Theory of Computing Systems (2022) 66:911–956
https://doi.org/10.1007/s00224-022-10093-w

A Framework for Memory Efficient Context-Sensitive Program Analysis

Mathias Hedenborg[1] · Jonas Lundberg[1] · Welf Löwe[1] · Martin Trapp[2]

Accepted: 13 June 2022 / Published online: 18 July 2022

Abstract

Static program analysis is in general more precise if it is sensitive to execution contexts (execution paths). But then it is also more expensive in terms of memory consumption. For languages with conditions and iterations, the number of contexts grows exponentially with the program size. This problem is not just a theoretical issue. Several papers evaluating inter-procedural context-sensitive data-flow analysis report severe memory problems, and the path-explosion problem is a major issue in program verification and model checking. In this paper we propose χ-terms as a means to capture and manipulate context-sensitive program information in a data-flow analysis. χ-terms are implemented as directed acyclic graphs without any redundant subgraphs. We introduce the *k-approximation* and the *l-loop-approximation* that limit the size of the context-sensitive information at the cost of analysis precision. We prove that every context-insensitive data-flow analysis has a corresponding *k, l-approximated* context-sensitive analysis, and that these analyses are sound and guaranteed to reach a fixed point. We also present detailed algorithms outlining a compact, redundancy-free, and DAG-based implementation of χ-terms.

Keywords Static program analysis · Data-flow analysis · Context-sensitivity

✉ Mathias Hedenborg
Mathias.Hedenborg@lnu.se

Jonas Lundberg
Jonas.Lundberg@lnu.se

Welf Löwe
Welf.Lowe@lnu.se

Martin Trapp
Martin.Trapp@senacor.com

[1] Linnaeus University, Växjö, Sweden

[2] Senacor Technologies AG, Nuremberg, Germany

1 Introduction

Static program analysis is an important part of both optimizing compilers and software engineering tools for program verification and model checking. Static analyses approximate the run-time behavior of a given program. This is done by abstracting from the concrete semantics of programs and from concrete values. Such analyses can be context-sensitive or -insensitive, i.e., an analysis may or may not distinguish different analysis results for different execution paths, i.e. different *contexts*, e.g., different call contexts of a method or alternative intra-procedural executions paths due to control statements. Context-sensitive analyses are, in general, more precise than their context-insensitive counterparts, but also more expensive in terms of time and memory consumption.

With conditional execution, the number of different contexts grows, in general, exponentially with the program size. Adding iterations leads, in general, to countable infinitely many contexts. With the number of contexts grow the analysis time and the memory to capture a mapping of contexts to analysis results. Merging the analyzed information of different contexts reduces the memory consumption at the cost of analysis precision. We should therefore aim for compact representations of the mapping of context to analysis information.

The memory usage problem related to context-sensitive analyses is not just a theoretical issue; several papers, e.g., [19, 25], evaluating various inter-procedural context-sensitive data-flow approaches report severe memory problems when using call context sensitivity with a call-depths $k \geq 2$, and the path-explosion problem, e.g., [5, 8], has for a long time been a major issue in program verification and model checking.

In this paper, we present a technique to capture context-sensitive analysis information in general. We do not distinguish between inter-procedural call context sensitivity [34] and intra-procedural trace or path sensitivity [32]. In both cases we map contexts to analysis values for each program point in a memory efficient way. Our approach is based on so-called χ-terms [39, 40] that capture the analysis results of different contexts. The main benefit of χ-terms is that it avoids all syntactic redundancies of sub-terms and values and is, hence, a memory efficient representation of context-sensitive analysis information [12].

We assume a Static Single Assignment (SSA) [10, 29] representation of the programs. We further assume a program analysis following a standard data-flow analysis approach as given, e.g., in [26]. It iterates over an SSA representation of the program and updates the analysis information at each node using the node's transfer functions until a fixed point is reached. It can be implemented using a worklist initialized with a start node. Iteratively, a node is taken from the worklist, its transfer function is applied, and, if its analysis value has changed, all control-flow dependent nodes are added to the worklist until it is empty.

In SSA graphs, ϕ-nodes are used to select between different definitions of a variable assigned in different control paths of the execution. There is a close relation between ϕ-nodes and χ-terms. We create a new χ-term as the result of the transfer function of a ϕ-node. Each χ-term encodes the various control-flow dependent analysis value options that were available when the transfer function was applied.

The reason why χ-terms are expected to be so memory efficient—and actually turn out to be in practice [12]—in capturing program analysis values is the following. A ϕ-node defines a χ-term that merges the analysis values from $p \geq 2$ predecessor paths or contexts, each encoded in one of the p sub-terms. They have been created in the p contexts and, in general, need to be captured as part of the respective context's analysis result. Instead of creating a whole new χ-term representation, the new term is represented by a new χ-node referring to the p existing χ-sub-term representations. Another advantage of χ-terms is that they capitalize on the redundancy savings *during their construction*, i.e., while executing the transfer function of the corresponding ϕ-node. This avoids using extensive memory first before redundancies are possibly eventually eliminated. The memory footprint of the χ-term representations just grows monotonically and expensive explicit memory management, i.e, redundancy elimination and garbage collection, is avoided.

Traditionally context-sensitivity is used for both intra- and inter-procedural dataflow analysis. However, for the sake of simplicity, our presentation of χ-terms in Sections 2–5 will focus on intra-procedural context-sensitivity where the various contexts are due to control statements. It is, however, important to notice that χ-terms can be used for any type of SSA-based context-sensitive static program analysis, e.g., for an intra-procedural compiler optimization as well as for an intra-procedural program verification.

There are also similarities between our χ-terms (represented as directed acyclic graphs) and the (directed, cyclic) value graphs that are used in *Global Value Numbering* [2], since they also capture the history of control flow. The main difference between the two approaches is the memory efficiency we achieve by avoiding data redundancy. Our approach can be seen as a generalization of Global Value Numbering where we trade precision for memory efficiency. This will be further discussed in Section 6.

χ-terms can also be understood as a generalization of *Binary Decision Diagrams* (BDD) [1] used to represent logical functions. An BDD is a redundancy-free representation of Boolean functions as a directed acyclic graph (DAG) that allows an efficient computation of frequently used operations such as disjunction, conjunction and function composition. A logical function $f(A, B, C) = \neg AB\neg C \vee AC$, e.g., with the truth table given in Fig. 1 (left), can be represented as a decision tree (middle). Removing and reusing redundant subtrees can simplify the decision tree and result in a much smaller BDD (right). Enforcing the same order of the parameters in a BDD on each path from the root to a leaf leads to *Ordered* BDDs. The main idea from OBDDs that we use in our implementation of χ-terms is the DAG representation and the redundancy elimination for memory efficiency. Furthermore, the leaves on OBDDs represent decisions based on the logical values (true/false) whereas χ-terms allow for multiple decisions in the leaves (the leaves represent context-insensitive analysis values).

It is NP-hard to find a parameter order leading to a minimal representation of the corresponding OBDD [6]. While memory efficiency is one of the main objectives of our approach, optimizing the order of nodes in χ-terms is not a part of this paper. Instead we simply use an order that is defined by the order in which the corresponding

A	B	C	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

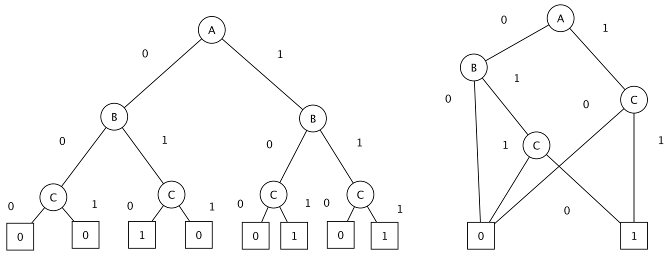

Fig. 1 Different representations of the function f(A,B,C) = $\neg AB\neg C \vee AC$

ϕ-nodes are analyzed. Related program analysis approaches exploiting (O)BDD ideas will be further discussed in Section 6.

This paper is a complement to our paper *Memory Efficient Context-Sensitive Program Analysis* [12]. In [12] we give an informal presentation of χ-terms and evaluate the memory efficiency by comparing the memory foot-prints of χ-terms with four other data structures (context table, context tree in two variants, and double hash map) often used to capture context-sensitive analysis information. The experiments use context-sensitive points-to analysis information taken from ten different benchmark programs. The results show that χ-terms are indeed memory efficient. They use on average only 13% of the memory used by the second best approach. A closer look at the context-sensitive analysis values used in the experiments show that they in general have a rather complex structure when presented using the other data structures, it is only the χ-terms with their redundancy elimination that can deduce (and make use of) the fact that a majority of these structures can be simplified, and made much smaller. Hence, [12] provides the experimental evidence that χ-terms are memory efficient, this paper presents χ-terms as a general framework for memory efficient context-sensitive program analyses.

Our contributions in this paper are the following:

1. We propose χ-terms, a generalization of OBDDs, as a memory efficient representation, to capture context-sensitive analysis values in an SSA-based program analysis.
2. We propose approximations in the representation of χ-terms to control memory explosion when handling context-sensitive analysis information in conditional control-flow and iterations (control-flow cycles), including the handling of unbounded iterations.
3. We prove that any sound context-insensitive analysis has a corresponding sound context-sensitive analysis based on χ-terms that is guaranteed to reach a fixed point.

The remainder of the paper is structured as follows:

– In Section 2, we define concrete and abstract analysis semantics, and introduce the term representation of context-insensitive analysis results.

- In Section 3, we introduce χ-terms and operations on χ-terms. We also prove that a sound context-insensitive analysis can be generalized to a sound context-sensitive analysis using χ-terms.
- In Section 4, we address the problem of terminating a data-flow analysis. The problem with the potentially infinite size of a χ-term can be avoided by introducing different widening approximations. We introduce such approximations and show that the resulting approximated analysis is sound and guaranteed to reach a fix point.
- In Section 5, we introduce and discuss the compact representation of χ-terms using redundancy-free direct acyclic graphs (DAGs), a generalization of Ordered Binary Decision Diagrams (OBDDs). Moreover, we present a fast and memory efficient algorithm for creating approximated χ-term in DAG format that avoids redundant sub-terms. We conclude the section by discussing memory management and its effect on the memory footprint.
- Section 6 discusses related works.
- Section 7 concludes the paper.

2 The Term Representation of Context-Insensitive Analysis

In this section, we introduce assumptions and notations that will be used throughout this paper. We start by outlining concrete and abstract analysis semantics for a data-flow analysis followed by a term representation for a context-insensitive analysis, which we then take as a point of departure for defining context-sensitive analysis values.

2.1 Concrete Analysis Semantics

Let a program $P \in \mathcal{P}$ ($\mathcal{P}$ the set of all syntactically correct programs) be represented by its program graph $G = (N, E, n^0) \in \mathcal{G}$ ($\mathcal{G}$ the set of all such program graphs G of programs in $\mathcal{P}$) in Static Single Assignment (SSA) form [10, 29] with

- N a set of nodes taken from a finite set of node kinds including ϕ nodes representing selections of values computed in different program paths,
- E a set of data- and control-dependency edges, and
- $n^0 \in N$ the unique start node without any predecessors in G.[1]

The nodes in an SSA graph represent data- and control-operations, and the edges their dependencies. The specific node type ϕ is used to represent a selection of one of the dynamic predecessor. An SSA graph with only forward edges represents a program with no cycles. A program with control flow cycles includes backward edges (loops).

A program state $(n, v) \in (N \times V)$ is defined as a pair of the current node $n \in N$ (the program pointer) and variable values $v \in V$. Let $[\![]\!]^* : \mathcal{G} \times V \mapsto 2^{(N \times V)^*}$ be a

[1]While we do not see any principle restriction, we assume forward analyses in the following definitions, theorems, and examples.

trace semantics function defining the semantics of all $P \in \mathcal{P}$. For a program $G \in \mathcal{G}$, it maps an initial state (n^0, v^0) to traces, i.e., a set of possibly infinite sequences of states:

$$[(n^0, v^0), \ldots, (n, v), (n', v'), \ldots],$$

where each such trace represents a possible program execution according to the semantics of the language. $(n', v') \in \mathit{next}(n, v)$ represents the next possible state(s) after a current state (n, v). Notice that $[\![]\!]^*$ gives a set of possible sequence of states (traces) whereas $\mathit{next}(n, v)$ just gives a set of possible states to choose the next single current state from. Due to non-determinism in the programming language semantics, there are, in general, more than one states in $\mathit{next}(n, v)$. Hence, there are, in general, several traces for one and the same program graph G and initial state v^0.

$[\![]\!]^*$ is defined as a composition of semantics functions $[\![]\!]_k^{\mathit{data}} : V \mapsto V$ and $[\![]\!]_k^{\mathit{ctrl}} : N \mapsto N$, one such pair for each of the different node kinds k, defining the mapping of a current state to the next state(s): $(n', v') \in \mathit{next}(n, v)$ with $v' \in [\![v]\!]_k^{\mathit{data}}$, $n' \in [\![n]\!]_k^{\mathit{ctrl}}$, $\mathit{kind}(n) = k$, and $(n, n') \in E$.

The update function $[\![]\!]_\phi^{\mathit{data}}$ for ϕ nodes is defined as $[\![v]\!]_\phi^{\mathit{data}} = v$. With $[\![n]\!]_\phi^{\mathit{ctrl}}$ the control flow edge successor of n, we have $\mathit{next}(n, v) = (n', v)$ if n is a ϕ node and $(n, n') \in E$. Notice that a ϕ-node does not change the input value; its semantics is a non-strict function that just selects the predecessor value computed at the dynamic predecessor block and provides it to its dynamic successors.

2.2 Abstract Analysis Semantics

Let a Monotone Data-flow Framework [26] $(A, \sqsubseteq, F, \iota)$ define *every* sound, context-*in*sensitive program analysis I with

- A is an abstraction of V.
- $\mathit{CPO}_A = (A, \sqsubseteq)$ a complete partial order fulfilling the ascending chain property representing the analysis values. $\bot$ is the smallest such value. CPO_A induces a semi-lattice $\mathcal{L}_A = \{A, \sqcup, \bot\}$ with $a \sqsubseteq b \Leftrightarrow a \sqcup b = b$. $(n, a) \in (N, A)$ defines an abstract program state in n.
- F a set of transfer functions one for each kind k of nodes $f_k^p : A^p \mapsto A$ including a special transfer or join function for ϕ nodes $f_\phi^p : A^p \mapsto A$; $f_\phi = \sqcup(a_1, \ldots, a_p)$, with p the number of predecessors of a node.
- $\iota \in A$ the initial abstract value of n^0.

Assume I is a *valid* abstraction of $[\![]\!]^*$ for each $G \in \mathcal{G}$ and the corresponding initial state v^0 in the sense of Abstract Interpretation [9]. Then corresponding abstraction and concretization functions map traces to abstract program states and vice versa. More specifically, we assume that I abstracts each trace ending in (n, v) with $(n, \alpha(v))$ and interprets an abstract state (n, a) as traces ending in (n, v), $v \in \gamma(a)$.

For the result of a valid abstraction of $[\![]\!]^*$, it holds for any abstract state (n, a) and $\mathit{kind}(n) = k$:

$$\bigcup_{v \in \gamma(a)} : [\![v]\!]_k^{\mathit{data}} \subseteq (\gamma \circ f_k)(a)$$

i.e., the union of the output values of applying the semantics function of n to any concrete input value $v \in \gamma(a)$ (left-hand side) is not larger than (is abstracted by) applying the corresponding transfer function to the abstract value a and then concretizing the abstract output (right-hand side).

Moreover, any fair sequence of transfer function updates, beginning with initial abstract states (n^0, ι) in the start node and $(n, \bot)$, otherwise, terminates in a valid fixed point, i.e., the concretization of the fixed point abstract state in n for a program (graph) G is an over-approximation (abstraction) of the possible traces ending in n.

2.3 Term Based Context-Insensitive Analysis

Definition 1 ($\sqcup$-terms) Let $(A, \sqsubseteq, F, \iota)$ define a context-*in*sensitive program analysis I. The set of all $\sqcup$-terms U_A over the abstract values A is defined recursively:

$$\begin{aligned} a \in A &\Rightarrow a \in U_A \\ t_1, \ldots, t_p \in U_A \wedge f \in F &\Rightarrow \mathtt{f}(t_1, \ldots, t_p) \in U_A \\ t_1, \ldots, t_p \in U_A &\Rightarrow \dot{\sqcup}(t_1, \ldots, t_p) \in U_A \end{aligned}$$

where $\mathtt{f}$ is a unique function symbol representing f. For the transfer function $\sqcup$ of ϕ-nodes, we use the function symbol $\dot{\sqcup}$.

Notice, even if the set of abstract values A were finite, U_A is infinite since there is no upper limit on the depth of the terms.

