Modular Analysis of Executables using On-Demand Heyting Completion

Julian Kranz[1] and Axel Simon[2]

[1] Technische Universität München, Garching b. München, Germany, julian.kranz@in.tum.de
[2] Google Inc., Mountain View, CA, USA, axelsimon@google.com

Abstract A function-modular analysis is presented that computes precise function summaries in the presence of pointers and indirect calls. Our approach computes several summaries for a function, each specialized to a particular input property. A call site combines the effect of several summaries, based on what properties hold. The key novelty is that the properties are tailored to the function being analyzed. Moreover, they are represented in a domain-agnostic way by using Herbrand terms with variables. Callers instantiate these variables, based on their state. For each variable instantiation, a new summary is computed. Since the computed summaries are exact with respect to the property, our fixpoint computation resembles the process of Heyting completion where a domain is iteratively refined to be complete wrt. the intersection with a property. Our approach combines the advantages of a modular analysis, such as scalability and context-sensitivity, with the ability to compute meaningful summaries for functions that call other functions via pointers that were passed as arguments. We illustrate our framework in the context of inferring indirect callees in x86 executables.

Keywords: executable analysis, modular analysis, domain refinement

1 Introduction

One challenge in static analysis is the sheer size of the input program. This is particularly true for the analysis of executables that have easily an order of magnitude more statements than the corresponding source program. One key to scalability is the treatment of functions: On the one hand, the highest precision needed to prove the absence of run-time errors [2] can be obtained by inlining functions at each call-site with the cost of increasing the code to be analyzed dramatically. On the other hand, the duplicate evaluation of code can be avoided by performing a context-insensitive analysis in which all calling contexts of a function are merged and the return state is propagated to all call sites. A context-sensitive analysis without duplicate evaluation of functions can be obtained by inferring an input/output relation for each function. These function summaries are then combined to obtain a solution to the whole program using a global fixpoint computation. This approach is known as modular analysis [6].

```
struct Parity {
  virtual bool IsEven() = 0;
  virtual bool IsOdd() = 0;
};
struct Even : public Parity {
  bool IsEven() { return true;}
  bool IsOdd() { return false;}
};
struct Odd : public Parity {
  bool IsEven() {
    even_call++; return false;}
  bool IsOdd() { return true;}
  int even_call = 0;
};

void CheckEven() {
  Even even;
  Check(&even);
}
void CheckOdd() {
  Odd odd;
  Check(&odd);
  assert(odd.even_call > 0);
}
void Check(Parity* parity) {
  assert(parity->IsEven()
      != parity->IsOdd());
}
```

Figure 1. The running example C++ program.

We illustrate the challenges of a modular analysis using the code in Fig. 1. Here, the tests `CheckEven` and `CheckOdd` rely on the helper function `Check` to test an invariant of the two sub-classes `Even` and `Odd`. In a modular analysis, the methods `Even::IsEven` and `Even::IsOdd` are summarized by their effect of returning a constant value. The `Odd::IsEven` method modifies the `even_call` field pointed-to by `this`. A summary for this method must therefore assume the existence of a memory region at `*this` containing an **`int`** field. A precise summary of this function can be expressed by $x' = x + 1$ where x, x' is the value of the field before, resp. after, the call. A more challenging task is the summary of `Check`. Invoking the virtual methods accessed through the `parity` pointer amounts to an indirect function call. Without knowing which functions can be dispatched to, a summary of this function would have to make worst case assumptions: the invoked function may modify any memory reachable from global variables or the `this` pointer. Without any additional information, a summary of a function f_i containing indirect calls provides little or no information.

One way to ensure that no precision loss occurs, even in the presence of higher-order functions, is to limit the precision of the analysis up front. Specifically, by using only abstract domains that are able to condense the effect of a function without loss of precision, it is possible to compute a summary of a function even if it takes other functions as parameters (examples are type inference for functional programs [20], groundness analysis in Prolog [14] and instances of the IFDS framework [16]). These so-called condensing domains [9] are too imprecise to distinguish function behaviors based on pointer aliasing and numeric properties.

One particular kind of condensing domains are those whose meet distributes over the join of the lattice, i.e. $s \sqcap (t \sqcup u) = (s \sqcap t) \sqcup (s \sqcap u)$. Giacobazzi and Scozzari propose Heyting completion to make an existing domain meet-distributive [10]. This process adds new elements to a domain and may thereby refine an abstract domain until it is isomorphic to the concrete domain (which is a set of states and thus forms a distributive lattice). Heyting completion is therefore not generally

practical. In this paper, we use Heyting completion on-demand, namely when the analysis of a function requires it to avoid a severe loss of precision. In particular, once a particular property p is identified for which we want to avoid the lossy approximation $\{p\} \sqcap (s \sqcup t) \sqsupset (\{p\} \sqcap s) \sqcup (\{p\} \sqcap t)$, we track a new abstract state $p \to (\{p\} \sqcap s) \sqcup (\{p\} \sqcap t)$ and postpone the computation of a state in which p does not hold until a call-site is encountered that requires it. Ultimately, a function is summarized by a table $[p_1 \mapsto \{p_1\} \sqcap s_1, \ldots p_n \mapsto \{p_n\} \sqcap s_n]$ and a call site c applies this summary by computing $\bigsqcup_i c \sqcap \{p_i\} \sqcap s_i$. We present an analysis whose predicates p state that an input to a function is equal to a function address. For instance, analyzing `CheckEven` creates a summary s_E of `Check` and stores the mapping (`parity->vtable[0]` $= a_E) \mapsto s_E$ where a_E is the address of `Even::IsEven`. A second summary of `Check` is created for the call site in `CheckOdd`. A call site such as `Check(rand() ? new Odd() : new Even())` can thereafter be evaluated by instantiating the two summaries and without re-analyzing `Check`.

Given a function with the predicated summary $[p_1 \mapsto s_1, \ldots p_n \mapsto s_n]$ and a call-site with state c, the question arises if the predicates cover the state described by c, i.e. if $\gamma(c) \subseteq \gamma(p_1) \cup \ldots \cup \gamma(p_n)$. If not, new predicates $p_{n+1}, \ldots p_{n+k}$ must be identified and a new summary must be computed for each predicate. For instance, calling `Check` with a new sub-class `Mark` whose method `Mark::IsEven` has address a_M, the computation of a new summary s_M of `Check` is needed, giving the table entry (`parity->vtable[0]` $= a_M) \mapsto s_M$. The challenge here is how to observe when a new predicate is needed and how to obtain it. Our contribution to this end is to represent predicates as a Herbrand abstraction (uninterpreted terms with variables as placeholder for other terms) which gives the analyzer the flexibility to express cross-cutting properties from several abstract domains. By evaluating these predicates wrt. a call-site state, the variables in the predicates will be instantiated with values that make the predicate true. Each variable assignment of a predicate gives a ground (i.e. fully instantiated) Herbrand term. A summary of the function is computed for each ground Herbrand term.

In summary, we make the following contributions towards modular analysis:

- We apply Heyting completion [10] on-demand in order to make the summary of a function complete for some predicate. Predicates are created on-demand, namely when incompleteness would lead to an unusably imprecise result.
- We propose Herbrand abstractions to express symbolic predicates that functions postulate and that call-sites instantiate, thereby providing an abstract interface between the base analysis and the completion mechanism.
- We present an implementation of this framework using an inter-procedural control-flow-graph analysis that is able to resolve function calls in an x86 executable compiled from a higher-order functional language.

The remainder of this paper is organized as follows: The next section defines a collecting and abstract semantics for an imperative language with indirect function calls. Section 3 generalizes these semantics to one that relates function inputs to outputs. Section 4 enhances this abstract interpretation with the generation of Herbrand terms and presents how a fixpoint is obtained in a modular way. Section 5 discusses our implementation before Sect. 6 presents related work.

$$
\begin{aligned}
\mathit{Prog} &::= \mathit{FDecl}^* & \mathit{Stmt} &::= \mathit{Loc}_S : \mathit{br}\ \underline{(}\mathit{Expr} : \mathit{Loc}_S;\underline{)?}\ \mathit{Loc}_S \\
\mathit{FDecl} &::= \texttt{ident}()\{\mathit{Stmt}^*\} & &\mid \mathit{Loc}_S : \mathit{Lhs} = \mathit{Expr} \\
\mathit{Lhs} &::= \texttt{ident.field}\underline{(\rightarrow\texttt{field})?} & &\mid \mathit{Loc}_S : \mathit{call}\ \mathit{Expr} \\
\mathit{Expr} &::= \mathit{Lhs} \mid \mathit{Loc} & &\mid \mathit{Loc}_S : \mathit{return}
\end{aligned}
$$

Figure 2. The abstract grammar of the analyzed program. $\underline{(}E\underline{)?}$ denotes zero or one E.

2 Preliminary Definitions

In this section we define a language with functions and define a collecting semantics for it. Let $[]$ denote an empty map, $m := [k_1 \mapsto v_1, \ldots k_n \mapsto v_n]$ a map where n values can be looked up with $m[k_i] = v_i$, let $m \setminus k$ denote a map without a mapping for k and let $m[k \mapsto v]$ denote an update at k. Let $\mathrm{dom}(m)$ denote the keys in m. Let $\mathit{Loc} = \mathit{Loc}_S \uplus \mathit{Loc}_M$ be the set of memory locations of a program P that is partitioned into statement labels Loc_S and statically and dynamically allocated memory regions Loc_M. Define $\mathit{Loc}_F \subseteq \mathit{Loc}_S$ to be the set of function entry points which coincide with the first statement in each function. We assume a C-like language where a variable v is stored at address $\&v \in \mathit{Loc}_M$. Let $\sigma \in \Sigma : \mathit{Loc}_M \to (\mathcal{F} \to \mathbb{V})$ define the program state with $\sigma(m)$ being a field map of the memory at address $m \in \mathit{Loc}_M$. A field map takes field names $\mathcal{F}$ to their content $\mathbb{V}$ where $\mathbb{V} := \mathit{Loc} \cup \mathbb{Z}$ denotes numeric values and addresses. The ability to partition a memory region into fields allows our analysis to express that a function call only accesses some but not all fields of a memory region.

