TPLP **19** (5–6): 990–1005, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000310

Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis[*]

JESÚS J. DOMÉNECH
Universidad Complutense de Madrid, Spain
(*e-mail:* jdomenech@ucm.es)

JOHN P. GALLAGHER
Roskilde University, Denmark and IMDEA Software Institute, Spain
(*e-mail:* jpg@ruc.dk)

SAMIR GENAIM
Universidad Complutense de Madrid, Spain
(*e-mail:* sgenaim@ucm.es)

submitted 30 July 2019; accepted 31 July 2019

Abstract

Control-flow refinement refers to program transformations whose purpose is to make implicit control-flow explicit, and is used in the context of program analysis to increase precision. Several techniques have been suggested for different programming models, typically tailored to improving precision for a particular analysis. In this paper we explore the use of partial evaluation of Horn clauses as a general-purpose technique for control-flow refinement for integer transitions systems. These are control-flow graphs where edges are annotated with linear constraints describing transitions between corresponding nodes, and they are used in many program analysis tools. Using partial evaluation for control-flow refinement has the clear advantage over other approaches in that soundness follows from the general properties of partial evaluation; in particular, properties such as termination and complexity are preserved. We use a partial evaluation algorithm incorporating property-based abstraction, and show how the right choice of properties allows us to prove termination and to infer complexity of challenging programs that cannot be handled by state-of-the-art tools. We report on the integration of the technique in a termination analyzer, and its use as a preprocessing step for several cost analyzers.

KEYWORDS: Control-flow refinement, partial evaluation, termination analysis, cost analysis

1 Introduction

Control-flow refinement (CFR) is used in program analysis to make the implicit control-flow of a given program explicit. Consider for example the program on the left:

```
while ( x > 0 )                       while (x > 0 && y < z) y++;
  if ( y < z ) y++; else x--;         while (x > 0 && y >= z) x--;
```

[*] This work was funded partially by the Spanish MICINN/FEDER, UE project RTI2018-094403-B-C31, the MINECO project TIN2015-69175-C4-2-R, the CM project S2018/TCS-4314 and by the predoctoral UCM grant CT27/16-CT28/16.

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

Its execution has two implicit phases: in the first one, `y` is incremented until it reaches the value of `z`, and in the second phase `x` is decremented until it reaches `0`. CFR techniques transform this program into the one on the right, in which the two phases are explicit. For termination analysis, the transformation simplifies the termination proof; while the original program requires a lexicographic termination argument, the transformed one requires only linear ones. There are also cases where it is not possible to prove termination without such transformations. For cost analysis, tools based on bounding loop iterations using linear ranking functions fail to infer the cost of the first program, while a linear upper-bound can be inferred for the second. CFR also helps in inferring more precise invariants, without the need for disjunctive abstract domains.

Gulwani et al. (2009) and Flores-Montoya and Hähnle (2014) used CFR to improve the precision of cost analysis. Roughly, they explore different combinations of the paths of a given loop in order to discover execution patterns, and then transform the loop to follow these patterns. Sharma et al. (2011) used CFR to improve invariants in order to prove assertions. Their technique is based on finding a *splitter predicate* such that when it holds one part of the loop is executed and when it does not hold another part is executed. The loop is then rewritten as two consecutive loops, where the predicate is required to hold in one and not to hold in the other. For the loop above, $y < z$ is a splitter predicate.

Since CFR is, in principle, a program transformation that specializes a programs to distinguish different execution scenarios, a natural question to ask is whether such a specialization can be achieved by partial evaluation, which is a general-purpose specialization technique. Using partial evaluation for CFR has the advantage that soundness comes for free, and that it is not tailored for a particular purpose but rather can be tuned depending on the application domain. In this paper we apply partial evaluation for CFR of integer transition systems, which are control-flow graphs where edges are annotated with formulas describing transitions, and can be represented as Horn clause programs. This is a very popular model that is often used in static analysis, and thus many tools would benefit from a CFR procedure for this setting. We use the partial evaluation technique of Gallagher (2019) that specializes Horn clause programs with respect to a set of predefined properties. For example, partial evaluation of the loop discussed above using the properties $x > 0$ and $y < z$ automatically discovers the two phases. Note that the right choice of properties is crucial for achieving the desired refinements.

The contributions are: (1) we suggest heuristics for inferring properties for partial evaluation automatically; (2) we discuss the use of CFR for termination analysis, using an algorithm that applies CFR only when needed, and for cost analysis as a preprocessing step; (3) we suggest the use of termination witnesses (ranking functions) as properties for CFR in order to improve precision of cost analysis; and (4) we provide a publicly available implementation and corresponding experimental evaluation that demonstrates the usefulness of our CFR procedure.

Sect. 2 gives some necessary background; Sect. 3 describes a CFR procedure that is based on using partial evaluation; Sect. 4 discusses the use of our CFR procedure in the context of termination and cost analyses; Sect. 5 discusses an implementation and experimental evaluation, and Sect. 6 concludes and discusses related and future work.

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

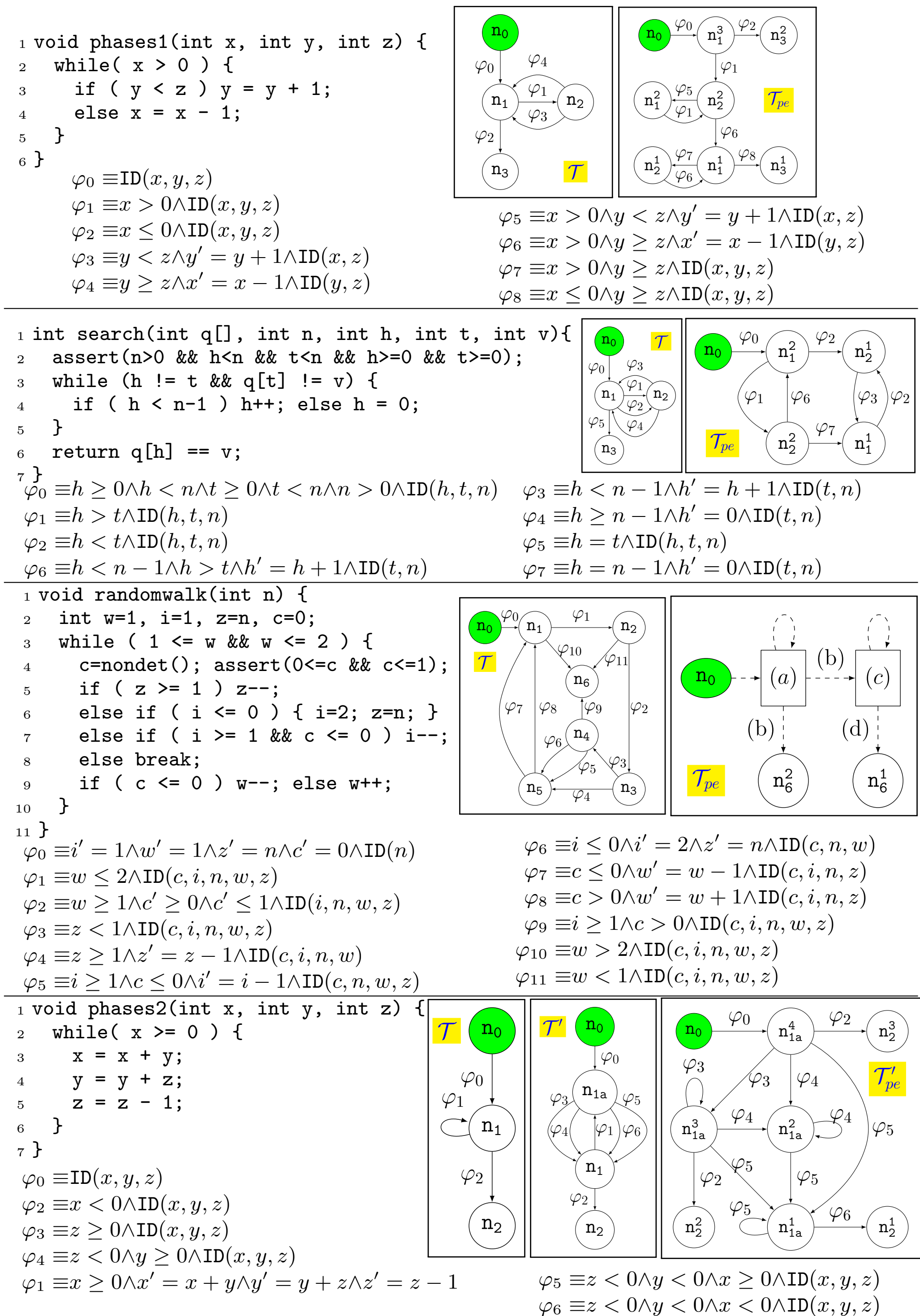

Fig. 1. Example programs. (a) **phases1**: a loop with 2 phases; (b) **search**: search in a circular queue; (c) **randomwalk**: a random walk simulation; (d) **phases2**: a loop with 3 phases.