In Fig. 2, we illustrate a simple code sequence with corresponding $\sqcup$-term representation for variable f. In the code, we have marked block numbers (#0 - #6) to identify the related $\sqcup$-term nodes. For example, the variable a is assigned a value in each branch (#1, #2) of the first if-statement. Therefore, the corresponding ϕ node's

```
if (...) {       (#0)
    a = 1;       (#1)
    b = 4;
}
else {
    a = 3;       (#2)
    b = 2;
}
c = a + b;       (#3)
d = 0; e = 0;
if (...){        (#4)
    d = c + 1;   (#5)
    e = c - 1;
}
f = d + e;       (#6)
...
```

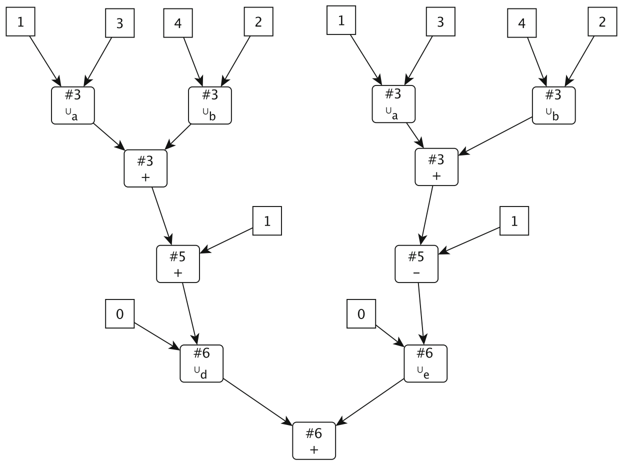

Fig. 2 A source code example with the corresponding tree representation for the $\sqcup$-term for variable f

transfer function at the beginning of Block #3 creates a term $\dot{\sqcup}(1,3)$. Similarly, the term for variable b is $\dot{\sqcup}(4,2)$. Consequently, the term for variable c in Block #3 is

$$c = +(a,b) = +(\dot{\sqcup}(1,3), \dot{\sqcup}(4,2)). \tag{1}$$

Finally, the term for variable f in Block #6 becomes:

$$\begin{aligned} f &= +(d,e) = +(\dot{\sqcup}(0,+(c,1)), \dot{\sqcup}(0,-(c,1))) \\ &= +(\dot{\sqcup}(0,+(+(\dot{\sqcup}(1,3),\dot{\sqcup}(4,2)),1)), (\dot{\sqcup}(0,-(+(\dot{\sqcup}(1,3),\dot{\sqcup}(4,2)),1)))). \end{aligned} \tag{2}$$

From Fig. 2, we can notice that all $\sqcup$-terms have a natural tree representation and that (sub-) terms/trees can be used to express/construct larger terms/trees. The abstract analysis values A form the leaves of the tree and the function symbols (including the join function symbol $\dot{\sqcup}$) the interior nodes. Notice also that the tree representation contains a lot of redundancies. For example, the identical sub-trees in the upper left and right corners both represent the value for variable c. Finally, a program containing loops will be represented by a $\sqcup$-term tree with an infinite depth.

The context-insensitive transfer functions $f, \sqcup$ of function symbols $\texttt{f}, \dot{\sqcup}$ are defined on A, not on U_A. However, we can generalize them by recursively applying them on sub-terms and evaluating them at the leaves of a $\sqcup$-term. This approach defines an evaluation of $\sqcup$-terms.

$$eval_{\sqcup} : U_A \mapsto A$$
$$eval_{\sqcup}(t) = \begin{cases} t, & \text{if } t \in A \\ f(eval_{\sqcup}(t_1), \ldots, eval_{\sqcup}(t_p)), & \text{if } t = \texttt{f}(t_1, \ldots, t_p) \\ \sqcup(eval_{\sqcup}(t_1), \ldots, eval_{\sqcup}(t_p)), & \text{if } t = \dot{\sqcup}(t_1, \ldots, t_p) \end{cases}$$

For example, assume that we use set union $\cup$ as our $\sqcup$-operator and that the transfer functions of the operators $+$ and $-$ are applied element by element on sets (e.g. $+(\{1,2\},\{3,4\}) = \{4,5,6\}$). In this case, the evaluation of the $\sqcup$-term expressions for variables c and f defined by (1) and (2) can be evaluated as:

$$\begin{aligned} eval_{\sqcup}(c) &= +(\dot{\sqcup}(1,3), \dot{\sqcup}(4,2) = +(\{1,3\},\{4,2\}) = \{3,5,7\} \\ eval_{\sqcup}(f) &= +(\dot{\sqcup}(0,+(c,1)), \dot{\sqcup}(0,-(c,1))) \\ &= +(\dot{\sqcup}(0,+(\{3,5,7\},1)), \dot{\sqcup}(0,-(\{3,5,7\},1))) \\ &= +(\dot{\sqcup}(0,\{4,6,8\}), \dot{\sqcup}(0,\{2,4,6\})) \\ &= +(\{0,4,6,8\},\{0,2,4,6\}) = \{0,2,4,6,8,10,12,14\} \end{aligned}$$

According to this context-insensitive analysis, the possible values for variables c and f are $\{3,5,7\}$ and $\{0,2,4,6,8,10,12,14\}$, respectively. The $eval_{\sqcup}$ approach outlined above will not terminate for programs containing loops. Loop handling will be addressed in Section 4.1.

Definition 2 ($\equiv_{\sqcup}$) is an equivalence relation on $\sqcup$-terms $t_1, t_2 \in U_A$:

$$t_1 \equiv_{\sqcup} t_2 \Leftrightarrow eval_{\sqcup}(t_1) = eval_{\sqcup}(t_1)$$

Definition 3 ($\sqsubseteq_\sqcup$) is a partial order relation on $\sqcup$-terms $t_1, t_2 \in U_A$:

$$t_1 \sqsubseteq_\sqcup t_2 \Leftrightarrow eval_\sqcup(t_1) \sqsubseteq eval_\sqcup(t_2)$$

Obviously, if $(A, \sqsubseteq, F, \iota)$ defines a sound analysis, then so does $(U_A, \sqsubseteq_\sqcup, F, \iota)$. It just separates the construction of transfer function terms from their evaluation. This exercise only becomes meaningful when we substitute the evaluation of the $\sqcup$ function, i.e., the transfer function of ϕ-nodes, with a less abstract evaluation, as done in the next section.

3 Using χ-Terms for Saving Context-Sensitive Information

In this section, we introduce χ-function symbols and χ-terms as a context-sensitive replacement of $\dot\sqcup$ and $\sqcup$-terms. We give an intuition on how to interpret χ-function symbols as a selection operator over possible control-flow paths. Given this interpretation, we can identify certain properties (e.g. switching behavior) which in turn motivates a rewrite rule for syntactic manipulation of χ-terms without changing the context-sensitive information they represent called the Shannon expansion. By introducing the Shannon expansion, we can finally define equivalence $\equiv_\chi$ (the basis for a formal interpretation of χ-terms), partial order $\sqsubseteq_\chi$, and $(X_A, \sqsubseteq_\chi, F, \iota)$ as a sound context-sensitive analysis based on χ-terms, which will follow next.

3.1 χ-Function Symbols

In the context-insensitive analyses introduced so far, each SSA ϕ-node of a program corresponds to one or more $\dot\sqcup$-function symbols in the $\sqcup$-terms generated by an analysis. In the context-sensitive analyses we introduce next, a ϕ-node of a program corresponds to one or more χ-function symbols (short χ-functions), which we use to introduce context-sensitivity. Each χ-function $\chi_j^b \in \mathrm{X}$ is identified by a pair (b, j) where:

1. The *block number* $b = block(\chi_j^b), b \in [1, B]$ indicates the basic block containing its corresponding ϕ-node. Here B is the number of basic blocks in a program P.
2. The *iteration index* $j = index(\chi_j^b)$ is an integer starting at index 0. The index j corresponds to the interpretation of ϕ-node in the jth run-time iteration over block b. The use of iteration indices will be discussed in detail in Section 3.4. For now, it just syntactically distinguishes χ-function symbols that correspond to the same ϕ-node in a block b.
3. The *forward* edges of an SSA graph induce a partial order $\prec$ of its ϕ-nodes. The block numbering respects this partial order, i.e.,

$$\phi_1 \prec \phi_2 \Rightarrow block(\chi_{j_1}^{b_1}) < block(\chi_{j_2}^{b_2})$$

where $\chi_{j_1}^{b_1}$ and $\chi_{j_2}^{b_2}$ are χ-functions corresponding to ϕ_1 and ϕ_2, resp.

4. The *arity* of a χ-function, denoted $arity(\chi_j^b)$, is the number p of predecessor blocks of the corresponding ϕ-node. It is the same regardless of the iteration index.

Notice, in order to separate iterations occurring at run-time when a program is executed from iterations of the analysis to establish a fixed point we will explicitly use the notion *run-time iterations* when referring to the former type of iteration.

3.2 χ-Terms

In this section, we formally define χ-terms along with some related general concepts.

Definition 4 (χ-terms) Let $(A, \sqsubseteq, F, \iota)$ define a context-*in*sensitive program analysis I. The set of all χ-terms X_A over the abstract values A is defined recursively:

$$a \in A \Rightarrow a \in X_A \tag{3}$$

$$t_1, \dots, t_p \in X_A \wedge f \in F \Rightarrow \mathtt{f}(t_1, \dots, t_p) \in X_A \tag{4}$$

$$t_1, \dots, t_p \in X_A \wedge \chi_j^b \in \mathrm{X} \Rightarrow \chi_j^b(t_1, \dots, t_p) \in X_A \tag{5}$$

Similar to the $\sqcup$-terms, each abstract value $a \in A$ is a χ-term (3) and a transfer function symbol $\mathtt{f}$ applied to χ-terms is a χ-term (4). ϕ correspond χ_j^b functions that also induce χ-terms (5).

We have previously associated all χ-functions χ_j^b with a block number $b = block(\chi_j^b)$ and arity $arity(\chi_j^b)$. These notations can be extended to χ-terms as well.

Definition 5 (Block number and arity) The *block number* of a χ-term $t \in X_A$, denoted $block(t)$, is defined as:

$$block(t) = \begin{cases} b & \text{if } t = \chi_{\_}^b(t_1, \dots, t_p). \\ 0 & \text{otherwise} \end{cases}$$

$$arity(t) = \begin{cases} p & \text{if } t = \chi_{\_}^b(t_1, \dots, t_p). \\ 0 & \text{otherwise} \end{cases}$$

From now on we will often skip the iteration index to simplify the notations. In these cases, we assume that all involved χ-functions, from a certain block b, have the same iteration index.

As an example of χ-terms, we once again take a look at the values for variables c and f from Fig. 2:

$$c = +(\chi^3(1, 3), \chi^3(4, 2)) \tag{6}$$

$$\begin{aligned} f &= +(\chi^6(0, +(c, 1)), \chi^6(0, -(c, 1))) \\ &= +(\chi^6(0, +(+(\chi^3(1, 3), \chi^3(4, 2)), 1)), \chi^6(0, -(+(\chi^3(1, 3), \chi^3(4, 2)), 1))) \end{aligned} \tag{7}$$

So far, χ-terms are elements of a term algebra. While we will give a formal interpretation later, an intuitive understanding of their semantics will help. Following the semantics of ϕ-nodes, χ-terms represent how different control-flow predecessors define the value of a variable. For example, we denote the value of variable a in block #3 in Fig. 2 using χ-terms as $a = \chi^3(1, 3)$ with the following interpretation: variable a has the value 1 if block #3 was reached from the first predecessor block (block #1) in the program execution; a has the value 3 if it was reached from second predecessor block (block #2). Similar (but more complex) are the interpretations of the χ-terms representing the variables c and f above. In short, in addition to a set of possible values, a χ-term also contains information about the control-flow path that generated each of these values. It abstracts, however, from the conditions leading to the different paths.

Definition 6 (Function set) The *function set* of a χ-term $t \in X_A$, denoted *func*(t), is the set of all χ-function symbols $\chi_j^b \in \mathrm{X}$ used to construct t.

For example, *func*$(f) = \{\chi^3, \chi^6\}$ for the χ-term defined in (7).

3.3 The Tree Representation of χ-Terms

Terms in general (as well as $\sqcup$- and χ-terms) have a tree representation. Many notions and χ-term operations are easiest to understand as tree properties. Basic notions include:

- The *leaves* of a χ-term $t \in X_A$, denoted *leaves*$(t) \subseteq A$, correspond to the set of all leaves in the tree representation of t.
- The *subterms* of a χ-term $t \in X_A$, denoted *subterms*$(t) \subseteq X_A$, correspond to the set of all χ-functions-rooted subtrees of t in the tree representation of t. Transfer-function-symbol-rooted subtrees and the leaves are *not* included. Also, t itself is *not* included if t is a χ-function-rooted term whereas all χ-function-rooted subterms are included if t is a transfer-function-symbol-rooted term.
- The *children* of χ-term $t \in X_A$, denoted *children*$(t) \subseteq X_A$, is defined as:

$$children(t) = \begin{cases} \{t_1, \ldots, t_p\} & \text{if } t = \chi_j^b(t_1, \ldots, t_p) \\ \emptyset & \text{otherwise} \end{cases}$$

- The *depth* of a χ-term $t \in X_A$, denoted *depth*(t), is the max number of χ functions on any path from the leaves to the root of the tree representation of t.
- Let $n \in X_A$ be a node in the tree representation of a χ-term $t \in X_A$. The *depth* of subterm s in t, denoted $depth(t, s) = depth(t) - depth(s)$, is the depth of the node representing s in the tree representation of t.

We exemplify the basic tree notations on the χ-term of variable f introduced in (7):

$$f = +(\chi^6(0, +(+(\chi^3(1, 3), \chi^3(4, 2)), 1)), \chi^6(0, -(+(\chi^3(1, 3), \chi^3(4, 2)), 1)))$$

$$\begin{aligned} leaves(f) &= \{0, 1, 2, 3, 4\} \\ subterms(f) &= \{\chi^6(0, +(+(\chi^3(1, 3), \chi^3(4, 2)), 1)), \\ &\quad \chi^6(0, -(+(\chi^3(1, 3), \chi^3(4, 2)), 1)), \\ &\quad \chi^3(1, 3), \chi^3(4, 2)\} \end{aligned}$$

$$\begin{aligned}
children(f) &= \{\chi^6(0, +(+(\chi^3(1,3), \chi^3(4,2)), 1)),\\
&\quad\ \chi^6(0, -(+(\chi^3(1,3), \chi^3(4,2)), 1))\}\\
depth(f) &= 2\\
depth(f, +(\chi^3(1,3), \chi^3(4,2))) &= 1
\end{aligned}$$

3.4 Basic χ-Term Operations

In this section, we present basic operations on χ-terms as a prerequisite for defining an equivalence relation and an evaluation function of χ-terms.

We observe that two ϕ nodes n, n' in the same block b and interpreted in the same run-time iteration j have the *same switching behavior*. We do not know statically which dynamic values are selected and which of the static predecessors of n, n' computes these values. However, we do know that it is the *same* predecessor for both ϕ nodes n and n'.

Recall that χ-function symbols represented the transfer function of ϕ nodes. Given two ϕ-nodes $\phi(x_1, \ldots, x_p)$ and $\phi'(y_1, \ldots, y_p)$ from the same block b (for selecting the values of variables x and y, respectively, valid in b). For any execution of the program it holds that in the same run-time iteration j over b

$$\forall k \in [1, p] : [\![\phi(x_1, \ldots, x_p)]\!] = x_k \Leftrightarrow [\![\phi'(y_1, \ldots, y_p)]\!] = y_k$$

Thus, the selection behavior of a ϕ-node is determined by a pair (b, j) where b is the block number and j is the iteration index. We abstract from this property of the ϕ-node interpretation in the insensitive transfer function $\sqcup$ and its interpretation in $\sqcup$-terms. In contrast, we will keep this property of ϕ-nodes in the context-sensitive transfer function χ_j^b and its interpretation in χ-terms.

This property is the basis for defining the most important operation, the *Shannon expansion*, used to manipulate χ-terms without affecting their value. To define this operation, we first introduce *restriction* of χ-terms as an auxiliary operation.

Definition 7 (Restriction) The *restriction* of a χ-term $t \in X_A$ to the k:th branch of χ_j^b, denoted $t|_{(b,j):k}$, is a new χ-term where every sub-term $\chi_j^b(t_1, \ldots, t_p)$ in t has been replaced by its k:th child t_k.