Figure 2 defines the grammar of $P \in \mathcal{L}(\mathit{Prog})$. A function is a sequence of statements consisting of conditional jumps, assignments, function calls, and returns. Note that every statement is preceded by its unique address $l \in \mathit{Loc}_S$. The statement $\mathit{Lhs} = \mathit{Expr}$ updates the specified field of a memory or, via the optional C arrow notation, a field in the pointed-to memory region. For brevity, we write `even_call` for $\texttt{Even::IsEven.this} \rightarrow \texttt{even\_call} \in \mathcal{L}(\mathit{Lhs})$ (where `Even::IsEven` is the method in Fig. 1). The concrete semantics of a statement takes an input program state $\sigma \in \Sigma$ and returns a tuple consisting of the output state and the location where execution continues. The individual rules are explained below.

$$
\begin{aligned}
[\![\cdot]\!]^\natural &: \mathcal{L}(\mathit{Stmt}) \times \Sigma \to (\mathit{Loc}_S \times \Sigma) \\
[\![l_s\colon \mathit{br}\ e : l_t; l_f]\!]^\natural\sigma &= \begin{cases} \langle l_t, \sigma\rangle & \text{if } [\![e]\!]^\natural_{\mathit{Expr}}\sigma = 0 \\ \langle l_f, \sigma\rangle & \text{otherwise} \end{cases} && (1) \\
[\![l_s\colon \texttt{m.f} = e]\!]^\natural\sigma &= \langle \mathit{next}(l_s), \sigma[m \mapsto \sigma(m)[f \mapsto v]]\rangle \text{ where } v = [\![e]\!]^\natural_{\mathit{Expr}}\sigma && (2) \\
[\![l_s\colon \texttt{m.f} \rightarrow \texttt{f}' = e]\!]^\natural\sigma &= [\![l_s\colon \texttt{m}'\texttt{.f}' = e]\!]^\natural\sigma \text{ where } \&m' = [\![\texttt{m.f}]\!]^\natural_{\mathit{Expr}}\sigma && (3) \\
[\![l_s\colon \mathit{call}\ e]\!]^\natural\sigma &= \langle \&\mathrm{f}, \sigma[\texttt{f} \mapsto [\texttt{ret} \mapsto \mathit{next}(l_s)]]\rangle \text{ where } \&\texttt{f} = [\![e]\!]^\natural_{\mathit{Expr}}\sigma && (4) \\
[\![l_s\colon \mathit{return}]\!]^\natural\sigma &= \langle l_r, \sigma \setminus \texttt{f}\rangle \text{ where } l_r = [\![\texttt{f.ret}]\!]^\natural_{\mathit{Expr}}\sigma && (5)
\end{aligned}
$$

The evaluation of an expression $e \in \mathcal{L}(\mathit{Expr})$ is defined as follows:

$$[\![\cdot]\!]^\natural_{Expr} \ : \ \mathcal{L}(Expr) \times \Sigma \to \mathbb{V}$$

$$[\![\texttt{m.f}]\!]^\natural_{Expr}\sigma = \sigma(m)(f) \tag{6}$$

$$[\![\texttt{m.f} \to \texttt{f}']\!]^\natural_{Expr}\sigma = [\![\texttt{m}'.\texttt{f}']\!]^\natural_{Expr}\sigma \text{ where } \&m' = [\![\texttt{m.f}]\!]^\natural_{Expr}\sigma \tag{7}$$

$$[\![l]\!]^\natural_{Expr}\sigma = l \qquad \text{(rule } Expr ::= Loc\text{)} \tag{8}$$

Jumps, defined by Eqn. 1, are unconditional if e is omitted. Equation 2 updates the field f in $\sigma(m)$. It returns the program location following this statement using a function $next : Loc_S \to Loc_S$ that we assume to be suitably defined for all non-branching statements. A write through a pointer in Eqn. 3 assumes that the pointer value `m.f` matches the beginning of a memory region m' and is undefined otherwise. Thus, we do not model general pointer arithmetic and array accesses but assume that `parity->vtable[0]` is interpreted such that `vtable[0]` is a field name. Our implementation supports general pointer arithmetic.

The call instruction in Eqn. 4 continues execution at the called function. It also creates a memory region with the same name as the function. This memory region serves as the stack frame. The return instruction in Eqn. 5 jumps to the location in the local variable `f.ret`, where `f` is the current function. (Note that supporting recursion requires the use of unique names for stack frames as done in the implementation.) Moreover, we assume that function arguments are copied by the caller into the stack frame of the callee. The semantics of expressions in Eqns. 6 to 8 is straightforward.

A suitable collecting semantics is the classic merge-over-all-path solution. Let $\Sigma_s \subseteq \Sigma$ be the initial state at the program entry point l_{main}. We define:

Definition 1. *The collecting semantics of P is a map $col_P : Loc_S \to \wp(\Sigma)$ satisfying $\Sigma_s \subseteq col_P(l_{main})$ and for all $l : stmt \in P$, $\sigma \in col_P(l)$, and $\langle \sigma', l' \rangle = [\![l\colon stmt]\!]^\natural(\sigma)$ it holds that $\sigma' \in col_P(l')$.*

The structure $\langle Loc_S \to \wp(\Sigma), \dot{\subseteq}, \dot{\cup} \rangle$ is the cpo of the concrete domain where $\dot{\subseteq}$ and $\dot{\cup}$ are the point-wise liftings of the corresponding operations on the images of the map. The next section details how it is approximated by an abstract domain.

2.1 Abstract Interpretation of the Collecting Semantics

The segregation of memory into distinct regions lies at the heart of a modular analysis where a function summary leaves all but a small set of memory regions untouched. We therefore lift the concept of a memory region to the abstract.

Specifically, an abstract interpretation of the collecting semantics abstracts the unbounded set of memory regions in the concrete environments Σ by a bounded set of abstract memory regions $\mathcal{M}$. The memory regions define a set of non-overlapping areas of memory. The structure of a memory region $r \in \mathcal{M}$ is defined by a map $MS = \mathcal{M} \to (\mathcal{F} \to \mathcal{X})$ whose mappings are written $[r_1 \mapsto \phi^\sharp_1, \ldots, r_n \mapsto \phi^\sharp_n]$ where each $\phi^\sharp_i$ maps fields of a memory region r_i to a value domain variable $x \in \mathcal{X}$ that takes on values in $\mathbb{V} = \mathbb{Z} \cup Loc$.

The values of $X \subseteq \mathcal{X}$ are given by a domain $\mathcal{D}_X = \langle D_X, \sqsubseteq_{D_X}, \sqcup_{D_X}, \sqcap_{D_X}, \bot_D \rangle$. Here, X is the support set of $\mathcal{D}_X$, that is, the variables that D_X restricts. In our implementation, $\mathcal{D}_X$ is a reduced product [4] of several abstract domains. Since the inference of summaries requires the ability to express relations between input and output variables, a domain $d \in D_X$ must be concretized in a way that retains these relations. Thus, the concretization $\gamma_{\mathcal{D}_\mathcal{X}} : D_X \to \wp(\mathbb{V}^*)$ maps $d \in D_X$ to $\gamma_{\mathcal{D}_X}(d) = \{\boldsymbol{v}_1, \ldots\}$ where each vector $\boldsymbol{v}_i$ has one dimension for each abstract variable $x \in \mathcal{X}$. For instance, let $d \in D_{\{x,y\}}$ have its variables restricted by the interval constraint $x \in [3, 5]$ and the equality $x + 1 = y$ then $\langle x, y \rangle \in \gamma_\mathcal{D}(d) = \{\langle 3, 4\rangle, \langle 4, 5\rangle, \langle 5, 6\rangle\}$. We write $\boldsymbol{v}(x)$ to extract the value from the vector corresponding to the dimension $x \in \mathcal{X}$. Changes to the support set X of a domain $\mathcal{D}_X$ are implemented by two functions $addVar_x : \mathcal{D}_X \to \mathcal{D}_{X \cup \{x\}}$ (leaving x unrestricted) and $delVar_x : \mathcal{D}_{X \cup \{x\}} \to \mathcal{D}_X$ that are defined iff $x \notin X$.

Combining Memory Structure and Value Domain We now describe how MS and $\mathcal{D}_X$ are combined. For the sake of this section, let $vars(ms) \subseteq \mathcal{X}$ denote the variables occurring in $ms \in MS$. The lattice of our analysis contains elements $\langle m, d \rangle \in MS \times \{\mathcal{D}_X \mid X \subseteq \mathcal{X}\}$ such that $d \in \mathcal{D}_{vars(m)}$. We denote this universe as $MS \times D$. The concretization of $MS \times D$ to environments Σ proceeds in three steps: First, we define a function *embed* that updates an environment $\sigma \in \Sigma$ with the values in a vector $\boldsymbol{v} \in \mathbb{V}^*$ based on the fields of a memory region. The function recursively processes each mapping by pattern matching against the empty map and a map $\{r \mapsto \phi^\sharp\} \uplus m$ containing a mapping for region r and other mappings m:

$$
\begin{aligned}
&\mathit{embed} : MS \times (\mathit{Loc} \cup \mathbb{Z})^* \times \Sigma \to \Sigma \\
&\mathit{embed}([], \boldsymbol{v}, \sigma) = \sigma \\
&\mathit{embed}([r \mapsto \phi^\sharp] \uplus m, \boldsymbol{v}, \sigma) = \mathit{embed}(m, \boldsymbol{v}, \sigma[r \mapsto \mathit{embed}_\phi(\sigma(r), \phi^\sharp, \boldsymbol{v})]) \\
&\text{where } \mathit{embed}_\phi(m, \phi^\sharp, \boldsymbol{v}) = m[f \mapsto \boldsymbol{v}(\phi^\sharp(f)) \mid f \in \mathrm{dom}(\phi^\sharp)]
\end{aligned}
$$

Second, we apply *embed* to the set of all concrete stores Σ, thereby obtaining $\{\mathit{embed}(m, \boldsymbol{v}, \sigma) \mid \sigma \in \Sigma\}$, the set of concrete stores in which the fields tracked by the abstract domain are restricted to values in $\boldsymbol{v}$. The final step is to compute this set for each value vector, giving the concretization function:

$$
\begin{aligned}
\gamma_{MS \times D} \;&:\; MS \times D \to \wp(\Sigma) \\
\gamma_{MS \times D}(\langle m, d \rangle) &= \bigcup_{\boldsymbol{v} \in \gamma_\mathcal{D}(d)} \{\mathit{embed}(m, \boldsymbol{v}, \sigma) \mid \sigma \in \Sigma\}
\end{aligned}
$$

We now address the task of defining the lattice operations on $MS \times D$. The problem to address is that two structures $m_1, m_2 \in MS$, that are propagated to the same program point, are associated with domains $d_i \in \mathcal{D}_{vars(m_i)}$, $i = 1, 2$, so that d_1 and d_2 range over different variables and cannot be compared or joined.