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

2 Preliminaries

In this section we define the programming model, integer transition systems, that we will use throughout this paper and provide some necessary related background.

We let V be a fixed set of integer-valued variables $\{x_1, \ldots, x_n\}$. A linear constraint ψ is of the form $a_0 + a_1x_1 + \cdots + a_nx_n \diamond 0$ where $\diamond \in \{>, <, \geq, \leq, =\}$, a_i are integer coefficients, and $x_i \in V$. An assignment $\sigma : V \mapsto \mathbb{Z}$ is a mapping that assigns integer values to variables. We say that σ is a solution for ψ if $a_0 + \sum_i a_i\sigma(x_i) \diamond 0$ is *true*. The set of all solutions for ψ is denoted by $[\![\psi]\!]$. A *formula* φ is a conjunction of linear constraints of the form $\psi_1 \wedge \cdots \wedge \psi_k$, and its set of solutions is $[\![\varphi]\!] = [\![\psi_1]\!] \cap \cdots \cap [\![\psi_k]\!]$. We write $\varphi_1 \models \varphi_2$ for $[\![\varphi_1]\!] \subseteq [\![\varphi_2]\!]$. The projection of a formula φ onto variables $X \subseteq V$, denoted as $\mathtt{proj}_X(\varphi)$, is a formula φ' such that $[\![\varphi']\!]$ coincide with $\{\sigma|_X \mid \sigma \in [\![\varphi]\!]\}$ where $\sigma|_X$ restricts the domain of σ to X. For example, restricting $x_1 \leq x_2 \wedge x_2 \leq 100 \wedge x_3 = x_2 + 10$ to $\{x_1, x_3\}$ results in $x_1 \leq 100 \wedge x_3 \leq 110$. Note that φ' can be effectively computed using off-the-shelf libraries for manipulating linear constraints (Bagnara et al. 2008). We use $\varphi[x_i/x_j]$ to denote the formula obtained from φ by renaming variable x_i to x_j.

An integer transition system $\mathcal{T}$ (ITS for short) is a control-flow graph where edges are labeled with formulas. Formally, $\mathcal{T} = \langle V, N, \mathtt{n}_0, E\rangle$ where V is a set of program variables, N is a set of nodes, $\mathtt{n}_0 \in N$ is the entry node, and E is a set of labeled edges. An edge is of the form $\mathtt{n_i} \xrightarrow{\varphi} \mathtt{n_j}$ such that $\mathtt{n_i}, \mathtt{n_j} \in N$ and φ is *formula* over variables $V \cup V'$ where $V' = \{x_i' \mid x_i \in V\}$. We assume that the entry node $\mathtt{n}_0$ has no incoming edges. We write $\mathtt{ID}(x_1, \ldots, x_n)$ for $x_1 = x_1' \wedge \ldots \wedge x_n = x_n'$, to indicate that variables $x_1, \ldots, x_n$ do not change their values in a given edge.

A program state is a pair $(\mathtt{n_i}, \sigma)$, where $\mathtt{n_i} \in N$ and $\sigma : V \mapsto \mathbb{Z}$ is an assignment. There is a transition (i.e. a valid execution step) from $s_1 = (\mathtt{n_i}, \sigma)$ to $s_2 = (\mathtt{n_j}, \sigma')$, denoted as $s_1 \rightarrow s_2$, if there is an edge $\mathtt{n_i} \xrightarrow{\varphi} \mathtt{n_j} \in E$ such that $\wedge_{x_i \in V}(x_i = \sigma(x_i) \wedge x_i' = \sigma'(x_i)) \models \varphi$. The primed variables V' in φ refer to the program state following the transition.

A trace is a sequence of states $s_0 \rightarrow s_1 \rightarrow \cdots$ such that $s_i \rightarrow s_{i+1}$ is a transition. We write $s_i \rightarrow^* s_j$ to indicate that state s_j is reachable from state s_i. An invariant (wrt. an initial state s_0) for a given node $\mathtt{n_i} \in N$ is a formula φ over the program variables, and is guaranteed to holds whenever the execution reaches $\mathtt{n_i}$, i.e., if $s_0 \rightarrow^* (\mathtt{n_i}, \sigma)$ then $\sigma \in [\![\varphi]\!]$. An ITS is *terminating* if there are no traces of infinite length starting in $(\mathtt{n}_0, \sigma)$.

The complexity (also called cost) of an ITS is typically defined in terms of the length of its traces. In order to model cost in a way that is amenable to program transformations, we can add an extra special variable x_{n+1} whose value is 0 in the initial state and which is incremented by 1 in every transition (x_{n+1} can be modified in other ways to capture different cost models). A function $f : \mathbb{Z}^n \mapsto \mathbb{R}_+$ is an upper-bound on the cost of $\mathcal{T}$ if for any trace $(\mathtt{n}_0, \sigma) \rightarrow^* (\mathtt{n}, \sigma')$, we have $f(\sigma(x_1), \ldots, \sigma(x_n)) \geq \sigma'(x_{n+1})$.

3 Partial Evaluation of Integer Transition Systems

In this section we give a brief description of the partial evaluation technique (for Horn clauses) of Gallagher (2019), and discuss its use for CFR of ITSs. A constrained Horn clause (CHC) has the form $q_0(\bar{x}_0) \leftarrow \varphi, q_1(\bar{x}_1), \ldots, q_k(\bar{x}_k)$, where q_i are predicate names (all of arity n for simplicity) and φ is a constraint. Here we assume that the $\bar{x}_i$ are tuples of integer variables, and φ is a formula over these variables, where the formula

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

is a conjunction of linear constraints as defined in Sect. 2. We call $q_0(\bar{x}_0)$ the head, and "$\varphi, q_1(\bar{x}_1), \ldots, q_k(\bar{x}_k)$" the body. We say that the set of CHCs with head predicate q_0 defines q_0. A CHC program is a set of CHCs such that one predicate is marked as the entry. The call graph induced by a CHC program is a directed graph whose nodes are the predicate names, and there is an edge from q_i to q_j if q_i is defined by a CHC including q_j in its body. We say that q_i is a *loop head* if it has a *back edge* in the corresponding call-graph wrt. depth-first traversal from the entry predicate.

The technique of Gallagher (2019) takes as input a CHC program and a constraint on the entry predicate, and returns a partially evaluated CHC program that preserves the computational behavior of the original program with the given entry. In particular, it preserves termination and complexity of the original. It is an online partial evaluation algorithm, meaning that control decisions are made on the fly, and it yields a *polyvariant* partially evaluated program, meaning that a single predicate q_i from the input program can result in several "versions" of q_i in the output program.

It is not possible to create versions for all reachable contexts as there may be infinitely many. During partial evaluation, when reaching a call $q(\bar{x})$ with a context φ (i.e. constraints on variables $\bar{x}$) the algorithm decides whether to create a version of $q(\bar{x})$ for this context, and replace the corresponding call by one to this version, or to *abstract* the context.

Gallagher (2019) suggests the use of property-based abstraction in order to guarantee the generation of a finite number of versions. For a predicate $q(\bar{x})$ it assigns beforehand a finite set of properties $\{\varphi_1, \ldots, \varphi_k\}$, where each φ_i is a formula over $\bar{x}$, and when reaching $q(\bar{x})$ with a context φ, instead of creating a version for φ it creates one for $\alpha(\varphi) = \wedge\{\varphi_i \mid \varphi \models \varphi_i\}$. This guarantees that there are at most 2^k versions for $q(\bar{x})$. For example, if predicate $p(x, y, z)$ is assigned properties $x > 0$ and $y \geq z$, it would then have up to 4 versions corresponding to (abstract) contexts $x > 0$, $y \geq z$, $x > 0 \wedge y \geq z$ and *true*. In practice, it is enough to apply property-based abstraction to loop head predicates as they cut all cycles; other predicates can be specialized wrt. to all contexts that are encountered during the evaluation. In addition, partial evaluation might unfold deterministic sequences of calls into a single CHC.