For example from (7):

$$f = +(\chi^6(0, +(+(\chi^3(1,3), \chi^3(4,2)), 1)), \chi^6(0, -(+(\chi^3(1,3), \chi^3(4,2)), 1)))$$

and we might have the following restrictions:

$$\begin{aligned}
f|_{(6,\_):1} &= +(0, 0)\\
f|_{(3,\_):2} &= +(\chi^6(0, +(+(3,2), 1)), \chi^6(0, -(+(3,2), 1)))
\end{aligned}$$

The χ-term restriction defines a new χ-term $t|_{(b,j):k}$ by only considering the value from the k:th predecessor of block b (in run-time iteration j) in the construction of a new term t'. It should also be noticed that if we restrict to the k:th branch of χ_j^b and $\chi_j^b \notin func(t)$ then $t = t|_{(b,j):k}$, i.e., the term t is left unaffected.

Definition 8 (Shannon expansion) The *Shannon expansion* of a χ-term $t \in X_A$ over χ_j^b is a new χ-term t' defined as:

$$t' = \chi_j^b(t|_{(b,j):1}, t|_{(b,j):2}, \dots, t|_{(b,j):n}) \quad \text{where } p = arity(\chi_j^b)$$

Notice that Shannon expansion over any sub-term with root function symbol χ_j^b is just a new term with the root $\chi_j^b(...)$ of all possible restrictions of χ_j^b. It allows to revert the order of χ-functions in a χ-term. Notice also that the Shannon expansion creates a new χ-term that encodes the same context-sensitive information as the original χ-term. It is just a rewrite rule that can be used to manipulate χ-terms.

Below we illustrate the Shannon expansion over χ^3 using the χ-term for variable c defined in (6).

$$\begin{aligned} c &= +(\chi^3(1,3), \chi^3(4,2)) \\ &\equiv \chi^3(+(1,4), +(3,2)) \end{aligned}$$

and over χ^6 and χ^3 using as before the χ-term for variable f defined in (7):

$$\begin{aligned} f &= +(\chi^6(0, +(+(\chi^3(1,3), \chi^3(4,2)), 1)), \chi^6(0, -(+(\chi^3(1,3), \chi^3(4,2)), 1))) \\ &\equiv \chi^6(+(0,0), +(+(+(\chi^3(1,3), \chi^3(4,2)), 1), -(+(\chi^3(1,3), \chi^3(4,2)), 1))) \quad (8) \\ &\equiv \chi^3(\chi^6(+(0,0), +(+(+(1,4), 1), -(+(1,4), 1))), \\ &\quad\ \chi^6(+(0,0), +(+(+(3,2), 1), -(+(3,2), 1)))) \quad (9) \end{aligned}$$

The first equivalence is reached by expansion over χ^6, cf. line (8), the second by expansion over χ^3, cf. line (9). Recall that the example did not contain any loop; the loop index 0 is the same and could therefore be removed.

Figure 3 shows the tree representation of variable c before and after the Shannon expansion over χ^3. The effect of the Shannon expansion is to create a new term/tree with χ^3 as root. This also has the effect of pushing the node corresponding to the transfer functions of operation $+$ closer to the leaves.

In general, repeated Shannon expansions over all available χ-functions (as we did for the variable f above) will result in a term/tree with all χ-terms being positioned close to the root, e.g., $\chi^3(\chi^6(\dots), \chi^6(\dots))$, and all nodes corresponding to regular transfer functions being pushed to the leaves.

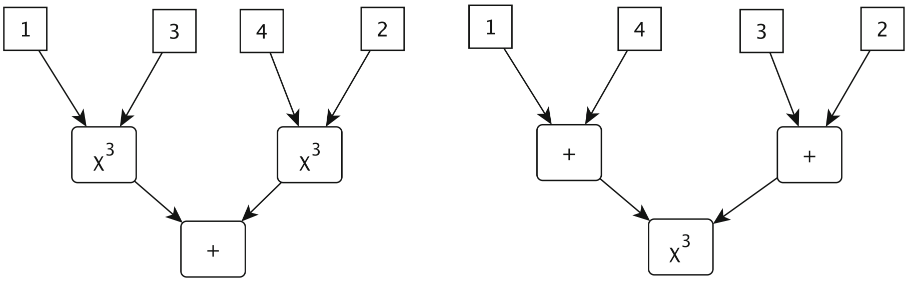

Fig. 3 Two tree representations of variable c: Before (left) and after (right) Shannon expansion

In the following we define the equivalence and redundancy properties of χ-terms, which allow to evaluate and to simplify them.

Definition 9 (equivalence, redundancy) Two χ-terms are *equivalent* ($\equiv$) iff

1. They are the equivalent context-insensitive values

$$v \equiv w \Leftrightarrow v \equiv_{\sqcup} w, \text{ where } v, w \in U_A$$

2. They have (syntactically) the same root and equivalent subterms

$$\chi_j^b(t_1, \dots, t_p) \equiv \chi_j^b(t_1', \dots, t_p') \Leftrightarrow t_1 \equiv t_1', \dots, t_p \equiv t_p'$$
$$\mathtt{f}(t_1, \dots, t_p) \equiv \mathtt{f}(t_1', \dots, t_p') \Leftrightarrow t_1 \equiv t_1', \dots, t_p \equiv t_p'$$

3. They are a Shannon expansion of each other (see Definition 8)
4. The root is a *redundant* χ operation that can be eliminated (*redundancy elimination*) if all its sub-terms are equivalent.

$$\chi_i^b(t_1, \dots, t_p) \equiv t \Leftrightarrow t_1 \equiv t_2 \equiv \dots \equiv t_p \equiv t$$

5. If the block represented by b is a loop head and the loop index is 0, the value of the corresponding χ-term is equal to the loop-entry value.

$$\chi_0^b(t_1, t_2) \equiv t_1 \text{ iff } b \text{ block number of a loop head}$$

The basic idea formalized in the definition above is that two χ-terms are equivalent if we can *guarantee* that they encode the same context-sensitive information. Some additional remarks related to the five parts of this definition:

1. Recall from Definition 4 that χ-terms can also be values of the context-insensitive analysis lattice. Hence, $v \equiv w$ above refers to the equivalence $\equiv_{\sqcup}$ of context-insensitive analysis values.
2. χ-terms can contain function symbols with χ-subterms and $\mathtt{f}(\chi_i, \dots, \chi_k) \equiv \mathtt{f}(\chi_1', \dots, \chi_k')$ requires the syntactic equivalence of the function symbols and the equivalence of the operands.
3. Shannon expansions generate a new χ-term representing exactly the same context-sensitive information as argued before.
4. Redundancy elimination is, just as the Shannon expansion, a rewrite rule leading to syntactically new χ-terms that encode the same context-sensitive information. They are, thus, a necessary part of the equivalence definition. Redundancy implies that a χ-term t containing a redundant sub-term χ^r can be reduced without any loss of information. The process of removing redundant χ-sub-terms is called *redundancy elimination* and uses the following pattern

$$\dots \chi^i(\dots, \chi^r(t', \dots, t'), \dots) \dots \equiv \dots \chi^i(\dots, t', \dots) \dots$$

   In the tree view of a χ-term, this corresponds to replacing a sub-tree rooted χ^r by any of its (all equivalent) sub-terms.
5. This property can be trivially derived from the semantics of ϕ-nodes in loop heads. It defines the initial value of the fixed-point analysis of loops.

3.5 χ-Term Evaluation

In Section 2.3, we used $eval_{\sqcup} : U_A \mapsto A$ to evaluate a context-insensitive $\sqcup$-term. That is, to associate each $\sqcup$-term with a concrete analysis result $a \in A$. We are now ready for a similar evaluation $eval_\chi$ for context-insensitive χ-terms: $eval_\chi : X_A \mapsto X_A$ is performed in three steps:

1. Repeatedly apply Shannon expansion over all χ-functions in $func(t)$. That is, push the non χ-term nodes to the leaves and separate $\sqcup$-sub-terms (with only $\mathtt{f}$ operation symbols and leaves A) and the χ_j^b function symbols using Shannon expansion.
2. Evaluate the $\sqcup$-sub-terms with $eval_{\sqcup}$.
3. Apply redundancy elimination until no further simplification is possible.

We call the evaluation process, $eval_\chi$ an "equivalence conversion", short EC. It simplifies the χ-terms representation of context-sensitive information without losing any control-flow information. Algorithm 1 is a recursive implementation of EC. Additionally, it enforces the same order of χ-functions from the leaves to the root in all χ-terms by picking the highest (b, j)-ranked χ-function (with largest j and largest b as secondary ranking criterion) in each Shannon expansion step. This leads to normalized χ-terms that are free from $\mathtt{f}$ operation and $\dot{\sqcup}$ symbols.

Algorithm 1 *normalize*$(t) \mapsto t'$.

```
if $t \in A$ then
    return $t$
else if $t = \chi_j^b(t_1, \ldots, t_p)$ then
    return $\chi_j^b(normalize(t_1), \ldots, normalize(t_p))$
else if $t = \mathtt{r}(t_1, \ldots, t_p)$, $\mathtt{r} \in \{\dot{\sqcup}, \mathtt{f}\}$ then
    $func\_set \leftarrow func(t_1) \cup \ldots \cup func(t_p)$
    if $func\_set = \emptyset$ then
        //apply the corresponding context-insensitive function $\sqcup$ or $f$, resp.
        return $eval_{\sqcup}(t)$
    else
        //enforce the same order of $\chi$-functions from the leaves to the root in all
        $\chi$-terms
        $\chi_j^b \leftarrow$ pick the one with the highest $(b, j)$-rank from $func\_set$
        return $normalize(\chi_j^b(t|_{(b,j):1}, \ldots, t|_{(b,j):arity(\chi_j^b)})$
    end if
end if
```

The result of $eval_\chi$ is in general a χ-term. To get a context-insensitive value that can be compared to $eval_{\sqcup}$ we introduce a second transformation $collapse_\chi : X_A \mapsto A$. It maps a χ-term not containing any function symbol to an element of the underlying context-insensitive analysis A CPO.

1. Substitute all χ_j^b function symbols with $\dot{\sqcup}$.

2. Apply $eval_{\sqcup}$.

We refer to this widening the “termination conversion”, short TC. It removes any remaining control-flow information by applying the context-insensitive join operator $\dot{\sqcup}$ in place of all remaining χ-functions. More precisely, we substituted all χ_j^b-functions with $\dot{\sqcup}$ whose evaluation replaces the control-flow dependencies of the result encoded by the χ-functions with a conservative approximation that merges the results of all possible control-flow options.

As an example of applying $eval_\chi$, we evaluate the χ-term for c defined in Equation (6):

$$\begin{aligned} c &= +(\chi^3(1,3), \chi^3(4,2)) \\ eval_\chi(c) &\equiv \chi^3(+(1,4), +(3,2)) && \text{(EC 1, expansion over } \chi^3\text{)} \\ &\equiv \chi^3(5,5) && \text{(EC 2, apply operators)} \\ &\equiv 5 && \text{(EC 3, redundancy elimination)} \end{aligned}$$

As an example of applying $eval_\chi$ and $collapse_\chi$, we evaluate the χ-terms for f defined in (7). The first EC 1 steps, the Shannon expansion over the χ-terms, have been shown already in (8) and (9):

$$\begin{aligned} f &= +(\chi^6(0, +(+(\chi^3(1,3), \chi^3(4,2)), 1)), \chi^6(0, -(+(\chi^3(1,3), \chi^3(4,2)), 1))) \\ eval_\chi(f) &\equiv \chi^6(+(0,0), +(+(+(\chi^3(1,3), \chi^3(4,2)), 1), -(+(\chi^3(1,3), \chi^3(4,2)), 1))) \\ &\equiv \chi^3(\chi^6(+(0,0), +(+(+(1,4),1), -(+(1,4),1))), \\ &\quad\ \chi^6(+(0,0), +(+(+(3,2),1), -(+(3,2),1)))) && \text{(EC 1)} \\ &\equiv \chi^3(\chi^6(0, +(+(5,1), -(5,1))), \chi^6(0, +(+(5,1), -(5,1)))) && \text{(EC 2)} \\ &\equiv \chi^3(\chi^6(0, +(6,4)), \chi^6(0, +(6,4))) && \text{(EC 2)} \\ &\equiv \chi^3(\chi^6(0,10), \chi^6(0,10)) && \text{(EC 2)} \\ &\equiv \chi^6(0,10) && \text{(EC 3)} \end{aligned}$$

$$\begin{aligned} collapse_\chi(eval_\chi(f)) &\equiv \dot{\sqcup}(0,10) && \text{(TC 1)} \\ &\equiv \{0,10\} && \text{(TC 2)} \end{aligned}$$

As these two examples show, the effect of applying $eval_\chi$ (push operators to the leaves, evaluate $\sqcup$-terms, apply redundancy elimination) often simplifies a χ-term expression considerably by removing all superfluous redundancies. The final result $c = 5$ and the intermediate result $f = \chi^6(0, 10)$ represent the full context-sensitive information where all effective control-flow decisions are kept. The interpretation of these results is the following: variable c in block #3 (Fig. 2) will always take the value 5 regardless of the control-flow, and variable f in block #6 will take the values 0 or 10 depending on whether we enter the final if statement or not. The example also shows, that we lose information when we apply $collapse_\chi$: $f = \{0, 10\}$ does not refer to the control-flow context any longer. Still, it is more precise than the context-insensitive result $\{0, 2, 4, 6, 8, 10, 12, 14\}$ of the evaluation of the corresponding $\sqcup$-term discussed earlier.

Notice that for each finite χ-term, $collapse_\chi \circ eval_\chi$ terminates with a value from A, the abstract analysis domain from the context-insensitive analysis. This is obvious as $eval_\chi$ removes all `f` operations and $collapse_\chi$ removes all χ-function symbols and applies $eval_\sqcup$ to compute an A value.

Analogously to the partial order relation on $\sqcup$-terms (Definition 3), we define such a relation on χ-terms:

Definition 10 ($\sqsubseteq_\chi$) is a partial order relation on χ-terms. Let $t_1, t_2 \in X_A$

$$t_1 \sqsubseteq_\chi t_2 \Leftrightarrow \begin{cases} t_1, t_2 \in U_A, eval_\sqcup(t_1) \sqsubseteq eval_\sqcup(t_2) & \text{(i)} \\ t_1 \equiv_\chi t_1', t_2 \equiv_\chi t_2', t_1' \sqsubseteq_\chi t_2' & \text{(ii)} \\ s_i \sqsubseteq_\chi s_i', i \in [1,p], t_1 = \chi_j^b(s_1,\ldots,s_p), t_2 = \dot{\sqcup}(s_1',\ldots,s_p') & \text{(iii)} \\ s_i \sqsubseteq_\chi s_i', i \in [1,p], t_1 = \chi_j^b(s_1,\ldots,s_p), t_2 = \chi_j^b(s_1',\ldots,s_p') & \text{(iv)} \\ t_1 \sqsubseteq_\chi t_1', t_1' \in subterms(eval_\chi(t_2)) & \text{(v)} \end{cases}$$

That is, $t_1 \sqsubseteq_\chi t_2$ if the following cases recursively apply: (base case i) $t_1, t_2 \in U_A$ and evaluate to $a_1, a_2 \in A$, resp., with $a_1 \sqsubseteq a_2$; (ii) equivalence conversions do not have an impact; (iii) Widening (i.e. replacing χ_j^b with $\dot{\sqcup}$) gives a larger χ-term, e.g., $\chi_j^b(t_1,\ldots,t_p) \sqsubseteq_\chi \dot{\sqcup}(t_1,\ldots,t_p)$; (iv) both terms t_1, t_2 are rooted with the same χ-function symbol and the respective children s_i, s_i' are pairwise ordered $s_i \sqsubseteq s_i'$; (v) t_1 is smaller than a sub-term of t_2.

An important part of $collapse_\chi$ was to substitute χ-function symbols with $\dot{\sqcup}$. We refer to this type of approximation as the $\sqcup$-approximation. In Section 4, we use $\sqcup$-approximations to introduce two alternative approaches guaranteeing the termination of χ-term based context-sensitive analyses.

Definition 11 ($\sqcup$-approximation) Let $t \in X_A$ be a χ-term and let s be a sub-term of t rooted by a χ-symbol χ_j^b. The $\sqcup$-approximation of t with respect to s, denoted t_s^*, is a new term where the root χ-symbol χ_j^b in s is replaced by $\dot{\sqcup}$.