We address this problem using a cofibered abstract domains [22] and define three sound morphisms[3] $addRegion_r, addField_{r,f}, renameField_f : MS \times D \to$

[3] In categorical terms, $MS \times D$ is a Grothendiek construction $F \rtimes C$ using functor $F : C \to \mathbf{Cat}$ where C is a small category with $obj(C) = MS$ and $\mathbf{Cat}$ is a category

$MS \times D$ that are applied if the memory structures ms_1, ms_2 differ:

$$\langle m, d\rangle \xrightarrow{addRegion_r} \langle m[r \mapsto []], d\rangle \tag{9}$$

$$\langle [r \mapsto \phi^\sharp] \uplus m, d\rangle \xrightarrow{addField_{r,f}} \langle [r \mapsto \phi^\sharp[f \mapsto x]] \uplus m, addVar_x(d)\rangle \tag{10}$$

$$\langle [r \mapsto \phi^\sharp[f \mapsto x]] \uplus m, d\rangle \xrightarrow{renameField_{f,x,y}} \langle r \mapsto \phi^\sharp[f \mapsto y]] \uplus m, \; delVar_x([\![y := x]\!]^\sharp addVar_y(d))\rangle \tag{11}$$

Here, $[\![y := x]\!]^\sharp$ in Eqn. 11 is the update transformer on $\mathcal{D}_X$. By applying a composition of the three morphisms on the domain tuples $\langle m_i, d_i\rangle$, $i = 1, 2$, one can obtain tuples $\langle m_i', d_i'\rangle$ with $m_1' = m_2'$ so that the lattice operations $\sqsubseteq_{D_X}, \sqcup_{D_X}$ can be applied to d_i'. The morphisms can be shown as sound wrt. $\gamma_{MS\times D}$ and we obtain the abstract lattice $\langle MS \times D, \sqsubseteq_{MS\times D}, \sqcup_{MS\times D}, \bot_{MS\times D}\rangle$.

Example 1. We give an intuition on where the above morphisms are applied using an alias domain with universe $D_X = X \to \wp(Loc_M \cup \{a_{bad}\})$. It implements $addVar_x$ adding the mapping $x \mapsto \{a_{bad}\}$ where a_{bad} is a symbolic constant that represents all illegal addresses. Consider the following two functions:

```
void foo() {struct { int* a; } s;  if (rand()) s.a = &f; }
void bar() {struct{int*a;} s; if(rand()) s.a=&f; else s.a=&g;}
```

Assume that `s` initially points to a region without fields, i.e. $s \mapsto []$. Assume further that, in `foo` and `bar`, the then-branch updates `s` such that $s \mapsto [a \mapsto x_1]$. For `foo`, we have to apply the $addField_{s,a}$ morphism on the else-branch state before the join; the join, consequently, results in the alias set $x_1 \mapsto \{a_{bad}, \texttt{\&f}\}$. In the else-branch of bar `bar`, the update creates, e.g., $s \mapsto [a \mapsto x_2]$. In this case, we have to apply $renameField_{a,x_2,x_1}$ so that the states to be joined have the same support set. The join results in $x_1 \mapsto \{\texttt{\&f}, \texttt{\&g}\}$ for the field `a`.

The presented memory structures MS do not allow for summarized memory regions as every abstract memory region $r \in \mathcal{M}$ corresponds to exactly one concrete memory region in σ, albeit at varying addresses. Although this suffices to illustrate our modular analysis, our implementation requires a simple form of summaries in form of weak updates. A concretization that caters for summarized memory regions [19] would complicate the presentation unnecessarily.

---

of small categories with $obj(\mathbf{Cat}) = \{\langle \mathcal{D}_X, \rho\rangle \mid X \subseteq \mathcal{X}, \rho : X \to (Loc_M \times \mathcal{F})\}$. Here, the translation ρ provides information on how X relates to the field names of memory regions. F maps a category of memory structures to a category of domains over variables in that memory structure. Thus, the category $F \rtimes C$ contains tuples $\langle m, \langle d, \rho\rangle\rangle \in obj(F \rtimes C)$ where $m \in MS$ and $d \in \mathcal{D}_{vars(m)}$. The morphisms $\langle m_1, \langle d_1, \rho_1\rangle\rangle \xrightarrow{(f,g)} \langle m_2, \langle d_2, \rho_2\rangle\rangle \in hom_{F\rtimes C}$ are pairs (f, g) where $m_1 \xrightarrow{f} m_2$ is a functor in C and g is a morphism $F(f)(\langle d_1, \rho_1\rangle) \xrightarrow{g} \langle d_2, \rho_2\rangle$ in $\mathbf{Cat}$. A morphism is sound if g defines an inclusion relation between its arguments [22] which is given if the values of d_1 are a subset of those in d_2 modulo the translation of variables: $g(\langle d_1, \rho_1\rangle, \langle d_2, \rho_2\rangle)$ iff $\forall \boldsymbol{v_1} \in \gamma_{\mathcal{D}_\mathcal{X}}(d_1).\exists \boldsymbol{v_2} \in \gamma_{\mathcal{D}_\mathcal{X}}(d_2).\forall x \in \mathrm{dom}(\rho_1) \wedge \rho_1(x) \in \mathrm{dom}(\rho_2^{-1})\,.\, \boldsymbol{v_1}(x) = \boldsymbol{v_2}(\rho_2^{-1}(\rho_1(x)))$. We omit ρ when defining morphisms as it is not needed.

3 Modular Program Semantics

In this section we generalize the collecting semantics and its abstract interpretation to function summaries. Specifically, we summarize the behavior of a function by a set of tuples $\langle \sigma, \bar{\sigma} \rangle$ that state how an input environment σ is mapped to an output environment $\bar{\sigma}$ and lift this relation to an abstract input/output relation.

We first define the input/output function semantics for a single input state. Recall that the semantics of calling f and returning from f in Eqns. 4 and 5 use the field `f.ret` to store the return address. In order to define the semantics of f independently of a caller, we evaluate it in an environment $\sigma = [\texttt{f.ret} \mapsto l_f^{res}]]$ where $l_f^{res} \in \mathit{Loc}$ is a location that is not used in P.

Definition 2. *The semantics of f at $l_f \in \mathit{Loc}_S$ and executing in state σ is a map $\mathit{col}_f^\sigma : \mathit{Loc}_S \to \wp(\Sigma)$ satisfying $\sigma[\texttt{f.ret} \mapsto l_f^{res}] \subseteq \mathit{col}_f^\sigma(l_f)$ and for all $l : \mathit{stmt} \in P$, $\sigma' \in \mathit{col}_f^\sigma(l)$, and $\langle \sigma', l' \rangle = [\![l\colon \mathit{stmt}]\!]^\natural(\sigma')$ it holds that $\sigma' \in \mathit{col}_f^\sigma(l')$.*

We use the previous definition to define the relational semantics of f, that is, how each input state relates to the states at each statement of f:

Definition 3. *The relational semantics $\mathit{rel}_f : \mathit{Loc}_S \to \wp(\Sigma \times \Sigma)$ of a function f is given by $\mathit{rel}_f(l) = \{\langle \sigma, \bar{\sigma} \rangle \mid \sigma \in \Sigma \wedge \bar{\sigma} \in \mathit{col}_f^\sigma(l)\}$.*

Observe that rel_f is defined in terms of Equ. 4 which defines the semantics of a call to evaluate the called function rather than using the summary rel_f. We therefore use the following definition from now on:

$$[\![l_s\colon \mathit{call}\ e]\!]^\natural\sigma = \langle \mathit{next}(l_s), \bar{\sigma} \rangle \text{ where } \&\mathrm{f} = [\![e]\!]^\natural_{\mathit{Expr}}\sigma \wedge \langle \sigma, \bar{\sigma} \rangle \in \mathit{rel}_f(l_f^{res}) \quad (12)$$

3.1 Abstract Interpretation of the Relational Semantics

The relational semantics of a function is approximated by an abstract domain $\mathit{MS}^2 \times D$ that is used to abstract $\mathit{rel}_f(l)$ for all locations $l \in \mathit{Loc}_S$ within function f. Here, $\mathit{MS}^2 = \mathit{MS} \times \mathit{MS}$ are two memory structures, the first describing the memory at the entry point of f, the second describing the memory at l. The relation between abstract and the concrete domain is given by $\gamma_{\mathit{MS}^2 \times D}$:

$$\gamma_{\mathit{MS}^2 \times D} \ : \ \mathit{MS}^2 \times D \to \wp(\Sigma \times \Sigma)$$
$$\gamma_{\mathit{MS}^2 \times D}(\langle m_{in}, m_{out}, d \rangle) = \bigcup_{\boldsymbol{v} \in \gamma_{\mathcal{D}}(d)} \{\langle \mathit{embed}(m_{in}, \boldsymbol{v}, \sigma), \mathit{embed}(m_{out}, \boldsymbol{v}, \sigma) \rangle \mid \sigma \in \Sigma\}$$

