Approximating Context-Sensitive Program Information

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

[1] Linnaeus University, Software Technology Group,
{Mathias.Hedenborg|Jonas.Lundberg|Welf.Lowe}@lnu.se
[2] Senacor Technologies AG, Martin.Trapp@senacor.com

Abstract. Static program analysis is in general more precise if it is sensitive to execution contexts (execution paths). In this paper we propose χ-terms as a mean to capture and manipulate context-sensitive program information in a data-flow analysis. We introduce *finite k-approximation* and *loop approximation* that limit the size of the context-sensitive information. These approximated χ-terms form a lattice with a finite depth, thus guaranteeing every data-flow analysis to reach a fixed point.

1 Introduction

Static program analysis is an important part of optimizing compilers and software engineering tools. These analyses predict properties of any execution of a given program, referred to as *program information*, by abstracting from its concrete execution semantics and its potential input values. Analyses can be context-sensitive or -insensitive, i.e., an analysis may or may not distinguish program information for different execution paths, i.e. for different *contexts*, e.g., the call contexts of a method. Context-sensitive analyses are, in general, more precise than their context-insensitive counterparts but also more expensive in terms of time and memory consumption.

In an iterative program, there are countably (infinitely) many contexts. Hence, merging the program information of some contexts is needed for the analysis to terminate. This, however, makes the analysis less context-sensitive, hence, less precise.

In Trapp et al. [THLL15], we focussed on capturing context-sensitive analysis information, i.e. contexts and program information for each program point, in a memory efficient way. In other words, we strived to delay merging the program information of different context for keeping analysis precision high. In the present paper, we discuss how to handle the approximations that sacrifies precision for memory. We propose two solutions: *finite k-approximation* and *loop approximation*, and we prove that any context-insensitive data-flow analysis problem have a *k-approximated* context-sensitive counterpart that is guaranteed to reach a fixed point.

The remainder of the paper is structured as follows: Section 2 summarise the notions of χ-terms and some of its fundamental operations. Sections 3 and 4 presents some theoretical results that will be useful later on. Section 5 presents two types of χ-term approximations and discusses how they relate to loop handling and analysis termination. Section 5.5 presents an approximative approach that is a normalized combination of these two χ-term approximations. Section 6 discusses related work, and section 7 concludes the paper and discusses future work.

2 Background

This section summarise the notions of χ-terms and some of its fundamental operations. It is basically a brief summary of [THLL15] included for the completness and understandability of this paper.

2.1 SSA Representation

We assume the analysis to be based on a *Static Single Assignment* (SSA) graph representation [CFR+91] of a program. Nodes in the SSA graph represent program points; special ϕ-nodes represent merge points of the execution paths, i.e., contexts. Here we distinguish the program information of incoming paths by creating a χ-term connected to sub-terms, each representing the program information analyzed for the respective incoming execution path.

Figure 1 shows an example code with corresponding basic block and SSA-graph representations.

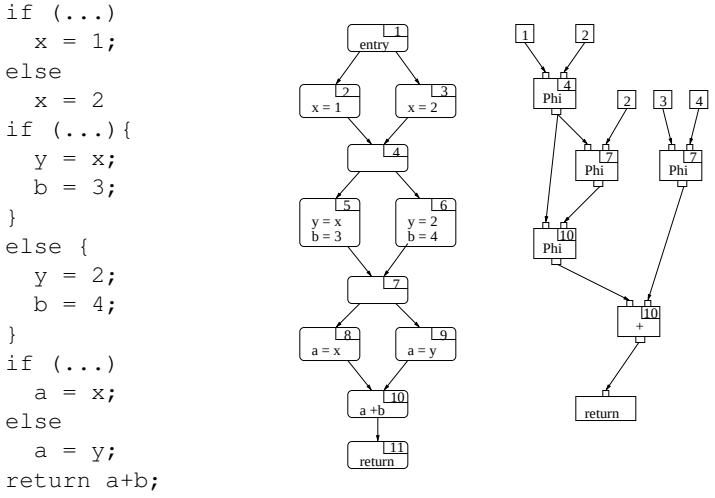


Fig. 1. A source code example with corresponding basic block and SSA graph structures.

In the figure the source code is transfered into numbered basic blocks (middle), and based on this a ϕ-node based SSA-graph have been generated (right). The ϕ-nodes will there be the merging point for different definitions of values for a given variable.

2.2 χ-terms

A χ-function is a representation of how different control-flow options affect the value of a variable. For example, we can write down the value of variable b in block 7 in Figure 1 using χ-functions as $b = \chi^7(3, 4)$. Interpretation: variable b has the value 3 if

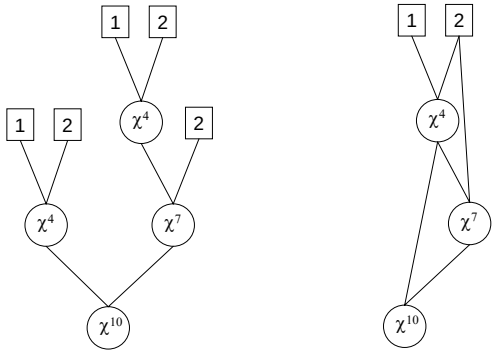


Fig. 2. Tree view of $\chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2))$ and its graph representation.

it was reached from the first predecessor to block 7 in the control-flow graph, and the value 4 if it was reached from the second predecessor block. That is, a value expressed using χ-functions (a so-called χ-term) does not only contain all possible values, it also contains which control-flow options that generated each of these values.

The construction of the χ-term values and the numbering of the χ-functions is a part of a context-sensitive analysis. Every ϕ-node in an SSA graph represents a join point for several possible definitions of a single variable, say x. When the analysis reaches a block b containing a ϕ-node for x it "asks" all the predecessor blocks to give their definition of x and constructs a new χ-term $\chi^b(x_1, \ldots, x_n)$ where x_i is the χ-term value for x given by the i:th predecessor. If the i:th predecessor block does not define x by itself, it "asks" its predecessor for the value. This process continues recursively until each predecessor has presented a χ-term value for x. The process will terminate if any use of a value also has a corresponding definition.

Iteration in the code will generate loops in the control-flow and this need to be handled in the χ-term. Loop handling will be elaborated in section 5.4.

In summary, a χ-term is a composition of χ-functions and analysis values $a, b, \ldots \in V$. Each program p has a (possibly infinite) set of χ-functions $\mathcal{X}(p)$ and each χ-function $\chi^b_j \in \mathcal{X}(p)$ is identified by a pair (b, j) where the *block number* b indicates in what basic block its generating ϕ-node is contained, and the *iteration index* j indicates on what analysis iteration over block b the χ-function was generated.

Two χ-terms $\chi^b_i(x_1, \ldots, x_n)$ and $\chi^b_j(y_1, \ldots, y_n)$ from the same block b have the same *switching behavior* if they have the same iteration index (i.e. $i = j$). That is, for any execution of the program it holds that

$$\text{branch } x_k \text{ is selected } \Leftrightarrow \text{ branch } y_k \text{ is selected}$$

Thus, the switching behavior of a χ-function is determined by a pair (b, i) where b is the block number and i is the iteration index. In what follows, we will often skip the iteration index to simplify the notations. In these cases we assume that all involved χ-function have the same iteration index.

```
public int m(int a, int b) {
    if ( ... ) {
      a = a+1;
      b = b-1;
    }
    else {
      a = a-1;
      b = b+1;
    }
    return a + b;
}
```

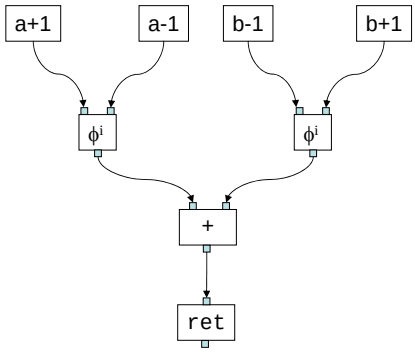


Fig. 3. An method with the corresponding SSA graph of the basic block containing the `return`-statement.

