a few commonly used methods (like `System.Array` members) for which we used manually written stubs, we treated calls to methods for which code was unavailable (such as native, GUI and database libraries) heuristically . Hence, our analysis could be unsound in the presence of such calls.

Performance and Purity Classification. SEAL classifies every method into 4 categories. A *pure* method does not have any externally visible side-effects and does not have any unresolved calls. A *conditionally pure* method has no side-effects but has one or more unresolved calls and hence its purity depends on the calling-context. An *impure* method has side-effects but has no unresolved calls. An *impure & incomplete* method has side-effects and unresolved calls.

Fig. 9 shows the results of running SEAL on our benchmarks on a 2.83 GHz, 4 core, Intel Xeon server running Windows Server 2008. We observe that SEAL scales to large, real world C♯ libraries with thousands of methods within reasonable time and memory overhead. Also observe that there exists a significant number of procedures whose purity and side-effects depends on unresolved calls, highlighting the need for a sound and precise treatment of call-backs.

Fig. 10 presents statistics that provide interesting insights into the analysis. The first column in Fig. 10 shows the average number of unresolved calls in the summary of a method, i.e., the size of the ω component of the summaries (the absolute deviation, i.e., the average of differences of the each of the values from mean is shown within parenthesis). It can be seen that, across all benchmarks, SEAL finds at least 2 unresolved calls per method on an average. In fact, many methods have many more unresolved calls, as indicated by the large absolute deviation. (up to 7 unresolved calls per method on average in *json* and *sys*).

Bench-mark	Unresolved calls	Completely resolved calls	Non-escaping internal nodes
dx	4.05 (5.42)	7% (10%)	33% (36%)
fb	2.55 (4.07)	6% (10%)	9% (15%)
ddd	2.10 (3.22)	1% (2%)	30% (37%)
test	2.52 (3.61)	5% (9%)	27% (34%)
json	7.32 (10.61)	6% (9%)	31% (35%)
qg	2.06 (3.13)	1% (3%)	10% (17%)
nr	4.04 (5.04)	1% (2%)	24% (32%)
cul	2.14 (2.84)	6% (11%)	19% (28%)
pdf	3.50 (5.13)	2% (3%)	37% (34%)
sys	6.87 (10.42)	4% (7%)	41% (35%)
ds	5.93 (8.77)	3% (5%)	10% (11%)

Fig. 10. Prevalence of unresolved calls and utility of $dropResolvedCalls$/ garbage collection

The second column of Fig. 10 shows the average percentage of indirect calls in *unsimplified* method summaries that are classified as completely resolved (and removed) by $simplify$. The third column shows the average percentage of internal nodes allocated by a method (and its callees) that are non-escaping. This shows that in spite of unresolved calls, the analysis is able to identify a significant percentage (25% on average) of locally allocated objects as non-escaping and eliminate them from the summaries.

A Comparison with CHA Callgraph Based Modular Analysis. We now compare SEAL with an alternative call-graph-based compositional heap analysis, which we refer to as CCC. CCC works by first constructing a call-graph using *Class Hierarchy Analysis (CHA)*. It then processes procedures in bottom-up order over this call-graph, using our first-order compositional heap analysis technique [4].

Our implementation of CCC is unsound for reasons explained below. However, our intention is solely to use the reported numbers as an upper bound for precision and lower bound for analysis time for a sound version of CCC. Our CCC implementation constructs the call-graph of libraries independent of the application (or client), which is potentially unsound due to callbacks. We exclude *DotSpatial* from this experiment due to the complications in constructing a reasonably sound call-graph spanning multiple DLLs. We found that conservatively modelling calls to virtual methods like *equals* and *hashCode* (defined in the root class *Object*) as dispatching to any of their overridden implementations doesn't scale to even a 10 line program within reasonable time limits when the referenced libraries are also included. For this reason, CCC heuristically (and unsoundly) treats calls to such top-level interface methods (as having no side-effect). (In contrast, SEAL does not resort to any such heuristics.)

The results in Fig. 11 show that CCC is dramatically slower than SEAL. The table includes statistics about the call-graphs in the two cases which suggest that the performance difference is likely due to the imprecision of the CCC call-graph. The SEAL call-graph is a bit different from a conventional static call-graph, as it includes some (but not all) transitive caller-callee relationships because of the way it inlines summaries. However, these numbers capture (in both cases) the dependences that exist between the summaries of different procedures. These numbers indicates that decoupling call-graph construction from the heap analysis leads to over-estimating these dependences and larger SCCs, which make the analysis slower and make a case for integrating the call-graph construction with the heap analysis, as we do in a compositional fashion.