The concretization retains the relational character of rel_l in two ways: first, the *embed* functions are applied on the same numeric vector $\boldsymbol{v} \in \mathbb{Z}^*$ so that relational information between numeric variables are manifest in the concrete states. Second, the information of the abstract domain is embedded into the same $\sigma \in \Sigma$. As a consequence, a field in any concrete memory region in σ that is not present in either m_{in} nor m_{out} is not altered. These relational properties are illustrated in the following example:

Example 2. Let $l_e \in Loc_F$ be the entry point of method `Odd::IsEven()` in Fig. 1. The relational semantics at l_e is the identity, i.e. $rel_{l_e}(l_e) = \{\langle\sigma,\sigma\rangle \mid \sigma \in \Sigma\} = \gamma_{MS^2\times D}(\langle m_{in}, m_{out}, d\rangle)$ where $m_{in} = m_{out} = []$ and $d \in \mathcal{D}_\emptyset$. Let $l_i \in Loc_S$ denote the location after the `even_call++` statement, then $\langle\sigma_{in},\sigma_{out}\rangle \in rel_{l_e}(l_i)$ contains a memory region o at $l_o \in Loc_M$ that contains the object instance. An abstract state $s = \langle m_1, m_2, d\rangle \in MS^2 \times D$ with $rel_{l_e}(l_i) \in \gamma_{MS^2\times D}(s)$ is $m_i = [\texttt{Odd::IsEven} \mapsto [\texttt{this} \mapsto y^i_{val}], o \mapsto [\texttt{even\_call} \mapsto x^i_{val}]]$, $i = 1, 2$ and a value domain $d \in D$ containing the constraints $y^1_{val} = y^2_{val} = l_o$ and $x^1_{val} + 1 = x^2_{val}$.

The algebra $\langle MS^2 \times D, \sqsubseteq_{MS^2\times D}, \sqcup_{MS^2\times D}, \bot_{MS^2\times D}, \bowtie_{MS^2\times D}\rangle$ defines the abstract domain. Here, $\bowtie_{MS^2\times D}$ is a special meet operator that combines the current state in a caller with the input state of a function summary. It is explained below. Other operations can be reduced to $\mathcal{D}$ using the following morphisms:

$$\langle m_1, m_2, d\rangle \xrightarrow{addRegion_r} \langle m_1[r \mapsto []], m_2[r \mapsto []], d\rangle \tag{13}$$

$$\begin{aligned}\langle [r \mapsto \phi^\sharp_1] \uplus m_1,\\ [r \mapsto \phi^\sharp_2] \uplus m_2, d\rangle\end{aligned} \xrightarrow{addField_{r,f}} \begin{aligned}\langle [r \mapsto \phi^\sharp_1[f \mapsto x_1]] \uplus m_1,\\ [r \mapsto \phi^\sharp_2[f \mapsto x_2]] \uplus m_2,\\ [\![x_2 := x_1]\!]^\sharp addVar_{x_1}(addVar_{x_2}(d))\rangle\end{aligned} \tag{14}$$

$$\langle [r \mapsto \phi^\sharp[f \mapsto x]] \uplus m_1, m_2, d\rangle \xrightarrow{renameField^1_f} \begin{aligned}\langle r \mapsto \phi^\sharp[f \mapsto y]] \uplus m_1, m_2,\\ delVar_x([\![y := x]\!]^\sharp addVar_y(d))\rangle\end{aligned} \tag{15}$$

One obvious difference between these morphisms and those in Eqns. 9-11 is that they operate on two memory structures, namely the input ms_1 and the current state m_2 that eventually becomes the output state. For the sake of brevity, we do not handle cases where a function allocates new memory regions and can therefore assume that $\mathrm{dom}(m_1) = \mathrm{dom}(m_2)$ at all times. Under this assumption, we define Eqn. 13 and 14 that allow to add a region, resp., a field. The morphism $addField_{r,f}$ adds variables $x_1, x_2 \in \mathcal{X}$ that are made equal in the numeric domain, so that the domain maps each value of the field in the input to the same value in the output. Analogous to Eqn. 11, Eqn. 15 renames a field in m_1. We omit the symmetric definition $renameField^2_f$ that renames a field in m_2 for brevity.

3.2 Abstract Semantics of Memory Accesses

This section details the abstract semantics of memory accesses and illustrates how to deal with accesses to unknown locations. Figure 3 presents the abstract semantics for expressions (abstracting Eqns. 6 and 7 by Eqns. 16 and 17, respectively) and assignments (abstracting Eqns. 2 and 3 by Eqns. 18 and 19, respectively).

The expression semantics returns a set of variables or locations so that Eqn. 17 can return one variable for each dereferenced pointer. Note here that $\gamma_\mathcal{D}$ returns vectors of possible values and that $ms_2(m)(f)$ returns the domain variable that is used to index into the vector. Each element returned by the expression semantics is assigned by Eqn. 2 and the various results are joined. Equation 19 computes the assignment via a pointer as the join of writing to all possible locations $\&m'$.

$$[\![\cdot]\!]^\sharp_{Expr} \;:\; \mathcal{L}(Expr) \times (MS^2 \times D) \to \wp(\mathcal{X} \cup Loc)$$

$$[\![\texttt{m.f}]\!]^\sharp_{Expr}\langle ms_1, ms_2, d\rangle = \{ms_2(m)(f)\} \tag{16}$$

$$[\![\texttt{m.f} \to \texttt{f}']\!]^\sharp_{Expr}\langle ms_1, ms_2, d\rangle = \bigcup_{\&m' \in \gamma_{\mathcal{D}}(d)(ms_2(m)(f))} [\![\texttt{m}'.\texttt{f}']\!]^\sharp_{Expr}\langle ms_1, ms_2, d\rangle \tag{17}$$

$$[\![\cdot]\!]^\sharp \;:\; \mathcal{L}(Stmt) \times (MS^2 \times D) \to Loc_S \times MS^2 \times D$$

$$[\![l_s\colon \texttt{m.f} = e]\!]^\sharp\langle ms_1, ms_2, d\rangle = \langle next(l_s), \bigsqcup_{e' \in [\![e]\!]^\sharp_{Expr}\langle ms_1, ms_2, d\rangle} \langle ms_1, ms_2, [\![ms_2(m)(f) = e']\!]^\sharp d\rangle\rangle \tag{18}$$

$$[\![l_s\colon \texttt{m.f} \to \texttt{f}' := e]\!]^\sharp\langle ms_1, ms_2, d\rangle = \langle next(l_s), \bigsqcup_{\&m' \in \gamma_{\mathcal{D}}(d)(ms_2(m)(f))} [\![l_s\colon \texttt{m}'.\texttt{f}' = e]\!]^\sharp\langle ms_1, ms_2, d\rangle\rangle \tag{19}$$

Figure 3. Abstract Expression Semantics.

Note that the expression $ms_2(m)(f)$ is undefined when either the memory region m does not exist in ms_2 or it does not contain a field f. Rather than handling this case in the semantic definition, we assume that the morphisms in Eqn. 13 and 14 are applied to prevent undefinedness. In case the transformer would access an unknown location through a pointer (i.e. `m.f` in Eqn. 17 or 19), a new region r is added using Eqn. 13 and `m.f` is restricted to point to it. Note that this behavior is not sound as it assumes that `m.f` does not alias with any other function inputs which may be wrong. We discuss this design choice in Sect. 5.

Example 3. We analyze `even_count++` in `Odd::IsEven` of Fig 1. Let `f.this` $== \&i$ be a test that forces `this` to point to the object instance i. For brevity, we use `f` for `Odd::IsEven`, `ev` for `even_count`, and write $d \in \mathcal{D}$ as set of constraints:

$$\langle [f \mapsto []], [f \mapsto []], \emptyset\rangle \xrightarrow{addField_{f,this}} \langle [f \mapsto [this \mapsto x_1]], [f \mapsto [this \mapsto x_2]], \{x_1 = x_2\}\rangle$$
$$\xrightarrow{addRegion_i} \xrightarrow{\texttt{f.this}==\&i} \langle [\ldots, i \mapsto []], [\ldots, i \mapsto []], \{x_1 = x_2 = l_i\}\rangle \xrightarrow{addField_{i,ev}}$$
$$\langle [\ldots, i \mapsto [ev \mapsto x_3]], [\ldots, i \mapsto [ev \mapsto x_4]], \{x_1 = x_2 = l_i, x_3 = x_4\}\rangle \xrightarrow{[\![\texttt{f.this}\to\texttt{ev++}]\!]^\sharp}$$
$$\langle [\ldots, i \mapsto [ev \mapsto x_3]], [\ldots, i \mapsto [ev \mapsto x_4]], \{x_1 = x_2 = l_i, x_3 + 1 = x_4\}\rangle$$

The idea of applying morphisms as a precursor to a domain operation is also key to concisely define the $\bowtie_{MS^2 \times D}$ operation that combines a call-site state $\langle m^1_{in}, m^1_{out}, d_1\rangle$ with the summary of a function $\langle m^2_{in}, m^2_{out}, d_2\rangle$. Assuming that morphisms were applied so that m^1_{out}, the current state at the caller, and m^2_{in}, the input state of the callee summary, contain the same fields with the same variables while m^1_{in} and m^2_{out} share no variables, $\bowtie_{MS^2 \times D}$ reduces to $\sqcap_D$:

$$\langle m^1_{in}, m^1_{out}, d_1\rangle \bowtie_{MS^2 \times D} \langle m^2_{in}, m^2_{out}, d_2\rangle = \langle m^1_{in}, m^2_{out}, d'\rangle \text{ where} \tag{20}$$
$$d' = delVar_{vars(m^1_{out}) \cup vars(m^2_{in})}(addVar_{vars(m^2_{out})}(d_1) \sqcap_D addVar_{vars(m^1_{in})}(d_2))$$

Here, *addVar* and *delVar* are used to add/remove a set of domain variables so that $\sqcap_D$ is applied to domains mapping variables of $m^1_{in}, m^1_{out}, m^2_{in}, m^2_{out}$ while

d' only contains variables relevant to the result. Ensuring that m^1_{out} and m^2_{in} contain the same fields amounts to matching the memory regions at the caller with those of the callee. Recall that the latter were created on-demand when computing the summary of the callee so that they have arbitrary names. We therefore compute a relation $R \subseteq \mathcal{M} \times \mathcal{M}$ between the caller's and the callee's memory regions by iteratively following pointers, starting with the actual and formal function arguments. If a pointer `s.p` can be followed to a region r in the callee but not in the caller, we apply $\mathit{addRegion}_r$, $\mathit{addField}_{s,p}$, to the caller and add $\langle r, r\rangle$ to R. For each pair added to R, we apply $\mathit{addField}_f$ in each region until both have the same fields. We appropriately name fields using $\mathit{renameField}_f$.