2.3 Tree and Graph Representation of χ-terms

Every χ- term can be naturally viewed as a tree. This is illustrated in Figure 2 (left) where we show the tree representation of the χ-term $\chi^{10}(\chi^4(1,2),\chi^7(\chi^4(1,2),2))$. Each edge represents a particular control flow option in this view 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 come into play. The tree representation of a χ-term is easy to understand and important from theoretical point of view: many of the notations to be presented are easiest to understand in terms of operations on the tree representation.

To actually represent each χ-term as a tree is in practice much to costly. A more compact graph representation can easily be found by reusing identical subtrees, cf. Figure 2 (right), thus avoiding redundancies. This is the approach we recommend for any χ-term implementation and it is similar to how OBDDs are handled in [Bry92], and how classifier information is handled in [DLL14].

2.4 Operations on χ-terms and Shannon Expansions

The code fragment on the left-hand side of Figure 3 assigns different values to variables a and b in the two different branches of the `if`-statement and then returns the sum $a+b$. Using χ-terms, we can express the values of a and b as $\chi(a+1,a-1)$ and $\chi(b-1,b+1)$ respectively, and the sum $a+b$ as

$$+(\chi(a+1,a-1),\chi(b-1,b+1)).$$

Furthermore, we know that any execution takes either the first or the second branch of the `if`-statement (but we do not know which). This observation leads us to the following rewriting:

$$\begin{aligned} +(\chi(a+1,a-1),\chi(b-1,b+1)) &= \chi(\text{ first branch }+,\ \text{second branch }+) \\ &= \chi(+(a+1,b-1),+(a-1,b+1)) \end{aligned}$$

That is, we can make use of the fact that both χ-terms have the same switching behavior and apply the $+$ operator on each of the two branches separately before we merge the result. This rewriting can be seen as that we are pushing the $+$ operator one step closer to the leaf values.

Finally, we are now in a position where we can apply $+$ on a set of leaf values. In this case $+$ is well defined and we can fall back on ordinary integer arithmetics. This manipulation, where we also use the redundancy rule $\chi(t,t) = t$, can symbolically be written as:

$$\begin{aligned}\chi(+(a+1,b-1),+(a-1,b+1)) &= \chi((a+1)+(b-1),(a-1)+(b+1)) \\ &= \chi(a+b,a+b) = a+b\end{aligned}$$

The result indicates that no matter what branch of the `if`-statement we use, we will always get the result $a+b$. This simple example illustrates one of the strengths of using χ-terms, we can by using a few simple rewrite-rules make use of having stored flow-path information and "compute" more precise results than would have been possible in a context-insensitive approach.

That we can rewrite the addition of two χ-terms as a χ-term over the addition of a and b for each individual branch is in this case quite obvious. This rewrite rule for χ-term expressions is called a *Shannon expansion* [3]. It does not change the information represented by that term and leads therefore to an *equivalent* ($\equiv$) term. It may, however, change the size needed for representing a χ-term. For example,

$$\begin{aligned}t_a &= \chi^{10}(\chi^4(1,2),\chi^7(\chi^4(1,2),2)) \\ &\equiv \chi^4(\chi^{10}(1,\chi^7(1,2)),\chi^{10}(2,\chi^7(2,2))) && \text{(expansion over } \chi^4) \\ &\equiv \chi^7(\chi^4(1,2),\chi^{10}(\chi^4(1,2),2)) && \text{(expansion over } \chi^7)\end{aligned}$$

The Shannon expansion is also used to define the result of *applying* an operation to χ-terms. For instance, assume $t_a = \chi^{10}(\chi^4(1,2),\chi^7(\chi^4(1,2),2))$ and $t_b = \chi^7(3,4)$, and assume further that $\texttt{apply}(+,i,j) = i+j$ for any two integers i and j. Then

$$\begin{aligned}\texttt{apply}(+,t_a,t_b) &= \texttt{apply}(+,\chi^{10}(\chi^4(1,2),\chi^7(\chi^4(1,2),2)),\chi^7(3,4)) \\ &= \chi^{10}(\texttt{apply}(+,\chi^4(1,2),\chi^7(3,4)), \\ &\qquad \texttt{apply}(+,\chi^7(\chi^4(1,2),2),\chi^7(3,4))) \\ &= \chi^{10}(\chi^7(\texttt{apply}(+,\chi^4(1,2),3),\texttt{apply}(+,\chi^4(1,2),4)), \\ &\qquad \chi^7(\texttt{apply}(+,\chi^4(1,2),3),\texttt{apply}(+,2,4))) \\ &= \chi^{10}(\chi^7(\chi^4(\texttt{apply}(+,1,3),\texttt{apply}(+,2,3)), \\ &\qquad\quad \chi^4(\texttt{apply}(+,1,4),\texttt{apply}(+,2,4))), \\ &\qquad \chi^7(\chi^4(\texttt{apply}(+,1,3),\texttt{apply}(+,2,3)),6)) \\ &= \chi^{10}(\chi^7(\chi^4(4,5),\chi^4(5,6)),\chi^7(\chi^4(4,5),6))\end{aligned}$$

[3] The word "Shannon expansion" is taken from the OBDD literature [Bry92] where a similar procedure is used to rewrite boolean functions represented as OBDDs.

The first step above, expansion over χ^{10}, can be seen as if we push the operator + one step towards the leaf values by computing + for each one of the branches of χ^{10} individually, and then merge these values using χ^{10}. This process can be repeated until we reach the leaf values where the context-insensitive version of + can be applied.

This idea generalizes to any operator implementing a context-insensitive transfer function. For each context-insensitive operator $\tau : A \times B \times \ldots \times N \mapsto V$ their is a corresponding χ-induced operator $\widetilde{\tau} : X_A \times X_B \times \ldots \times X_N \mapsto X_V$ defined as $\widetilde{\tau}(t_a, \ldots, t_n) = \texttt{apply}(\tau, t_a, \ldots, t_n)$.

3 Structural Induction on χ-terms

Many basic properties such as commutativity and associativity of a context-insensitive operator τ are directly mapped to the χ-induced counterpart $\widetilde{\tau}$. In what follows, we will often want to proof statements like: Assume that statement $S(v_1, \ldots, v_n)$ is true for all abstract values $v_i \in V$. Then $\widetilde{S}(t_1, \ldots, t_n)$ (the χ-induced counterpart to S) is true for all χ-terms $t_i \in X_V$. A typical example of such a statement is:

Theorem 1. *Let $\tau : V \times V \mapsto V$ be a* commutative *operation and let $\widetilde{\tau} : X_V \times X_V \mapsto X_V$ be the corresponding χ-induced operator. It then holds that*

$$\widetilde{\tau}(t_a, t_b) = \widetilde{\tau}(t_b, t_a) \qquad \forall t_a, t_b \in X_V.$$

This type of statement can in many cases be proved by something called *structural induction* on χ-terms. This type of induction is similar in spirit to ordinary structural induction as presented in most text books (e.g. [AU95]). That is, we do an induction on the depth of χ-terms. The following is an outline for such induction proofs:

1. *The Base Case*: We show that $\widetilde{S}(v_a, \ldots, v_n)$ is true when $v_a, \ldots, v_n \in V \subseteq X_V$. This step can in most cases be done by applying the property $S(v_a, \ldots, v_n)$.
2. *The Inductive Hypothesis*: Let

   $$t_a = \chi^a(a_1, \ldots, a_p),\ \ldots, t_n = \chi^n(n_1,\ \ldots, n_q)$$

   and assume that

   $$\widetilde{S}(a_i, \ldots, n_j) \text{ is true } \forall (a_i, \ldots, n_j) \in children(t_a) \times \ldots \times children(t_n).$$

3. *The Inductive Step*: Prove that $\widetilde{S}(t_a, \ldots, t_n)$ is true using the assumptions in the inductive hypothesis and the definition of operations on χ-terms including Shannon expansion.