	SEAL	*CCC*	*CCC call-graph*				*SEAL call-graph*			
	Time	Time	*#Edges*	*#SCCs*	*Avg SCC Size*	*Max SCC Size*	*#Edges*	*#SCCs*	*Avg SCC Size*	*Max SCC Size*
dx	1m17s	12m52s	684	0	NA	NA	1273	0	NA	NA
fb	1m59s	23m13s	4052	3	3.33	4	4090	1	2	2
ddd	3m58s	∞	9105	6	18.17	99	3666	1	2	2
test	2m50s	16m7s	2532	13	5	25	1891	7	4	7
json	27m34s	∞	10701	18	28.06	450	13033	8	4.63	12
qg	1m50s	∞	296982	11	66.73	658	3416	1	2	2
nr	21m49s	∞	20763	14	79.43	911	10976	10	14.4	55
cul	5m13s	2h34m12s	34231	11	35.82	354	4740	3	2.67	3
pdf	9m53s	23m31s	7339	21	3.62	19	14434	6	2.33	3
sys	11m28s	3h44m55s	56712	10	58.30	508	7292	11	8.45	45

Fig. 11. Comparison of SEAL and CCC. ∞ indicates timeout after 4 hours.

7 Related Work

This paper extends our previous work [4,5] on compositional heap analysis for first-order procedures. The problem of resolving indirect calls has attracted a lot of attention, ranging from various call-graph construction algorithms for object-oriented languages to control-flow analysis algorithms for functional languages, e.g., see [9,2,6]. Many of these algorithms, however, take a top-down, whole-program, analysis approach. In contrast, we have focused on a compositional, bottom-up, approach that can be used to compute summaries for libraries that can be reused for any client of the library.

Rountev *et al.* [7] present a framework for modular analysis of libraries in the presence of call-backs by extending Sharir and Pnueli's functional approach. For procedures containing indirect calls (directly or transitively), their approach constructs a simplified control flow graph as a (higher-order) summary, by simplifying paths that contain only direct calls to procedures that have a first-order summary to an edge labelled by its transformer. This is similar to the starting point of our approach, but we push this approach further. We show how to inline a higher-order summary at a call-site, simplify the resulting summary, resolve indirect calls when possible, and integrate a heap analysis within this approach. (E.g., they rely on other, separate, analyses to identify targets of indirect calls when a library's summary is instantiated in the context of a client.)

Vivien *et al.* [10] present an approach for analyzing an arbitrary set of procedures in a complete program. Their approach permits the summary computed for a procedure to be incrementally refined using the summaries of callees when they become available. However, this approach does not handle indirect calls, and assumes that a call-graph is available. In contrast, we deal with indirect calls and callbacks, and construct a call-graph during the analysis in a compositional fashion. Furthermore, the approach doesn't have a theoretical formalization or proof of correctness. We believe that our abstract interpretation formalization can be easily adapted to express Vivien *et al.*'s approach.

Lattner *et al.* [3] present a modular unification-based pointer analysis for C programs in the presence of function pointers. Our approach shares several elements with the Lattner *et al.* approach, most notably combining a (first-order) transformer with

a set of unresolved calls into a summary, but we use a more precise (non-unification) pointer analysis. [3] does not simplify summaries as aggressively as we do, does not explain identification/elimination of completely resolved calls, does not have an abstract interpretation formulation and is quite complex. We believe that our formalization can be adapted with minor modifications to formalize Lattner *et al.* analysis. An interesting aspect of [3] is the use of a context-sensitive heap abstraction (or heap cloning). Conceptually, it is straight forward to incorporate heap cloning into our analysis by altering the definition of N_a and the abstract semantics of call statements, in fact, our implementation (SEAL) supports heap cloning. However, it has far reaching implications on the precision and scalability of the analysis; initial evaluations indicate a dramatic increase in the sizes of the transformer graphs and the number of unresolved indirect calls. In the future, we plan to investigate ways of efficiently incorporating heap cloning into our analysis.