3.3 Computing Fixpoint of the Abstract Relational Semantics

This section details how the modular abstract semantics is used to compute a fixpoint of the whole program. A whole-program analysis populates a table $T \in \mathbb{T} = \mathit{Loc}_F \to \mathit{MS}^2 \times D$ that takes a function addresses to its summary. Since a function f may call other functions, a call statement in f will access T to obtain the most up-to-date summary for the called function. The semantics of the `call` statement is therefore parameterized by T:

$$[\![l_s\colon \mathit{call}\ e]\!]^\sharp_T\langle ms_1, ms_2, d\rangle = \langle \mathit{next}(l), \bigsqcup_{l_f \in \gamma_{\mathcal{D}}(d)([\![e]\!]^\sharp)} \langle ms_1, ms_2, d\rangle \bowtie_{\mathit{MS}^2\times D} T(l_f)\rangle \quad (21)$$

The resulting summary for f must therefore be re-computed if any summaries taken from T change. In the presence of recursive calls, widening [4] must be applied on the summaries to ensure termination.

The summary of f, given the table T and initial state s, is defined as follows:

Definition 4. *The abstract state of f is a map $\mathit{abs}^T_{f,s} : \mathit{Loc}_S \to \mathit{MS}^2 \times D$ with $s \sqsubseteq_{\mathit{MS}^2\times D} \mathit{abs}^T_f(l_f)$ and for all $l : \mathit{stmt} \in P$ and $\langle l', s'\rangle = [\![l : \mathit{stmt}]\!]^\sharp_T(\mathit{abs}^T_f(l))$ it holds that $s' \sqsubseteq_{\mathit{MS}^2\times D} \mathit{abs}^T_f(l')$. (Note: Eqns. 16-19 used $[\![l : \mathit{stmt}]\!]^\sharp \equiv [\![l : \mathit{stmt}]\!]^\sharp_T$.)*

Let $\mathit{init} = \langle m_1, m_2, \{x_1 = x_2 = l^{res}_f\}\rangle$ with $m_i = [f \mapsto [\mathit{ret} \mapsto x_i]]$, $i = 1, 2$ be the initial summary state. The summary semantics of f relates the first statement of the function at l_f with the location l^{res}_f that the return statement branches to:

Definition 5. *The abstract summary of f under T is $\mathit{sum}^T_f = \mathit{abs}^T_{f,\mathit{init}}(l^{res}_f)$.*

This concludes the presentation of the concrete relational semantics and the abstract summary domain and semantics. The next section tackles the challenge of computing precise summaries in the presence of indirect function calls.

4 On-Demand Heyting Completion

This section details how we use Herbrand terms to refine a function summary in cases where the most generic input would lead to an unacceptable precision loss. In particular, the next sections discuss the creation of Herbrand terms to express a need for refinement, the computation of a specialized function summary and the *call* semantics that combines specialized function summaries.

4.1 Extracting Refinement Information using Herbrand Terms

The challenge in specializing the summary of `Check` in Fig. 1 is that the variable over which to specialize is not known until the indirect call `parity->IsEven()` is analyzed. Our solution is that the analysis poses the question "What value can `parity->vtable[0]` take on?" to all callers of `Check` who may answer "The expression `parity->vtable[0]` may contain `&Odd::IsEven()`". (Recall that we use `vtable[0]` as a field name to fit our restricted grammar.) For each different answer, the summary of the analysis is specialized to the value in that answer. The analysis of `Check` can now proceed to the next indirect call `parity->IsOdd()` for which a new question is posed to the caller. Once the indirect function calls are resolved, `Check` can be summarized without posing further questions.

The "question" in the exposition above is represented by a Herbrand term that contains variables in places where the answer is expected. The answer to the question is given by a set of ground Herbrand terms, that is, Herbrand terms where the variables have been replaced by values.

Definition 6. *Herbrand terms* $Herb = \mathcal{L}(Term)$ *are defined by the grammar*

$$\begin{aligned} Term ::= {} & \texttt{constructor}\ Term^* \\ \mid {} & \texttt{variable} \end{aligned}$$

where `variable` *is drawn from* $\mathcal{X}_H$*. Note that* $\mathcal{X}_H$ *is distinct from* $\mathcal{X}$*. Let* $vars(h)$ *denote all variables in* $h \in Herb$*. Let* $GHerb = \{h \in Herb \mid vars(h) = \emptyset\}$ *denote ground Herbrand terms. A substitution* $\theta \in \Theta : \mathcal{X}_H \to Herb$ *is a total map with* $\theta(x) = x$ *except for a finite number of variables* $y \in \mathcal{X}_H$ *where* $\theta(y) \neq y$*. We write* $[x/y] \in \Theta$ *with* $[x/y](x) = y$ *and* $[x/y](v) = v$ *for all* $v \neq x$*. Given a term* $h \in Herb$*, we write* $\theta(h)$ *to denote the result of replacing all variables* x *in* h *by* $\theta(x)$*. Let* $\theta(H) = \{\theta(h) \mid h \in H\}$ *be the lifting to sets.*

The generic nature of Herbrand terms enables us to formulate questions that cut across several abstract domains in an abstract state $\langle m_1, m_2, d\rangle \in MS^2 \times D$.

Example 4. Suppose that the constructor *Deref* and *Field* are used by the memory domain $m \in MS$ to denote a pointer or field access, respectively, while *ConstPtr* is used by the numeric domain $d \in D$ to denote a function pointer. Then the term *ConstPtr* (*Field* (*Deref* `parity`) `vtable[0]`) a_E is the request to access the field `vtable[0]` of the memory region pointed-to by `parity` and to extract the value as a constant pointer, denoting the result by $a_E \in \mathcal{X}_H$. This query accesses $m(f) = [\texttt{parity} \mapsto x, ...]$ where f is the frame of the currently analyzed function to obtain the numeric variable $x \in \mathcal{X}$ that contains the points-to set of `parity`. The numeric domain d is queried for the points-to set of x which resolves to, say, the address of memory region `even` $\in \mathcal{M}$. Finally, the memory domain is used to look up $m(\texttt{even}) = [\texttt{vtable[0]} \mapsto vt_E, \ldots]$ and d is queried for the values of vt_E, the constant address `vtable[0]` of `Even`, which becomes the solution of a_E.

For the sake of readability, we leave the exact definition of the term structure open and write `var->field...->field=` a_E, that is, we use C-like access paths

that generalize $\mathcal{L}(Expr)$ by allowing several indirections. Moreover, we also omit the memory region (i.e. we write `this->vtable[0]` instead of `f.this->vtable[0]`) since a Herbrand term is always relative to the stack frame of the current function.

Herbrand terms are used in the abstract semantics when a precise value is needed. For instance, the *call e* instruction requires a precise value for the function address e that determines which function is being invoked. An answer is computed using a function *herbEval* that evaluates a term set (e.g. $\{\text{“}e = x''\} \subseteq Herb$ for the call) given an abstract state. *herbEval* has the following signature:

$$herbEval : \wp(Herb) \times MS^2 \times D \to \wp(\Theta) \times \wp(Herb)$$

For variables $a_1, \dots a_n$ in the input Herbrand terms, *herbEval* returns assignments in form of substitutions $\theta_1, \dots \theta_k$ where each $\theta_j = [a_1/c_1^j, \dots a_n/c_n^j]$ maps variables to constants $c_1^j, \dots c_n^j \in \mathbb{V}$, $j = 1, \dots k$, or it rewrites the Herbrand terms into terms over the function's input arguments. In order to illustrate this, we say that a Herbrand term h_i matches a domain variable x_i if h_i represents a field access (possibly via one or more pointer indirections) whose value is given by the domain variable x_i. We give an intuitive overview of *herbEval* by describing the four cases it distinguishes:

A set of values for tabulation can be constructed. The term h_i with variable a_i matches a domain variable x_i, $i = 1, \dots n$. In case x_i are finite in the value domain state d, *herbEval* returns a set of constant value vectors $\boldsymbol{c}^1, \dots \boldsymbol{c}^k \in \{\langle \boldsymbol{v}(x_1), \dots \boldsymbol{v}(x_n)\rangle \mid \boldsymbol{v} \in \gamma_{\mathcal{D}_X}(d)\}$ in the form of k substitutions $\theta_j = [a_1/\boldsymbol{c}^j(x_1), \dots a_n/\boldsymbol{c}^j(x_n)] \in \Theta$. For example, $herbEval(\{\mathtt{m}.f = a\}, \langle[\mathtt{m} \mapsto [f \mapsto x_1]], [\mathtt{m} \mapsto [f \mapsto x_2]], d\rangle)$ evaluates to $\langle\{[a/42]\}, \emptyset\rangle$ where $d = \{x_2 = 42\}$ represents the value domain.

An exact precondition can be synthesized. A term h_i matches a variable x_i. There exists $x_i' = x_i$ where x_i' is a domain variable of a field in the input memory region. For each x_i', we return a Herbrand term h_i' that matches x_i'. For example, $herbEval(\{\mathtt{m}.f = a\}, \langle[\mathtt{m} \mapsto [f \mapsto x_1], \mathtt{r} \mapsto [g \mapsto x_2]], [\mathtt{m} \mapsto [f \mapsto x_3], \mathtt{r} \mapsto [g \mapsto x_4]]\rangle, d) = \langle\emptyset, \{\mathtt{r}.g = a\}\rangle$ if $d = \{x_2 = x_3\}$ is the value domain.

