The Journal of Systems & Software 177 (2021) 110952

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Memory efficient context-sensitive program analysis[☆]

Mathias Hedenborg[*], Jonas Lundberg, Welf Löwe

Department of Computer Science and Media Technology, Linnaeus University, Sweden

ARTICLE INFO

Article history:
Received 7 September 2020
Received in revised form 14 March 2021
Accepted 16 March 2021
Available online 24 March 2021

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

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.

To show the efficiency of our approach we run experiments comparing the memory usage of χ-terms with four alternative data structures. Our experiments show that χ-terms clearly outperform all the alternatives in terms of memory efficiency.

1. Introduction

Static program analyses approximate the run-time behavior of a given program. This is done by abstracting from the concrete semantics of programs and from concrete data 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 countably (infinitely) many contexts. Merging the analyzed information of different contexts reduces memory consumption at the cost of analysis precision. We should therefore aim for compact representations of the mappings of context information to analysis information.

The memory usage problem related to context-sensitive analyses is not just a theoretical issue; several papers, e.g., Lhoták and Hendren (2008a), Lundberg and Löwe (2013), evaluating various inter-procedural context-sensitive data-flow approaches report severe memory problems when using call-depths $k \geq 2$, and the path-explosion problem, e.g., Cadar and Sen (2013), Boonstoppel et al. (2008) 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. Hence, we do not distinguish between inter-procedural call context sensitivity (Shivers, 1991) and intra-procedural trace or path sensitivity (Rival and Mauborgne, 2007). 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 that capture the analysis results of different contexts.

We assume a Static Single Assignment (SSA) (Cytron et al., 1991; Muchnick, 1997) representation of a program. We further assume a program analysis following a standard data-flow analysis approach as given, e.g., in Marlowe and Ryder (1990). 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.

In SSA graphs, ϕ-nodes are used to select between different definitions of a variable. There is a close relation between ϕ-nodes and χ-terms. We create a new χ-term each time we apply the transfer function of a ϕ-node and each χ-term encodes the various control-flow dependent analysis value options that were available when the transfer function was applied. We distinguish the χ-term operator symbols (χ^b) by the static block number b of the related ϕ-nodes. If a ϕ-node occurs in a loop and is, hence, analyzed several times, a numerical index j is added to the operator symbols (χ_j^b), and a new set of χ-terms is generated

☆ Editor: Gabriele Bavota.

* Correspondence to: Department of Computer Science and Media Technology, Office Number B3020, Linnaeus University, 351 95, Växjö, Sweden.

E-mail addresses: Mathias.Hedenborg@lnu.se (M. Hedenborg), Jonas.Lundberg@lnu.se (J. Lundberg), Welf.Lowe@lnu.se (W. Löwe).

for each analysis iteration j over block b containing ϕ-nodes. This index increases each time a ϕ-node is updated, i.e., each time its transfer function is applied in the analysis.

Our approach is an extension and formalization of the ideas first presented by Martin Trapp in his dissertation (Trapp, 1999). 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* (Alpern et al., 1988), since they also capture the history of control flow. The main difference between these two approaches is the memory efficiency we achieve by avoiding data redundancy.

χ-terms can also be understood as a generalization of *Binary Decision Diagrams* (BDD) (Akers, 1978) used to represent logical functions. The main idea from BDDs that we use in our χ-terms is the graph representation and the redundancy elimination for memory efficiency. The represented decisions based on the logical values (true/false) are used to represent analysis after conditional executions. To this end, χ-terms are ordered decision diagrams (like OBDDs) but allow for multiple decisions in the nodes (not just true/false of a parameter). Subtrees represent analysis values from multiple control-flow predecessors of a ϕ-node. They also allow for multiple target values in the leaves (not true/false as a result). Leaves represent analysis (lattice) values.

Traditionally context-sensitivity is used for inter-procedural data-flow analysis. For the sake of simplicity, however, our presentation of χ-terms in Section 2 will focus on intra-procedural context-sensitivity where the various contexts are due to control statements. In our experiments in Section 3, we will use context-sensitive data from an inter-procedural data-flow analysis. 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.

This paper is a complement to our paper *A Framework for Memory Efficient Context-Sensitive Program Analysis* (Hedenborg et al., 2021). In Hedenborg et al. (2021) we give a formal presentation of χ-terms as context-sensitive analysis values that forms an abstract value lattice. We prove that any conservative context-insensitive analysis can be transformed into an approximated conservative context-sensitive analysis with a finite number of χ-terms. This implies that the resulting context-sensitive analysis is guaranteed to reach a fixed point. Hence, χ-terms are not only a memory efficient data structure for context-sensitive analysis information, they are also the backbone of a theoretical framework for context-sensitive program analysis.