If we do so, then we can conclude that $\widetilde{S}(t_a, \ldots, t_n)$ is true for all χ-terms $t_a, \ldots, t_n \in X_V$. In order to exemplify this type of proof by structural induction we prove Theorem 1.

1. *The Base Case*: We are given that τ is commutative and we know that for every χ-induced operator it holds that

   $$\widetilde{\tau}(v_a, \ldots, v_n) = \tau(v_a, \ldots, v_n), \forall v_i \in V \subseteq X_V.$$

Thus, let $t_a = v_a, t_b = v_b$

$$\widetilde{\tau}(t_a, t_b) = \tau(v_a, v_b) = \tau(v_b, v_a) = \widetilde{\tau}(t_b, t_a), \qquad \forall v_a, v_b \in V \subseteq X_V.$$

This completes the base case.

2. *The Inductive Hypothesis*: Let $t_a = \chi^a(a_1, \ldots, a_p)$ and $t_b = \chi^b(b_1, \ldots, b_q)$ and assume that

$$\widetilde{\tau}(a_i, b_j) = \widetilde{\tau}(b_j, a_i) \qquad \forall (a_i, b_j) \in children(t_a) \times children(t_b).$$

3. *The Inductive Step*: Prove that $\widetilde{\tau}(t_a, t_b) = \widetilde{\tau}(t_b, t_a)$ is true using the assumptions in the inductive hypothesis. We start with the left-hand side:

$$\begin{aligned}
\widetilde{\tau}(t_a, t_b) &= \widetilde{\tau}(\chi^a(a_1, \ldots, a_p), \chi^b(b_1, \ldots, b_q)) \\
&= \chi^a(\widetilde{\tau}(a_1, \chi^b(b_1, \ldots, b_q)), \ldots, \widetilde{\tau}(a_p, \chi^b(b_1, \ldots, b_q)) \\
&= \chi^b(\chi^a(\widetilde{\tau}(a_1, b_1)), \widetilde{\tau}(a_2, b_1)), \ldots, \widetilde{\tau}(a_p, b_1)), \\
&\qquad \ldots, \\
&\qquad \chi^a(\widetilde{\tau}(a_1, b_q)), \widetilde{\tau}(a_2, b_q)), \ldots, \widetilde{\tau}(a_p, b_q))) \\
&= \chi^b(\chi^a(\widetilde{\tau}(b_1, a_1)), \widetilde{\tau}(b_1, a_2)), \ldots, \widetilde{\tau}(b_1, a_p)), \\
&\qquad \ldots, \\
&\qquad \chi^a(\widetilde{\tau}(b_q, a_1)), \widetilde{\tau}(b_q, a_2)), \ldots, \widetilde{\tau}(b_q, a_p)))
\end{aligned}$$

The first two rewritings are Shannon expansions in χ^a and χ^b, respectively. In the final rewriting we have used the inductive hypothesis $\widetilde{\tau}(a_i, b_j) = \widetilde{\tau}(b_j, a_i)$. Attacking the right-hand side with a similar approach (but no use of the inductive hypothesis) we find that:

$$\begin{aligned}
\widetilde{\tau}(t_b, t_a) &= \widetilde{\tau}(\chi^b(b_1, \ldots, b_q), \chi^a(a_1, \ldots, a_p)) \\
&= \chi^a(\widetilde{\tau}(\chi^b(b_1, \ldots, b_q), a_1), \ldots, \widetilde{\tau}(\chi^b(b_1, \ldots, b_q), a_p) \\
&= \chi^b(\chi^a(\widetilde{\tau}(b_1, a_1)), \widetilde{\tau}(b_1, a_2)), \ldots, \widetilde{\tau}(b_1, a_p)), \\
&\qquad \ldots, \\
&\qquad \chi^a(\widetilde{\tau}(b_q, a_1)), \widetilde{\tau}(b_q, a_2)), \ldots, \widetilde{\tau}(b_q, a_p)))
\end{aligned}$$

Thus, $\widetilde{\tau}(t_a, t_b) = \widetilde{\tau}(t_b, t_a)$ and we have completed the proof of Theorem 1.

In what follows, we will not show any proofs that are straight forward applications of this type of structural induction since they pretty much look the same. For example, it is straight forward to show that properties like associativity, distributivity, and closure are conserved for any χ-induced operator $\widetilde{\tau} : X_V \times X_V \mapsto X_V$ by using a similar approach as in the previous proof.

4 The χ-term lattice

In Section 2, we learned that every context-insensitive operator τ has a χ-induced counterpart $\widetilde{\tau}$. This also holds for the abstract value lattice operators $\sqcap$ and $\sqcup$. The interesting point here is that these operators can be used to define a χ-induced χ-term lattice $\widetilde{\mathcal{L}}_V$ over the elements in X_V, the set of all χ-terms. This χ-induced lattice is important since it will be the value lattice in a χ-term based context-insensitive analysis.

Theorem 2. *For each lattice of abstract values $\mathcal{L}_V = \{V, \sqcap, \sqcup, \top, \bot\}$ there is a corresponding χ-induced lattice $\widetilde{\mathcal{L}}_V = \{X_V, \widetilde{\sqcap}, \widetilde{\sqcup}, \top, \bot\}$ where $\widetilde{\sqcap}$ and $\widetilde{\sqcup}$ are the χ-induced versions of $\sqcap$ and $\sqcup$.*

Showing that $\widetilde{\sqcap}$ and $\widetilde{\sqcup}$ have the properties *commutative*, *associative*, and *closure*, is a straight forward exercise in structural induction as presented in Section 3. The same holds for $t \,\widetilde{\sqcap}\, \bot = \bot$, $t \,\widetilde{\sqcup}\, \top = \top$ for all $t \in X_V$. We will not show it here.

Furthermore, we can use the χ-induced lattice operator $\widetilde{\sqcup}$ to define a partial ordering relation between χ-terms. The Connecting Theorem in [DP02] implies that

Theorem 3. *Let $\widetilde{\mathcal{L}}_V = \{X_V, \widetilde{\sqcap}, \widetilde{\sqcup}, \top, \bot\}$ be a χ-induced lattice for some abstract values V and let $\widetilde{\sqsubseteq} : \widetilde{\mathcal{L}}_V \times \widetilde{\mathcal{L}}_V \mapsto \{\texttt{true}, \texttt{false}\}$ be an operator defined as:*

$$t_1 \,\widetilde{\sqsubseteq}\, t_2 \iff t_1 \,\widetilde{\sqcup}\, t_2 = t_2, \qquad \forall t_1, t_2 \in X_V.$$

Then $\widetilde{\mathcal{P}}_V = \{\widetilde{\sqsubseteq}, X_V\}$ is a (χ-induced) partial ordering over X_V.

Due to the iteration indicis the χ-induced lattice $\widetilde{\mathcal{L}}_V$ has an infinite height. Thus implying that a data-flow analysis based on this lattice will not be guaranted to terminate. Further approximations are needed (cf. Section 5).