The definition is easy to understand using the tree representation of t. It simply means that we replace the root χ-symbol χ_j^b in a sub-term s by $\dot{\sqcup}$. Intuitively, any $\sqcup$-approximation is conservative, since it replaces the selective χ-term with an approximation that merges all possible control-flow options. With Definition 10, the partial order relations on χ-terms, we can formulated this intuition in the following

Theorem 1 *Let $t = \chi(\ldots,s,\ldots) \in X_A$ be a χ-term and let $s = \chi'(\ldots) \in X_A$ be a sub-term of t then $t \sqsubseteq_\chi t_s^*$.*

Proof This follows directly from cases (iii) and (iv) in Definition 10. We define q as a new χ-term, the result of applying $\dot{\sqcup}$ on sub-term s, then

$$\begin{aligned} \dot{\sqcup}(s) &\equiv q & \\ s &\sqsubseteq_\chi q & \text{(iii)} \\ t = \chi(\ldots,s,\ldots) &\sqsubseteq_\chi t_s^* = \chi(\ldots,q,\ldots) & \text{(iv)} \end{aligned}$$

□

Theorem 1 is important, since it tells us how to make conservative approximations of χ-terms. That is, we can in any phase of the analysis replace a χ-term $\chi(t_1, \ldots, t_p)$ with $\dot{\sqcup}(t_1, \ldots, t_p)$ and still maintain a conservative approach.

3.6 Analysis Soundness

In this section, we prove the soundness of our context-sensitive analysis assuming it terminates. We end the section discussing issues related to termination (reaching a fixed point) and analysis accuracy.

The following two lemmata state that the induced context-sensitive transfer functions on X_A are monotone, provided that the initial context-insensitive transfer functions on A were monotone.

Lemma 2 *Let $(A, \sqsubseteq, F, \iota)$ define a sound context-insensitive program analysis and $(X_A, \sqsubseteq_\chi, F, \iota)$ define the corresponding context-sensitive program analysis using χ-terms. Then*

$$\forall t_i \in X_A, i \in [1, p] : t_i \sqsubseteq_\chi \chi_j^b(t_1, \ldots, t_p).$$

Proof As $t_i \in \mathit{subterms}(\chi_j^b(t_1, \ldots, t_p))$, the claim follows directly from case (v) of Definition 10. □

Lemma 3 *Let $(A, \sqsubseteq, F, \iota)$ define a sound context-insensitive program analysis and $(X_A, \sqsubseteq_\chi, F, \iota)$ define the corresponding context-sensitive program analysis using χ-terms. Let $\mathtt{f}$ denote the function symbol of the context-insensitive transfer function f. Then $\forall f \in F, t_i \in X_A, t_i' \in X_A, i \in [1, p]$*

$$t_i \sqsubseteq_\chi t_i', i \in [1, p] \Rightarrow \mathtt{f}(t_1, \ldots, t_p) \sqsubseteq_\chi \mathtt{f}(t_1', \ldots, t_p').$$

Proof We prove the claim by structural induction on the depth of the χ-terms.
Induction basis: The claim holds if $\mathit{depth}(t_i) = \mathit{depth}(t_i') = 0$. This implies $t_i, t_i' \in A$, $t_i \sqsubseteq t_i'$, and

$$\begin{aligned} \mathtt{f}(t_1, \ldots, t_p) &\equiv_\chi f(t_1, \ldots, t_p) = a \\ \mathtt{f}(t_1', \ldots, t_p')) &\equiv_\chi f(t_1', \ldots, t_p') = a' \\ a &\sqsubseteq a', \end{aligned}$$

since the context-insensitive transfer functions $f : A^p \mapsto A$ are monotone. Hence, $\mathtt{f}(t_1, \ldots, t_p) \sqsubseteq_\chi \mathtt{f}(t_1', \ldots, t_p')$ for the induction basis according to (i) of Definition 10.
Induction step: Provided it holds

$$s_i \sqsubseteq_\chi s_i', i \in [1, p] \Rightarrow \mathtt{f}(s_1, \ldots, s_p) \sqsubseteq_\chi \mathtt{f}(s_1', \ldots, s_p').$$

if $\mathit{depth}(s_l) \leq k$, $\mathit{depth}(s_l') \leq k$. Then the claim holds for arguments of $\mathit{depth}(t_i) \leq k+1$, $\mathit{depth}(t_i') \leq k+1$. W.l.o.g. let $t_i = \chi_j^b(s_{i_1}, \ldots, s_{i_{p'}})$ and $t_i' = \chi_j^b(s_{i_1}', \ldots, s_{i_{p'}}')$.

Since, $t_i \sqsubseteq_\chi t'_i$, it follows from (iv) of Definition 10 and from the induction step's precondition $\forall i \in [1, p], l \in [1, p']$ that

$$\begin{aligned} s_{i_l} &\sqsubseteq_\chi s_{i_l} \\ \mathtt{f}(s_{1_l}, \ldots, s_{p_l}) &\sqsubseteq_\chi \mathtt{f}(s'_{1_l}, \ldots, s'_{p_l}). \end{aligned}$$

Further

$$\begin{aligned} \mathtt{f}(t_1, ..., t_p) &\equiv_\chi \chi^b_j(\mathtt{f}(s_{1_1}, ..., s_{p_1}), ..., \mathtt{f}(s_{1_{p'}}, ..., s_{p_{p'}})) \\ \mathtt{f}(t'_1, ..., t'_p) &\equiv_\chi \chi^b_j(\mathtt{f}(s'_{1_1}, ..., s'_{p_1}), ..., \mathtt{f}(s'_{1_{p'}}, ..., s'_{p_{p'}})) \\ \chi^b_j(\mathtt{f}(s_{1_1}, ..., s_{p_1}), ..., \mathtt{f}(s_{1_{p'}}, ..., s_{p_{p'}})) &\sqsubseteq_\chi \chi^b_j(\mathtt{f}(s'_{1_1}, ..., s'_{p_1}), ..., \mathtt{f}(s'_{1_{p'}}, ..., s'_{p_{p'}})) \\ \mathtt{f}(t_1, ..., t_p) &\sqsubseteq_\chi \mathtt{f}(t'_1, ..., t'_p) \end{aligned}$$

because of cases (ii) and (iv) of of Definition 10, which concludes the induction step and proves the claim. □

Hence, according to Lemmas 2 and 3, if $(A, \sqsubseteq, F, \iota)$ defines a sound analysis, monotone transfer functions are guaranteed for the context-sensitive analysis $(X_A, \sqsubseteq_\chi, F, \iota)$. Still it does only little more than separating the construction of transfer function terms from their evaluation. Compared to the previously discussed insensitive analysis $(U_A, \sqsubseteq_\sqcup, F, \iota)$, it only performs some equivalence transformations between the construction of transfer function terms and their evaluation. However, we can state the following theorem:

Theorem 4 *Let $(A, \sqsubseteq, F, \iota)$ define a sound context-insensitive program analysis and $(X_A, \sqsubseteq_\chi, F, \iota)$ define the corresponding context-sensitive program analysis using χ-terms. If the corresponding context-sensitive analysis $(X_A, \sqsubseteq_\chi, F, \iota)$ terminates, then it is sound.*

We acknowledge some issues that need to be discussed. First, the context-sensitive analysis is not guaranteed to terminate, in general, as the ascending chain property does not hold in the CPO $(X_A, \sqsubseteq_\chi)$. Although an analysis χ-term fixed-point still *exists* as the transfer functions are monotone, it is *not iteratively computable* any longer. Instead, χ-terms may grow infinitely when applying standard iterative data-flow analysis. This is a problem that will be addressed in Section 4.

Second, while $(A, \sqsubseteq, F, \iota)$ and $(U_A, \sqsubseteq_\sqcup, F, \iota)$ produce the same result, the result of $(X_A, \sqsubseteq_\chi, F, \iota)$ may be more accurate (smaller, still conservative) than the original analysis results. This is because we distribute the transfer functions over the meet operation and can apply them to more concrete (smaller) input values in different control flow predecessors, which, in turn, potentially allows for more concrete results.

As an example of this change in analysis precision, we can take a look at $eval_\sqcup$ and $eval_\chi$ previously computed for the variables c and f:

$$\begin{aligned} eval_\sqcup(c) &= \{3, 5, 7\}, eval_\chi(c) = 5 \\ eval_\sqcup(f) &= \{0, 2, 4, 6, 8, 10, 12, 14\}, collapse_\chi(eval_\chi(f)) = \{0, 10\} \end{aligned}$$

By pushing the transfer functions of operators $+$ and $-$ to the leaves, we can deduce that c in block #3 always takes the value 5, and that f is either 0 or 10 exploiting the improved precision of $eval_\chi(c)$.

This does not only work in the example. In general, $collapse_\chi(eval_\chi())$ gives more precise (smaller) analysis results than $eval_\sqcup$. We notice that the same accuracy could be achieved by a semantic-preserving source code transformation that duplicates code after a control flow conjunction into the different branches before this conjunction (code in-lining). This transformation would obviously not terminate for loop conjunctions, neither does our χ-term construction in these cases. This would lead to an exponential growth for sequential code with conditional statements; just as our χ-term construction lets them grow exponentially.

To exemplify this, cf. to the code in Fig. 2 again replicated in Fig. 4 (left) and assume that the blocks #3 − #6 were copied into both branches #1 and #2 of the first if statement, and in both copies, block #6 were moved into block #5 and into a new else block of the second if statement. The result of such a semantics preserving transformation is given in Fig. 4 (right). It is easy to see that now even a context-insensitive analysis using $eval_\sqcup$ would achieve the same accuracy as a context-sensitive analysis based on $eval_\chi$ on the original program: $c = 5$ for both copies in blocks #3a and #3b, $f = 10$ for the copies in blocks #6aa and #6ba, and $f = 0$ for the copies in blocks #6ab and #6bb.

The tree-based χ-term representation is (arguably) already better than representations based on the cloned code. However, we will introduce a more compact χ-term representation that does not contain any redundancies in Section 5.

Finally, in order to simplify our notations, we will from now on present the results in their normal form in our examples. That is, rather than presenting χ-terms involving complex combinations of context-insensitive values, $\sqcup$-operators, and χ-functions, we will show them after having applied *normalize*, cf. Algorithm 1, i.e., push operator transfer functions to the leaves, evaluate $\sqcup$-terms, apply redundancy elimination, and at the same time give an ordered structure based on the χ-term ranking. For example, instead of the complete χ-terms for c and f in given (6) and (7), respectively, we will show them as $c = 5$ and $f = \chi^6(0, 10)$.

4 Termination and Approximations

As observed in Section 3.4, an analysis based on fixed-point iteration starting with $\perp$ does not terminate since the ascending chain property does not hold in the CPO $(X_A, \sqsubseteq_\chi)$. The χ-terms represent different values that are context-dependent on the different control-flows options and there are countable many such options. More specifically, χ-terms will grow infinitely when applying a standard iterative data-flow analysis on a program involving loops or any other cycles in the control-flow graph. In this section, we will handle this problem by introducing one type of widening applying $\sqcup$-approximation to control-flow cycles (referred to as the l-loop-approximation). It will guarantee finitely sized χ-terms and analysis termination. We will also introduce another type of widening (referred to as the k-approximation)

```
if (...) {      (#0)
    a = 1;      (#1)
    b = 4;
}
else {
    a = 3;      (#2)
    b = 2;
}
c = a + b;      (#3)
d = 0; e = 0;
if (...){       (#4)
    d = c + 1;(#5)
    e = c - 1;
}
f = d + e;      (#6)
```

```
if (...) {              (#0)
    a = 1;              (#1)
    b = 4;
    c = a + b;          (#3a)
    d = 0; e = 0;
    if (...){           (#4a)
        d = c + 1;(#5a)
        e = c - 1;
        f = d + e;(#6aa)
    }
    else {
        f = d + e;(#6ab)
    }
}
else {
    a = 3;              (#2)
    b = 2;
    c = a + b;          (#3b)
    d = 0; e = 0;
    if (...){           (#4b)
        d = c + 1;(#5b)
        e = c - 1;
        f = d + e;(#6ba)
    }
    else {
        f = d + e;(#6bb)
    }
}
```

Fig. 4 Source code example from Fig. 2 before and after a semantics preserving code cloning transformation

limiting the depths of χ-terms and allowing the analysis precision to be adjusted by an integer parameter k.

4.1 Loop Handling

Control-flow cycles, e.g. loops, generate χ-terms with an infinite depth and prevent analysis termination. For guaranteeing termination, we define a loop-approximated CPO in which the the ascending chain property holds. Therefore, we introduce a widening step, the $\sqcup$-approximation, in the transfer functions of ϕ nodes generated for the loop heads.

We illustrated the idea with an introductory example. Assume a $(X_A, \sqsubseteq_\chi, F, \iota)$ based value analysis that, in turn, generalizes an context-insensitive value analysis $(A, \sqsubseteq, F, \iota)$ with $(A, \sqsubseteq)$ defining a integer constant (flat) lattice and F straightforward transfer functions for operators $+$ and $-$: if arguments are constants they calculate the constant result, if one argument is $\bot(\top)$ the result is $\bot(\top)$.

For the code fragment and the corresponding SSA sub-graph in Fig. 5, the analysis generates χ-terms for the loop-escaping variable x assigned to y (cf. Fig. 5) as follows:

$$\begin{aligned}
x_0^w &= \chi_0^w(1, \bot) \equiv 1 \\
x_0^i &= \chi_0^i(+(x_0^w, 1), -(x_0^w, 1)) \\
&\equiv \chi_0^i(+(1, 1), -(1, 1)) \\
&\equiv \chi_0^i(2, 0) \\
x_1^w &= \chi_1^w(1, x_0^i) \\
&\equiv \chi_1^w(1, \chi_0^i(2, 0)) \\
x_1^i &= \chi_1^i(+(x_1^w, 1), -(x_1^w, 1)) \\
&\equiv \chi_1^i(+(\chi_1^w(1, \chi_0^i(2, 0)), 1), -(\chi_1^w(1, \chi_0^i(2, 0)), 1)) \\
&\equiv \chi_1^i(\chi_1^w(+(1, 1), +(\chi_0^i(2, 0), 1)), \chi_1^w(-(1, 1), -(\chi_0^i(2, 0), 1))) \\
&\equiv \chi_1^i(\chi_1^w(+(1, 1), \chi_0^i(+(2, 1), +(0, 1))), \chi_1^w(-(1, 1), \chi_0^i(-(2, 1), -(0, 1)))) \\
&= \chi_1^i(\chi_1^w(2, \chi_0^i(3, 1)), \chi_1^w(0, \chi_0^i(1, -1))) \\
x_2^w &= \chi_2^w(1, x_1^i) \\
&\equiv \chi_2^w(1, \chi_1^i(\chi_1^w(2, \chi_0^i(3, 1)), \chi_1^w(0, \chi_0^i(1, -1)))) \\
x_2^i &= \chi_2^i(+(x_2^w, 1), -(x_2^w, 1)) \\
&\equiv \chi_2^i(+(\chi_2^w(1, \chi_1^i(\chi_1^w(2, \chi_0^i(3, 1)), \chi_1^w(0, \chi_0^i(1, -1)))), 1), \\
&\quad -(\chi_2^w(1, \chi_1^i(\chi_1^w(2, \chi_0^i(3, 1)), \chi_1^w(0, \chi_0^i(1, -1)))), 1)) \\
&\equiv \chi_2^i(\chi_2^w(2, \chi_1^i(\chi_1^w(3, \chi_0^i(4, 2)), \chi_1^w(1, \chi_0^i(2, 0)))), \\
&\quad \chi_2^w(0, \chi_1^i(\chi_1^w(1, \chi_0^i(2, 0)), \chi_1^w(-1, \chi_0^i(0, -2))))) \\
x_3^w &= \chi_3^w(1, x_2^i) \\
&\equiv \chi_3^w(1, \chi_2^i(\chi_2^w(2, \chi_1^i(\chi_1^w(3, \chi_0^i(4, 2)), \chi_1^w(1, \chi_0^i(2, 0)))), \\
&\quad \chi_2^w(0, \chi_1^i(\chi_1^w(1, \chi_0^i(2, 0)), \chi_1^w(-1, \chi_0^i(0, -2)))))) \\
&\quad \ldots
\end{aligned}$$

In the example, χ_j^w (χ_j^i) represents the (selection) semantics of the ϕ node corresponding to the while (if) block after $0 \ldots j$ completed run-time iterations; x_j^w is the term representing the analysis value of x after $0 \ldots j$ completed run-time iterations conservatively assuming that such a program execution is realizable in a concrete run. Especially, x_0^w represents x for the control flow option that does not iterate over the loop body, and, trivially, it is $x = 1$, as defined in point 5 of

```
x=1
while (...)
    if (...)
        x++
    else
        x--
y=x
```

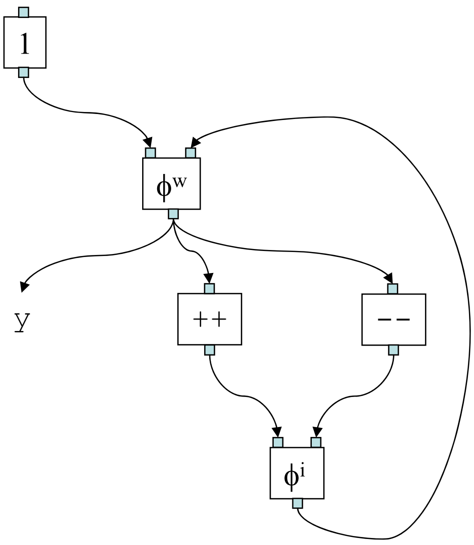

Fig. 5 Loop approximation example, code fragment (left) and SSA sub-graph (right). The basic blocks and the control flow between them was omitted here

Definition 9. x_1^w represents x for the control flow option that does zero or one run-time iteration over the loop body. After analyzing the loop body (and the included conditional code) and a few transformations x_1^w, can be interpreted as follows: $x = 1$ for no iteration or $\chi_0^i(2, 0)$ for one run-time iteration, which, in turn is 2 or 0 depending on the executed branch of the if statement in the body. x_2^w represents x for the control flow option that iterates zero, once, or twice, and the χ-term gets large already. The final result after $0 \ldots 2$ run-time iterations and normalization $\chi_2^w(1, \chi_1^i(\chi_1^w(2, \chi_0^i(3, 1)), \chi_1^w(0, \chi_0^i(1, -1))))$ can be interpreted as $x = 1$ for no run-time iteration, or $x = \chi_0^i(2, 0)$ for one run-time iteration. For two run-time iterations, $x = \chi_1^i(\chi_0^i(3, 1), \chi_0^i(1, -1))$, i.e., 3, 1, or -1 depending on the four possible combinations of control flow options in the two run-time iterations over the if statement in the loop body.