A sufficient precondition can be synthesized. The term h_i matches a variable x_i. There exist several variables $\{x_i^1, \dots x_i^{k_i}\}$ from which there is a flow of information to x_i. We translate the single term h_i to Herbrand terms $h_i^1, \dots h_i^{k_i}$ that match $x_i^1, \dots x_i^{k_i}$ and return the term $\mathtt{Set}\ h_i^1 \dots h_i^{k_i}$. For example, $herbEval(\{\mathtt{t}.q = a\}, [\mathtt{u} \mapsto [r \mapsto x_1], \mathtt{v} \mapsto [s \mapsto x_2]], [\dots, \mathtt{t} \mapsto [q \mapsto x_3]], d) = \langle\emptyset, \{\mathtt{Set}\ \mathtt{u}.r = a_1\ \mathtt{v}.s = a_2\}\rangle$ where $d = [x_1 \mapsto \{\&p_1\}, x_2 \mapsto \{\&p_2\}, x_3 \mapsto \{\&p_1, \&p_2\}]$ represents the information of our aliasing domain $D_X = X \to \wp(Loc_M \cup \{a_{bad}\})$ used in Ex. 1. We will disregard this case until our discussion in Sect. 5.

No values can be synthesized. The term h_i matches no variable x_i nor can a field variable be added using $addField_{r,f}$. Thus, the values of variables in h_i are neither finite nor traceable to the input. An empty set of substitutions and Herbrand terms is returned. A warning is generated so that the analysis is sound if no warnings are emitted.

```
bool Case1() {
  Odd odd;
  Even even;
  Parity* parity =
    rnd() ? &odd : &even;
  return Check(parity);
}
```

```
void Case2(Parity *p) {
  Check(p);
}
void Case3(Parity *p,
           Parity *q) {
  Check(rnd() ? p : q);
}
```

Figure 4. Creating Herbrand terms for calls to `Check` in Fig. 1.

Example 5. We illustrate cases 1 to 3 using the functions in Fig. 4. We assume that `Check` has been analyzed with no specialization such that the first indirect call cannot be resolved. The resulting summary state is $\langle \bot_{MS^2 \times D}, H \rangle$ where $H = \{\texttt{parity->vtable[0]} = a\}$. As a consequence, H is evaluated at each call site using *herbEval*.

Consider the code of `Case1` in Fig. 4. When reaching the call to `Check` with summary state $s \in MS^2 \times D$, we evaluate $\mathit{herbEval}(H, s)$ which amounts to evaluating the value of `parity->vtable[0]` in s. In this case, the state at the call site contains a finite set of values for this field, namely $\boldsymbol{v}_1 = \langle \texttt{\&Odd::IsEven} \rangle$ and $\boldsymbol{v}_2 = \langle \texttt{\&Even::IsEven} \rangle$. Thus, two new table entries have to be generated for `Check`, one for $H_1 = \{\texttt{parity->vtable[0]} = \texttt{\&Odd::IsEven}\}$ and $H_2 = \{\texttt{parity->vtable[0]} = \texttt{\&Even::IsEven}\}$. No further queries are raised. In `Case2`, the state at the call site of `Check` does not contain a finite set of values for the queried fields. However, there exists an equality relation with the parameter `p`. Thus, *herbEval* rewrites H to $H' = \{\texttt{p->vtable[0]} = a\}$ in terms of the parameter and propagates it to the callers of `Case2`. Finally, in `Case3`, *herbEval* is able to use the flow information computed by the points-to domain to determine that the l-values in `parity->vtable[0]` is a superset of the values in `p->vtable[0]` and `q->vtable[0]`. Thus, *herbEval* returns a single Herbrand term $\mathit{Set}\ h_p\ h_q$ where $h_i \equiv \{i\texttt{->vtable[0]} = a_i\}$.

We omit a formal definition of *herbEval* as it is parametric in the value domain it operates on: In this case, *herbEval* extracts finite value sets and equalities between variables from the value domain, but other information can exploited as well. The next section discusses how *herbEval* is used to compute specialized summaries.

4.2 Specializing Summaries with Herbrand Terms

This section illustrates how a function summary is computed that is specialized wrt. a set of ground terms $H_g \in \mathit{GHerb}$. To this end, we first define the lattice of an abstract domain where transformers can generate Herbrand terms whenever the function context needs to be refined. The lattice of this analysis is a product of $MS^2 \times D$ and a set of Herbrand terms *Herb* that we write as $\langle MS^2 \times D \times \wp(\mathit{Herb}), \sqsubseteq_H, \sqcup_H, \bot_H \rangle$. All lattice operations are the point-wise liftings, i.e. $\langle s_1, H_1 \rangle \sqsubseteq_H \langle s_2, H_2 \rangle \equiv s_1 \sqsubseteq_{MS^2 \times D} s_2 \wedge H_1 \subseteq H_2$, etc. In particular,

note that the product is not reduced [5], so that $\langle \bot_{MS^2 \times D}, H \rangle \neq \bot_H$ unless $H = \emptyset$.

The analysis populates a table in $\mathbb{T}_{Herb} = Loc_F \times GHerb \to MS^2 \times D \times \wp(Herb)$. Each entry $\langle f, H_g \rangle \mapsto \langle s, H \rangle$ states that f, when specialized by H_g, has the summary s and requires further specializations by instantiating H in its callers. We define the following transformer to impose H_g on an abstract state:

$$[\![test\ H_g]\!]^\sharp : (MS^2 \times D) \to MS^2 \times D \tag{22}$$

For example, given the terms $H_g = \{\texttt{field} = 42\}$, the initial state *init* in Sect. 3.3 is refined to $[\![test\ H_g]\!]^\sharp init = \langle m_1, m_2, \{x_1 = x_2 = l_f^{res}, x_3 = x_4 = 42\} \rangle$ where $m_i = [f \mapsto [ret \mapsto x_i, \texttt{field} \mapsto x_{i+2}]]$ for $i = 1, 2$. The semantics of a function f for a specialization H_g is defined by $sum_f^{T_H}$ that generalizes Def. 5:

Definition 7. *The specialized abstract summary of f under $T_H \in \mathbb{T}_{Herb}$ is given by $sum_f^{T_H} : GHerb \to (MS^2 \times D) \times \wp(Herb)$ where $sum_f^{T_H}(H_g) = abs^{T_H}_{f, [\![test\ H_g]\!]^\sharp init}$.*

Here, $T_H \in \mathbb{T}_{Herb}$ is the table of specialized summaries. Its elements are defined in terms of $sum_f^{T_H}$:

Definition 8. *$T_H \in \mathbb{T}_{Herb}$ is a well-formed table if $T_H(\langle f, H_g \rangle) = sum_f^{T_H}(H_g)$ for all $\langle f, H_g \rangle \in \mathrm{dom}(T_H)$.*

The analysis bootstraps by computing a summary for each function f with no specialization, thus providing the table entries with key $\langle f, \emptyset \rangle$. For any specialization H_g, a result $\langle s, H \rangle \in T_H(\langle f, H_g \rangle)$ may contain a non-empty set $H \in Herb$ that states how the function input must be specialized further so that the summary is an over-approximation of the function's concrete semantics. We now define how a call site of f instantiates H to a set of ground Herbrand terms $H_g \in GHerb$ that can be used to compute a specialized function summary $\langle f, H_g \rangle$ in T_H.

4.3 Combining Specialized Function Summaries

We now explain the differences between the semantics of the *call*-statement in Def. 21 and the following definition over the $(MS^2 \times D) \times \wp(Herb)$ domain:

$$\begin{aligned}[\![l_s\colon\ call\ e]\!]^\sharp_{T_H}\ \langle s, H \rangle = \langle \bot_{MS^2 \times D}, H \cup H_f \rangle \sqcup_{MS^2 \times D} \bigsqcup_{l_f \in \{l_f^1, \ldots l_f^n\}} applyEntries^{T_H}_{l_f}(s, \emptyset, \emptyset) \\ \langle \{[a/l_f^1], \ldots [a/l_f^n]\}, H_f \rangle = herbEval(\{\text{“}e = a''\}, s)\end{aligned} \tag{23}$$

Rather than using the concretization function $\gamma_{MS^2 \times D}$ to obtain the callee addresses l_f, we evaluate a Herbrand term $e = a$ in the current state $s \in MS^2 \times D$ where e is the called expression. We obtain a set of function addresses l_f^i, $i \in [1, n]$ and/or Herbrand terms H_f. Recall that a non-empty H_f are predicates over the inputs of this function that need to be restricted to a finite set of callers before

$$applyEntries_f^{T_H} : ((MS^2 \times D) \times \wp(Herb) \times \wp(GHerb)) \to (MS^2 \times D) \times \wp(Herb)$$

$$applyEntries_f^{T_H}(s, H, H_g) =$$

$$\text{let } \langle s', H' \rangle \in T_H(\langle f, H_g \rangle) \text{ in} \tag{24}$$

$$\text{if } H' = \emptyset \text{ then } \langle s \bowtie_{MS^2 \times D} s', \emptyset \rangle \text{ else} \tag{25}$$

$$\text{let } \langle \Theta, H^{new} \rangle = herbEval(H \cup H', s) \text{ in} \tag{26}$$

$$\text{let } \overline{H'_g} = \{H'_g \mid H'_g = \theta(H \cup H') \cap GHerb, \theta \in \Theta, H_g \subseteq H'_g\} \text{ in} \tag{27}$$

$$\langle \bot_{MS^2 \times D}, H^{new} \rangle \sqcup_{MS^2 \times D} \bigsqcup_{H'_g \in \overline{H'_g}} applyEntries_f^{T_H}(s, H \cup H', H'_g) \tag{28}$$

Figure 5. Applying a specialized function summary in $T_H \in \mathbb{T}_{Herb}$

this call has an effect. Thus, the predicates $H \cup H_f$ are returned with a bottom summary $\bot_{MS^2 \times D}$. The effect of each known callee at l_f^i is composed with the current state s using a helper function *applyEntries* that is defined in Fig. 5.