An ITS $\mathcal{T}$ can be translated into a CHC program $\texttt{HC}(\mathcal{T})$ by translating each edge $\texttt{n}_\texttt{i} \xrightarrow{\varphi} \texttt{n}_\texttt{j}$ into a CHC $q_{\texttt{n}_\texttt{i}}(\bar{x}) \leftarrow \varphi, q_{\texttt{n}_\texttt{j}}(\bar{x}')$, and marking $q_{\texttt{n}_0}$ as the entry. Note that the control-flow graph of $\mathcal{T}$ is equivalent to the call-graph of $\texttt{HC}(\mathcal{T})$. Observe that $\texttt{HC}(\mathcal{T})$ is linear, that is, each clause has only one call in the body. Linearity guarantees that the specialized version is also linear, and thus can be converted back into an ITS $\mathcal{T}_{pe}$ in a similar way. Soundness of partial evaluation guarantees equivalence between the traces of $\mathcal{T}$ and $\mathcal{T}_{pe}$, in particular, there is a infinite trace in $\mathcal{T}$ iff there is one in $\mathcal{T}_{pe}$. The complexity is preserved as well (when modeled as discussed in Sect. 2).

Example 3.1

We now describe the partial evaluation algorithm for Horn clauses using a worked example, assuming that the clauses are linear, as they are generated from an ITS as just described.

Consider the method `phases1` of Fig. 1(a) and its corresponding ITS $\mathcal{T}$. As discussed in Sect. 1, the execution of this loop has two phases. Our aim is to use partial evaluation to split the loop into two explicit phases (see the corresponding $\mathcal{T}_{pe}$ of Fig. 1(a)). Translating

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

this ITS into a CHC program results in:

$$q_{\mathtt{n}_0}(\bar{x}) \leftarrow \varphi_0, q_{\mathtt{n}_1}(\bar{x}'). \qquad q_{\mathtt{n}_1}(\bar{x}) \leftarrow \varphi_1, q_{\mathtt{n}_2}(\bar{x}'). \qquad q_{\mathtt{n}_1}(\bar{x}) \leftarrow \varphi_2, q_{\mathtt{n}_3}(\bar{x}').$$
$$q_{\mathtt{n}_2}(\bar{x}) \leftarrow \varphi_3, q_{\mathtt{n}_1}(\bar{x}'). \qquad q_{\mathtt{n}_2}(\bar{x}) \leftarrow \varphi_4, q_{\mathtt{n}_1}(\bar{x}').$$

where $\bar{x} = \langle x, y, z\rangle$, $\bar{x}' = \langle x', y', z'\rangle$, and each φ_i is as in Fig. 1(a). Note that the only loop head predicate is $q_{\mathtt{n}_1}$ ($\mathtt{n}_1$ has a back edge when traversing in $\mathcal{T}$ starting at $\mathtt{n}_0$). Let us fix the properties of $q_{\mathtt{n}_1}(x, y, z)$ to be $\Psi = \{x > 0, y < z, y \geq z\}$, which are the properties of the variables at loop entry that determine which loop path or loop exit is taken.

The partial evaluation algorithm performs a sequence of iterations. The input to each iteration is a set of "versions" of predicates, where each version is a pair $\langle q(\bar{x}), \varphi\rangle$, where q is a program predicate and φ is a constraint on its arguments. Each iteration performs an *unfolding* for each version, followed by a property-based *abstraction* of the resulting calls using Ψ, yielding a new set of versions that is input to the following iteration. The iterations end when no new versions are produced by an unfolding. The unfold operation for a version $\langle q(\bar{x}), \varphi\rangle$ consists of expanding calls in the body of clauses with head $q(\bar{x})$, so long as the call is deterministic and not a loop head, and the constraint φ is satisfied.

Let the initial version be $\langle q_{\mathtt{n}_0}(x, y, z), \mathit{true}\rangle$. Then the iterations of the algorithm yield the following sequence of new versions.

1. $\{\langle q_{\mathtt{n}_1}(x, y, z), \mathit{true}\rangle\}$.
2. $\{\langle q_{\mathtt{n}_2}(x, y, z), x > 0\rangle, \langle q_{\mathtt{n}_3}(x, y, z), x \leq 0\rangle\}$.
3. $\{\langle q_{\mathtt{n}_1}(x, y, z), x > 0\rangle, \langle q_{\mathtt{n}_1}(x, y, z), y \geq z\rangle\}$.
4. $\{\langle q_{\mathtt{n}_2}(x, y, z), x > 0 \wedge y \geq z\rangle, \langle q_{\mathtt{n}_3}(x, y, z), x \leq 0, y \geq z\rangle\}\}$.
5. $\emptyset$.

We look in detail at iteration 3; the versions $\langle q_{\mathtt{n}_2}(x, y, z), x > \mathit{0}\rangle$ and $\langle q_{\mathtt{n}_3}(x, y, z), x \leq 0\rangle$ resulting from iteration 2 are unfolded. The latter yields no clauses since there is no clause with head predicate $q_{\mathtt{n}_3}$. The version $\langle q_{\mathtt{n}_2}(x, y, z), x > \mathit{0}\rangle$ is unfolded giving the clauses

$$q_{\mathtt{n}_2}(x, y, z) \leftarrow x > 0, z > y, y' = y - 1, q_{\mathtt{n}_1}(x, y', z)$$
$$q_{\mathtt{n}_2}(x, y, z) \leftarrow x > 0, y \geq z, x' = x - 1, q_{\mathtt{n}_1}(x', y, z).$$

The constraints on the body calls are then collected. The constraints on $q_{\mathtt{n}_1}(x, y', z)$, projected onto $\{x, y', z\}$, are $x > 0$. The constraints on $q_{\mathtt{n}_1}(x', y, z)$, projected onto $\{x', y, z\}$, are $x' > -1, y \geq z$. Using property-based abstraction with Ψ, the set of properties for $q_{\mathtt{n}_1}$, we obtain (after renaming x' to x)

$$\alpha(x > 0) = x > 0 \qquad\qquad \alpha(x > -1 \wedge y \geq z) = y \geq z.$$

Note that the constraint $x > -1 \wedge y \geq z$ entails $y \geq z$ but no other member of Ψ and hence the constraint $x > -1$ is abstracted away. This yields the two versions $\langle q_{\mathtt{n}_1}(x, y, z), x > 0\rangle$ and $\langle q_{\mathtt{n}_1}(x, y, z), y \geq z\rangle$ that are shown as the result of iteration 3. Note that without abstraction, an infinite number of iterations could result. In this example, versions $\langle q_{\mathtt{n}_1}(x, y, z), x > -1, y \geq z\rangle, \langle q_{\mathtt{n}_1}(x, y, z), x > -2, y \geq z\rangle, \ldots$ would be generated.

For each version, a new predicate is generated. We use predicate names that carry information on the different versions: $q_{\mathtt{n}_\mathtt{i}^\mathtt{j}}$ is the jth version of predicate $q_{\mathtt{n}_\mathtt{i}}$.The entry predicate $q_{\mathtt{n}_0}$ is not renamed. For our example a suitable renaming is as follows.

$\langle q_{\mathtt{n}_1}(x, y, z), \mathit{true}\rangle$	$\Rightarrow$	$q_{\mathtt{n}_1^3}(x, y, z)$	$\langle q_{\mathtt{n}_2}(x, y, z), x > 0\rangle$	$\Rightarrow$	$q_{\mathtt{n}_2^2}(x, y, z)$
$\langle q_{\mathtt{n}_1}(x, y, z), x > 0\rangle$	$\Rightarrow$	$q_{\mathtt{n}_1^2}(x, y, z)$	$\langle q_{\mathtt{n}_2}(x, y, z), x > 0 \wedge y \geq z\rangle$	$\Rightarrow$	$q_{\mathtt{n}_1^2}(x, y, z)$
$\langle q_{\mathtt{n}_1}(x, y, z), y \geq z\rangle$	$\Rightarrow$	$q_{\mathtt{n}_1^1}(x, y, z)$	$\langle q_{\mathtt{n}_3}(x, y, z), x \leq 0\rangle$	$\Rightarrow$	$q_{\mathtt{n}_3^2}(x, y, z)$
			$\langle q_{\mathtt{n}_3}(x, y, z), x \leq 0, y \geq z\rangle$	$\Rightarrow$	$q_{\mathtt{n}_3^1}(x, y, z)$

The head and body calls of the unfolded clauses for each version are then renamed,