The general pattern is $x_j^b = \chi_j^b(\ldots \chi_{j-1}^b(\ldots \chi_0^b(\ldots) \ldots) \ldots)$. That is, a χ-term for the j-th loop contains and depend on decisions $\chi_0^b, \ldots, \chi_{j-1}^b$ with the same block number b and iteration indices $0, \ldots, j-1$. This pattern occurs over and over again, since each loop iteration results in a new composition of χ^b with itself. This results in countably many χ-terms and, hence, in a non-terminating analysis if no measure is taken to stop the analysis.

One possible way to handle this problem is, for a χ-term $t = \chi_j^b(t_1, \ldots, t_p)$, to replace every sub-term of t that has the same block number as t by its $\sqcup$-approximation. Using this approach, the approximated term(s) for the example are:

$$\begin{aligned} x_0^w &= 1 \\ x_1^w &= \chi_1^w(1, \chi_0^i(2, 0)) \end{aligned}$$

$$\begin{aligned}
x_2^w &= \chi_2^w(1, \chi_1^i(\chi_1^w(2, \chi_0^i(3, 1)), \chi_1^w(0, \chi_0^i(1, -1)))) \\
&\sqsubseteq \chi_2^w(1, \chi_1^i(\dot{\sqcup}(2, \dot{\sqcup}(3, 1)), \dot{\sqcup}(0, \dot{\sqcup}(1, -1)))) \\
&= \chi_2^w(1, \chi_1^i(\dot{\sqcup}(2, \top), \dot{\sqcup}(0, \top))) \\
&= \chi_2^w(1, \chi_1^i(\top, \top)) \\
&= \chi_2^w(1, \top) \\
x_3^w &= \ldots = \chi_3^w(1, \top) \\
&\quad \ldots
\end{aligned}$$

The terms $x_j^w = \chi_j^w(t_1, t_2) = \chi_j^w(1, \top)$ represent the value of x after $0 \ldots j$ run-time iterations and $t_1 = x_0^w = 1$ the value after the 0-th iteration, actually no iteration. However, $t_2 = \top$ represents *all* abstract values after $1, \ldots, j$ run-time iterations. Since a $\sqcup$-approximation is always larger than the term it is applied to, this widening does not sacrifice the monotonicity of updates. Notice, disregarding loop indices in χ_2^w and χ_3^w, the terms are the same and will remain so in all the following run-time iterations.

In general, assume program graphs $\mathcal{G}_0$ with a loop nesting depth of 0, i.e., they do not contain any loops, only sequential and conditional code. Trivially, all χ-terms analyzed for the nodes of $\mathcal{G}_0$ have a finite depth. Further, assume program graphs $\mathcal{G}_1$ with a loop nesting depth of 1, i.e., all with loops contain program graphs of $\mathcal{G}_0$. If we applied the described $\sqcup$-approximation on all χ-terms generated by the ϕ nodes of loop heads, the χ-terms would have a final depth, i.e., for each w.l.o.g.

$$depth(t_1, \ldots, t_p) = \max(depth(t_1), \ldots, depth(t_p)) + 1$$

Structural induction on all such sets of program graphs $\mathcal{G}_0, \mathcal{G}_1, \ldots \mathcal{G}_i, \mathcal{G}_{i+1}, \ldots$ shows that this $\sqcup$-approximation on χ-terms stemming from loop heads leads to final depths of all χ-terms in an analysis of arbitrary program graphs $\mathcal{G}$.

As the loop indices j are arbitrary, we reached a fixed point when the χ-terms do not change disregarding these loop indices and we can stop the analysis of the loops.

Under this widening abstraction, we extend the equivalence Definition 9 by:

$$\chi_j^b(t_1, \ldots, t_p) \equiv \chi_{j'}^b(t'_1, \ldots, t'_p) \Leftrightarrow t_1 \equiv t'_1, \ldots, t_p \equiv t'_p \tag{10}$$

which makes (all) the loop indices j, j' obsolete. As the updates are monotone, the depths of χ-terms is finite, and the set of χ-functions is finite too, the analysis is guaranteed to terminate in a fixed point.

In the above example, we get the fixed point

$$x^w \equiv x_3^w \equiv x_2^w \equiv \chi^w(1, \top).$$

The interpretation is that $x = 1$ if there has been no run-time iteration and $x = \top$, i.e., unknown, if there has been one or more run-time iterations.

We refer to the above analysis as 1-loop-analysis: for a given loop head ϕ-node, it replaces every sub-term of the corresponding χ-term $\chi_j^b(t_1, \ldots, t_p)$ with indices $j - 1, \ldots, 1$ by their $\sqcup$-approximation. In the next section, we generalize the 1-loop-analysis to l-loop-analysis and conclude with a general widening abstraction that replaces the fixed point condition in (10).

4.2 The *l*-Approximation

As stated above, we can get a more precise analysis if we instead used a 2-loop-analysis where we $\sqcup$-approximated all sub-terms of $\chi_j^b(t_1, \ldots, t_p)$ with indices $j - 2, \ldots, 1$. This idea can of course be generalized to an arbitrary number l:

Definition 12 (l-loop-approximated χ-term) Let $t \in X_A$ be a χ-term rooted by a χ-function χ_j^b and let $F \subseteq \mathit{func}(t)$ be all χ-functions of t, such that

$$\chi_y^x \in F \Leftrightarrow b = x \wedge j - y \geq l.$$

The *l-loop-approximation* of t is then a new χ-term where every $\chi_y^x \in F$ has been replaced by $\dot{\sqcup}$.

An analysis where every newly created χ-term related to a loop head ϕ-node is immediately loop approximated is said to be a *l-loop-approximated analysis*.

This l-loop-approximation is easy to understand as a tree manipulation. We make a post order traversal of the tree and replace each χ-term having the same block number as the root node χ_j^b, and an iteration index in range $1, \ldots, j - l$, with their corresponding $\sqcup$-approximation. This approach is implemented in Algorithm 7 and discussed in Section 5.4.

The l-loop-approximation is conservative, since any $\sqcup$-approximation is conservative. As the special case of a 1-loop-approximation, it also guarantees that every χ-term has a finite depth. However, as for this special case, we need in general guarantee that we do not get new χ-functions with new loop indices j and, hence, never reach a fixed point for the (finite depth) χ-terms.

Before we present a general solution, we recap the example of Fig. 5 once again. For the χ-term of x_3^w, the 2-loop-approximation is given below

$$\begin{aligned}
x_3^w \equiv\ & \chi_3^w(1, \chi_2^i(\chi_2^w(2, \chi_1^i(\chi_1^w(3, \chi_0^i(4, 2)), \chi_1^w(1, \chi_0^i(2, 0)))),\\
& \chi_2^w(0, \chi_1^i(\chi_1^w(1, \chi_0^i(2, 0)), \chi_1^w(-1, \chi_0^i(0, -2))))))\\
\sqsubseteq\ & \chi_3^w(1, \chi_2^i(\chi_2^w(2, \chi_1^i(\dot{\sqcup}(3, \chi_0^i(4, 2)), \dot{\sqcup}(1, \chi_0^i(2, 0)))),\\
& \chi_2^w(0, \chi_1^i(\dot{\sqcup}(1, \chi_0^i(2, 0)), \dot{\sqcup}(-1, \chi_0^i(0, -2))))))\\
\equiv\ & \chi_3^w(1, \chi_2^i(\chi_2^w(2, \chi_1^i(\chi_0^i(\top, \top)), \chi_0^i(\top, \top)),\\
& \chi_2^w(0, \chi_1^i(\chi_0^i(\top, \top)), \chi_0^i(\top, \top))))\\
\equiv\ & \chi_3^w(1, \chi_2^i(\chi_2^w(2, \top), \chi_2^w(0, \top)))
\end{aligned}$$

This analysis result can be interpreted as $x = 1$ for no iteration, or $x = \chi_2^i(2, 0)$ for one run-time iteration, depending on the if condition in this run-time iteration. For two and three run-time iterations, the result is $\top$, i.e., we do not know.

Continuing with the analysis of the loop using this 2-loop-approximated term gives:

$$\begin{aligned}
x_3^i &= \chi_3^i(+(x_3^w, 1), -(x_3^w, 1))\\
&= \chi_3^i(+(\chi_3^w(1, \chi_2^i(\chi_2^w(2, \top), \chi_2^w(0, \top))), 1),
\end{aligned}$$

$$\begin{aligned}
&\quad -(\chi_3^w(1, \chi_2^i(\chi_2^w(2, \top), \chi_2^w(0, \top))), 1)) \\
&\equiv \chi_3^i(\chi_3^w(2, \chi_2^i(\chi_2^w(3, \top), \chi_2^w(1, \top))), \\
&\quad \chi_3^w(0, \chi_2^i(\chi_2^w(1, \top), \chi_2^w(-1, \top)))) \\
x_4^w &= \chi_4^w(1, x_3^i) \\
&= \chi_4^w(1, \chi_3^i(\chi_3^w(2, \chi_2^i(\chi_2^w(3, \top), \chi_2^w(1, \top))), \\
&\quad \chi_3^w(0, \chi_2^i(\chi_2^w(1, \top), \chi_2^w(-1, \top))))) \\
&\sqsubseteq \chi_4^w(1, \chi_3^i(\chi_3^w(2, \chi_2^i(\dot{\sqcup}(3, \top), \dot{\sqcup}(1, \top))), \\
&\quad \chi_3^w(0, \chi_2^i(\dot{\sqcup}(1, \top), \dot{\sqcup}(-1, \top))))) \\
&\equiv \chi_4^w(1, \chi_3^i(\chi_3^w(2, \chi_2^i(\top, \top)), \\
&\quad \chi_3^w(0, \chi_2^i(\top, \top)))) \\
&\equiv \chi_4^w(1, \chi_3^i(\chi_3^w(2, \top), \chi_3^w(0, \top)))
\end{aligned}$$

The final analysis result can be interpreted as $x = 1$ for no iteration, or $x = \chi_3^i(2, 0)$ for one run-time iteration, depending on the if condition in this run-time iteration. For two, three, and four run-time iterations, it is $\top$, i.e., we do not know the result. Once again, disregarding loop indices, the χ-terms χ_3^w and χ_4^w are the same . We have reached a fixed point, if we abstract *two, three, and four* run-time iterations to *all further* run-time iterations.

To formalize and generalize this, we extend the equivalence Definition 9 generalizing on (10). We do this in two steps. First, we introduce a loop index substitution and second, we define terms under this substitution as equivalent.

Definition 13 (Loop index substitution) The loop-index-substituted χ-term of a χ-term t, denoted $loop\_index\_sub(t)$, is the term identical to t except for all loop indices j of the χ-functions in $func(t)$ are consistently replaced by $j - 1$.

Under the l-loop-approximation (a widening abstraction), we extend the equivalence Definition 9 by:

$$t \equiv loop\_index\_sub(t) \tag{11}$$

If we apply loop index substitution with any update, the updates are still monotone, the depths of χ-terms is finite, and the set of χ-functions is finite. Hence, the analysis is guaranteed to terminate in a fixed point.

4.3 The *k*-Approximation

The l-loop-approximation is sufficient to guarantee a conservative analysis that terminates, and thereby be sound. However, in practice, it is likely to be both slow and memory costly since the χ-terms, although finite, are likely to get large. A straightforward way to reduce the size of the χ-terms is to limit their maximum depth. This idea is similar to the finite call depth in a CFA analysis [34], or the context depth limitations in object-sensitive or this-sensitive points-to analysis [23, 24, 27].

In our case, we keep track of the last k control-flow options that might influence the value of a variable. Assuming a control-flow based block numbering as outlined in Section 3.1, this means that we keep more ”recent” control-flow options whereas more “remote” options are $\sqcup$-approximated.

The k-approximation of χ-terms is easy to understand using the tree representation $G_t = \{N, E, r\}$. Whenever a new χ-term t is generated, we replace all χ-terms $t_{sub} = \chi_i^b(t_1, \ldots, t_p)$ in *subterms*(t) that has $depth(t_{sub}, t) \geq k$ with $\dot{\sqcup}(t_1, \ldots, t_p)$. The process starts in the leaves and proceeds towards the root node. The result is a new χ-term $t^{(k)}$ with $depth(t^{(k)}) \leq k$ that only embodies the last k control-flow options that might influence the value. Notice also that in the case $k = 0$ all context-sensitive information is lost and we have a context-insensitive analysis. An algorithm for k-approximation is formalized and discussed in Section 5.3.

The following example shows the result of two different k-approximations of the same χ-term a and Fig. 6 shows the corresponding tree representations.

$$\begin{aligned} a &= \chi^3(\chi^1(1,2), \chi^2(\chi^1(3,4),2)) = k\_approx(k \geq 3, a) \\ a^{(2)} &= \chi^3(\chi^1(1,2), \chi^2(\{3,4\},2)) = k\_approx(k = 2, a) \\ a^{(1)} &= \chi^3(\{1,2\},\{2,3,4\}) = k\_approx(k = 1, a) \end{aligned}$$

4.4 Approximation Summary

The l-loop-approximation and the k-approximation can be seen as two different strategies to apply the $\sqcup$-approximation. The aim of the l-approximation is to avoid generating infinite χ-terms when analyzing loops and other cyclic control-flow dependencies. It also guarantees analysis termination. The purpose of the k-approximation is to, at the cost of analysis precision, speed up the analysis and reduce the memory costs. They can be used separately or combined into what

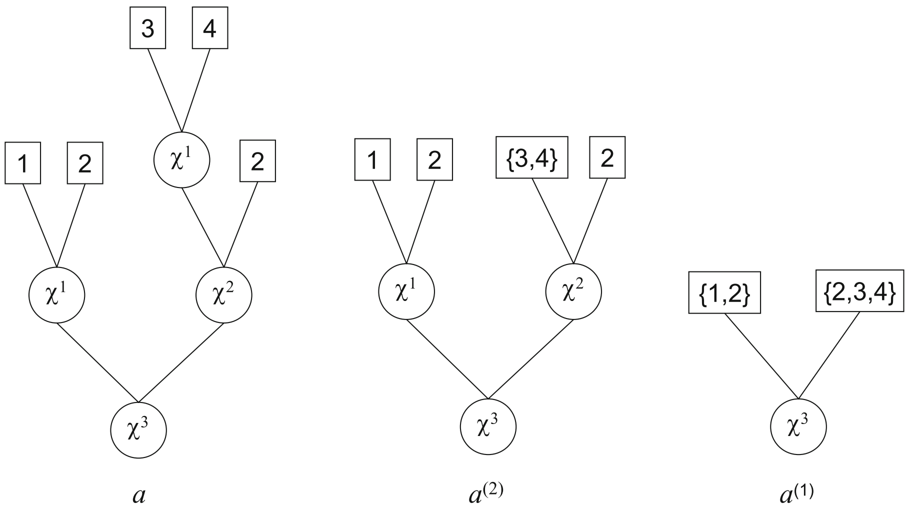

Fig. 6 Two different finite k approximations $a^{(2)}$ and $a^{(1)}$ and of the same χ-term a

we refer to as a k, l-approximated analysis where loop related χ-terms are l-loop -approximated, and where all sub-terms having a depth $\geq k$ are $\sqcup$-approximated.