References

1. Cousot, P., Cousot, R.: Modular Static Program Analysis. In: CC 2002. LNCS, vol. 2304, pp. 159–179. Springer, Heidelberg (2002)
2. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: OOPSLA, pp. 108–124 (1997)
3. Lattner, C., Lenharth, A., Adve, V.S.: Making context-sensitive points-to analysis with heap cloning practical for the real world. In: PLDI, pp. 278–289 (2007)
4. Madhavan, R., Ramalingam, G., Vaswani, K.: Purity Analysis: An Abstract Interpretation Formulation. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 7–24. Springer, Heidelberg (2011)
5. Madhavan, R., Ramalingam, G., Vaswani, K.: Purity analysis: An abstract interpretation formulation. Tech. rep., Microsoft Research (2011)
6. Might, M., Smaragdakis, Y., Horn, D.V.: Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis. In: PLDI, Toronto, Canada, pp. 305–315 (June 2010)
7. Rountev, A., Kagan, S., Marlowe, T.: Interprocedural Dataflow Analysis in the Presence of Large Libraries. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 2–16. Springer, Heidelberg (2006)
8. Sălcianu, A., Rinard, M.: Purity and Side Effect Analysis for Java Programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005)
9. Shivers, O.G.: Control-Flow Analysis of Higher-Order Languages or Taming Lambda. Ph.D. thesis, Carnegie-Mellon Univeristy (May 1991)
10. Vivien, F., Rinard, M.: Incrementalized pointer and escape analysis. In: PLDI, pp. 35–46 (2001)
11. Whaley, J., Rinard, M.C.: Compositional pointer and escape analysis for java programs. In: OOPSLA, pp. 187–206 (1999)

A Appendix

Definition of $Call_S$**.** Let S be a procedure call statement with arguments a_1,...,a_k. Let $Param(i)$ denote the i-the formal parameter. Define the functions $push_S \in \Sigma_c \mapsto \Sigma_c$,

$pop_S \in \Sigma_c \times \Sigma_c \mapsto \Sigma_c$, and $Call_S$ as follows:

$$push_S(\sigma) = \lambda v.\ v \in Globals \rightarrow \sigma(v) \mid v \in Locals \rightarrow null \mid v = Param(i) \rightarrow \sigma(a_i)$$
$$pop_S(\sigma, \sigma') = \lambda v.\ v \in Globals \rightarrow \sigma'(v) \mid v \in Locals \cup Params \rightarrow \sigma(v)$$
$$Call_S(f) = \lambda((\mathsf{V}, \mathsf{E}, \sigma), t).\{\ ((\mathsf{V}', \mathsf{E}', pop_S(\sigma, \sigma')), t') \mid ((\mathsf{V}', \mathsf{E}', \sigma'), t') \in f\ ((\mathsf{V}, \mathsf{E}, push_S(\sigma)), t)\ \}$$

Definition of η**.** Let $\tau \in \mathcal{F}_a$ be $(\mathsf{EV}, \mathsf{EE}, \pi, \mathsf{IV}, \mathsf{IE}, \sigma)$ and $g_c \in \mathbb{G}_c$ be $(\mathsf{V}_c, \mathsf{E}_c, \sigma_c)$. The function η used in the definition of $\tau\langle g_c \rangle$ is defined as follows. Define a mapping $\eta : (\mathsf{EV} \cup \mathsf{IV} \cup \mathsf{PV}_a) \mapsto (\mathsf{IV} \cup \mathsf{V}_c)$ that maps every vertex (and procedure ids) in τ to a set of values, as follows: Let $Escaping(\tau) = \{x \mid \exists w \in range(\pi).\ x$ is reachable from w via $\mathsf{EE} \cup \mathsf{IE}$ edges $\}$.

$$v \in \pi(x) \Rightarrow \sigma_c(x) \subseteq \mu(v)$$
$$v \in \mathsf{IV} \cup \mathsf{PV}_a \Rightarrow v \in \mu(v)$$
$$\langle u, f, v\rangle \in \mathsf{EE}, u' \in \mu(u), \langle u', f, v'\rangle \in \mathsf{E}_c \Rightarrow v' \in \mu(v)$$
$$\langle u, f, v\rangle \in \mathsf{EE}, (\mu(u) \cap \mu(u') \setminus \mathsf{PV}_c \setminus \mathsf{PV}_a) \neq \emptyset, \langle u', f, v'\rangle \in \mathsf{IE}, u \in Escaping(\tau) \Rightarrow \mu(v') \subseteq \mu(v)$$