The idea of *applyEntries* is to find those specializations of callee f that match the caller state s and to combine those specializations with s. The arguments H and H_g always contain the same number of terms, where H_g is one specialization of H in s. In Eqn 24, we assume the table T_H contains an entry for the specialization $\langle f, H_g \rangle$. It is up to the fixpoint engine to compute a missing entry on-the-fly or to resume the evaluation of the caller once the entry is available. If the retrieved summary s' requires no new specializations, i.e. if $H' = \emptyset$, the summary s' is composed with the caller state in Eqn. 25 and returned. In case $H' \neq \emptyset$, the summary s' is an under-approximation and a more specialized summary must be consulted by instantiating $H \cup H'$ in the caller's state as done in Eqn. 26. The evaluation has two outcomes (which are not necessarily mutually exclusive): if $H^{new} \neq \emptyset$ then *herbEval* was able to translate the terms H' of the callee to inputs of the caller. These terms are therefore returned with the bottom summary $\bot_{MS^2 \times D}$ so that the caller will be refined. The second case is that $H \cup H'$ could be instantiated to concrete values in form of a set of substitutions Θ. Equation 27 applies Θ to obtain sets of ground terms $\overline{H'_g} \in \wp(\wp(GHerb))$ of which only those are returned that match the current specialization H_g. Each set $H'_g \in \overline{H'_g}$ is used to look up a more specialized summary of f by calling *applyEntries* recursively. We illustrate these definitions with an example.

Example 6. We illustrate the call semantics using the call to `Check` in `Case1` in Fig. 4. Assume that T_H has the following entries (`vt` is short for `parity->vtable`):

1	$\langle$`&Check`, $\emptyset\rangle$	$\langle \bot_{MS^2 \times D}, \{$`vt[0]` $= a_0\}\rangle$
2	$\langle$`&Check`, $\{$`vt[0]=&Even::IsEven`$\}\rangle$	$\langle s_1, \{$`vt[1]` $= a_1\}\rangle$
3	$\langle$`&Check`, $\{$`vt[0]=&Odd::IsEven`$\}\rangle$	$\langle s_2, \{$`vt[1]` $= a_2\}\rangle$
4	$\langle$`&Check`, $\{$`vt[0]` = `&Even::IsEven`, `vt[1]` = `&Even::IsOdd`$\}\rangle$	$\langle s_3, \emptyset\rangle$
5	$\langle$`&Check`, $\{$`vt[0]` = `&Odd::IsEven`, `vt[1]` = `&Odd::IsOdd`$\}\rangle$	$\langle s_4, \emptyset\rangle$

The abstract call semantics in Eqn. 23 invokes $applyEntries^{T_H}_{\texttt{\&Check}}(s, \emptyset, \emptyset)$ where s is the caller state at the call site. The fact that Eqn. 24 returns a non-empty $H' = \{\texttt{vt[0]} = a_0\}$ means that a specialization needs to be computed, based on s which is done by Eqn. 26. Since s provides a finite set of values for a_0, $\Theta = \{[a_0/\texttt{\&Even::IsEven}], [a_0/\texttt{\&Odd::IsEven}]\}$ while H^{new} is empty. Applying these substitutions in Eqn. 27 gives two specializations in $\overline{H'_g}$, leading to two recursive calls in Eqn. 28, namely $applyEntries(s, \{\texttt{vt[0]} = a_0\}, \{\texttt{vt[0]} = \texttt{\&Even::IsEven}\})$ and $applyEntries(s, \{\texttt{vt[0]} = a_0\}, \{\texttt{vt[0]} = \texttt{\&Odd::IsEven}\})$. We only consider the first call as the second is analogous. Equation 24 extracts the 2nd table entry which, yet again, returns a non-empty H'. Equation 26 computes $\Theta = \{\theta_1, \theta_2\}$ where $\theta_1 = [a_0/\texttt{\&Even::IsEven}, a_1/\texttt{\&Even::IsOdd}]$, $\theta_2 = [a_0/\texttt{\&Odd::IsEven}, a_1/\texttt{\&Odd::IsOdd}]$ for the terms $H \cup H' = \{\texttt{vt[0]}= a_0, \texttt{vt[1]}= a_1\}$, thereby preserving the information at the call site that both, `vt[0]` and `vt[1]`, are taken from the same object instance. However, $\theta_2(H \cup H')$ is not a superset of H_g and is therefore discarded by Eqn. 27 as it is not a specialization of table entry 2. Thus, the only recursive call $applyEntries(s, H \cup H', \{\texttt{vt[0]} = \texttt{\&Even::IsEven}, \texttt{vt[1]} = \texttt{\&Even::IsOdd}\})$ consults table entry 4 and applies summary s_4 to the caller state using Eqn. 25.

4.4 Heyting Completion

In this section we show that the iterative tabulation of specialized function summaries is a Heyting completion, a well-known domain refinement technique [10]. A domain refinement adds new elements to an abstract domain. Our contribution is that completion is done on-demand, that is, only those elements are added to the lattice that are required by the program that is being analyzed.

Let $\langle L, \sqsubseteq_L, \sqcup_L, \sqcap_L \rangle$ be a complete lattice and $\alpha_X : L \to X$ a closure operator, i.e., monotone $Y \sqsubseteq_L Z \Rightarrow \alpha_X(Y) \sqsubseteq_L \alpha_X(Z)$, idempotent $\alpha_X(\alpha_X(Y)) = \alpha_X(Y)$, extensive $Y \sqsubseteq_L \alpha_X(Y)$, $\forall Y, Z \subseteq L$. Then $\langle L, \alpha, X, id \rangle$ is a Galois insertion [5].

Let $\Rightarrow \in L^2 \to L$ be a binary operator with $a \Rightarrow b = \bigsqcup_L \{c \in L \mid a \sqcap_L c \sqsubseteq_L b\}$. If $a \Rightarrow b \sqsubseteq_L b$ then $a \Rightarrow b$ is called the pseudo-complement of a relative to b. A lattice in which all pairs of elements have a pseudo-complement is called a Heyting algebra. We lift $\cdot \Rightarrow \cdot$ to sets $A, B \subseteq L$ as $A \Rightarrow B = \{a \Rightarrow b \in L \mid a \in A, b \in B\}$.

For any $X \subseteq L$ let $\curlywedge(X) = \{\bigsqcap_L Y \mid Y \subseteq X\}$ define the Moore closure of X. Let $A, B \in L$ such that α_A, α_B exist. Then the Heyting completion of A with respect to B is $\curlywedge(A \Rightarrow B)$. Let $\mathbb{H} = \bigcup_{H_G \subseteq GHerb} \{[\![test\ H_G]\!]^\sharp s \mid s \in MS^2 \times D\}$.

Theorem 1. *$\mathbb{H}$ is a Heyting completion of GHerb with respect to $MS^2 \times D$.*

Proof. First, show $\curlywedge(\mathbb{H}) = \mathbb{H}$. Let $[\![test\ H_i]\!]^\sharp s_i \in \mathbb{H}$ for $i = 1, 2$. Then $s := [\![test\ H_1]\!]^\sharp s_1 \sqcap_{MS^2 \times D} [\![test\ H_2]\!]^\sharp s_2 = [\![test\ H_1]\!]^\sharp [\![test\ H_2]\!]^\sharp (s_1 \sqcap_{MS^2 \times D} s_2) \in \mathbb{H}$ if there exists $H \in GHerb$ with $[\![test\ H]\!]^\sharp = [\![test\ H_1]\!]^\sharp \circ [\![test\ H_2]\!]^\sharp$. If "$e = c''_i \in H_i$ exists with $c_i \in \mathbb{Z}$ and $c_1 \neq c_2$ then $s = \bot_{MS^2 \times D}$. Otherwise, since $s_1 \sqcap_{MS^2 \times D} s_2$ has finitely many fields, there exists a finite $H \subseteq H_1 \cup H_2$. Thus, $H \in GHerb$.

Now show $\mathbb{H} = GHerb \Rightarrow MS^2 \times D$. Let $S_1 \sqsubseteq_{\mathbb{H}} S_2$ if for all $s_1 \in S_1$ there exists $s_2 \in S_2$ with $s_1 \sqsubseteq_{MS^2 \times D} s_2$ and note that $\sqcap_{\mathbb{H}}$ exists due to $\curlywedge(\mathbb{H}) = \mathbb{H}$. Choose $H \subseteq GHerb$, $b \in MS^2 \times D$. Let $a = [\![test\ H]\!]^\sharp \langle [], [], \top_D \rangle \in MS^2 \times D$.

For the sake of contradiction, assume there exist c_i with $\{a\} \sqcap_{\mathbb{H}} c_i \sqsubseteq_{\mathbb{H}} \{b\}$ and $(\{a\} \sqcap_{\mathbb{H}} c_1) \sqcup_{\mathbb{H}} (\{a\} \sqcap_{\mathbb{H}} c_2) \not\sqsubseteq_{\mathbb{H}} \{b\}$. Let $C_i = \{[\![test\ H]\!]^\sharp s \mid s \in c_i\}$. Then $C_i \subseteq \mathbb{H}$ but also $C_1 \cup C_2 \subseteq \mathbb{H}$, thus $C_1 \cup C_2 \sqsubseteq_{\mathbb{H}} \{b\}$ which is a contradiction.

Corollary 1. *The entries of the table $T_H \in \mathbb{T}_{Herb}$ defined in Def. 8 are a partial Heyting completion of GHerb with respect to $MS^2 \times D$.*

Proof. By observing that $s \in \mathbb{H}$ for all $\langle f, H_g \rangle \mapsto \langle s, \emptyset \rangle \in T_H$.

5 Implementation and Evaluation

In this section, we discuss the implementation of our analyzer. Our analyzer reconstructs the control flow- and call graph of an x86 binary. The input binary is decoded and translated into the RReil language using the GDSL toolkit [12], starting at all function entry points defined in the ELF header.

Inter-procedurally, the analysis computes summaries for all functions starting from the initial state *init* defined in Sect. 3.3. The fixpoint computation proceeds by computing the summary of a callee before continuing at a call site using a dynamically updated partial order on the caller/callee relation. Intra-procedurally, the basic blocks of a function are discovered on-the-fly and we identify loops by observing jumps from higher to lower machine addresses. Within each loop, we apply a combined widening and narrowing operator for faster convergence [1].