Theorem 4. *Let $\tau : \mathcal{L}_A \mapsto \mathcal{L}_B$ be a monotone function and let $\widetilde{\tau} : \widetilde{\mathcal{L}}_A \mapsto \widetilde{\mathcal{L}}_B$ be the corresponding χ-induced operator. It then holds that*

$$t_1 \,\widetilde{\sqsubseteq}\, t_2 \;\Rightarrow\; \widetilde{\tau}(t_1) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(t_2), \qquad \forall t_1, t_2 \in \widetilde{\mathcal{L}}_A$$

Once again, the proof of this theorem is a straight forward exercise in structural induction as presented in Section 3. The only tricky part is the final induction step where we must show that $t_a \,\widetilde{\sqsubseteq}\, t_b \;\Rightarrow\; \widetilde{\tau}(t_a) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(t_b)$ for two arbitrary χ-terms $t_a = \chi^a(a_1, \ldots, a_p)$, $t_b = \chi^b(b_1, \ldots, b_q)$ given the induction hypothesis

$$a_i \,\widetilde{\sqsubseteq}\, b_j \;\Leftrightarrow\; \widetilde{\tau}(a_i) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(b_j), \qquad \forall (a_i, b_j) \in children(t_a) \times children(t_b).$$

This can be done in two steps:

1. Show that

   $$t_a \,\widetilde{\sqcup}\, t_b = t_b \;\Rightarrow\; a_i \,\widetilde{\sqcup}\, b_j = b_j \qquad \forall (a_i, b_j) \in children(t_a) \times children(t_b).$$

   which corresponds to $t_a \,\widetilde{\sqsubseteq}\, t_b \;\Rightarrow\; a_i \,\widetilde{\sqsubset}\, b_j, \forall (a_i, b_j)$. This can be done by comparing $t_a \,\widetilde{\sqcup}\, t_b$ with t_b after both has been Shannon expanded over both χ^a and χ^b.
2. Show that

   $$\begin{aligned} \widetilde{\tau}(a_i) \,\widetilde{\sqcup}\, \widetilde{\tau}(b_j) = \widetilde{\tau}(b_j), \forall (a_i, b_j) \in children(t_a) \times children(t_b) \\ \Rightarrow \widetilde{\tau}(t_a) \,\widetilde{\sqcup}\, \widetilde{\tau}(t_b) = \widetilde{\tau}(t_b). \end{aligned}$$

   which corresponds to $\widetilde{\tau}(a_i) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(b_j), \forall (a_i, b_j) \Rightarrow \widetilde{\tau}(t_a) \,\widetilde{\sqsubseteq}\, \widetilde{\tau}(t_b)$. This can be done by a Shannon expansion of $\widetilde{\tau}(t_a) \,\widetilde{\sqcup}\, \widetilde{\tau}(t_b)$ over both χ^a and χ^b followed by repeated use of the identities $\widetilde{\tau}(a_i) \,\widetilde{\sqcup}\, \widetilde{\tau}(b_j) = \widetilde{\tau}(b_j)$.

These two steps, together with induction hypothesis, prove the inductive step.

5 χ-term Approximations

In this section, we present two different approximations to the fully context-sensitive approach outlined above. We refer to these two approximations as the *finite k-approximation* and the *loop approximation*. In the end of this section (Section 5.4), we show how these two approximations can be used to handle loops. These two approximations will in Section 5.5 be merged to something we refer to as a k-approximated analysis. This type of analysis is parametrized by a single precision parameter k. However, we start by introducing the concept of $\widetilde{\sqcup}$-approximations.

5.1 $\widetilde{\sqcup}$-Approximations

The aim of this section is to show that we always can replace any subterm $\chi_j^b(t_1, \ldots, t_n)$ in a χ-term t with $\widetilde{\sqcup}(t_1, \ldots, t_n)$. Hence, given that we interpret the χ-terms as values in a data-flow analysis, we still maintain a conservative approach. We refer to this type of χ-term manipulations as $\widetilde{\sqcup}$-approximations.

Theorem 5. *For any χ-term $\chi(t_1, \ldots, t_n) \in \widetilde{\mathcal{L}}_V$ it holds that*

$$\chi(t_1, \ldots, t_n) \mathrel{\widetilde{\sqsubseteq}} \widetilde{\sqcup}(t_1, \ldots, t_n)$$

Proof: Using Theorem 3 as a definition for $\widetilde{\sqsubseteq}$, it is sufficient to verify that

$$\widetilde{\sqcup}(\chi(t_1, \ldots, t_n),\ \widetilde{\sqcup}(t_1, \ldots, t_n)) = \widetilde{\sqcup}(t_1, \ldots, t_n).$$

This can be done in a few steps starting with a Shannon expansion over χ and ending by applying the redundancy rule

$$\begin{aligned}\widetilde{\sqcup}(\chi(t_1, \ldots, t_n),\ \widetilde{\sqcup}(t_1, \ldots, t_n)) &= \chi(\widetilde{\sqcup}(t_1,\ \widetilde{\sqcup}(t_1, \ldots, t_n)), \ldots, \\ &\qquad \widetilde{\sqcup}(t_n,\ \widetilde{\sqcup}(t_1, \ldots, t_n))) \\ &= \chi(\widetilde{\sqcup}(t_1, \ldots, t_n), \ldots, \widetilde{\sqcup}(t_1, \ldots, t_n)) \\ &= \widetilde{\sqcup}(t_1, \ldots, t_n)\end{aligned}$$

Theorem 6. *Let $t_a = \chi_j^b(a_1, \ldots, a_n)$ and $t_b = \chi_j^b(b_1, \ldots, b_n)$ be two χ-terms with the same switching behavior where $a_i \mathrel{\widetilde{\sqsubseteq}} b_i, \forall i \in [1, n]$. It then holds that $t_a \mathrel{\widetilde{\sqsubseteq}} t_b$.*

Proof outline: Start with a Shannon expansion of $t_a \mathbin{\widetilde{\sqcup}} t_b$ over χ_j^b and use $a_i \mathbin{\widetilde{\sqcup}} b_i = b_i, \forall i \in [1, n]$ to verify that $t_a \mathbin{\widetilde{\sqcup}} t_b = t_b$.

Definition 1 (**$\widetilde{\sqcup}$-approximation).** *Let $t \in X_V$ be a χ-term and let $a = \chi_j^b(a_1, \ldots, a_n)$ be a subterm of t. The $\widetilde{\sqcup}$-approximation of t with respect to a, denoted t_a^*, is a new term where a in t is replaced by $\widetilde{\sqcup}(a_1, \ldots, a_n)$.*

This definition is easy to understand using the tree representation of t. It simply means that we have replaced the subtree rooted by the node $\chi_j^b(a_1, \ldots, a_n)$ with $\widetilde{\sqcup}(a_1, \ldots, a_n)$.

Theorem 7. *Let $t \in X_V$ be a χ-term and let a be a subterm of t then $t \mathrel{\widetilde{\sqsubseteq}} t_a^*$.*

Proof outline: This follows directly from the Theorems 5 and 6. Theorem 5 says that the replaced subterm is less then (or equal to) the new subterm. Theorem 6 says that this property is propagated to the root.

Theorem 7 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 $\chi(t_1, \ldots, t_n)$ with $\widetilde{\sqcup}(t_1, \ldots, t_n)$ and still maintain a conservative approach.

5.2 The Finite k-Approximation

The construction of new χ-terms is a part of the context-sensitive analysis. When the analysis reaches a ϕ-node in block b for a variable x, it constructs a new χ-term by composing χ^b with all possible values for x. The newly constructed χ-term embodies all control-flow options that might influence the value of x at that point. The size of the χ-term representing x grows larger (without upper limit) as the analysis proceeds and more and more control-flow options influences the value of x. This represents a fully context-sensitive analysis where the effect of every control-flow option for every variable is kept at all times. In this section, we will present an approximation of the fully context-sensitive analysis where we only keep track of the last k control-flow options that might influence the value of a variable[4]. More "remote" control-flow options are merged using the ordinary context-insensitive merge operator $\sqcup$.

The finite 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 $t_{sub} = \chi_i^b(t_1, \ldots, t_n)$ in $subterms(t)$ that has $depth(t_{sub}, t) \geq k$ with $\sqcup(t_1, \ldots, t_n)$. The process starts in the leafs and proceeds towards the root node. The result is a new χ-term $t^{(k)}$ with $depth(t^{(k)}) \leq k$. The process is outlined in Algorithm 1 where we define a recursive function $\texttt{kApprox}(k, t)$ that transforms an input χ-term t into a k-approximated χ-term $t^{(k)}$.