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

yielding the following set of clauses as the result of partial evaluation (φ_i are as in Fig. 1(a)):

$$q_{\mathtt{n_0}}(\bar{x}) \leftarrow \varphi_0, q_{\mathtt{n_1^3}}(\bar{x}). \quad q_{\mathtt{n_1^3}}(\bar{x}) \leftarrow \varphi_2, q_{\mathtt{n_3^2}}(\bar{x}). \quad q_{\mathtt{n_1^3}}(\bar{x}) \leftarrow \varphi_1, q_{\mathtt{n_2^2}}(\bar{x}).$$
$$q_{\mathtt{n_3^2}}(\bar{x}) \leftarrow \varphi_5, q_{\mathtt{n_1^2}}(\bar{x}). \quad q_{\mathtt{n_2^2}}(\bar{x}) \leftarrow \varphi_6, q_{\mathtt{n_1^1}}(\bar{x}). \quad q_{\mathtt{n_1^2}}(\bar{x}) \leftarrow \varphi_1, q_{\mathtt{n_2^2}}(\bar{x}).$$
$$q_{\mathtt{n_1^1}}(\bar{x}) \leftarrow \varphi_8, q_{\mathtt{n_3^1}}(\bar{x}). \quad q_{\mathtt{n_1^1}}(\bar{x}) \leftarrow \varphi_7, q_{\mathtt{n_2^1}}(\bar{x}). \quad q_{\mathtt{n_2^1}}(\bar{x}) \leftarrow \varphi_6, q_{\mathtt{n_1^1}}(\bar{x}).$$

Translating these CHCs back to an ITS results in $\mathcal{T}_{pe}$, depicted in Fig. 1(a). □

Choice of properties. In the above example, the properties were chosen manually to give the appropriate versions and achieve CFR. While any choice of properties gives a sounds partial evaluation, heuristics are needed to infer appropriate properties automatically. We observe that properties relevant to CFR are closely related to the conditions in the loop head and within the loop body. This leads to the following heuristics that extract properties of $q_{\mathtt{n_i}}$ from the constraints of incoming and outgoing edges of node $\mathtt{n_i}$:

$$\begin{aligned}
\mathtt{PR_h}(q_{\mathtt{n_i}}) &= \{\psi \mid q_{\mathtt{n_i}}(\bar{x}) \leftarrow \varphi, q_{\mathtt{n_j}}(\bar{x}') \in \mathtt{HC}(\mathcal{T}), \psi = \mathtt{proj}_{\bar{x}}(\varphi)\}\\
\mathtt{PR_{hv}}(q_{\mathtt{n_i}}) &= \{x_\ell \diamond c \mid q_{\mathtt{n_i}}(\bar{x}) \leftarrow \varphi, q_{\mathtt{n_j}}(\bar{x}') \in \mathtt{HC}(\mathcal{T}), \ell \in [1..n], \diamond \in \{\leq, \geq\}, \varphi \models x_\ell \diamond c\}\\
\mathtt{PR_c}(q_{\mathtt{n_i}}) &= \{\psi[\bar{x}'/\bar{x}] \mid q_{\mathtt{n_j}}(\bar{x}) \leftarrow \varphi, q_{\mathtt{n_i}}(\bar{x}') \in \mathtt{HC}(\mathcal{T}), \psi = \mathtt{proj}_{\bar{x}'}(\varphi)\}\\
\mathtt{PR_{cv}}(q_{\mathtt{n_i}}) &= \{x_\ell \diamond c \mid q_{\mathtt{n_j}}(\bar{x}) \leftarrow \varphi, q_{\mathtt{n_i}}(\bar{x}') \in \mathtt{HC}(\mathcal{T}), \ell \in [1..n], \diamond \in \{\leq, \geq\}, \varphi \models x'_\ell \diamond c\}
\end{aligned}$$

$\mathtt{PR_h}$ infers properties by extracting conditions from the CHCs defining $q_{\mathtt{n_i}}$ (i.e., outgoing edge of $\mathtt{n_i}$); this is done by projecting the corresponding constraints on $\bar{x}$. Note that this also captures implicit conditions that do not appear syntactically in φ. $\mathtt{PR_{hv}}$ infers properties by extracting bounds, for each variable x_i, from the CHCs defining $q_{\mathtt{n_i}}$, inferring the minimal/maximal value for x_i is done in practice using linear programming. $\mathtt{PR_c}$ and $\mathtt{PR_{cv}}$ are similar but use calls to $q_{\mathtt{n_i}}$ (i.e., incoming edge of $\mathtt{n_i}$).

Example 3.2
Let us compute properties for $\mathtt{q_{n_1}}$. The corresponding node $\mathtt{n_1}$ has outgoing edges labeled with φ_1 and φ_2 and incoming edges labeled with φ_0, φ_3 and φ_4. $\mathtt{PR_h}(q_{\mathtt{n_1}}) = \{\mathtt{proj}_{\bar{x}}(\varphi_1), \mathtt{proj}_{\bar{x}}(\varphi_2)\} = \{x \geq 1, x \leq 0\}$, which is also what we get for $\mathtt{PR_{hv}}$ in this case. $\mathtt{PR_c}(q_{\mathtt{n_1}}) = \{\mathtt{proj}_{\bar{x}'}(\varphi_0), \mathtt{proj}_{\bar{x}'}(\varphi_3), \mathtt{proj}_{\bar{x}'}(\varphi_4)\}[\bar{x}'/\bar{x}] = \{y \leq z, y \geq z\}$ and $\mathtt{PR_{cv}} = \emptyset$. □

The above heuristics suffice for many cases in practice, but they may be inadequate when conditions in a loop body are not directly implied by the formulas of incoming/outgoing edges of the loop head. E.g., assume method `phase1` has an additional statement `w=w+1` at the end of the loop body. In this case, the outgoing edges of $\mathtt{n_2}$ will not go directly to $\mathtt{n_1}$, but to a new node connected to $\mathtt{n_1}$ with the formula $w' = w + 1 \wedge \mathtt{ID}(x, y, z)$. Thus, $y < z$ and $z \geq y$ are not implied by the constraints of incoming/outgoing edges of $\mathtt{n_1}$.

To overcome this limitation, we developed a new heuristic $\mathtt{PR_h^d}$ that propagates all constraints in the loop bodies backwards to loop heads as follows: (1) we remove all backedges (in $\mathtt{HC}(\mathcal{T})$), that is, we replace each "$q_{\mathtt{n_j}}(\bar{x}) \leftarrow \varphi, q_{\mathtt{n_i}}(\bar{x}') \in \mathtt{HC}(\mathcal{T})$" by "$q_{\mathtt{n_j}}(\bar{x}) \leftarrow \varphi$" if $q_{\mathtt{n_i}}$ is a loop head, which results in a recursion-free CHC program; (2) we compute the set of answers (the minimal model) of the resulting CHC program and take them as properties, i.e., if $\varphi_1, \ldots, \varphi_l$ are the answers for $q_{\mathtt{n_i}}(\bar{x})$, we define $\mathtt{PR_h^d}(q_{\mathtt{n_i}}) = \{\varphi_1, \ldots, \varphi_l\}$. For the modification of the program of Fig. 1(a) just described, we would get $\mathtt{PR_h^d} = \{y < z, y \geq z, x > 0\}$. We could propagate conditions in the loop bodies forwards as well; however, we do not do it since in practice we add invariants to all transitions (which has the effect of propagating conditions forward) before inferring properties.

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

In the rest of this article, we assume that we have a procedure PE that receives an ITS $\mathcal{T}$ and applies CFR via partial evaluation as follows: (1) optionally, it computes invariants for all nodes, and adds them to their outgoing edges, which makes more properties visible in the incoming edges of nodes as discussed above; (2) generates the CHC program $\mathtt{HC}(\mathcal{T})$; (3) computes properties for each loop head; (4) applies partial evaluation as described above; (5) translates the specialized CHCs into a new ITS; (6) optionally, computes invariants again in order to eliminate unreachable nodes in the new ITS; and (7) returns the new ITS. For simplicity, we assume that the choice of properties is provides via global configuration, and not received as a parameter. We might also call PE with a list of nodes N as a second argument to indicate that properties should be assigned to a loop head $q_{\mathtt{n_i}}$ only if $\mathtt{n_i} \in N$, this is used to apply CFR only to a subset of the ITS. Finally, procedure PE can be applied iteratively on its output, which might achieve further refinements.

4 Application to Termination and Cost Analysis

In this section we discuss how CFR via partial evaluation can improve the precision of termination and cost analysis of ITSs. After a high-level description of a typical termination analyzer (based on `iRankFinder` (iRank 2019)), we present an algorithm for termination analysis incorporating the CFR procedure of Sect. 3, and finally discuss some representative examples. At the end of the section we discuss cost analysis.