The value domain D of the analysis is implemented as a set of three domains. The equality domain tracks predicates of the form $x = y + c$ for $x, y \in \mathcal{X}$ and $c \in \mathbb{Z}$. The pointer domain $D_X = X \to \wp(Loc_M \cup \{a_{bad}\}) \times X$ tracks relationships of the form $x_p - x_o \in \{l_1, \ldots, l_n\}$ with $x_p, x_o \in \mathcal{X}$, $l_i \in Loc_M$. Here, x_p is the pointer variable that is being tracked, x_o contains the offset relative to the beginning of l_i, the addresses of a memory region. Finally, the value set domain is used to track finite subsets of $\mathbb{Z}$ and intervals. We impose no fixed bound on the size of the subsets (i.e. no k-limiting) but widen a growing set to an interval. The three domains form a hierarchy where a parent domain forwards any domain operation to its child. For instance, the pointer domain transforms operations on pointer variables to operations on pointer offsets and passes them on to its child domain.

Section 4.1 raised the possibility that only necessary preconditions can be synthesized that are represented by a Herbrand term *Set* ... . For instance, the call to `Case3Set` in Fig. 4 would generate the term `parity->vtable` $= a$ which is translated to a precondition *Set* {`p->vtable` $= a_1$, `q->vtable` $= a_2$}. In this case, instead of generating different table entries for each variable instantiation, we specialize $init_H$ with the join of all caller states, projected onto the variables in *Set* $\{\ldots\}$. When the caller state on these variables change, the function summaries need to be re-computed. Once necessary preconditions are generated, the analysis is no longer fully context-sensitive since the state at one caller can be propagated to the call site of a different caller. Our analysis also distinguishes summary memory regions that are created when accessing memory regions within loops. Any precondition generated in terms of summary memory regions are only necessary and never exact.

Binary	Exact H	Set H	None	Avg. H_G	Indirect	Resolved	Time	Size
`libgdsl_x86`	145	1	10	14.6	388	237	8.3h	1.1mb
`libgdsl_avr`	137	3	2	5.8	224	157	9.5m	303kb
`libgdsl_arm7`	82	0	4	3.4	153	96	13.4m	407kb
`echo`	0	0	0	N/A	7	2	134s	14.2kb
`cat`	0	0	0	N/A	7	2	8.2m	16.2kb

Table 1. Evaluation Results

Currently, each time a pointer is accessed that can be traced to the input, we create a fresh memory region. As a result, we implicitly assume that none of the pointer parameters alias. Thus, in general, our analysis is not sound. Future work will address how to incorporate the input aliasing configuration into the tabulation.

5.1 Evaluation

We have evaluated our implementation on the set of example binaries shown in Table 1. In particular, the benchmarks starting with `libgdsl` are libraries that are written in an ML-like functional programming language GDSL. GDSL is translated into idiomatic C code where higher-order functions are translated into C function pointers or heap-allocated closures containing function pointers [13].

Column **Exact** H contains the number of *call* / *br* statements that generated a Herbrand term with a single variable, i.e., terms that correspond to exactly one input memory field. The column **Set** H reports call sites that generate a term with a *Set* constructor, i.e. the cases where only necessary preconditions can be synthesized. The number of Herbrand terms that could not be translated to an input memory field is shown in **None**. Note that this number is low compared to the number of terms that can be translated to inputs, thereby showing that a summary abstract domain compares favorably against a backward analysis: the latter comes at the cost of implementing the backward semantic transformers. Column **Avg.** H_G contains the average number of instantiations for a Herbrand term. Columns **Indirect** and **Resolved** show the number of indirect *call*/*br* statements how many that were resolved to at least one target. Not all call sites can be resolved due to imprecision in our analysis as well as, for the libraries, due to the fact that many public functions take function pointers. Finally, columns **Time** and **Size** contain the analysis time and the size of the `.text` section.

Note that the gathering of the experimental data has been done using a preliminary prototype that tracks only a single summary per function by merging all requested function summaries. However, a more faithful implementation should only increase the precision of the analysis.

6 Related Work

One traditional approach of improving the precision of context-insensitive analysis is to only merge call sites whose last k parent call sites are the same (so-called

k-CFA) [18]. While the k-CFA approach improves the precision (i.e. Fig. 1 verifies with $k = 1$), it does so without consideration for the semantics of the program.

Modular analyses are context-sensitive by combining summaries of components/functions to a solution of the whole program. There are four principles [6]: compute a global fixpoint over some simplified semantics of each component, compute summaries under worst case assumptions, compute summaries using (possibly user-supplied) interfaces, and symbolic relational separate analysis (input/output abstractions). Most analyses combine some of these four principles.

Analyses that rely on condensing domains [9,14,15,16,20] perform a pure symbolic relational analysis based a restricted class of domains that comprise Herbrand terms with variables, Boolean functions and affine equalities.

The SeaHorn analyzer allows arguing over rich, numeric properties in a modular way [11]. It simplifies the input program into Horn clauses over predicates that are tailored to the analyzed program. These are then solved in a modular way. The downside is that no new invariants can be synthesized interprocedurally. Our tabulation over Herbrand terms is, in theory, less efficient than SeaHorn's Horn clauses since we store a summary state for each set of predicates. Yet, our summaries allow the computation of new invariants even interprocedurally.

Specializing the input of a summary falls into the category of summarizing with interfaces. One instance of this idea is the inference of preconditions that, when violated, lead to an error in the analyzed code [7]. An approach called "angelic verification" [8] goes further by restricting inputs to likely correct inputs.

Modular analyses that re-evaluate a component several times also adhere to the principle of computing summaries with interfaces, as each summary of a component is somehow specialized. The classic work on tabulation proposes to analyze a function for any possible input state and to combine table entries that match a call site [17]. Our approach is an on-demand tabulation that uses concrete values of function pointers as keys. Amato et al. perform tabulation based on the equality of the abstract input state [1]. Their tabulation approach may re-analyze a function unnecessarily, i.e. when a call site state has no match in the table but matches the join of several tabulated states. Moreover, matching tabulated states by equality may lead to non-monotone behavior [1, Example 1].

In the context of binary analysis, Xu et al. manually summarise functions using pre- and postconditions [23] that are similar to our Herbrand terms.

Finally, one "simplified semantics" idea is to break the program down so that it consists of parts that can be summarized with little precision loss (with the extreme of synthesizing transfer functions for groups of instructions [3,21]).

6.1 Conclusion

We presented a framework for modular analysis that judiciously computes multiple summaries. Each summary is specialized by Herbrand terms whose template is created by the function that is being analyzed and that is instantiated by its callees. We illustrated that this versatile approach corresponds to an on-demand Heyting completion of the domain and recovers indirect function calls.

References

1. G. Amato, F. Scozzari, H. Seidl, K. Apinis, and V. Vojdani. Efficiently intertwining widening and narrowing. Sci. Comput. Program., 120:1–24, 2016.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In Programming Language Design and Implementation, San Diego, California, USA, June 2003. ACM.
3. J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In R. Cousot and M. Martel, editors, Static Analysis Symposium, volume 6337 of LNCS, pages 182–196. Springer, September 2010.
4. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Principles of Programming Languages, pages 238–252, Los Angeles, California, USA, January 1977. ACM.
5. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In Principles of Programming Languages, pages 269–282, San Antonio, Texas, USA, January 1979. ACM.
6. P. Cousot and R. Cousot. Modular Static Program Analysis. In R. N. Horspool, editor, Compiler Construction, pages 159–178, Grenoble, France, April 2002. Springer. invited paper.
7. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic Inference of Necessary Preconditions, pages 128–148. Springer Berlin Heidelberg, Rome, Italy, 2013.
8. A. Das, S. Lahiri, A. Lal, and Y. Li. Angelic Verification: Precise Verification Modulo Unknowns, pages 324–342. Springer International Publishing, San Francisco, CA, USA, 2015.
9. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making Abstract Domains Condensing. Trans. Comput. Log., 6(1):33–60, 2005.
10. R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. Transactions on Programming Languages and Systems, 20(5):1067–1109, 1998.
11. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn Verification Framework, pages 343–361. Springer, San Francisco, California, USA, July 2015.
12. J. Kranz, A. Sepp, and A. Simon. GDSL: A Universal Toolkit for Giving Semantics to Machine Language. In C. Shan, editor, Asian Symposium on Programming Languages and Systems, Melbourne, Australia, December 2013. Springer.
13. J. Kranz and A. Simon. Structure-Preserving Compilation: Efficient Integration of Functional DSLs into Legacy Systems. In Principles and Practice of Declarative Programming. ACM, September 2014.
14. K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. ACM Lett. Program. Lang. Syst., 2:181–196, March 1993.
15. M. Müller-Olm and H. Seidl. Precise Interprocedural Analysis through Linear Algebra. In Principles of Programming Languages, pages 330–341. ACM, January 2004.
16. T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In Principles of Programming Languages, pages 49–61, San Francisco, California, USA, 1995. ACM.
17. M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

18. O. Shivers. Control-Flow Analysis of Higher-Order Languages. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.
19. H. Siegel and A. Simon. FESA: Fold- and Expand-based Shape Analysis. In Compiler Construction, volume 7791 of LNCS, pages 82–101, Rome, Italy, March 2013. Springer.
20. A. Simon. Deriving a Complete Type Inference for Hindley-Milner and Vector Sizes using Expansion. Science of Computer Programming, 95, Part 2(0):254–271, 2014.
21. A. Thakur and T. Reps. A Method for Symbolic Computation of Abstract Operations. In Computer Aided Verification, LNCS, pages 174–192, Berkeley, CA, 2012. Springer.
22. A. Venet. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In Static Analysis Symposium, LNCS, pages 366–382, London, UK, 1996. Springer.
23. Z. Xu, T. Reps, and B. Miller. Typestate Checking of Machine Code, pages 335–351. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.