Algorithm 1 $\texttt{kApprox}(k, t = \chi_j^b(t_1, \ldots t_n)) \mapsto t^{(k)}$

```
if k = 0 then
    t^(k) = collapse(t)
else
    for all t_i ∈ children(t) do
        t_i* = kApprox(k − 1, t_i)
    end for
    t^(k) = χ_j^b(t_1*, ..., t_n*)
end if
return t^(k)
```

The post-order traversal in `kApprox` guarantees that our approximation starts from the leafs and proceeds towards the root. Parts of the tree that has a depth greater than

[4] Other approaches to limit the size of the χ-terms are possible. We could, for example, limit the tree *size* (i.e. number of nodes) rather than the *depth*.

$$a = \chi^3(\chi^1(1,2), \chi^2(\chi^1(3,4), 2))$$
$$a^{(2)} = \chi^3(\chi^1(1,2), \chi^2(\{3,4\}, 2))$$
$$a^{(1)} = \chi^3(\{1,2\}, \{2,3,4\})$$

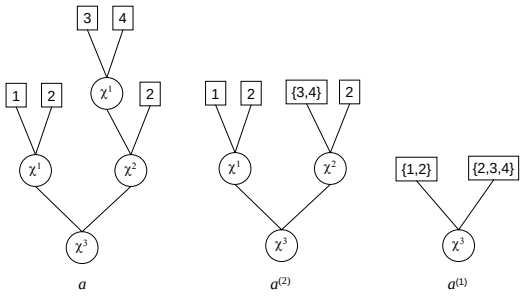


Fig. 4. Two different finite k approximations of the same χ-term a.

k is collapsed into leaf values by a process named `collapse` (see Algorithm 2). The process of merging leaf values proceeds until we have reached depth k of the input χ-term t. The result is a new χ-term $t^{(k)}$ 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.

Algorithm 2 `collapse`$(t) \mapsto v$

```
if t ∈ V then
    v = t
else
    let v = ⊥
    for all t_i ∈ children(t) do
        v = v ⊔ collapse(t_i)
    end for
end if
return v
```

Figure 4 shows the result of two different finite k approximations of the same χ-term a. On the left-hand side we have the result in print and on the right-hand side we have the same result depicted as an original tree and two trees where the depth have been reduced and the leaf values have been merged.

5.3 The Loop Approximation

According to Trapp at al. [THLL15] we know that the analysis of a loop will generate χ-terms like $x_n^b = \chi_n^b(\ldots \chi_{n-1}^b(\ldots)\ldots)$. That is, the newly created χ-term will have a subterm with the same block number and a lower iteration index. This pattern will probably occur over and over again since each loop iteration results in a new composition of χ^b with itself. This will result in χ-terms of infinite depth and a non-terminating

analysis if no measure is taken to stop the iterations. In this section we will show one approximation that is a first step in that process. Informally, a χ-term $t = \chi_i^b(t_1, ..., t_n)$ is loop-approximated if every subterm of t that has the same block number as t is replaced by its $\widetilde{\sqcup}$- approximation.

Definition 2 (Loop-approximated χ-term). *A χ-term t is* loop approximated *if*

$$t_{sub} \in subterms(t) \ \wedge \ block(t_{sub}) = block(t) \ \Rightarrow t \rightarrow t^*_{t_{sub}}$$

where $t^*_{t_{sub}}$ is the $\widetilde{\sqcup}$-approximation of t with respect to t_{sub}. An analysis where every newly created χ-term is immediately loop approximated is said to be a loop approximated analysis*.*

This 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 with their context-insensitive approximation. This approach is outlined in Algorithm 3 where we define a recursive function $\texttt{loopApprox}(t, b)$ that recursively visits all children before any merging takes place.

Algorithm 3 $\texttt{loopApprox}(t, b) \mapsto t^*$

```
for all t_i ∈ children(t) do {Visit all children}
    t*_i = loopApprox(t_i, b)
end for
if block(t) = b then
    let t* = apply(⊔, t*_1, ..., t*_n)
    return t*
else
    return χ^b(t*_1, ..., t*_n)
end if
```

The loop approximated analysis comes with a number of important observations:

1. Each χ-term will have a finite depth limited by the number of basic blocks that contain ϕ^b-nodes.
2. We can now drop the iteration index since only control-flow options from the last visit to any given block b will be recordered. Control-flow options from earlier visits have all been conservatively merged by $\widetilde{\sqcup}$- approximations. Thus, we will never have two χ-terms generated from the same block with different switching behavior.
3. If we drop the iteration index then the number of χ-functions at use will reduce to the number of basic blocks that contains a ϕ-node (a finite number).
4. A finite number of χ-functions and a finite depth of all χ-terms implies that we have a finite number of possible χ-terms. (Obvious if we think in terms of possible tree representations).

A consequence of the final observation is the following theorem:

Theorem 8. *The χ-induced lattice $\widetilde{\mathcal{L}}_V$ associated with a loop approximated analysis is a finite lattice with a finite height.*

One implication of this theorem is that every loop approximated analysis has a value lattice satisfying the ascending chain condition (see [DP02]) and that every analysis involving monotone transfer functions eventually will terminate (see [NNH99]). An example related to these properties is presented in the next section.

5.4 Analysis Loop Handling

The relation between the loop approximation and the loop handling is best illustrated with an example. In Figure 5, we show a hypothetical situation that illustrates a general case. On the left-hand side we have a piece of code that contains two variables `x` and `y` which values will be updated within the loop body. On the right-hand side we have the same situation depicted as an SSA graph. The two variable values entering the loop are represented as a tuple t_i and the updates within the loop body are represented by a mapping $f : \widetilde{\mathcal{L}}_T \mapsto \widetilde{\mathcal{L}}_T$. The loop approximated values generated by the ϕ-node in

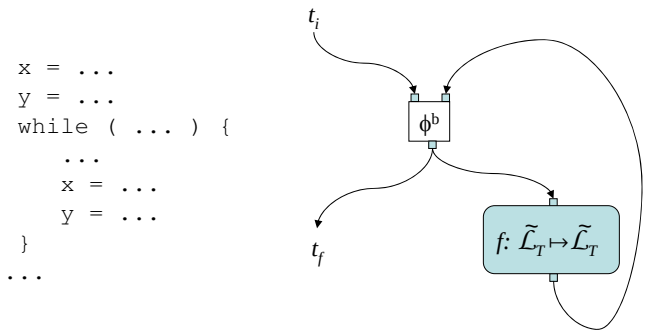


Fig. 5. A piece of source code and the corresponding graph. The mapping $f : \widetilde{\mathcal{L}}_T \mapsto \widetilde{\mathcal{L}}_T$ symbolizes the effect of the loop body on the variable tuple $[x, y]$.

block b can then be written as

$$
\begin{aligned}
t_0 &= \chi^b(t_i, \bot) \\
t_1 &= \chi^b(t_i, f(t_i)) \\
t_2 &= \chi^b(t_i, \widetilde{\sqcup}(f(t_i), f^2(t_i))) \\
&\dots \\
t_n &= \chi^b(t_i, \widetilde{\sqcup}(f(t_i), f^2(t_i), \dots, f^n(t_i)))
\end{aligned}
$$

where f^k is the composition of f with itself k times. Here t_i denotes the (loop approximated) value that t_f would get if we terminated the loop analysis after i iterations. We have derived these expressions by repeated use of $t_n = \chi^b(t_i, f(t_{n-1}))$ followed by

Shannon expansion of the "inner" χ^b, and a loop approximation. We have also assumed that $f(\bot) = \bot$. For example, t_1 was derived as

$$\begin{aligned} t_1 &= \chi^b(t_i, f(t_0)) = \chi^b(t_i, f(\chi^b(t_i, \bot))) \\ &= \chi^b(t_i, \chi^b(f(t_i), \bot)) \approx \chi^b(t_i, \widetilde{\sqcup}(f(t_i), \bot))) \\ &= \chi^b(t_i, f(t_i))) \end{aligned}$$

Notice also that we have dropped all iteration indices. This is possible in a loop approximated analysis where control-flow options from previous visits to a block b have all been merged by a $\widetilde{\sqcup}$-approximation.