The formal presentation in Hedenborg et al. (2021) outlines a theoretical framework of χ-terms without any supporting experimental evidence of being memory efficient. The aim of this paper is to complement that study with a set of experiments. Our contributions in this paper are the following:

1. We propose approximated χ-terms as a memory efficient representation, to capture context-sensitive analysis values in an SSA-based program analysis.
2. We show in a set of experiments the memory efficiency of using χ-terms compared to four other approaches that handle context-sensitive information. The four approaches are context table, context tree in two variants, and double hash map. These approaches will be explained in detail in Section 3.2.

The remainder of the paper is structured as follows: In Section 2, we give an informal introduction to χ-terms. We use examples from an intra-procedural analysis here as they provide an easy understanding of the theory. In Section 3 we present experiments and demonstrate the memory efficiency of χ-terms in relation to four other data structures. Here we apply the theory to inter-procedural points-to analysis. In Section 4 we present related works and Section 5 concludes the paper.

2. Introduction to χ-terms

In this section we give an informal introduction to χ-terms aiming for basic understanding rather than formality. As stated earlier, a formal presentation of χ-terms, as well as algorithms and proofs can be found in Hedenborg et al. (2021). Brief presentations of χ-terms can also be found in Trapp (1999) (published in German), Lundberg (2004), and Trapp et al. (2015). Algorithms for manipulating χ-terms only outlined in this section are described in Hedenborg et al. (2021).

We assume an SSA-based program representation and we will refer to the used program representation as the *SSA graph*. In Fig. 1, we show a simple piece of code, e.g., a method body (left), containing three `if`-statements and the corresponding basic block structure (middle) and SSA graph (right). In the SSA graph, we have annotated each ϕ-node Φ_x with their basic block number #b. The nodes 1, 2, 3 and 4 (in boxes) in the SSA graph represent integer values.

The SSA graphs are the method body parts of a program representation. They are *flow-sensitive*: every def-use relation of represented operations in a method is explicitly represented as an edge between the defining operation and the using operation (Hasti and Horwitz, 1998). An *SSA method graph* $G = \{N, E, Entry, Exit\}$ is a directed, ordered multi-graph where N is a set of SSA nodes, E is a set of SSA edges, *Entry* is a unique graph entry node satisfying $|pred(Entry)| = 0$, and *Exit* is a unique graph exit node satisfying $|succ(Exit)| = 0$. *Entry* dominates all nodes and *Exit* post-dominates all nodes in N.

In a context-insensitive data-flow analysis, the interpretation of the ϕ-function can be seen as a merger of all possible definitions of a given variable. This results in an approximation of the run-time values, e.g., by sets of possible values. For example, the approximated values of the variables x, y, and a in Fig. 1 is $\{1, 2\}$, and the value of b is $\{3, 4\}$.

A χ-term 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 Fig. 1 using χ-term as $b = \chi^7(3, 4)$. Interpretation: variable b has the value 3 if block #7 was reached from the predecessor block #5 in the control-flow graph; b has the value 4 if it was reached from the predecessor block #6. In addition to a set representation containing 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. Using χ-terms, the values for the variables in Fig. 1 are:

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

Notice that a variable value is depending on a sequence of control-flow conditions, e.g., the value of variable a, correspond to χ-terms over χ-terms, i.e., they are constructed as a composition of the χ-terms representing previous control-flow conditions. The variable s, defined by $a + b$, where a and b are two χ-terms, generates a new χ-term involving an operator $+$.