A termination analyzer receives an input ITS $\mathcal{T}$, and separately proves termination of each its strongly connected components (SCCs) using a procedure that we call TerminSCC. Proving termination of an SCC S is done by inferring a *ranking function* $f_{\mathtt{n_i}} : \mathbb{Z}^n \mapsto D$ for each node $\mathtt{n_i}$ mapping program states into a set D ordered by a well-founded relation $>$, such that if $\mathtt{n_i} \xrightarrow{\varphi} \mathtt{n_j}$ is an edge of S then $\varphi \models f_{\mathtt{n_i}}(\bar{x}) > f_{\mathtt{n_j}}(\bar{x}')$. Well-foundedness of $>$ guarantees the absence of infinite traces. In practice, we require $f_{\mathtt{n_i}}(\bar{x}) > f_{\mathtt{n_j}}(\bar{x}')$ to hold only on a subset of the edges that break all cycles; for other edges it is enough that $f_{\mathtt{n_i}}(\bar{x}) \geq f_{\mathtt{n_j}}(\bar{x}')$. Even if TerminSCC fails to prove termination of S, it might prove that some of its edges cannot be taken infinitely, and thus that any infinite trace has a suffix that uses only the other edges (which are returned by TerminSCC in such case). TerminSCC makes use of invariants as well, as they can increase its precision. For simplicity we assume that invariants are added to the input ITS and that procedure PE adds such invariants to the refined ITS as well.

The effectiveness of TerminSCC is affected by the kind of ranking functions and invariants used. Probably the best known kinds of ranking function are *linear ranking functions* (LRFs), which have the form $f(\bar{x}) = a_0 + \sum_{i=1}^{n} a_i x_i$; there are efficient algorithms for inferring them (Podelski and Rybalchenko 2004; Bagnara et al. 2012). However, LRFs do not suffice for all terminating programs; in such cases TerminSCC resorts to combinations of linear functions, e.g. *lexicographic ranking functions* (LLRFs) (Alias et al. 2010; Ben-Amram and Genaim 2014). Apart from performance, LRFs have the advantage that they induce bounds on the length of traces, and are thus suitable for complexity analysis. For invariants, analyzers often use abstract domains such as convex polyhedra (Cousot and Halbwachs 1978), which cannot capture more expressive, but expensive disjunctive invariants.

CFR can improve termination analysis by (1) enabling the inference of LRFs where without CFR one would need LLRFs; and (2) enabling automatic termination proof

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

```
1  TERMIN(T,CFR_B,CFR_A,CFR_S)
2  |  if CFR_B then T = PE(T)
3  |  F := TERMINITS(T)
4  |  while F ≠ ∅ and CFR_A > 0 do
5  |  |  N := nodes(F)
6  |  |  T := DELNONREACHING(T,N)
7  |  |  T := PE(T,N)
8  |  |  T' := DELTERMINATING(T,N)
9  |  |  CFR_A := CFR_A − 1
10 |  |_ F := TERMINITS(T',CFR_S)
11 |_ return F == ∅

12 TERMINITS(T,CFR_S)
13 |  F := ∅; Q := [⟨T, CFR_S⟩]
14 |  while Q ≠ ∅ do
15 |  |  ⟨T', i⟩ := Q.getFirst()
16 |  |  foreach SCC S of T' do
17 |  |  |  F_S := TERMINSCC(S)
18 |  |  |  if F_S ≠ ∅ and i > 0 then
19 |  |  |  |  T'' := PE(CONITS(F_S))
20 |  |  |  |_ Q.add(⟨T'', i − 1⟩)
21 |  |_ |_ else F := F ∪ F_S
22 |_ return F
```

Algorithm 1: Pseudocode of Termination Analysis with Control-Flow Refinement

where without CFR it is not possible. We suggest the following schemes for adding CFR to a termination analyzer, trading off ease of implementation with performance and precision.

- ($\mathtt{CFR_B}$): in this scheme CFR is applied directly to the input ITS. This is easy to implement but can perform unnecessary refinements that cause overhead in analysis.
- ($\mathtt{CFR_S}$): in this scheme, when TERMINSCC fails to prove termination of a SCC, CFR is applied only to the part of the SCC that it could not prove terminating.
- ($\mathtt{CFR_A}$): this scheme first collects all edges (from all SCCs) on which TERMINSCC has failed, and applies CFR to the input ITS taking into account only those edges.

For $\mathtt{CFR_S}$ and $\mathtt{CFR_A}$ schemes, CFR can be applied iteratively so that refinement is interleaved with termination proofs attempts. In this way, each step might introduce further refinements that could not be done in previous steps. The precision and performance of these schemes are compared experimentally in Sect. 5. Alg. 1 shows the pseudocode of a termination analysis algorithm that uses the above schemes for CFR. It consists of two procedures TERMIN and TERMINITS, and uses TERMINSCC as a black box. It also uses some auxiliary procedures that we explain below.

TERMIN receives an ITS $\mathcal{T}$, a Boolean $\mathtt{CFR_B}$ indicating whether $\mathtt{CFR_B}$ should be applied, and integers $\mathtt{CFR_A}$ and $\mathtt{CFR_S}$ giving the number of times that the respective schemes can be applied. At Line 2 (L2 for short) it calls PE($\mathcal{T}$) if $\mathtt{CFR_B}$ is **True**, and at L3 it calls TERMINITS to analyze the SCCs of $\mathcal{T}$ for the first time. TERMINITS returns the set F of edges for which it failed to show termination. If $F = \emptyset$ it halts, otherwise, it enters a loop as long as $\mathtt{CFR_A}$ can be applied ($\mathtt{CFR_A} > 0$) and termination has not been proven ($F \neq \emptyset$). At L11 it returns **True** if $F = \emptyset$, showing that $\mathcal{T}$ is terminating, and **False** otherwise.

In each iteration of the refine/analyze loop (L5-10) it applies CFR only to the parts corresponding to F. At L5 it computes the set of nodes N in F; at L6 it removes from $\mathcal{T}$ nodes that do not reach nodes in N (since those parts of $\mathcal{T}$ have been proven terminating and they do not affect the CFR of any node in N); at L7 it applies CFR considering only loop heads in N; at L8 it removes from $\mathcal{T}$ nodes not in N as they have been proven terminating already; at L9 it decreases the $\mathtt{CFR_A}$ scheme counter; and finally at L10 it calls TERMINITS again to analyze $\mathcal{T}'$. Note that the nodes removed at L8 are not removed before at L6 in order to (a) guarantee soundness, as CFR must consider all possible ways in which nodes in N are reached; (b) allow CFR to benefit from context information; and (c) allow other parts reachable from nodes in N to benefit from refinements.

TERMINITS analyzes the SCCs of $\mathcal{T}$, and applies CFR at the level of SCCs if needed.

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

At L13 it initializes local variables F and Q, where F is used to accumulate the edges that it fails to prove terminating (returned at the end at L22), and Q is a queue of pending (sub) ITSs to be analyzed. The elements of Q are pairs $\langle \mathcal{T}', i\rangle$, where $\mathcal{T}'$ is an ITS to be analyzed and i is the number of times left to apply CFR to its SCCs. The *while* loop is executed as long as there are pending ITSs in Q: at L15 it takes an ITS $\mathcal{T}'$ from Q, and at L16-21 tries to prove termination of its SCCs. This is done by calling TerminSCC on S at L17, and if it fails ($F_S \neq \emptyset$) it applies CFR at L19-20 to the problematic part F_S if possible ($i > 0$), otherwise it adds F_S to F at L21. Applying CFR to F_S is done as follows: at L19 it builds a new ITS from F_S, by calling ConITS(F_S), and applies CFR to it. ConITS builds an ITS that consists of the nodes and edges of F_S, together with a new entry node that has edges to all nodes of F_S that are reachable from nodes not in F_S. This is because CFR must consider all possible ways in which nodes of F_S are reached (we assume that ConITS has global access to the original ITS); and at L20 it adds the refined ITS to Q ($\mathcal{T}''$ might have several SCCs now).

Next we show some examples of how Alg. 1, can benefit termination analysis, using the different CFR schemes. For all examples we give programs and ITSs, but we omit CHCs as they are similar to ITSs. ITSs are sometimes simplified (e.g. by joining chains of nodes) for presentation; however, they are very similar to what we get in practice – see Sect. 5.