This example shows that the effect of using a loop approximated analysis is that the analysis of loops (or any other cycle in the dependency graph) will generate a sequence of χ-terms all following a similar pattern:

$$t_n = \chi^b(t_i, T_n) \text{ where } T_n = \widetilde{\sqcup}(f(t_i), f^2(t_i), \ldots, f^n(t_i)).$$

The term T_n clearly represents a conservative approximation of the contribution from n loop iterations and $t_n = \chi^b_n(t_i, T_n)$ can be interpreted as: we go into the loop (T_n) or we do not (t_i).

Another consequence of using a loop approximated analysis is that

$$t_i, f(t_i), f^2(t_i), \ldots, f^n(t_i)$$

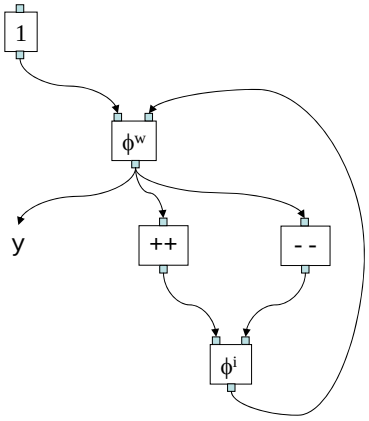
forms an ascending chain that will eventually get stabilized if f is a monotone function. Thus, after a finite number of loop iterations we will have $f^n(t_i) = f^{n-1}(t_i)$ and as result that $t_n = t_{n-1}$. This signals that the loop analysis can be terminated.

To gain a more concrete understanding of how the loop approximation can be used to terminate the analysis of a loop let us look into an intuitive example. If we have a `while`-loop that enclosed an `if`-statement that assigns a new value to a variable `x`. This situation is depicted in Figure 6 (SSA-graph left) where we also show the first three `x` values that might escape the loop (top right). The non-approximated values are given at the top of the figure and illustrates the problem of growth. That is, the set of control-flow options that might influence the value is growing larger and larger for each iteration. Furthermore, the values x^w_n and x^w_{n-1} returned from two consecutive iterations are not comparable (i.e. $x^w_n \mathrel{\widetilde{\not\sqsubseteq}} x^w_{n-1}$ and $x^w_{n-1} \mathrel{\widetilde{\not\sqsubseteq}} x^w_n$). This implies that we will never reach a stable situation where $x^w_n = x^w_{n-1}$ where we can terminate the loop analysis.

The situation is quite different in the loop approximated analysis of the loop where we after the second iteration get a result $x^w_2 = \chi^w(1, \top)$ that is not changed in the following iterations. (We have used a so-called "flat" lattice for integers where $n \sqcup \bot = n$ and $n \sqcup m = \top$ for any two lattice elements n and m, $n \neq m$. Moreover, we assume the following transfer functions for the $++(--)$ operations: $n++(--) = n + 1(n-1)$ for integers n, $\top++(--) = \top$ and $\bot++(--) = \bot$. We have also removed redundant subterms.) The stable situation will get recognized by the analysis after the third iteration and the loop analysis will terminate.

5.5 The k-Approximated Analysis

In the previous section, we introduced two different approximations that make sense in almost any type of analysis. The loop approximation is necessary to guarantee analysis



No Approximations

$$x_0^w = \chi_0^w(1, \bot)$$
$$x_1^w = \chi_1^w(1, \chi_0^i(\chi_0^w(2, \bot), \chi_0^w(0, \bot)))$$
$$x_2^w = \chi_2^w(1, \chi_1^i(\chi_1^w(2, \chi_0^i(\chi_0^w(3, \bot), \chi_0^w(1, \bot))), \chi_1^w(0, \chi_0^i(\chi_0^w(1, \bot), \chi_0^w(-1, \bot)))))$$

Loop Approximated

$$x_0^w = \chi^w(1, \bot)$$
$$x_1^w = \chi^w(1, \chi^i(2, 0))$$
$$x_2^w = \chi^w(1, \top)$$
$$x_3^w = \chi^w(1, \top) \qquad \text{(Loop analysis terminated)}$$

Fig. 6. A loop approximated version of the given example . It illustrates how the loop approximation can be used to terminate the analysis of a loop.

termination and k in the finite k approximation is a precision parameter that can be seen as the size of "context memory" which decides how many previous control-flow options that each χ-term should try to remember.

In this section, we present notations and results that are valid for χ-terms, and analyses, that are both loop and finite k approximated. We will for simplicity refer to such χ-terms as *k-approximated* and an analysis that uses this approach will be called a *k-approximated analysis*.

In what follows, we will present results and notations related to a k-approximated analysis. This will be a rather brief presentation since many of the concepts has earlier on been introduced in a non-approximated version. However, this is the approach we intend to use in the rest of this section and the results presented here can be seen as the "final" results of this rather lengthy section.

The Normalized Set $X_V^{(k)}$ The set of all k-approximated χ-terms forms a subset of all χ-terms. We will here introduce a normalized form of this subset where we require *increasing* block numbers of the subterms along all leaf-to-root paths in a χ-term. (We assume that all leaf values $v \in V$ has been assigned the block number 0.) We refer to χ-terms having this specific structure as *normalized*. This approach of describing $X_V^{(k)}$ has the advantage that all χ-term values now has a unique χ-term (tree) representation. In what follows will take great care in maintaining this ordering.

Furthermore, we will assume that we have a *control-flow numbering* of the basic blocks in the flow-graph. That is, when numbering the basic blocks, we try to assign each basic block a higher number than their control-flow predecessors. More precisely, for any two blocks B^1 and B^2 we *try* to assign block numbers b^1 and b^2 such that:

$$B^2 \text{ always executed after } B^1 \Rightarrow b^2 > b^1.$$

We have here emphasized the word try since this type of block numbering is, although possible within a method, not possible for a whole program. However, it serves as a

guide line for how to number the basic blocks in a program. The advantage of this approach is that subterms with a large depth can be considered as more "remote" than those closer to the root and that the finite k approximation can be defined more accurately.

ϕ-node Semantics A new χ^b-term is created whenever the analysis reaches a ϕ^b-node. The four steps involved in this process in case of k-approximated analysis are outlined in Algorithm 4. The first step is to remove every occurrence of χ^b in the input

Algorithm 4 $\phi^{\mathtt{b}}_{\mathtt{op}}(t_1^{(k)}, \ldots, t_n^{(k)}) \mapsto t^{(k)}$

for all $t_i^{(k)} \in \{t_1^{(k)}, \ldots, t_n^{(k)}\}$ **do**
  $t_i = \mathtt{loopApprox}(t_i^{(k)}, b)$
end for
$t = \chi^b(t_1, \ldots, t_n)$
$t = \mathtt{normalize}(t)$
$t^{(k)} = \mathtt{kApprox}(t, k)$

operands. This is done in `loopApprox` by replacing all subterms with block number b with their $\widetilde{\sqcup}$-approximation. Next, we construct a new χ-term that is now guaranteed to be loop approximated but neither normalized nor finite k approximated. The algorithm `normalize` takes care of the normalization (see Algorithm 5). It is a recursive process where we make repeated Shannon expansions over the χ-function with the highest block number. This process continues until we reach subterms having a block number that is less then b. Once `normalize` is applied we have a χ-term that is both loop approximated and normalized. We complete the ϕ-node handling by applying `kApprox` to make sure that the resulting χ-term has a maximum depth of k.