Definition of $push^\sharp_S$ ***and*** $pop^\sharp_S$**.** Let S be a direct/indirect call statement with arguments $a_1, \ldots, a_n$ and $\tau_1 = (\mathsf{EV}_1, \mathsf{EE}_1, \pi_1, \mathsf{IV}_1, \mathsf{IE}_1, \sigma_1)$, $\tau_2 = (\mathsf{EV}_2, \mathsf{EE}_2, \pi_2, \mathsf{IV}_2, \mathsf{IE}_2, \sigma_2)$ $push^\sharp_S(\tau_1) = (\mathsf{EV}_1, \mathsf{EE}_1, \pi_1, \mathsf{IV}_1, \mathsf{IE}_1, \sigma'_1)$ and $pop^\sharp_S(\tau_2, \tau_1) = (\mathsf{EV}_2, \mathsf{EE}_2, \pi_2, \mathsf{IV}_2, \mathsf{IE}_2, \sigma'_2)$ where, $\sigma'_1 = \lambda v.\ (v = Param(i) \rightarrow \sigma_1(a_i) \mid v \in Globals \rightarrow \sigma_1(v) \mid v \in Locals \rightarrow null)$ and $\sigma'_2 = \lambda v.\ (v \in Params \cup Locals \rightarrow \sigma_1(v) \cup \sigma_2(v) \mid v \in Globals \rightarrow \sigma_2(v))$

Definition of Relational Composition Operator $\langle\langle\rangle\rangle$**.** Let $\tau_1 = (\mathsf{EV}_1, \mathsf{EE}_1, \pi_1, \mathsf{IV}_1, \mathsf{IE}_1, \sigma_1)$, $\tau_2 = (\mathsf{EV}_2, \mathsf{EE}_2, \pi_2, \mathsf{IV}_2, \mathsf{IE}_2, \sigma_2)$. We define $\tau_2\langle\langle\tau_1\rangle\rangle_a$ to be $\tau_2\langle\langle\tau_1, \eta_a\rangle\rangle$, where η_a is the least solution of the following set of constraints over the variable μ_a.

$$u \in \pi_2(p) \Rightarrow \sigma_1(p) \subseteq \mu_a(u)$$
$$u \in (\mathsf{IV}_2 \cup \mathsf{PV}_a) \Rightarrow u \in \mu_a(u)$$
$$\langle u, f, v\rangle \in \mathsf{EE}_2, u' \in \mu_a(u), \langle u', f, v'\rangle \in \mathsf{IE}_1 \Rightarrow v' \in \mu_a(v)$$
$$\langle u, f, v\rangle \in \mathsf{EE}_2, (\mu_a(u) \cap \mu_a(u') \setminus \mathsf{PV}_a) \neq \{\}, \langle u', f, v'\rangle \in \mathsf{IE}_2 \Rightarrow \mu_a(v') \subseteq \mu_a(v)$$
$$\langle u, f, v\rangle \in \mathsf{EE}_2, \mu_a(u) \cap Escaping(\tau_2\langle\langle\tau_1, \mu_a\rangle\rangle) \neq \{\} \Rightarrow v \in \mu_a(v)$$

Define $\tau_2\langle\langle\tau_1, \nu\rangle\rangle$ to be $\tau' = (\mathsf{V}' \cap (\mathsf{IV}_1 \cup \mathsf{IV}_2), \mathsf{EE}', \pi', \mathsf{V}' \cap (\mathsf{EV}_1 \cup \mathsf{EV}_2), \mathsf{IE}', \sigma')$ where,

$$\mathsf{V}' = (\mathsf{IV}_1 \cup \mathsf{EV}_1) \cup \hat{\nu}(\mathsf{IV}_2 \cup \mathsf{EV}_2)$$
$$\mathsf{IE}' = \mathsf{IE}_1 \cup \{\langle v_1, f, v_2\rangle \mid \langle u, f, v\rangle \in \mathsf{IE}_2, v_1 \in \nu(u) \setminus \mathsf{PV}_a, v_2 \in \nu(v)\}$$
$$\mathsf{EE}' = \mathsf{EE}_1 \cup \{\langle u', f, v\rangle \mid \langle u, f, v\rangle \in \mathsf{EE}_2, u' \in \nu(u), u' \in Escaping(\tau')\}$$
$$\pi' = \lambda var.\ \pi_1(var) \cup \hat{\nu}(\pi_2(var))$$
$$\sigma' = \lambda var.\ \sigma_1(var) \cup \hat{\nu}(\sigma_2(var))$$