CFR can simplify the termination witness from LLRFs to LRFs, which is useful if the underlying analyzer does not use LLRFs, and can help in making cost analysis feasible as well. For example, consider again method `phases1` of Fig. 1(a) and its corresponding ITS $\mathcal{T}$. It is easy to prove that $\mathcal{T}$ terminates using the LLRF $\langle z - y, x\rangle$ for nodes $\mathtt{n}_1$ and $\mathtt{n}_2$, but it is not possible if we restrict ourselves to the use of LRFs. Calling Termin($\mathcal{T}$, *true*, 0, 0) applies CFR at L2 before trying to prove termination, yielding the ITS $\mathcal{T}_{pe}$ of Fig. 1(a). The two loop phases are now explicit: nodes $\mathtt{n}_1^2$ and $\mathtt{n}_2^2$ correspond to the first phase, and nodes $\mathtt{n}_1^1$ and $\mathtt{n}_2^1$ to the second phase. Using this refined ITS, TerminITS finds LRFs $z-y$ and x for the corresponding SCCs. The next example discusses cases for which CFR is essential for proving termination, not only for simplifying the form of the witness.

Example 4.1
Consider method `search` of Fig. 1(b). It receives an array q representing a circular queue, the size of the array n, the indexes h and t of the head and tail, and a value v to search for. The loop first (if $h > t$) searches for v in the interval $[h..n-1]$ and then in $[0..t]$. This defines two phases where moving from the first to the second is done by setting h to 0. It is easy to see that (a) in the first phase, $n - h$ is non-negative and decreasing in all iterations except the last, i.e., when setting h to 0 and (b) in the second phase, $n - h$ is decreasing and non-negative. The last iteration of the first phase makes automatic proofs subtle, because it breaks the conditions that a LRF or a LLRF has to satisfy as function $n - h$ increases when setting h to 0. If we succeed in splitting these phases into separate loops, such that the last iteration of the first phase is a transition that connects them, then proving termination should be possible with LRFs only. Unlike method `phases1` of Fig. 1(a), in this one the two phases execute the same code, i.e., the *then* branch at PP4 (short for program point 4).

The ITS $\mathcal{T}$ of this program is shown in Fig. 1(b). Node $\mathtt{n}_1$ corresponds to the loop head, and $\mathtt{n}_2$ to the *if* statement. Assume that this loop is the only loop in a larger program that we cannot prove terminating, so as to take advantage of applying CFR at the level

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

of SCCs. Calling $\textsc{Termin}(\mathcal{T}, \mathit{false}, 0, 1)$ we eventually reach L17 with SCC S containing nodes $\mathtt{n}_1$ and $\mathtt{n}_2$. $\textsc{Termin}$SCC(S) fails to prove termination of any edge of S and returns the set of edges of S. The new ITS $\mathcal{T}''$ at L19 is like the ITS $\mathcal{T}$ of Fig. 1(b), but without the exit node $\mathtt{n}_3$. Applying CFR at L19 using properties $\{h = 0, h \leq t\}$ (or $\mathtt{PR_{cv}}$ and $\mathtt{PR_{c}}$) we obtain $\mathcal{T}_{pe}$ shown in Fig. 1(b), which is added at L20 to Q in order to analyze it again. Node $\mathtt{n}_1$ of $\mathcal{T}$ now has 2 versions in $\mathcal{T}_{pe}$: $\mathtt{n}_1^2$ is for the first phase which excludes the last step that sets h to 0, which is now handled by the edge $\mathtt{n}_2^2 \overset{\varphi_7}{\to} \mathtt{n}_1^1$; and node $\mathtt{n}_1^1$ is for the second phase. When $\mathcal{T}_{pe}$ is analyzed, $\textsc{Termin}$SCC finds LRFs for both SCCs of $\mathcal{T}_{pe}$. □

The next example shows (1) how CFR is useful for inferring precise invariants, without using disjunctive abstract domains; and (2) the importance of the scheme $\mathtt{CFR_A}$.

Example 4.2
Let us modify method `search` (and its ITS $\mathcal{T}$) to include another variable w, that is initialized to $t - h + 1$ before the loop and is set to 1 in the *else* branch at PP4. The initial value of w is at least 1 if $h \leq t$, and it is at most 0 if $h > t$. However, in the later case the execution eventually passes through the *else* branch and sets w to 1. This means that $w \geq 1$ holds after the loop. Using the ITS $\mathcal{T}$ of this program, invariant generators would fail to infer this information without relying on disjunctive invariants, e.g. $(h \leq t \wedge w \geq 1) \vee (h > t \wedge w \leq 0)$ for node $\mathtt{n}_1$. However, when using $\mathcal{T}_{pe}$ they succeed because in $\mathcal{T}_{pe}$ it is explicit that the second phase is reached, in either way, with $w \geq 1$.

Precise invariants are essential for the precision of termination analysis. Assume that the loop of method `search` is followed by a second loop "`while (x>=1) x=x-w;`", then termination analysis of this loop would fail without the invariant $w \geq 1$ since the loop is non-terminating for $w \leq 0$. Note that in order to propagate $w \geq 1$ from the first loop to the second, we cannot use the scheme $\mathtt{CFR_S}$ since it analyzes the SCCs independently, and thus after applying CFR to the SCC of the first loop the new invariants are not propagated to the SCC of the second loop. Instead, we can use $\mathtt{CFR_A}$ by calling $\textsc{Termin}(\mathcal{T}, \mathit{false}, 1, 0)$. In this case, the first call to $\textsc{Termin}$ITS at L3 fails to prove any of the two loops terminating, and then CFR at L2 is applied on both loops together and now the constraint $w \geq 1$ is propagated to the second loop. One could also use $\mathtt{CFR_B}$, but it might be less efficient if the program is part of a larger one that does not need CFR to handle other SCCs. □

Next we discuss one of the examples that no tool could handle in the last termination competition (TERMCOMP 2019), except `iRankFinder` when using CFR.

Example 4.3
Consider method `randomwalk` of Fig. 1(c). It simulates a random walk where w is repeatedly increased or decreased at PP9 depending on the random choice for c at PP4. The code at PP5-8 ensures termination. Assuming that $n \geq 1$, the loop passes through the following phases: (a) $z \geq 1$, so z is decremented at PP5 until it reaches 0; (b) since $z = 0$ and $i = 1$ now, it either executes PP8 and exits the loop, or executes PP7 which decrements i to 0 and in the next iteration executes PP5 and sets i to 2 and z back to n; (c) since $z \geq 1$ now, it is decremented at PP5 until it reaches 0; (d) since $z = 0$ and $i = 2$ now, it either executes PP7 twice, which means that $c = 0$ and thus at PP9 w is decremented twice to 0 and exits the loop, or it executes PP8 at least once and exits the loop. The case of $n \leq 0$ passes in steps (b) and (d) only. Applying CFR to $\mathcal{T}$ of Fig. 1(c), e.g. using properties $\mathtt{PR_{cv}}$ and $\mathtt{PR_h^d}$, we obtain the ITS $\mathcal{T}_{pe}$ sketched in Fig. 1(c) (each box

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

is an SCC with several nodes and a single cycle that decrements z). Calling TERMIN on $\mathcal{T}$ with any of the schemes refines the graph to something like $\mathcal{T}_{pe}$, and then proves termination with LRF z for both SCCs. □

Cost analysis. In the rest of this section we discuss the use of CFR for cost analysis, namely the inference of upper-bound functions on the length of traces (see Sect. 2). There are several cost analysis tools for (variants of) ITSs, and they are based on similar techniques to those used in termination analysis. In particular, they use ranking functions to infer *visit-bounds*, which are upper-bounds on the number of visits to edges in the ITS.

`KoAT` (Brockschmidt et al. 2016) is a tool that works directly on ITSs as defined in Sect. 2 and uses quasi-LRFs (Alias et al. 2010) to infer visit-bounds. `CoFloCo` (Flores-Montoya 2017) is a tool that works on a form of ITSs that is called cost relations (CRSs), which are similar to recurrence relations. It mainly uses LRFs to infer visit-bounds, and it applies CFR (directly to CRSs) to increase precision. The CFR techniques of `CoFloCo` are very similar to those of Gulwani et al. (2009). `PUBs` (Albert et al. 2011) is the first tool to infer upper-bounds for CRSs. It uses LRFs to infer visit-bounds and it does not apply any kind of CFR. In terms of precision, `CoFloCo` can handle anything that `PUBs` can, and it is not directly comparable to `KoAT` as there are cases where one succeed and the other fail to infer upper-bounds.