Algorithm 5 $\mathtt{normalize}(t) \mapsto t^*$

$b = block(t)$
$max = maxBlock(children(t))$
if $max > b$ **then**
  for all $i \in [1, arity(\chi^{max})]$ **do**
    $t_i^* = \mathtt{normalize}(t|_{max:i})$
  end for
  $t^* = \chi^{max}(t_1^*, \ldots, t_{arity(\chi^{max})}^*)$
  return t^*
else
  return t
end if

$\chi^{(k)}$-induced operators We have previously shown that for each context-insensitive operator $\tau : A \times B \times \ldots \times N \mapsto V$ their is a corresponding χ-induced operator $\widetilde{\tau} : X_A \times X_B \times \ldots \times X_N \mapsto X_V$. In the definition of $\widetilde{\tau}$, we used an algorithm `apply` that performed repeated Shannon expansions until it reaches the leaf values where the context-insensitive operator can be applied.

The definition of a $\chi^{(k)}$-induced operator can be seen as `apply` followed by a normalization procedure and a finite k-approximation. Using this approach, it is obvious that algebraic properties of a context-insensitive operator τ, like commutativity and associativity, are preserved for $\chi^{(k)}$-induced operators since we showed in Section 3 that they where preserved for any non-approximated χ-induced operator.

In Algorithm 6, we present a recursive algorithm `kPush` that performs all three activities (`apply`, normalization, and finite k cut-off) in a single traversal of the input operands.

Algorithm 6 $\texttt{kPush}(k, \tau, t_1, \ldots, t_n) \mapsto t^{(k)}$

$b = \max(block(t_1), \ldots, block(t_n))$
if $k = 0$ **then**
  for all $i \in [1, n]$ **do**
    $v_i = \texttt{collapse}(t_i)$
  end for
  $t^{(k)} = \tau(v_1, \ldots, v_n)$
else if $b = 0$ **then**
  $t^{(k)} = \tau(t_1, \ldots, t_n)$
else
  for all $i \in [1, arity(\chi^b)]$ **do**
    $c_i = \texttt{kPush}(k-1, \tau, t_1|_{b:i}, \ldots, t_n|_{b:i})$
  end for
  $t^{(k)} = \chi^b(c_1, \ldots, c_{arity(\chi^b)})$
end if
return $t^{(k)}$

The default handling in this algorithm is to push the operator τ towards the leaf values by making a Shannon expansion over the root χ-function in the operands that has the highest block number. This process guarantees that the result is normalized if all the input χ-terms are normalized.

The test $k = 0$ identifies the cut-off case where we have reached the maximum depth k of the resulting χ-term. In this case, we use `collapse` to make a conservative approximation of the remaining subtrees and apply the context-insensitive operator τ on the results. The case $b = 0$ identifies the case where all input operands are leaf values and the context-insensitive operator τ can be applied.

Finally, by using `kPush`, we can properly define the $\chi^{(k)}$-induced operators.

Definition 3. *For each context-insensitive operator $\tau : A \times B \times \ldots \times N \mapsto V$ their is a corresponding $\chi^{(k)}$-induced operator $\widetilde{\tau}^k : X_A^{(k)} \times X_B^{(k)} \times \ldots \times X_N^{(k)} \mapsto X_V^{(k)}$ defined*

as

$$\widetilde{\tau}(t_a, \ldots, t_n) = \texttt{kPush}(k, \tau, t_a, \ldots, t_n) \qquad \forall t_a, \ldots, t_n \in X_V^{(k)}.$$

We noted in Section 5.2 that Shannon expansions in combination with finite k-approximations are a bit problematic. The basic observation was that the interpretation of finite k approximations as an approach "where we only keep track of the last k control-flow options" was disturbed when Shannon expansions was used since they rewrite the structure of the χ-term. In this section, we have tried to minimize this problem by introducing a heuristic control-flow numbering of the basic blocks and introduced algorithms (`normalize` and `kPush`) that uses this block ordering to minimize the mixing of "remote" and "recent" control-flow options due to Shannon expansion.

The Lattice $\widetilde{\mathcal{L}}_V^{(k)}$ In Section 4, we introduced a χ-induced lattice $\widetilde{\mathcal{L}}_V$ that in general has an infinite height. In this section, we present the $\chi^{(k)}$-induced lattice $\widetilde{\mathcal{L}}_V^{(k)}$ which in contrast has a finite height. The finite height result follows from the fact that we always have a finite number of χ-terms in any loop approximated analysis. (see Section 5.3).

Theorem 9. *For each lattice of abstract values $\mathcal{L}_V = \{V, \sqcap, \sqcup, \top, \bot\}$ there is a corresponding $\chi^{(k)}$-induced lattice $\widetilde{\mathcal{L}}_V^{(k)} = \{X_V^{(k)}, \widetilde{\sqcap}^{(k)}, \widetilde{\sqcup}^{(k)}, \top, \bot\}$ where $\widetilde{\sqcap}^{(k)}$ and $\widetilde{\sqcup}^{(k)}$ are the $\chi^{(k)}$-induced versions of $\sqcap$ and $\sqcup$ defined in terms of the algorithm `kPush` as:*

$$\widetilde{\sqcup}^{(k)}(t_1, \ldots, t_n) = \texttt{kPush}(k, \sqcup, t_1, \ldots, t_n)$$
$$\widetilde{\sqcap}^{(k)}(t_1, \ldots, t_n) = \texttt{kPush}(k, \sqcap, t_1, \ldots, t_n).$$

That $\widetilde{\sqcap}^{(k)}$ and $\widetilde{\sqcup}^{(k)}$ are both commutative and associative follows from the preservation of algebraic identities previously discussed. The same holds for $t \; \widetilde{\sqcap}^{(k)} \; \bot = \bot$, $t \; \widetilde{\sqcup}^{(k)} \; \top = \top$ for all $t \in X_V^{(k)}$. Finally, closure follows from the design of `kPush` that guarantees to generate a new k-approximated χ-term.

We can use the $\chi^{(k)}$-induced lattice operators $\widetilde{\sqcap}^{(k)}$ and $\widetilde{\sqcup}^{(k)}$ to define a partial ordering relation between k-approximated χ-terms. The Connecting Theorem (see [DP02], page 39) implies that

Theorem 10. *Let $\widetilde{\mathcal{L}}_V^{(k)} = \{X_V^{(k)}, \widetilde{\sqcap}^{(k)}, \widetilde{\sqcup}^{(k)}, \top, \bot\}$ be a χ-induced lattice for some abstract values V and let $\widetilde{\sqsubseteq}^{(k)} : \widetilde{\mathcal{L}}_V^{(k)} \times \widetilde{\mathcal{L}}_V^{(k)} \mapsto \{\texttt{true}, \texttt{false}\}$ be an operator defined as:*

$$t_1 \; \widetilde{\sqsubseteq}^{(k)} \; t_2 \iff t_1 \; \widetilde{\sqcup}^{(k)} \; t_2 = t_2, \qquad \forall t_1, t_2 \in X_V^{(k)}$$

Then $\widetilde{\mathcal{P}}_V^{(k)} = \{\widetilde{\sqsubseteq}^{(k)}, X_V^{(k)}\}$ is a ($\chi^{(k)}$-induced) partial ordering over $X_V^{(k)}$.