The k, l-approximation is conservative since any $\sqcup$-approximation is conservative. It also guarantees that every χ-term has a finite depth and that the analysis will terminate. Hence,

Theorem 5 *Let $(A, \sqsubseteq, F, \iota)$ define a sound context-insensitive program analysis. Then $(k, l, X_A, \sqsubseteq_\chi, F, \iota)$, the corresponding k, l-approximated context-sensitive program analysis using χ-terms, is sound as well.*

Theorem 5 concludes our formal presentation of χ-term based context-sensitive data-flow analysis. It shows that any sound context-insensitive analysis can be transformed into a sound context-sensitive χ-term based analysis that is guaranteed to terminate.

5 Compact Representations of χ-Terms

In this section, we discuss the compact representations of χ-terms along with some implementation details allowing for fast and memory efficient analyses. We start with a introductory example of compact χ-term representations in Section 5.1. In Section 5.2, we show how to create and maintain the redundant-free DAG-based representations of χ-terms without creating larger intermediate terms with redundancies that would then be in need of subsequent redundancy elimination. In Section 5.3, we introduce the creation of k-approximated χ-terms that limit their depth to an adjustable parameter k in order to trade precision against memory. In Section 5.4 we show an algorithm for l-loop-approximation to limit the size of loop-head χ-terms. Finally, in Section 5.5, we discuss the memory management of these two variants of χ-term representations.

5.1 Introductory Example of a Compact s of χ-Terms Representation

Consider the simple code fragment of Fig. 7. After the normalization with Algorithm 1 applying equivalence conversion (EC) but no termination conversion (TC) of $eval_\chi$, the χ-terms for the variables x, y, a, b in are:

$$\begin{aligned}
x &= \chi^4(1, 2),\\
y &= \chi^7(\chi^4(1, 2), 2),\\
b &= \chi^7(3, 4),\\
a &= \chi^{10}(\chi^4(1, 2), \chi^7(\chi^4(1, 2), 2)),\\
s &= a + b = +(\chi^{10}(\chi^4(1, 2), \chi^7(\chi^4(1, 2), 2)), \chi^7(3, 4))
\end{aligned}$$

As discussed in Section 3.3, every χ-term can naturally be viewed as a tree. This is illustrated in Fig. 8 (left) where we show the tree representation of the χ-term for the

```
if (...) (#1)
  x = 1; (#2)
else
  x = 2; (#3)
if (...){(#4)
  y = x; (#5)
  b = 3; (#5)
}
else {
  y = 2; (#6)
  b = 4; (#6)
}
if (...) (#7)
  a = x; (#8)
else
  a = y; (#9)
s = a+b; (#10)
return s;(#11)
```

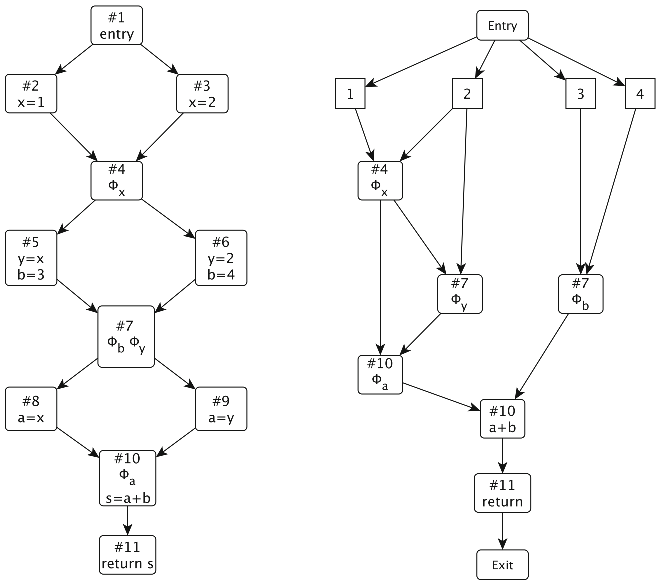

Fig. 7 A source code example with corresponding basic block and SSA graphs

variable a. Each edge represents a particular control-flow option and each path from the root node to a leaf value contains the sequence of control-flow decisions required for that particular leaf value to be calculated in the corresponding program run.

Obviously, representing χ-terms as trees is in practice by far too expensive. A more cost efficient representation is a directed acyclic graph (DAG), similar to Binary-Decision-Diagrams (BDDs) [6, 7], which avoids redundant sub-trees. This is illustrated in Fig. 8 (right).

The property of the suggested DAG representation of terms is that *every* sub-tree is only constructed once, and then referred to when needed elsewhere instead of being reconstructed whenever needed. We need to maintain this value semantics of sub-term DAGs in a redundancy-free, hence, memory efficient, representation (implementation) of χ-terms.

Still, we can interpret the χ-terms in the DAG representation as before in the tree representation. For example, both representations of the term analyzed for the variable a can be interpreted as the values 1 and 2, resp., depending on the same control flows of the program, e.g., $a = 1$ if the conditions in all blocks #1, #4, #7 are true and, consequently, the left predecessor provides the value of a in all blocks #4, #7, #10.

Not only the interpretation is the same for the two representations, tree or DAG. Also the operations can be applied to both representations. Especially, the DAG representation does not need to be unfolded to a tree, in order to apply $eval_\chi$ including Shannon expansion and redundancy elimination.

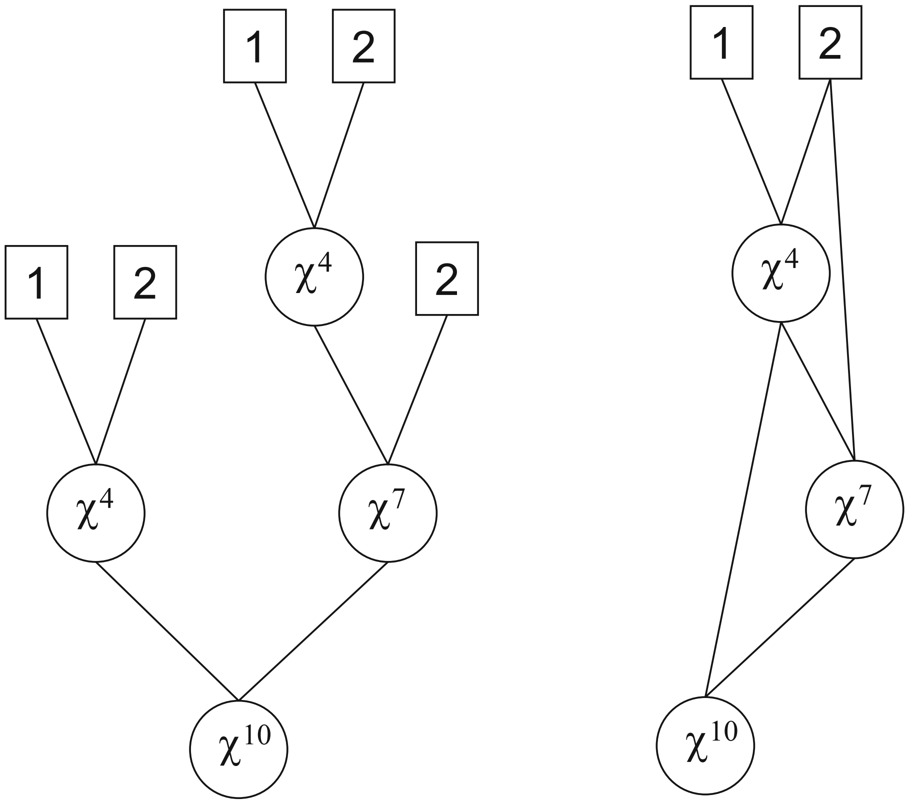

Fig. 8 Term for variable $a = \chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2))$ is illustrated by its tree and directed acyclic graph representations (root at the bottom)

For example, the below sequence shows the evaluation steps during normalization of Algorithm 1, applied to the χ-term for variable $s = a + b$.

$$\begin{aligned}
s &= +(\chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2)), \chi^7(3,4)) \\
&\equiv \chi^{10}(+(\chi^4(1,2), \chi^7(3,4)), +(\chi^7(\chi^4(1,2),2), \chi^7(3,4))) \\
&\equiv \chi^{10}(\chi^7(+(\chi^4(1,2),3), +(\chi^4(1,2),4)), \chi^7(+(\chi^4(1,2),3), +(2,4))) \\
&\equiv \chi^{10}(\chi^7(\chi^4(+(1,3),+(2,3)), \chi^4(+(1,4),+(2,4))), \\
&\quad\ \chi^7(\chi^4(+(1,3),+(2,3)),+(2,4))) \\
&\equiv \chi^{10}(\chi^7(\chi^4(4,5), \chi^4(5,6)), \chi^7(\chi^4(4,5),6))
\end{aligned}$$

The χ-term notation corresponds to the tree representation including some redundancy. For example, the sub-terms $\chi^4(1,2)$ (in lines 2 and 3 of the example transformation) and $\chi^4(4,5)$ (in the last line) occur several times in one and the same term, but cannot be removed by redundancy elimination.

Figure 9 shows the same evaluation steps directly transforming the DAG representation of the χ-term. The leftmost DAG in the figure represents the χ-term for s. The DAGs from left to the right correspond to new χ-terms after each evaluation step in

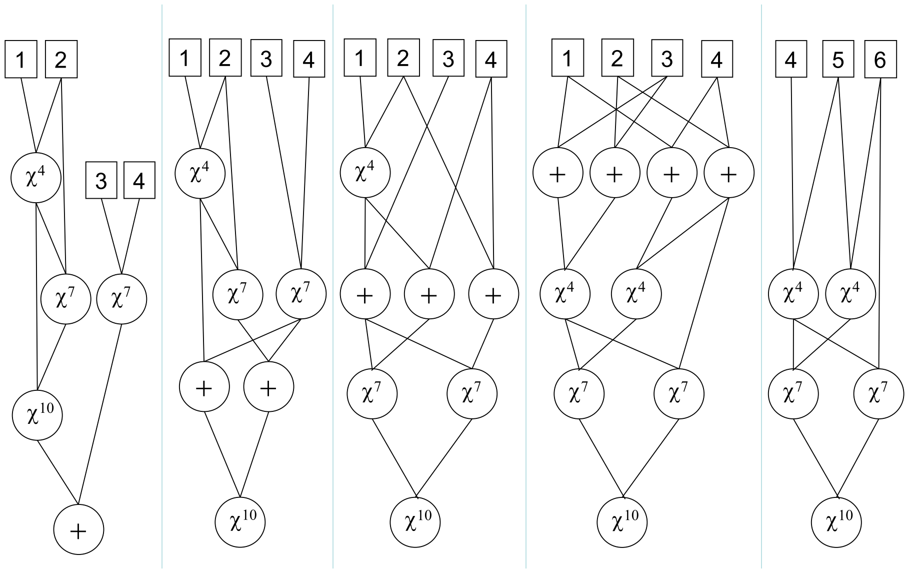

Fig. 9 Evaluation steps applying the equivalence conversions (EC) of $eval_\chi$ on the term generated for variable $s = a+b = +(\chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2)), \chi^7(3,4))$ in directed acyclic graph representation

the lines of the example (each equivalent to the original term). In each χ-term, any sub-term is only represented once and then referred to, e.g., the sub-terms $\chi^4(1,2)$ and $\chi^4(4,5)$ again.

It remains to show how a DAG based χ-term representation can be implemented efficiently maintaining the property that each sub-term is only constructed once, and than referred to when needed elsewhere.

5.2 Efficient Updates of Redundant-Free χ-Terms

We capture all unique χ-terms in a repository. When a new χ-term needs to be created during analysis, we check if there is an equivalent χ-term already captured in this repository. A straight-forward implementation of $\equiv_\chi$ would recursively apply the Shannon expansion to bring the χ-terms in a normal form with χ-nodes ordered the same way, then recursively apply redundancy elimination on the (sub-) terms, and finally check for the equivalence of χ-terms by comparing the root nodes and their children recursively. This is a too expensive implementation.

Our suggested implementation implements redundancy elimination on-the-fly. It is based on maintaining some properties of the terms captured in the repository, and on aggressive hashing as outlined below and presented in detail in Algorithms 2 and 3 that follows.

Property (i): By construction, χ-terms stored in the repository do not contain operation function symbols `f`. Instead, χ-terms are captured in normal form after applying all equivalence conversions (EC) steps of $eval_\chi$.

Property (ii): Each basic unique χ-term is a unique context-sensitive value a. We refer to a as the value number $vn = a$ of the basic χ-term.

Property (iii): Each non-basic χ-term represents a unique context-insensitive value; it is also identified with a unique value number vn. Each non-basic χ-term is rooted by a χ-function symbol χ_j^b, which is uniquely determined by a block number b and an iteration index j. Each non-basic χ-term is captured as a tuple $[\chi_j^b, vn_1, \ldots, vn_p]$ with the unique identifier of the root χ-function symbol χ_j^b and $vn_1, \ldots, vn_p$ the value numbers for its children.

Note that the roots of different non-basic χ-terms may have the same χ-function symbol but different vn; vice-versa, identical χ-terms always have the same vn and the same χ-function symbol.

Property (iv): We avoid χ-function symbols χ_j^b occurring in different orderings on the paths from the root to the leaves of a χ-term representation. This property trivially holds for the basic χ-terms (not containing χ-function symbols at all). For the non-basic χ-terms, it can easily be maintained by following the standard update order of a (forward) program analysis where SSA nodes are analyzed and updated in a data-driven way: nodes before their successors; inner loop nodes before outer loops. Otherwise, any consistent order scheme can be enforced if needed by Shannon expansion.

Property (v): We maintain a hash-map h mapping each χ-term created earlier—its root node `r` and its sub-terms' value numbers—to its corresponding value number vn:

$$h : [\mathtt{r}, vn_1 \ldots, vn_p] \mapsto vn$$

Here $\mathtt{r} \in \{\chi_j^b, \mathtt{f}\}$ is a χ- or an operation-function symbol. Note that we maintain the operation-function symbols `f` in this mapping, but not in the representation of χ-terms. We also maintain the inverse mapping:

$$h^{-1} : vn \mapsto [\mathtt{r}, vn_1 \ldots, vn_p]$$

Assuming (i)–(v), two cases need to be distinguished when analyzing and updating an SSA node *node*. Next we informally describe them before we formalize them in Algorithms 2 and 3.

Case 1 *node* is a ϕ node in block b analyzed under the current loop index j and has the predecessors $node_1, \ldots, node_p$. Recall that the (current) analysis values of these predecessors are χ-terms identified with their respective (current) value numbers $vn_1, \ldots, vn_p$. If h contains a value number vn for the tuple $(\chi_j^b, vn_1, \ldots, vn_p)$, a corresponding χ-term has been created earlier, which is then analysis update for this *node*. Otherwise, in the special case of $vn_1 = \ldots = vn_p$, we choose vn_1 as the value analyzed for *node* and book-keep this in h. This implements redundancy elimination. No new χ-term has to be created; the repository does not change. In general, if at least two predecessor value numbers are different, we create a new unique χ-term, i.e., a root χ_j^b with a references to its children $vn_1, \ldots vn_p$, and add it to the repository. We also create a new value number vn for this new unique term and book-keep this in h.

Algorithm 2 *update*(*node*).

```
for all $n_i \in preds(node)$ do
    $vn_i \leftarrow current\_analysis(n_i)$
end for
if $type(node) = \phi_j^b$ then
    // Case 1: update a $\phi$ SSA node
    $vn \leftarrow create(\chi_j^b, vn_1, \ldots, vn_p)$ // create a $\chi$-term
    //$vn \leftarrow k\_approx(k, [\chi_j^b, vn_1, \ldots, vn_p])$ // alternatively, create a $k$-
      approximated $\chi$-term
    //$vn \leftarrow l\_approx(l, [\chi_j^b, vn_1, \ldots, vn_p])$ //alt. for a loop head create a $l$-
      approximated $\chi$-term
else if $type(node) = \mathtt{f}$ then
    // Case 2: update an operation SSA node
    $vn \leftarrow create(\mathtt{f}, vn_1, \ldots, vn_p)$
end if
$current\_analysis(node) \leftarrow vn$
```

Case 2 *node* is an operation node with a function symbol f and with the predecessor (argument) values $vn_1, \ldots, vn_p$. Again, we check if the result of this update is known from before by consulting the hash-map $h(\mathtt{f}, vn_1, \ldots, vn_p)$. If h contains a corresponding value number vn, the new χ-term has been created earlier, which is then the analysis update for *node*. Again, no new χ-term has to be created; the repository does not change.

Otherwise, we need to recursively apply the Shannon expansion over the predecessor χ-terms $h^{-1}(vn_1), \ldots, h^{-1}(vn_p)$ to push f to the leaves of these terms.

In the base case, $vn_1, \ldots, vn_p$ represent basic context-insensitive values. In this base case, we apply the insensitive analysis function f of f to its insensitive arguments denoted by $vn_1, \ldots, vn_p$ and get a insensitive result denoted by vn. If it has not been computed earlier, we update h accordingly.