2.1. χ-term construction

The construction of the χ-terms including the numbering of the χ-symbols for their distinction is a part of a context-sensitive analysis, more precisely, the result of applying an analysis update function of a ϕ-node. When the analysis method selects a ϕ-node for variable x in block b, its transfer function "asks" all 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)
```

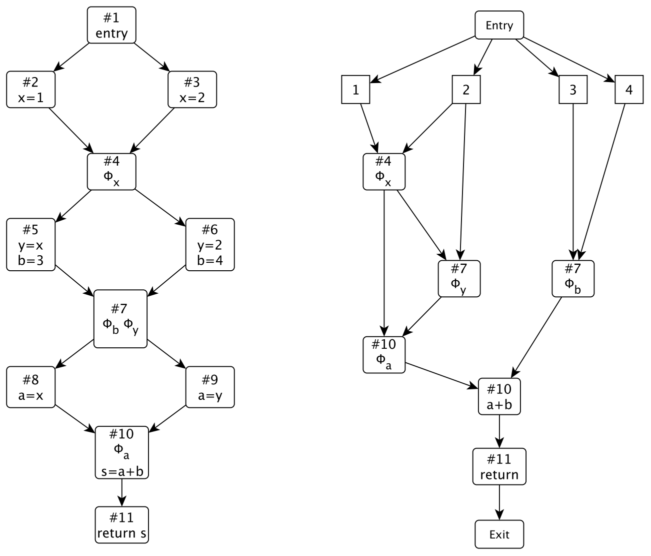


Fig. 1. A source code example with corresponding basic block and SSA graphs.

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

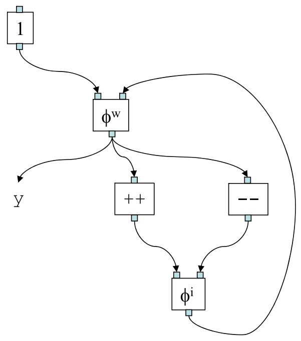


Fig. 2. A simple code fragment with a loop that encloses an if-statement and the corresponding SSA graph.

predecessor blocks to give their last definition of x and constructs a new χ-term $\chi^b(x_1, \dots, x_n)$ where x_i is the χ-term for x given by the i-th predecessor.

If the i-th predecessor block does not define x itself, it "asks" its predecessors for the value, which may lead to the construction of a χ-subterm. This process continues recursively until each predecessor has presented a χ-term for x. The process will terminate if any use of a value also has a corresponding definition and the SSA graphs are loop free.

Loops in the control-flow cause problems, as they would lead to ever growing χ-terms, i.e., analysis would not reach a fixed point. We will solve this termination problem by approximating χ-terms with values from the context-insensitive analysis lattice after a fixed number k of analysis iterations. For a first try, we may assume a countably infinite number of χ-terms with the same block number b since every ϕ-node in b generates a new χ-term when the analysis reaches that block.

This situation is illustrated in Fig. 2 where we show a `while`-loop that encloses an `if`-statement that assigns a new value to a variable `x`.

Initially, the ϕ^w-node in block w containing the `while`-loop header generates the value $\chi_0^w(1, \perp) = 1$ where the bottom element $\perp$ symbolizes "value undefined". This value is later propagated inside the loop body where the operators `+` and `-` has been pushed inside the χ-block by an operation called Shannon expansion, which will be explained in detail in Section 2.4. The ϕ^i node after the `if`-statement in block i generates the value $\chi_0^i(\chi_0^w(2, \perp), \chi_0^w(0, \perp)) = \chi_0^i(2, 0)$, a value that is propagated back to the loop header block w. Using this approach, analysis would generate all possible values for `x` after an arbitrary but fixed number of loop iterations. The values for `x` after at most three interactions that might escape the loop are:

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

There are a few things to notice. First, x_n^w is the value of `x` after at most n loop iteration. Secondly, each χ-term $x_n^w = \chi_n^w(1, \dots)$ can be interpreted as: Either `x` has the value 1 (no iterations), or the value $\dots$ after at most n loop iterations. Note that such an analysis would not terminate, but grow the terms indefinitely. Approximations will take care of termination; they are described in Section 2.6 below.

2.2. Common switching behavior of χ-terms with same index

A new set of χ-terms is generated for each analysis iteration j over block b containing ϕ-nodes. We denote this set with χ_j^b. All χ-terms generated in block b at iteration j have the same *switching behavior*. That is, for any two χ-terms $\chi_i^b(x_1, \dots, x_n)$ and $\chi_j^b(y_1, \dots, y_n)$, and for any execution of the program, it holds that if $i = j$ and branch x_k in $\chi_i^b(x_1, \dots, x_n)$ is selected, then also branch y_k in $\chi_j^b(y_1, \dots, y_n)$ is selected. This property will be important later on when we introduce operations on χ-terms. It allows us to reduce the terms significantly even if we do not know actual values statically.

2.3. χ-term representation

Every χ-term can be naturally viewed as a tree. This is illustrated in Fig. 3 (left) where we show the tree representation of the χ-term for the variable a in Fig. 1. 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 come into play.

Actually representing χ-terms as trees is in practice by far too expensive. A more cost efficient representation is a directed