To motivate the following result we can use the same line of arguments that we used when discussing the preservation of algebraic identities. That is, a $\chi^{(k)}$-induced operator $\widetilde{\tau}^{(k)}$ is just a finite k approximated χ-induced operator $\widetilde{\tau}$. From this it follows that

$$\widetilde{\tau}(t_1) \; \widetilde{\sqsubseteq} \; \widetilde{\tau}(t_2) \; \Rightarrow \widetilde{\tau}^{(k)}(t_1) \; \widetilde{\sqsubseteq}^{(k)} \; \widetilde{\tau}^{(k)}(t_2), \qquad \forall t_1, t_2 \in \widetilde{\mathcal{L}}_A^{(k)}$$

and consequently that

Theorem 11. *Let $\tau : \mathcal{L}_A \mapsto \mathcal{L}_B$ be a monotone function and let $\widetilde{\tau} : \widetilde{\mathcal{L}}_A^{(k)} \mapsto \widetilde{\mathcal{L}}_B^{(k)}$ be the corresponding $\chi^{(k)}$-induced operator. It then holds that*

$$t_1 \mathbin{\widetilde{\sqsubseteq}} t_2 \;\Rightarrow\; \widetilde{\tau}^{(k)}(t_1) \mathbin{\widetilde{\sqsubseteq}}^{(k)} \widetilde{\tau}^{(k)}(t_2), \qquad \forall t_1, t_2 \in \widetilde{\mathcal{L}}_A^{(k)}.$$

Thus, for any context-insensitive data-flow problem with value lattice $\mathcal{L}_V$ and transfer function $\tau^{(k)}$, we have $\chi^{(k)}$-induced counter parts $\widetilde{\mathcal{L}}_V$ and $\widetilde{\tau}^{(k)}$ that, since $\widetilde{\mathcal{L}}_V$ has a finite hight, is guaranteed to reach a fixed point.

Finally we know from the fundamental theory of data-flow frameworks [NNH99] that the time complexity for an analysis is proportional to the lattice height h^k. A rough estimate of h^k can be motivated as follows: let p be a program, let a be the maximum arity in any χ-function in $\mathcal{X}(p)$, and let h^v be the height of the context-insensitive value lattice $\mathcal{L}_V$. Furthermore, the tree representation of an arbitrary χ-term with depth k has about a^k leafs and the same number of subterms. Each leaf has a height of h^v. Thus, just by choosing different leaf values for this particular tree structure we can construct an ascending chain $x_1, \ldots, x_n$ that has length $O(a^k \cdot h^v)$. Furthermore, each element x_i in this chain can be further divided into an ascending subchain $x_{i1}, \ldots, x_{iN}$ by replacing each one of the a^k subterms by their $\widetilde{\sqcup}$-approximation. This gives this subchain a length of about a^k. Thus, a rough estimate of the maximum ascending chain length, and therefore the height h^k of the lattice $\widetilde{\mathcal{L}}_V^{(k)}$, is $O(a^k \cdot a^k \cdot h^v) = O(a^{2k} \cdot h^v)$.

6 Relation to Previous Work

As mentioned before, this paper is very much inspired by the ideas first presented by Martin Trapp in his dissertation [Tra99]. In that work, he presents an approach that is very similar to the k-approximated analysis. The major difference is that he presents his loop and finite k approximated approach as a *monolithic construct* without discussing the non-approximated case. To put it very short, he presents the set of normalized χ-terms $X_V^{(k)}$ together with rules for how to compute $\widetilde{\sqcup}$-approximations and $\chi^{(k)}$-induced operators $\widetilde{\tau}^{(k)}$. Furthermore, he states that $\chi(t_1, \ldots, t_n) \mathbin{\widetilde{\sqsubseteq}} \widetilde{\sqcup}(t_1, \ldots, t_n)$ and that we, in any phase of the analysis, can replace a χ-term $\chi(t_1, \ldots, t_n)$ with $\widetilde{\sqcup}(t_1, \ldots, t_n)$ and still maintain a conservative approach. The additional work that we have done is decribed in next section.

In [RKS99] and [KR00] Rütting et al. demonstrate an efficient and powerful approach by the usage of *value graphs*, which have initial similarities to our χ-terms representations. Both representations are based on a SSA representation of a program, and are using a graph representing the control flow in the program. The focus of their usage of *value graphs* is to find a solution for *Constant propagation* problem, while in our case we focus on value propagation for any data-flow analysis problem. Another difference is that we are using Shannon expansion to force our operators out to the leaves, where the operation can be evaluated. This is not the case in the *value graphs*, the operator nodes are scattered out in the *value graph*-representation, and therefore the evaluation of the result has another approach.

Lundberg and Löwe have in [LL13] been looking into the possibility of saving more information for doing a more precise points-to analysis. Their approach was to increase

the call-depth ($k \geq 1$) for each new context. Their result of not getting any substantial precision improvement when the depth was increased ($k > 1$) implicates that increasing k in our *k-approximation* is not a sure way to get a significant precision improvement.

7 Summary and Future Work

Taken all together, our presentation of χ-terms is an attempt to verify many of the results that was presented, hinted, and implicitly assumed by Martin Trapp. It is also an attempt to verify (and understand) many of his "stated" results and definitions. In addition to this we have focused on the non-approximated χ-term expressions that we think is missing, and the reason for this is:

1. In order to properly motivate the introduction of the $\widetilde{\sqcup}$-approximation as a "conservative" approximation satisfying $t \mathrel{\widetilde{\sqsubseteq}} t_a^*$, for any $\widetilde{\sqcup}$-approximation t_a^* of a χ-term t, we need to introduce the non-approximated χ-term lattice $\widetilde{\mathcal{L}}_V$, and the corresponding partial ordering $\mathrel{\widetilde{\sqsubseteq}}$. It is only in this context that we can verify that $t \mathrel{\widetilde{\sqsubseteq}} t_a^*$ (Theorem 7) and properly interpret a $\widetilde{\sqcup}$-approximation as "conservative".
2. We have been able to prove that many basic properties, such as commutativity and associativity, of a context-insensitive operator τ are directly mapped to the $\chi^{(k)}$-induced counterpart $\widetilde{\tau}^{(k)}$. We did this in two steps: i) We proved that it holds in the non-approximated case using structural induction, ii) We concluded that any identity that holds in the non-approximated case also must hold in the k-approximated case since the k-approximation is just a simple tree manipulation.
3. The two most important results are that any abstract value lattice $\mathcal{L}_V$ has a $\chi^{(k)}$-induced counterpart $\widetilde{\mathcal{L}}_V^{(k)}$, and that $\tau : \mathcal{L}_A \mapsto \mathcal{L}_B$ is monotone implies that $\widetilde{\tau}^{(k)} : \widetilde{\mathcal{L}}_A^{(k)} \mapsto \widetilde{\mathcal{L}}_B^{(k)}$ is monotone. These results make it possible to say that any context-insensitive data-flow framework has a $\chi^{(k)}$-induced context-sensitive counterpart.

The ideas from this paper will be used in future work to explore details about concrete context-sensitive dataflow problems such as constant folding and points-to analysis.

References

[AU95] A.V. Aho and J.D. Ullman. *Principles of Computer Science, (Second C edition)*. Computer Science Press, 1995.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[DLL14] Antonina Danylenko, Jonas Lundberg, and Welf Löwe. Decisions: Algebra, implementation, and first experiments. *Journal of Universal Computer Science*, 20(9):1174–1231, September 2014.

[DP02] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order, Second Edition*. Cambridge University Press, 2002.

[KR00] Jens Knoop and Oliver Rüthing. Constant propagation on the value graph: Simple constants and beyond. In DavidA. Watt, editor, *Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 94–110. Springer Berlin Heidelberg, 2000.

[LL13] Jonas Lundberg and Welf Löwe. Points-to analysis: A fine-grained evaluation. *Journal of Universal Computer Science*, 18(20):2851 – 2878, 2013.

[NNH99] F. Nielsen, H.R. Nielsen, and C. Hankin. *Priciples of Program Analysis*. Springer, 1999.

[RKS99] Oliver Rüthing, Jens Knoop, and Bernhard Steffen. Detecting equalities of variables: Combining efficiency with precision. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 232–247. Springer Berlin Heidelberg, 1999.

[THLL15] Martin Trapp, Mathias Hedenborg, Jonas Lundberg, and Welf Löwe. Capturing and manipulating context-sensitive program information. *Software Engineering Workshops 2015*, 1337:154–163, 2015.

[Tra99] Martin Trapp. *Optimerung Objektorientierter Programme*. PhD thesis, Universität Karlsruhe, December 1999.