In the recursive case, at least one of the χ-terms $h^{-1}(vn_1), \ldots, h^{-1}(vn_p)$ has a χ-function symbol as its root. Assume χ_j^b is the root of all these χ-terms with the highest iteration index j and block number b as the secondary ordering criterion, i.e., the highest (b, j)-rank. Accordingly, we implement the following Shannon expansion as a recursion step:

$$\begin{aligned} t &= \mathtt{f}(h^{-1}(vn_1), \ldots, h^{-1}(vn_p)) \\ t' &= \chi_j^b(t|_{(b,j):1}, t|_{(b,j):2}, \ldots, t|_{(b,j):p'}) \quad \text{where } p' = arity(\chi_j^b) \end{aligned}$$

Each restriction $t|_{(b,j):l}, l \in [1 \ldots p']$ constitutes a new χ-term creation request, i.e., a recursive application of *Case 2* until the insensitive base cased is reached. Each such request leads to a χ-term with a corresponding value number vn^l. The final χ-term creation

$$\chi_j^b(vn^1, \ldots, vn^{p'})$$

is an application of *Case 1*.

The analysis update of SSA nodes is given in Algorithm 2. It calls the redundancy free χ-term creation formalized in Algorithm 3. Both algorithms follow the two cases described above.

Algorithm 3 *create*$(fs, vn_1, \ldots, vn_p) \mapsto vn$.

```
if $fs = \chi_j^b$ then
    // Case 1: invoked to update a $\phi$ SSA node or in upwards recursion
    $vn \leftarrow h(\chi_j^b, vn_1, \ldots, vn_p)$ // check hash map if $\chi$-term exists in repository
    if $vn = null$ then
        // $\chi$-term does not exist in repository
        if $vn_1 = \ldots = vn_p$ then
            $vn \leftarrow vn_1$ // implicit redundancy elimination
        else
            $vn \leftarrow new([\chi_j^b, vn_1, \ldots, vn_p])$ // create a new $\chi$-node
        end if
        $h([\chi_j^b, vn_1, \ldots, vn_p]) \leftarrow vn$
        $h^{-1}(vn) \leftarrow [\chi_j^b, vn_1, \ldots, vn_p]$
    end if
else if $fs = \mathtt{f}$ then
    // Case 2: invoked to update an f operation SSA node or in downwards
      recursion
    $vn \leftarrow h(\mathtt{f}, vn_1, \ldots, vn_p)$ // check hash map if $\chi$-term exists in repository
    if $vn = null$ then
        // $\chi$-term does not exist in repository
        if basic$(vn_1) \wedge \ldots \wedge$ basic$(vn_p)$ then
            $vn \leftarrow apply(\mathtt{f}, vn_1, \ldots, vn_p)$ // apply context-insensitive transfer
              function
        else
            $\chi_{j'}^{b'} \leftarrow \chi$-node with highest $(b', j')$-rank in $vn_1, \ldots, vn_p$
            $p' \leftarrow arity(\chi_{j'}^{b'})$
            for $l \leftarrow 1 \ldots p'$ do
                $vn^l \leftarrow create(\mathtt{f}, vn_1|_{(b,j):l}, \ldots, vn_k|_{(b,j):l})$ // recursively create
                  new $\chi$-terms (Case 2)
            end for
        end if
        $vn \leftarrow create(\chi_{j'}^{b'}, vn^1, \ldots, vn^{p'})$ // recursively create a new $\chi$-term (Case 1)
        $h([\mathtt{f}, vn_1, \ldots, vn_p]) \leftarrow vn$
        $h^{-1}(vn) \leftarrow [\mathtt{f}, vn_1, \ldots, vn_p]$
    end if
end if
return $vn$
```

The algorithms use the functions *preds*, *current_analysis*, *type*, *basic*, *apply*, *arity*, and *new*. They are informally defined as follows. The function *preds* applied to an SSA *node* returns the predecessors of *node* in the SSA graph. The function *current_analysis* applied to an SSA *node* returns the value number of the χ-term analyzed for *node*. The function *type* applied to an SSA node returns its node type, i.e., ϕ_j^b or `f`. The test function *basic*(vn) checks if the term $h^{-1}(vn)$ is an context-insensitive value (*true*) or contains a χ-function symbol (*false*). Assume *basic*(vn_i) is *true* for all parameters $i \in [1, p]$, i.e., $h^{-1}(vn_i) = a_i \in A$, a context insensitive analysis value, then we $apply(\mathtt{f}, vn_1, \ldots, vn_p)$. This evaluates the context-insensitive transfer function f of `f` to the context-insensitive analysis values:

$$\begin{aligned} vn &= f(h^{-1}(vn_1), \ldots, h^{-1}(vn_p)) \\ &= f(a_1, \ldots, a_p) \end{aligned}$$

returning the value number vn equal to the resulting basic context-insensitive value and equal to $eval_\sqcup$ in Section 2. According to its definition, the function *arity*(χ_j^b) gives the number of children of a χ-node. Finally, the function *new* just creates a new χ-node of the DAG representing all χ-terms.

5.3 The *k*-Approximated χ-Term Creation

In case we want to limit the size of χ-terms by limiting their depths to k, we can replace the call to *create*$(\chi_j^b, vn_1, \ldots, vn_p)$ with a call to the widening function *k_approx*$(k, [\chi_j^b, vn_1, \ldots, vn_p])$, cf. the out-commented alternative line in Algorithm 2. This alternative assures that the terms $h^{-1}(vn_1), \ldots, h^{-1}(vn_p)$ have depths of at most $k - 1$ by merging too deep leaves applying the insensitive meet function $\sqcup$ instead of keeping context separated with the corresponding χ functions. The function *k_approx* and its subroutine *collapse* are defined in the Algorithms 4 and 5, respectively.

The two cases $n = a \in A$ and $k = 0$ of *k_approx* defined in Algorithm 4 are base cases. The recursive step creates a new χ-term guaranteed to have sub-terms with maximum depth $k - 1$.

The function *collapse* defined in Algorithm 5 aggressively collapses any sub-terms by $\sqcup$-approximating any remaining non-basic χ-term.

5.4 The *l*-Approximated χ-Term Creation

Section 4.2 introduce the l-approximation as a mean to stabilize the analysis of loops and to guarantee analysis termination. The basic idea is to replace every subterm $\chi_q^b(t_1, \ldots, t_p)$, with indices $q = j - l, \ldots, 0$, of a loop head χ-term χ_j^b by their $\sqcup$-approximation. The function *l_approx* and its subroutine *loop_approx* are defined in the Algorithms 6 and 7, respectively.

Algorithms 6 starts by checking if the node is a loop head. A loop head χ-term always looks like $[\chi_j^b, vn_1, vn_2]$, a χ-function χ_j^b, and two abstract values representing the loop entry value (represented by vn_1) and the values after one or more run-time iterations (represented by vn_2). We have two special cases where no approximation takes place: 1) At first entry ($j = 0$) we know that vn_1 is the value to use (point 5 of Definition 9). 2) The first l run-time iterations are not approximated and we apply the default handling for new χ-terms (Algorithm *create*). The actual approximation is handled in function *loop_approx*.

Algorithm 4 *k_approx*$(k, n) \mapsto vn$.

```
if $n = a \in A$ then
    // $n$ is a basic $\chi$ node
    $vn \leftarrow a$
else if $k = 0$ then
    // collaps subgraphs when max depth reached
    $vn \leftarrow collapse(n)$
else
    //$n = [\chi_j^b, vn_1, \dots, vn_p]$ is a non-basic $\chi$ node
    for all $vn_i \in vn_1, \dots, vn_p$ do
        if $depth(vn_i) < k$ then
            $vn_i^* \leftarrow vn_i$
        else
            $vn_i^* \leftarrow k\_approx(k-1, h^{-1}(vn_i))$
        end if
    end for
    // Create a $\chi$-term and add it to the repository if it does not exist already
    $vn \leftarrow create(\chi_j^b, vn_1^*, \dots, vn_p^*)$
end if
return $vn$
```

Algorithm 5 *collapse*$(n) \mapsto a \in A$.

```
if $n \neq a \in A$ then
    //$n = [x_j^b, vn_1, \dots, vn_p]$ is a non-basic $\chi$ node
    for all $vn_i \in vn_1, \dots, vn_p$ do
        $a_i \leftarrow collapse(h^{-1}(vn_i))$
    end for
    $a \leftarrow \sqcup(a_1, \dots, a_p)$
end ifreturn $a$
```

Algorithm 7, *loop_approx*(l, b, j, vn), recursively traverses vn in search of subterms χ_x^y with block number $y = b$ and iteration index $x = j - l$ which then are $\sqcup$-approximated using *create*$(\sqcup, vn'_1, \dots, vn'_p)$. Notice: 1) we search for subterms with iterations index $x = j - l$ since more remote subterms ($x > j - l$) will not exist if the l-approximation is applied consistently, and 2) *create*$(\sqcup, vn'_1, \dots, vn'_p)$ makes use of Case 2 of the create algorithm and pushes $\sqcup$ to the leaves where it is applied.

5.5 Memory Management of χ-Terms During Analysis

Finally, we show that the memory consumed by the intermediate results of χ-terms can be controlled on-the-fly by identifying unused χ-(sub-)terms that can be freed directly. This means that we do not create large redundant-free but *unused* intermediate analysis results and that the size of the finally analyzed χ-terms is actually an upper bound of the memory footprint consumed during the whole analysis. We argued for this property of our χ-term representation in [12] and exploited it there by assessing and comparing only the memory sizes of the final analysis results of χ-term-based and alternative representations. Below we describe the "garbage collection" that guarantees this property.

Algorithm 6 $l\_approx(l, n) \mapsto vn$.

```
if $n$ is loop head then
    // n = $[\chi_j^b, vn_1, vn_2]$
    if $j = 0$ then
        // Special case when first entering a loop, see point 5, Definition 9
        $vn \leftarrow vn_1$
        $h([\chi_j^b, vn_1, vn_2]) \leftarrow vn$
        $h^{-1}(vn) \leftarrow [\chi_j^b, vn_1, vn_2]$
    else if $j \leq l$ then
        // First $l$ subterms are not approximated
        $vn \leftarrow create(\chi_j^b, vn_1, vn_2)$
        //$vn \leftarrow k\_approx(k, [\chi_j^b, vn_1, vn_2])$ // alternatively, create a $k$-approximated $\chi$-term
    else
        // loop approximate $vn_2$
        $vn \leftarrow loop\_approx(l, b, j, vn_2)$
    end if
else
    // No loop head, apply default
    $vn \leftarrow create(\chi_j^b, vn_1, \ldots, vn_p)$
    //$vn \leftarrow k\_approx(k, [\chi_j^b, vn_1, vn_2])$ // alternatively, create a $k$-approximated $\chi$-term
end if
return $vn$
```

Since χ-terms are directed *acyclic* graphs, we can use simple reference counting to free unused χ-(sub-)terms. Since value numbers are unique for each χ-(sub-)term, we can book-keep the references to a value number in order to count the references to each χ-(sub-)term and maintain a simple function:

$$count : vn \mapsto int$$

There are only few locations in our analysis algorithms where new references to value numbers are added. One is the call $vn \leftarrow new([\chi_j^b, vn_1, \ldots, vn_p])$ in Algorithm 3. Each execution of this call to *new*, we set $count(vn) \leftarrow 0$ (no reference to vn is added yet) and $\forall i \in [1, p] : count(vn_i) \leftarrow count(vn_i) + 1$.

The roots references of the χ-terms are the analysis values for each SSA *node* captured in $current\_analysis(node)$. Hence, on each execution of an analysis value update $current\_analysis(node) \leftarrow vn$ in Algorithm 2, we might need to update the reference $count(vn)$, as well. More specifically, if a value number vn returned by the *create* call is different from the value number vn' analyzed so far for the current *node*, we need to increase the count for vn and decrease it for vn'. To do so, we substitute the last line in Algorithm 2 by

Algorithm 7 $loop\_approx(l, b, j, vn) \mapsto vn^*$.

```
if $basic(vn)$ then
    $vn^* \leftarrow vn$
else
    $\chi_x^y, vn'_1, \ldots, vn'_p = h^{-1}(vn)$
    if $y = b \wedge x = j - l$ then
        // Subterm found, apply $\sqcup$ approximation
        $vn^* \leftarrow create(\sqcup, vn'_1, \ldots, vn'_p)$
    else
        for all $vn'_i \in vn'_1, \ldots, vn'_p$ do
            $vn^*_i \leftarrow loop\_approx(l, b, j, vn'_i)$
        end for
        $vn^* \leftarrow create(\chi_x^y, vn^*_1, \ldots, vn^*_p)$
    end if
end if
return $vn^*$
```

```
$vn' \leftarrow current\_analysis(node)$ // read the old analysis value
if $vn' \neq vn$ then
    $current\_analysis(node) \leftarrow vn$ // update to the new analysis value
    $count(vn) \leftarrow count(vn) + 1$
    $maybe\_free(vn')$
end if
```

The function *maybe_free*, detailed in Algorithm 8, is straight forward: it decreases the reference count for the argument value number vn', checks if there are any references left and, if not, it frees the corresponding χ-term n, removes its footprint from the hash-maps, and, if n is not basic, recursively calls itself to the sub-terms' value numbers.

Finally, *free*(n) is implemented using free-lists. Assume n is a basic χ-term representing a context-insensitive value, e.g., $\bot$, $\top$, constants or elements a of a power set lattice A. All context-insensitive values of the same analysis lattice require the same amount of memory, e.g., a symbolic constant or a bit vector. Hence, we can maintain separate free-lists for each context-insensitive analysis lattice.

Assume n is a χ-terms representing a context-sensitive value. Each node in this term captures its constant-size χ- or operation function symbol, and p value numbers vn_i referring to its children. While p can be different for different ϕ or operation SSA nodes, we can maintain separate free-lists for each fixed p occurring in the SSA representation of the program under analysis.

Algorithm 8 *maybe_free*(vn).

```
$count(vn) \leftarrow count(vn) - 1$
$n \leftarrow h^{-1}(vn)$
**if** $count(vn) = 0$ **then**
    $h(n) \leftarrow null$
    $h^{-1}(vn) \leftarrow null$
    **if** $n \neq a \in A$ **then**
        $//n = [x_j^b, vn_1, \ldots, vn_p]$ is a non-basic $\chi$ node
        **for all** $vn_i \in vn_1, \ldots, vn_p$ **do**
            $maybe\_free(vn_i)$
        **end for**
    **end if**
    $free(n)$
**end if**
```

6 Relation to Previous Work

Global Value Numbering (GVN) is an approach to capture semantically equivalent values, (sub-) expressions, and assignments used, e.g., to propagate constants, to simplify expressions, and to eliminate redundancy, with the goals of optimizing the internal representation of a computer program [2]. So-called value graphs represent the symbolic executions of statements and expressions in a program. Splitting the value graphs into congruent partitions allows each partition to be replaced with the same program representation.

There are similarities between our χ-terms (represented with directed acyclic graphs) and the (directed, cyclic) value graphs. Both representations are based on an SSA representation of a program and use a graph to model how control-flow decisions affect the analysis values.

However, our main goal is to save representation space during analysis: χ-terms are therefore reduced to their normal form online during their construction while analyzing the value of each statement and expression. Instead of trying to conservatively proof the equivalence of (sub-) expressions, GVN uses an optimistic fixed-point iteration approach. Value graphs must be constructed first before partitioning can reduce them, which does not save analysis space. Moreover, the optimistic approach makes it difficult to find algebraic identities, e.g., that $a = b + 1$, $c = a - 1$ implies $b == c$, while our χ-term normalization naturally exploits algebraic identities.

Rütting et al. [17, 33] present an efficient approach to constant propagation using value graphs. Their approach is restricted to constant propagation whereas ours can be applied on any data-flow analysis problem. We use the Shannon expansion to push operators to the leaves, where the operation can be evaluated. This is not the case in the value graphs, where the operator nodes remain scattered over the value graphs. Moreover, by using χ-terms and pushing the evaluation to the leaves, we automatically remove the redundancies.

Harris et al. [11] use Satisfiability Modulo Theory (SMT) to create a path-sensitive analysis to verify safety properties of C programs. Their approach, called Satisfiability Module Path Programs, enumerates the existing paths in the program by using the Satisfiability Theory (SAT) formulas given by the control-flow in the program. Their approach is more precise than ours but does not scale to larger programs.[2]

Heinze et al. [13] take a similar approach. They apply data-flow analysis on an SSA representation of the program to derive variable path predicates for each SSA variable in the program. The path predicate contains detailed information (predicates) about every control decision that might influence a given variable value. These variable predicates can later be fed to an SMT solver to verify certain program properties.