In the context of cost analysis, CFR can improve the precision of inferring visit-bounds in a way that is similar to simplifying the witness of a termination proof. The next example shows this for all examples that we have discusses so far. For CFR in this context we simply apply procedure `PE` of Sect. 3 to the input ITS.

Example 4.4
For Fig 1(a), `KoAT` infers a linear upper-bound without CFR thanks to the use of quasi-LRFs, which allow the inference of x and $z-y$ as visit-bounds for $\mathtt{n_2} \xrightarrow{\varphi_4} \mathtt{n_1}$ and $\mathtt{n_2} \xrightarrow{\varphi_3} \mathtt{n_1}$, respectively; `CoFloCo` infers a linear upper-bound since it applies CFR that splits the loop into phases; `PUBs` fails to infer any upper-bound, however, it infers a linear upper-bound when applied to the ITS after CFR. For Ex. 4.1, `KoAT` and `PUBs` fail to infer any upper-bound, but they infer a linear upper-bound after applying CFR; `CoFloCo` infers a linear upper-bound since it applies its own CFR that splits the loop into two phases. For Ex. 4.2, all tools fail to infer an upper-bound, and all infer a linear upper-bound after applying CFR (note that `CoFloCo`'s own CFR is insufficient here). For Ex. 4.3, all tools fail to infer any upper-bound, and they infer a linear upper-bound for the refined ITS. □

Next we discuss another direction where ranking functions are used as properties for CFR. Sometimes we can prove termination of an ITS but cannot infer an upper-bound on its cost. One reason is that in termination analysis we can use ranking functions that do not necessarily imply an upper-bound on the length of traces, e.g. LLRFs. However, for some kind of LLRFs, even if they do not imply an upper-bound, they provide useful information about loop control flow. Our goal is to use these ranking functions to help CFR to make this control flow explicit, which helps cost analysis to infer upper-bounds.

Example 4.5
Consider method `phases2` of Fig 1(d) and its ITS $\mathcal{T}$. Termination analysis succeeds in proving termination, assigning node $\mathtt{n_1}$ the LLRF $\langle z,y,x\rangle$. This LLRF is in a sub-class

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

called *multi-phase* linear ranking functions (MΦRFs) (Ben-Amram and Genaim 2017); it has the following properties (considering consecutive visits to $\mathtt{n_1}$): z decreases, and when it becomes negative y starts to decrease, and when y becomes negative x starts to decrease. This means that on reaching $\mathtt{n_1}$ one of the following must hold: $z \geq 0$, $z < 0 \wedge y \geq 0$, $z < 0 \wedge y < 0 \wedge x \geq 0$ or $z < 0 \wedge y < 0 \wedge x < 0$. The ITS $\mathcal{T}'$ of Fig 1(d) is a modification of $\mathcal{T}$ making this information explicit: we added a new node $\mathtt{n_{1a}}$, changed all incoming edges of $\mathtt{n_1}$ to go to $\mathtt{n_{1a}}$, and added 4 edges from $\mathtt{n_{1a}}$ to $\mathtt{n_1}$, each annotated with one of the above constraints. Applying CFR to $\mathcal{T}'$ results in the ITS $\mathcal{T}'_{pe}$ in which the phases are explicit (nodes $\mathtt{n^1_{1a}}$, $\mathtt{n^2_{1a}}$, and $\mathtt{n^3_{1a}}$). Cost analysis tools can infer a linear upper-bound for $\mathcal{T}'_{pe}$. Note that the addition of node $\mathtt{n_{1a}}$ is essential.

The above example can be generalized as follows. If a loop-head node $\mathtt{n_i}$ is assigned a MΦRF $\langle f_1, \ldots, f_k \rangle$ during termination analysis: we add a new node $\mathtt{n_{ia}}$ and change all incoming edges of $\mathtt{n_i}$ to $\mathtt{n_{ia}}$; and add $k+1$ edges from $\mathtt{n_{ia}}$ to $\mathtt{n_i}$ where the ith edge has the constraints $f_1(\bar{x}) < 0 \wedge \cdots \wedge f_{i-1}(\bar{x}) < 0 \wedge f_i(\bar{x}) \geq 0 \wedge \mathtt{ID}(\bar{x})$. Applying CFR to the new ITS takes advantage of the new edges, and splits the loop into corresponding phases.

5 Implementation and Experimental Evaluation

In this section we discuss implementation and experimental evaluation. More details are available in the technical report (Doménech et al. 2019) and at the companion website `http://irankfinder.loopkiller.com/papers/extra/iclp19`.

We provide a standalone tool that receives an ITS (in several formats) and generates an ITS after applying CFR using partial evaluation (Gallagher 2019). It accepts any of the properties discussed in Sect. 3 (the user selects the heuristics, and then they are inferred automatically), as well as user-supplied properties and can be applied to a subset of the ITS. In addition, it includes an invariants generator that can be optionally used. Apart from the standalone tool, we modified `iRankFinder` to incorporate Alg. 1.

We have evaluated the effect of Alg. 1 on `iRankFinder` using standard sets of benchmarks (TPDB 2019; Flores-Montoya 2017). We analyzed all benchmarks (except those known to be nonterminating) in two *settings*, using LRFs and LLRFs, and using different schemes and properties. Out of 556 (resp. 143) ITSs that it cannot handle using LRFs (resp. LLRFs) without CFR, it can handle 251 (resp. 48) with CFR. As expected, CFR has performance overhead that depends on the scheme and the kind of properties used. Experiments show that properties $\mathtt{PR^d_h}$ and $\mathtt{PR_c}$ are important for precision, and that using them with $\mathtt{CFR_S}$ provides the best performance/precision trade-off. The time spent on CFR can be up to 42% of the total time, depending on the scheme and properties used (as expected, $\mathtt{CFR_B}$ takes more time than $\mathtt{CFR_A}$ and $\mathtt{CFR_S}$).

We have also analyzed this set of benchmarks using other termination analyzers of ITSs: VeryMax (Borralleras et al. 2017) and T2 (Brockschmidt et al. 2016). There are 8 (resp. 14) examples that we can handle due to the use of CFR and VeryMax (resp. T2) cannot handle without CFR. For benchmarks where VeryMax (resp. T2) failed, we have applied them again on the ITSs after CFR (using scheme $\mathtt{CFR_B}$) and it could prove termination of 6 (resp. 39) more benchmarks and nontermination of 80 (resp. 40) benchmarks.

For cost analysis, we applied `KoAT` on a set of 200 benchmarks. With CFR we got tighter upper-bounds for 15 benchmarks. Using $\mathtt{PR^d_h}$ and $\mathtt{PR_c}$ gave the best precision. We did not

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

run PUBs on this set since it is not directly designed for ITSs, and more work is required to transform ITSs to CRSs without losing precision (i.e., inferring loop summaries which is usually done by the frontend that uses PUBs). We did not run CoFloCo on this set since it includes a CFR component for CRSs. However, as we have seen in the examples of Sect. 4, our CFR procedure can improve the precision of both PUBs and CoFloCo.

We used the invariant generator of iRankFinder to prove the assertions in 13 programs from Sharma et al. (2011). Without CFR, it proved the assertions for 5 of them, and with CFR it did so for all benchmarks. Also here, $\mathtt{PR}^{\mathtt{d}}_{\mathtt{h}}$ and $\mathtt{PR}_{\mathtt{c}}$ provided the most precise results. We have also evaluated our CFR procedure on all examples of Fig. 1 and Fig. 2 of Gulwani et al. (2009), and got similar refinements. The tools of Sharma et al. (2011) and Gulwani et al. (2009) are not available so we cannot experimentally compare to them.

6 Conclusions, Related and Future Work

In this work we proposed the use of partial evaluation as a general purpose technique to achieve CFR. Our CFR procedure is developed for ITSs, and used a partial evaluation technique for Horn clause programs based on specializing programs wrt. a set of supplied properties. The right choice of properties is a key factor for achieving the desired CFR, and we suggested several heuristics for inferring properties automatically. We provide an implementation that can be used as a preprocessing step for any static analysis tool that uses ITSs, and integrated it in iRankFinder in a way that allows application of CFR only if needed. Experimental evaluation demonstrated that our CFR procedure enables termination proofs and cost inference of many ITSs that we could not handle before.