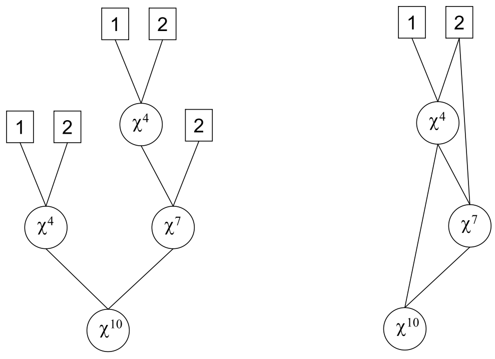


Fig. 3. Term for variable $a = \chi^{10}(\chi^4(1,2), \chi^7(\chi^4(1,2),2))$ is illustrated by its tree and directed acyclic graph representations (edge direction from bottom to top).

graph, similar to how BDDs are represented in Bryant (1986, 1992), which avoid redundant subtrees (cf. Fig. 3 (right)).

2.4. Basic χ-term operations

In this section we present basic operations on χ-terms based on Trapp et al. (2015). We focus on *restrictions, Shannon expansion* and *redundancy*. Most important is the *Shannon expansion* that can be used to manipulate χ-terms without affecting their value.

The *restriction* of a χ-term $t \in X_V$ to the k:th branch of χ_j^b, denoted $t|_{(b,j):k}$, is a new χ-term where every subterm $\chi_j^b(t_1, \dots, t_n)$ with switching behavior (b, j) in t has been replaced by its k:th child t_k.

For example:

$$t = \chi^3(\chi^1(1,2), \chi^2(\chi^1(1,2),2)) \Rightarrow \begin{cases} t|_{(3,\_):2} = \chi^2(\chi^1(1,2),2) \\ t|_{(1,\_):1} = \chi^3(1, \chi^2(1,2)) \end{cases}$$

It should also be noticed that if we restrict us to the k:th branch of χ_j^b, when χ_j^b does not occur in t, t is left unaffected.

The *Shannon expansion* of a χ-term $t \in X_V$ over χ_j^b is a new χ-term defined as:

$$t = \chi_j^b(t|_{(b,j):1}, t|_{(b,j):2}, \dots, t|_{(b,j):n}) \qquad \text{where } n = arity(\chi_j^b)$$

Notice, the Shannon expansion creates a new χ-term but not a new χ-term value. It is just a rewrite rule that can be used to manipulate a χ-term expression. This expansion is valid for terms with the same switching behavior in this manipulation, and according to the definition of switching behavior the index needs to be the same and can therefore be removed.

We illustrate the Shannon expansion (without any redundancy elimination) with the following example

$$\begin{aligned} t &= \chi^3(\chi^1(1,2), \chi^2(\chi^1(1,2),2)) \\ &\equiv \chi^1(\chi^3(\chi^2(1,1), \chi^2(1,2)), \chi^3(\chi^2(2,2), \chi^2(2,2))) && \text{(expansion over } \chi^1) \\ &\equiv \chi^2(\chi^3(\chi^1(1,2), \chi^1(1,2)), \chi^3(\chi^1(1,2), \chi^1(2,2))) && \text{(expansion over } \chi^2) \end{aligned}$$

A χ-term is *redundant* and can be simplified (*redundancy elimination*) if all its sub-terms are equivalent.

$$\chi_i^b(\chi_1, \dots, \chi_k) \equiv t \Leftrightarrow \chi_1 \equiv \chi_2 \equiv \dots \equiv \chi_k \equiv t$$

Redundancy elimination is a rewrite rule producing new χ-terms encoding the same context-sensitive information, and implies that a χ-term t containing a redundant subterm χ_r can be reduced without any loss of information. The process of removing redundant χ-subterms is called *redundancy elimination* and uses the pattern

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

In the tree view of a χ-term, this corresponds to replace a subtree rooted by χ^r by any of its subterms (which are all equivalent).

We illustrate redundancy elimination with the following example.

$$\begin{aligned} t &= \chi^1(\chi^3(1, \chi^2(1,2)), \chi^3(2, \chi^2(2,2))) \\ &\equiv \chi^1(\chi^3(1, \chi^2(1,2)), \chi^3(2,2)) && \text{(since } \chi^2(2,2) \equiv 2) \\ &\equiv \chi^1(\chi^3(1, \chi^2(1,2)), 2) && \text{(since } \chi^3(2,2) \equiv 2) \end{aligned}$$

Redundancy elimination, e.g., $\chi^3(2,2) \equiv 2$, removes redundant control-flow information from a χ-term. Repeated redundancy eliminations can remove all superfluous control-flow information from a χ-term. This results in a compact representation of context-sensitive information where only control-flow decisions that actually have an effect on a certain value are kept. Redundancy elimination is the major reason why χ-terms outperform all the presented alternatives in Section 3 in terms of memory efficiency.

2.5. Operations on χ-terms

The code fragment at the left-hand side of Fig. 4 assigns different values to the two 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 before we add them as:

$$a = \chi(a+1, a-1), \qquad b = \chi(b-1, b+1)$$

The corresponding χ-term is illustrated in the center of Fig. 4 where we consider the addition as an operator $+ : X_{int} \times X_{int} \mapsto X_{int}$ defined on χ-terms over (sets of) integers rather than on integers directly.