Our χ-terms is an abstraction of the path-sensitive approaches used in Rütting et al., Harris et al. and Heinze et al. We only keep track of the last k contexts that might influence a given variable value, and we disregard detailed information under which control-flow predicates these contexts get active. Thus, we trade precision for performance allowing us to handle much larger programs.

In general, higher precision can be reached by using context-sensitive analysis at the cost of a larger memory consumption. This implies the need of data structures with efficient memory usage and operations that makes quick manipulations on these structures. Here the usage of Binary Decision Diagrams (BDD) offer such an approach, which was exploited before, especially, for Points-to Analysis:

- Zhu and Calman [42, 43] present an approach to points-to analysis that uses Symbolic Pointers. Blocks of memory are source (domain) and target (range) of references (pointers). Points-to relations are modelled as directed graph of such blocks. Each block has an id corresponding to a unique Boolean formula. An edge is represented as a pair of domain and range block ids, i.e., pairs of Boolean domain and range functions. These Boolean functions are captured as BDDs and updated during analysis using BDD operations. This saves memory and preserves update performance.
- Berndl et al. [4] use BDDs to minimize the representation of the points-to data. The points-to relations between variables and sets of abstract objects are represented by binary strings. For large programs, the number of such sets can be very large. The BDD approach reduces the memory of partially redundant points-to sets. An evaluation shows that the approach is beneficial for both execution time and memory consumption.
- Whaley and Lam [41] create a clone for different invocations of a method call (call paths). A context-insensitive analysis on the extended call graph representing all clones results in a context-sensitive analysis. The context-sensitive relations are captured using BDDs leading to an efficient memory usage.
- Based on benchmarks on different context-insensitive and context-sensitive analysis variants, Lhotak and Hendren [18] conclude that the best method for points-to analysis is object-sensitive [28]. The usage of BDDs in capturing the

[2]Their analyzes of programs with about 60KLOC take more than three and half hours.

analysis data allows to increase the size of the analyzed programs. The same authors also discuss the effect on precision and efficiency of context-sensitive points-to analyses [20]. It shows that the precision is application dependent and that the efficiency depends on the used analysis method. The analysis framework, PADDLE is based on object-sensitivity and uses a BDD representation for the context-sensitive information. The resulting reduction of memory usage opens up for a more sensitive analyses to get a better accuracy.

– Ball and Rajamani introduce Bebop [3], a path-sensitive inter-procedural data-flow analysis tool for C programs. It adds data-flow facts to each vertex in a control-flow graph allowing to rule out paths that are not feasible in the analysis. For memory efficiency, the set of facts are captured in BDDs.

All above papers show that the usage of BDD provides memory efficient approaches to capture context-sensitive points-to information. Our paper introduces a systematic approach to generalize context-insensitive to context-sensitive analyses using BDDs.

Kim et al. [16] state the importance of utilizing different dimensions of sensitivity in static program analysis to improve the analysis precision. They present a general framework to capture many different dimensions of sensitivity, such as, context-, flow-, trace-sensitivity, that encompass a large variety of well-known analyses in all these dimensions and combination thereof. Using abstract interpretation, they show that each context-sensitive analysis instance of their framework is sound. Furthermore, they point out the importance of using a “sparse representation” based on “dictionaries” to handle the analysis data without providing any further implementation details. We focus on a data structure that have a sparse representation, and therefore suggest χ-terms as an efficient and flexible data structure to handle analysis information from any analysis generated by this framework.

Adding context-sensitivity to a context-insensitive analysis increases the precision of the results, but makes it more expensive in space and time. Therefore, it is a problem to select the variant and the depth of context-sensitivity. The ideal is a variant with positive effect on the result precision, but without using too much memory and time. In the following discussion, we call an approach *selective* if it selects among different context-sensitive analysis variants and *adaptive* if it selects the depth k of context-sensitivity:

– Kastrinis et al. [15] describe a selective approach creating an extended points-to analysis by combining the two different context-sensitive approaches (call-site- and object-sensitivity). The selection is guided by the available information at different analysis points. Such information includes knowledge about the type of calls and the type of the program. The evaluation shows that a combined approach (using additional depth of information) is successful concerning precision without introducing high performance and memory costs.
– Li et al. [22] present the so-called SCALAR framework implementing a selective approach, where different context-sensitive methods are compared and selected based on an estimation of the scalability risk. Pre-analysis using a first context-insensitive approach gives an approximation of the needed resources (total scalability threshold). This guides the automatic selection of the most suitable method amongst the available object- or type-sensitive methods in a second

context-sensitive analysis. This approach meets scalability needs reducing the risk of timeouts.

- For better scalability allowing the analysis of larger programs and/or a better precision, an adaptive approach for context-sensitive analysis can be used. Smaragdakis et al. [35] suggest a two-step approach including a context-insensitive analysis to find procedures that can be analyzed with a context-sensitive method to increase precision without drawbacks on the performance (time and space). The first step selects methods or objects that should be analyzed a second time based on a mix of costs, e.g., the size of points-to sets or the maximum size of fields' points-to sets relative to the number of fields. The idea is to avoid the methods or objects with high cost in the context-sensitive analysis phase. The evaluation shows that the approach gives good results in precision and scalability in the comparison with object-sensitivity, call-site-sensitivity, and type-sensitivity.
- Lee et al. [30] suggest the selection of a reasonable depth of context-sensitivity for k-call-string sensitivity. Their adaptive approach makes an impact estimation of the use of k-call-string-sensitivity balancing the expected precision and the analysis costs. The first step is a pre-analysis for finding the least value of k, fulfilling the context demand of context-sensitivity depending on the analysis queries. In a second step, each procedure is analyzed with the recommended depth k of context-sensitivity. This approach has been evaluated against a context-insensitive approach using ten software packages written in C. The result shows an improvement in precision.
- Jeong et al. [14] present a data-driven approach. Using machine learning on a large code base, their adaptive approach can learn the heuristic rules for selecting the depth of contexts. This learned heuristic is then used in the analysis of new programs. The results of analyzing new programs show that the approach is a fast alternative in comparison to the state-of-the-art points-to analyses.
- Li et al. [21] describe an adaptive approach called ZIPPER that strives to reduce the imprecision in points-to analysis by selecting methods that should use context-sensitive analysis. This selection relies on previously constructed Precision Flow Graphs (PFG) that are based on Object Flow Graphs [38]. Using the PFG of a class and based on the value flow of each method, the approach selects which methods to analyze in a context-(in)sensitive way. The results show that the precision is similar to 2-object-sensitive pointer analysis [28].

Thiessen and Lhotak [37] present another pointer analysis variant related to our k approximation. It uses transformation strings instead of the traditional context-strings. The idea combines the context-free language (CFL) reachability formulation of points-to analysis [31, 36] with k-object-sensitivity [28]. In CFL-reachability, abstract points-to relations are represented by paths (strings of node labels) in the call graph. Such a path relates sub-paths at the caller and callee sites, i.e., caller and callee contexts, referred to as a context transformation. Points-to analysis is formulated as an algebraic structure of context transformations. This abstraction can then be k-limited. The evaluation shows that this string abstraction instead of traditional context strings has a positive effect in most cases on the analysis time and on the overall precision.

The present paper formalizes the informal descriptions of χ-terms from our conference paper [40]. The paper is inspired by the ideas first presented, hinted, and implicitly assumed by Martin Trapp in his dissertation [39] (published in German). In our presentation of χ-terms, we have refined many of the notations that he introduced. We have also been able to prove many of the statements that he presented and implicitly assumed. The unique contributions in this paper are:

1. We proved that a sound context-insensitive program analysis with the partial order relation $\sqsubseteq$ has a corresponding sound context-sensitive program analysis using the partial order relation $\sqsubseteq_\chi$.
2. We proved that a $\sqcup$-approximation of a χ-term is always conservative. This means that we in any phase of the analysis can widen a χ-term by its $\sqcup$-approximation. This saves memory and still guarantees the soundness of the approach.
3. We have presented two different parameterized approximations (k-approximation and l-loop-approximation) and proved that any analysis based on these approximations are sound and guaranteed to reach a fixed point.
4. We give algorithms implementing an efficient construction of redundancy-free DAG-based χ-terms. The DAG representation and aggressive reuse of already existing χ-(sub-)terms give a very memory efficient representation of context-sensitive information. We also show how to manage the (theoretically) exponential memory consumption, using both widening operations and garbage collection.

Finally, this paper is a complement to our paper *Memory Efficient Context-Sensitive Program Analysis* [12] that evaluates the memory efficiency by comparing the memory foot-prints of χ-terms with four other data structures. The experiments use context-sensitive points-to analysis information taken from ten different Java benchmark programs. The results show that χ-terms are indeed memory efficient. They use on average only 13% of the memory used by the second best approach. Hence, [12] provides the experimental evidence, this paper presents χ-terms as a general framework for memory efficient context-sensitive program analyses.

7 Summary

Static program analysis is an important part of both optimizing compilers and software engineering tools for program verification and model checking. Such analyses can be context-sensitive or -insensitive, i.e., an analysis may or may not distinguish different analysis results for different execution paths. Context-sensitive analyses are, in general, more precise than their context-insensitive counterparts but also more expensive in terms of time and memory consumption.

This paper presents χ-terms as a means to capture context-sensitive analysis values for programs represented as SSA-graphs. Each meet point of execution paths in the program, i.e., each ϕ-node, is mapped to a χ-term whose children represent the alternative analysis values of these paths. The χ-terms are represented by graphs without any redundancy, which generalizes the idea behind OBDDs [6, 7].

For languages with conditional execution, the number of different contexts grows, in general, exponentially with the program size. Adding run-time iterations lead, in general, to countably (infinitely) many contexts. To handle this path-explosion problem, we introduce *k-approximation* and *l-loop-approximation* that limit the size of the context-sensitive information. We prove that each context-insensitive data-flow analysis has a corresponding k, l-approximated context-sensitive analysis, and we also prove that these k, l-approximated χ-terms form a partial ordered relation with a finite depth, thus, guaranteeing every data-flow analysis to reach a fixed point if their insensitive counterpart did.

The paper also give algorithms for how to implement redundancy-free, DAG-based, χ-terms. The DAG representation and aggressive reuse of already existing χ-(sub-)terms give a very memory efficient representation of context-sensitive information.

Finally, context-sensitive program analysis often comes with memory problems. We propose χ-terms as presented in this paper as a solution to this problem. The theory presented here is supported by experiments in [12] showing the memory efficiency of using χ-terms.

Funding Open access funding provided by Linnaeus University.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **27**(6), 509–516 (1978)
2. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '88. ACM (1988)
3. Ball, T., Rajamani, S.K.: Bebop: a path-sensitive interprocedural dataflow engine. In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01, pp. 97–103. ACM, New York. https://doi.org/10.1145/379605.379690 (2001)
4. Berndl, M., Lhotak, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: Proceedings of the Conference on Programmimg Language Design and Implementation (PLDI'03), pp. 103–114 (2003)
5. Boonstoppel, P., Cadar, C., Engler, D.: RWset: attacking path explosion in constraint-based test generation. In: 14th International Conference, TACAS 2008, pp. 351–366. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-78800-3_27 (2008)
6. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
7. Bryant, R.E.: Symbolic boolean manipulation with ordered binary decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)

8. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013). https://doi.org/10.1145/2408776.2408795
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximations of fixed points. In: Conference Record of the Fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, pp. 238–252 (1977)
10. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
11. Harris, W.R., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: Proceedings of the Conference on Principles of Programming Languages (POPL '10) (2010)
12. Hedenborg, M., Lundberg, J., Löwe, W.: Memory efficient context-sensitive program analysis. Elsevier J. Syst. Softw. **177**. https://doi.org/10.1016/j.jss.2021.110952 (2021)
13. Heinze, T.S., Amme, W.: Sparse analysis of variable path predicates based upon SSA-form. In: 7th International Symposium, ISoLA 2016, pp. 227–242. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-47166-2_16 (2016)
14. Jeong, S., Jeon, M., Cha, S., Oh, H.: Data-driven context-sensitivity for points-to analysis. Proc. ACM Program Lang. **1**(OOPSLA), 100:1–100:28 (2017). https://doi.org/10.1145/3133924
15. Kastrinis, G., Smaragdakis, Y.: Hybrid context-sensitivity for points-to analysis. SIGPLAN Not. **48**(6), 423–434 (2013). https://doi.org/10.1145/2499370.2462191
16. Kim, S.W., Rival, X., Ryu, S.: A theoretical foundation of sensitivity in an abstract interpretation framework. ACM Trans. Program. Lang. Syst. **40**(3), 13:1–13:44 (2018). https://doi.org/10.1145/3230624
17. Knoop, J., Rüthing, O.: Constant propagation on the value graph: simple constants and beyond. In: Watt, D. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 1781, pp. 94–110. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-46423-9_7 (2000)
18. Lhoták, O., Hendren, L.: Context-sensitive points-to analysis: is it worth it? In: Proceedings of the 15th International Conference on Compiler Construction, CC'06, pp. 47–64. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11688839_5 (2006)
19. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Methodol. **18**(1), 1–53 (2008). https://doi.org/10.1145/1391984.1391987
20. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. ACM Trans. Softw. Eng. Methodol. **18**(1), 3:1–3:53 (2008). https://doi.org/10.1145/1391984.1391987
21. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: Precision-guided context sensitivity for pointer analysis. Proc. ACM Program. Lang. **2**(OOPSLA), 141:1–141:29 (2018). https://doi.org/10.1145/3276511
22. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: Scalability-first pointer analysis with self-tuning context-sensitivity. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pp. 129–140. ACM, New York. https://doi.org/10.1145/3236024.3236041 (2018)
23. Lundberg, J.: Fast and precise points-to analysis. Ph.D. thesis, Linnaeus University (2014)
24. Lundberg, J., Gutzmann, T., Edvinsson, M., Löwe, W.: Fast and precise points-to analysis. J. Inf. Softw. Technol. **51**(10), 1428–1439 (2009)
25. Lundberg, J., Löwe, W.: Points-to analysis: a fine-grained evaluation. J. Univers. Comput. Sci. **18**(20), 2851–2878 (2013)
26. Marlowe, T., Ryder, B.: Properties of data flow frameworks: a unified model. Acta Inform. **28**, 121–163 (1990)
27. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. **14**(1), 1–41 (2005)
28. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. **14**(1), 1–41 (2005). https://doi.org/10.1145/1044834.1044835
29. Muchnick, S.S.: Advanced Compiler Design Implementation. Morgan Kaufmann Publishers, San Francisco (1997)
30. Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. SIGPLAN Not. **49**(6), 475–484 (2014). https://doi.org/10.1145/2666356.2594318

31. Reps, T.: Undecidability of context-sensitive data-dependence analysis. ACM Trans. Program. Lang. Syst. **22**(1), 162–186 (2000). https://doi.org/10.1145/345099.345137
32. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. **29**(5). https://doi.org/10.1145/1275497.1275501 (2007)
33. Rüthing, O., Knoop, J., Steffen, B.: Detecting equalities of variables: combining efficiency with precision. In: Cortesi, A., Filé, G. (eds.) Static Analysis, Lecture Notes in Computer Science, vol. 1694, pp. 232–247. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-48294-6_15 (1999)
34. Shivers, O.: Control-flow analysis of higher-order languages. Tech. rep., PhD thesis, Carnegie-Mellon University, CMU-CS-91-145 (1991)
35. Smaragdakis, Y., Kastrinis, G., Balatsouras, G.: Introspective analysis: context-sensitivity, across the board. SIGPLAN Not. **49**(6), 485–495 (2014). https://doi.org/10.1145/2666356.2594320
36. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for java. SIGPLAN Not. **41**(6), 387–400 (2006). https://doi.org/10.1145/1133255.1134027
37. Thiessen, R., Lhoták, O.: Context transformations for pointer analysis. SIGPLAN Not. **52**(6), 263–277 (2017). https://doi.org/10.1145/3140587.3062359
38. Tonella, P.: Reverse engineering of object oriented code. In: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, pp. 724–725. ACM, New York. https://doi.org/10.1145/1062455.1062637 (2005)
39. Trapp, M.: Optimerung Objektorientierter Programme. Ph.D. thesis, Universität Karlsruhe (1999)
40. Trapp, M., Hedenborg, M., Lundberg, J., Löwe, W.: Capturing and manipulating context-sensitive program information. Software Engineering Workshops 2015 **1337**, 154–163 (2015). http://ceur-ws.org/Vol-1337/
41. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'04) (2004)
42. Zhu, J.: Symbolic pointer analysis. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, ICCAD '02, pp. 150–157. ACM, New York. https://doi.org/10.1145/774572.774594 (2002)
43. Zhu, J., Calman, S.: Symbolic pointer analysis revisited. SIGPLAN Not. **39**(6), 145–157 (2004). https://doi.org/10.1145/996893.996860