Closely related work has been discussed in Sect. 1. The use of partial evaluation for CFR is not directly comparable to these works; however, experiments show that we achieve similar results. In particular in the examples of Sect. 4 we discussed some cases that Flores-Montoya and Hähnle (2014) cannot handle, and in Sect. 5 we have seen that we achieve similar refinements for the examples of Gulwani et al. (2009). In addition, we could handle all examples of Sharma et al. (2011). We conjecture that the partial evaluation algorithm achieves the same refinement as the technique of Sharma et al. (2011), provided that the splitter predicates (or their negations) are among the properties provided to the algorithm. Sharma et al. generate candidate splitter predicates from the conditions occurring in loops, using the weakest precondition operator to project them onto the loop head, which corresponds closely to our property generation $\mathtt{PR}^{\mathtt{d}}_{\mathtt{h}}$.

There is a close relationship between partial evaluation of logic programs (sometimes called partial deduction) and abstract interpretation of logic program with respect to a goal (top-down abstract interpretation); both can be expressed in a single generic framework parameterised by an abstract domain and an unfolding strategy (Puebla et al. 1999; Leuschel 2004; Puebla et al. 2006). The combination of the two techniques can be mutually beneficial, as shown in (Puebla et al. 2006). In top-down abstract interpretation, the aim is to derive call- and answer-patterns, which are described by abstract substitutions over program variables, whereas in partial evaluation, the aim is to unfold parts of the computation and derive a program specialised for the given goal. The versions in a polyvariant partial evaluation correspond to multiple call-patterns in top-down abstract interpretation. Viewing CFR as an instance of a generic framework, the choice of abstract domain and unfolding strategy are crucial, and therefore this paper focusses on

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

those aspects. There are many ways to achieve polyvariance in partial evaluation, and obtaining a good set of versions leading to useful refinements requires careful choice of the abstract domain, which in our case is the power set of the given set of properties. The control of unfolding is also critical, as unfolding choice predicates leads to a trade-off between specialisation and blow-up in the size of the specialised program.

Logic program specialisation has been previously applied as a component in program verification tools, with a goal similar to the one in this paper. Namely, the specialisation of a program with respect to a goal (corresponding to a property to be proved) can enable the derivation of more precise invariants, contributing to a proof of the property (Leuschel and Massart 2000; De Angelis et al. 2012; Fioravanti et al. 2012; Kafle et al. 2018). Polyvariant specialisation is often crucial, allowing (in effect) the inference of disjunctive invariants. Again, for CFR the choices for abstraction and unfolding strategy are important, to achieve the right balance between precision and the size of the specialised programs. In (De Angelis et al. 2012), an operation called constrained generalisation is used; this identifies constraints on a predicate that determine control flow. A generalisation operator on constraints is designed to preserve the control flow. This has a relation to property-based abstraction but we find it more natural and controllable to let the properties determine the control flow rather than the other way round. However, further evaluation of different abstractions is needed. We performed some experiments on examples from this paper using a general-purpose logic program specialiser (ECCE (Leuschel et al. 2006)) but the abstraction used in that tool did not result in any useful CFR.

For future work, we plan to explore further the automatic inference of properties, in particular the use of ranking functions as discussed in Sect. 4. Moreover, we want to explore the use of CFR for other static analysis tools, where the data is not numerical. It is also worth investigating whether the insights from CFR could be used to improve general-purpose specialisation tools.

References

Albert, E., Arenas, P., Genaim, S., and Puebla, G. 2011. Closed-form upper bounds in static cost analysis. *J. Autom. Reasoning 46,* 2, 161–203.

Alias, C., Darte, A., Feautrier, P., and Gonnord, L. 2010. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Static Analysis Symposium, SAS'10*, R. Cousot and M. Martel, Eds. LNCS, vol. 6337. Springer, 117–133.

Bagnara, R., Hill, P. M., and Zaffanella, E. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program. 72,* 1-2, 3–21.

Bagnara, R., Mesnard, F., Pescetti, A., and Zaffanella, E. 2012. A new look at the automatic synthesis of linear ranking functions. *Inf. Comput. 215*, 47–67.

Ben-Amram, A. M. and Genaim, S. 2014. Ranking functions for linear-constraint loops. *Journal of the ACM 61,* 4 (July), 26:1–26:55.

Ben-Amram, A. M. and Genaim, S. 2017. On multiphase-linear ranking functions. In *Computer Aided Verification, CAV 2017*, R. Majumdar and V. Kuncak, Eds. LNCS, vol. 10427. Springer, 601–620.

Borralleras, C., Brockschmidt, M., Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., and Rubio, A. 2017. Proving termination through conditional termination. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'17*, A. Legay and T. Margaria, Eds. LNCS, vol. 10205. 99–117.

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press

Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., and Piterman, N. 2016. T2: temporal property verification. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS 2016*, M. Chechik and J. Raskin, Eds. LNCS, vol. 9636. Springer, 387–393.

Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., and Giesl, J. 2016. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst. 38,* 4, 13:1–13:50.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Fifth Annual ACM Symposium on Principles of Programming Languages, POPL'78*, A. V. Aho, S. N. Zilles, and T. G. Szymanski, Eds. ACM Press, 84–96.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2012. Specialization with constrained generalization for software model checking. In *LOPSTR 2012*, E. Albert, Ed. LNCS, vol. 7844. Springer, 51–70.

Doménech, J. J., Gallagher, J. P., and Genaim, S. 2019. Control-flow refinement by partial evaluation, and its application to termination and cost analysis. *CoRR abs/1907.12345.* https://arxiv.org/abs/1907.12345.

Fioravanti, F., Pettorossi, A., Proietti, M., and Senni, V. 2012. Improving reachability analysis of infinite state systems by specialization. *Fundam. Inform. 119,* 3-4, 281–300.

Flores-Montoya, A. 2017. Cost analysis of programs based on the refinement of cost relations. Ph.D. thesis, Darmstadt University of Technology, Germany.

Flores-Montoya, A. and Hähnle, R. 2014. Resource analysis of complex programs with cost equations. In *Asian Symposium on Programming Languages and Systems, APLAS 2014*, J. Garrigue, Ed. LNCS, vol. 8858. Springer, 275–295.

Gallagher, J. P. 2019. Polyvariant program specialisation with property-based abstraction. In *Pre-proceedings of Verification and Program Transformation, VPT'19*, A. Lisitsa and A. P. Nemytykh, Eds. Available at http://refal.botik.ru/vpt/vpt2019/VPT2019_paper_5.pdf. Accepted for EPTCS.

Gulwani, S., Jain, S., and Koskinen, E. 2009. Control-flow refinement and progress invariants for bound analysis. In *Programming Language Design and Implementation, PLDI'09*, M. Hind and A. Diwan, Eds. ACM, 375–385.

iRank 2019. iRankFinder. http://irankfinder.loopkiller.com.

Kafle, B., Gallagher, J. P., Gange, G., Schachte, P., Søndergaard, H., and Stuckey, P. J. 2018. An iterative approach to precondition inference using constrained Horn clauses. *TPLP 18,* 3-4, 553–570.

Leuschel, M. 2004. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst. 26,* 3, 413–463.

Leuschel, M., Elphick, D., Varea, M., Craig, S., and Fontaine, M. 2006. The Ecce and Logen partial evaluators and their web interfaces. In *PEPM 2006*, J. Hatcliff and F. Tip, Eds. ACM, 88–94.

Leuschel, M. and Massart, T. 2000. Infinite state model checking by abstract interpretation and program specialisation. In *LOPSTR'99*, A. Bossi, Ed. LNCS, vol. 1817. 63–82.

Podelski, A. and Rybalchenko, A. 2004. A complete method for the synthesis of linear ranking functions. In *Verification, Model Checking, and Abstract Interpretation, VMCAI'04*, B. Steffen and G. Levi, Eds. LNCS, vol. 2937. Springer, 239–251.

Puebla, G., Albert, E., and Hermenegildo, M. V. 2006. Abstract interpretation with specialized definitions. In *SAS 2006*, K. Yi, Ed. LNCS, vol. 4134. Springer, 107–126.

Puebla, G., Hermenegildo, M., and Gallagher, J. P. 1999. An integration of partial evaluation in a generic abstract interpretation framework. In *PEPM'99*, O. Danvy, Ed. Technical report BRICS-NS-99-1. University of Aarhus, 75–84.

Sharma, R., Dillig, I., Dillig, T., and Aiken, A. 2011. Simplifying loop invariant generation using splitter predicates. In *Computer Aided Verification, CAV 2011*, G. Gopalakrishnan and S. Qadeer, Eds. LNCS, vol. 6806. Springer, 703–719.

TERMCOMP 2019. http://termination-portal.org/wiki/Termination_Competition_2019.

TPDB 2019. http://termination-portal.org/wiki/TPDB.

https://doi.org/10.1017/S1471068419000310 Published online by Cambridge University Press