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

$$\begin{aligned} +(\chi(a+1, a-1), \chi(b-1, b+1)) &= \chi(\text{ add } \texttt{if} - \text{branch values}, \\ &\qquad \text{add } \texttt{else} - \text{branch values}) \\ &= \chi(+(a+1, b-1), +(a-1, b+1)) \end{aligned}$$

That is, we can in this case 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 transformation is illustrated in the right-most part of Fig. 4. The fact 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. However, this rule can be applied on any operator τ and any χ-term χ^b.

This rewrite rule for χ-term expressions makes use of *Shannon expansion* (see 2.4), and is also taken from the OBDD literature where it is used to symbolically operate over Boolean functions represented as OBDDs.

Finally, once we applied this rewrite rule for the operator $+$ in Fig. 4, we can apply $+$ on a set of leaf values in which case we can fall back on ordinary integer arithmetic. The resulting simplifications together with the redundancy rule[1] $\chi(t,t) = t$ can symbolically be written as:

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

The result indicates that no matter what branch of the `if`-statement we use, we will always get the result $a + b$. This simple

[1] The rewrite rule $\chi(t, \dots, t) = t$ can be used when every control-flow option gives the same value t, cf. 2.4.

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

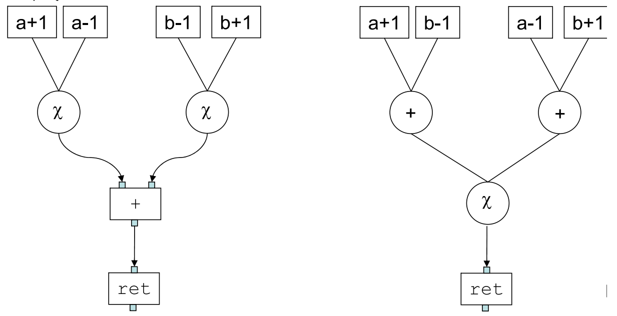


Fig. 4. A method with the corresponding χ-term representation (center) of the `return`-expression $a + b$. The right-most figure illustrates how the $+$-operator is "pushed" to the leaves together with the exploitation of switching behavior.

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.

2.6. χ-term approximations

We need to introduce some approximations to handle the ever growing χ-term generated during analysis, and to ensure analysis termination when handling loops in the control-flow. The first approximation we need is to state how many control-flow options a χ-term shall remember. This will be handled by introducing a precision parameter k, which can be seen as the size of "context-memory". Secondly we will approximate loops to guarantee the termination of the analysis.

In the analysis process a χ-term is created when analyzing a ϕ-node in the SSA graph. The χ-term represents how different control-flow options affect the value of a variable, and the size of the term grows larger as the analysis proceeds. To reduce the size of a χ-term we make a **finite** k **approximation** by restricting the number of control-flow options to k. This approximation can be seen as a tree manipulation in which we make a post order traversal and, starting at the leaf nodes, replace each χ-term by its context-insensitive approximation until we have a χ-term of height k.

Fig. 5 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.

From the loop example in Section 2.1 we can see that an analysis of a loop will generate χ-terms like $x_j^b = \chi_j^b(\ldots\chi_{j-1}^b(\ldots)\ldots)$. That is, the newly created χ-term will have a subterm with the same block number and a lower iteration index. This pattern will occur over and over again since each loop iteration results in a new composition of χ^b with itself. This will result in a χ-term of infinite depth and a non-terminating analysis if no measure is taken to stop the iterations. To be able to guarantee a termination of the analysis we need to make **loop approximations**.

During the analysis of a loop we approximate each newly generated χ-term by approximating each subterm (with same block number as the root) by the union of the context-insensitive values for the subterms. 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. We can then drop iteration index since the only control-flow option we use is the from the last visit to the block. Options from earlier visits is merged by the approximation.

Let us use 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 $++$ (resp. $--$) operation: $n++(--) = n+1(n-1)$ for integers n, $\top++(--) = \top$ and $\bot++(--) = \bot$. This will give the following loop approximation for the loop example given in Section 2.1

$$\begin{aligned}
x_0^w &= \chi^w(1, \bot) = 1\\
x_1^w &= \chi^w(1, \chi^i(2, 0))\\
x_2^w &= \chi^w(1, \top)\\
x_3^w &= \chi^w(1, \top)
\end{aligned}$$

After three iterations we can see that we have a stable value. That will end the analysis of the given loop and guarantee a termination of the analysis. Thus, by introducing the finite k approximation and the loop approximation we can reduce the memory demand and ensure analysis termination. We also get a precision parameter k that we can adjust the get a reasonable balance between analysis precision and memory costs.

In summary, χ-terms as presented in this section is a data structure for context-sensitive analysis information that (similar to BDDs) avoid redundancies. They can be used to capture intra-procedural (this section) as well as inter-procedural (in the evaluation, Section 3) context information. In addition to BBDs, they form the backbone of a theoretical framework guaranteeing that any conservative context-insensitive analysis can be transformed into an approximated conservative context-sensitive analysis guaranteed to reach a fixed point (Hedenborg et al., 2021).

2.7. Size of χ-terms during analysis

In this section we show that the intermediate results of χ-terms never consume more memory than the final result. This means that the final result of the χ-terms is actually an upper bound approximation of the memory footprint during the whole analysis. In Section 3, we will exploit this result by assessing and comparing only memory sizes the final analysis results for χ-terms and its alternatives.

We understand the χ-terms as directed acyclic graphs; their size is determined by their number of nodes N and edges E. Without loss of generality, we assume *forward* and *may* analyses constructing the χ-terms. SSA nodes are analyzed and updated in a data-driven way: nodes before their successors; inner loop nodes

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

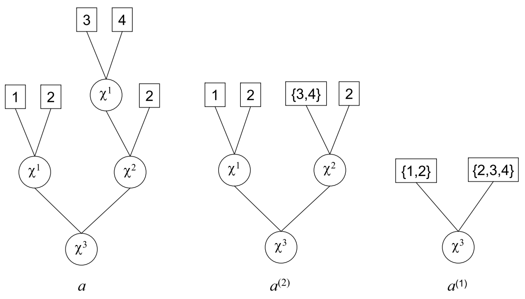


Fig. 5. Two different finite k approximations of the same χ-term a.

before outer loops. Each point in the (SSA based) program representation is eventually attached with the χ-term representing the context-sensitive value analyzed for this program point as a fixed point; initially with the smallest χ-term $\perp$. Leafs of χ-terms represent context-insensitive values, e.g., $\perp$, $\top$, constants or elements of a power set lattice. Recall that these context-insensitive values are special χ-terms (without χ-nodes). All elements of the same analysis lattice require the same amount of memory, e.g., a symbolic constant or a bit vector. Inner nodes of χ-terms represent context-sensitive values. They require a unique χ-node (root) and references to their child χ- or leaf-nodes.

Each ϕ node of the program contributes to the analysis with a join function $\sqcup$; all other SSA nodes n with a transfer function f_n. Updating a ϕ-node during analysis updates the corresponding χ-term. Either of the following situations might occur:

1. The χ-terms analyzed for the predecessors are all the same, in which case the current output value points to this one and the same χ-term.
2. At least two predecessors χ-terms are different, in which case the current output value points to a χ-node corresponding to this ϕ-node (a new such χ-node is created the first time this situation occurs). Its children pointers are set or updated to the χ-terms analyzed for the predecessor SSA nodes (so far).

Updating another node n during analysis updates the corresponding χ-term according to its transfer function f_n. Either of the following situations might occur:

3. The predecessor χ-terms are all context-insensitive values, in which case the current output is updated to the context-insensitive value according to f_n, which is selected if exists or created, otherwise.
4. At least one predecessors is a χ-term representing an context-insensitive value (with a χ-node as a root), in which case the current output value is updated to the χ-term of the Shannon expansion of f_n applied to the predecessor χ-terms.

It is important to note that each of the cases only keeps the χ-term sizes constant or increases it. This is obvious for the cases 1–3. In case 4, the Shannon expansion pushes the f_n over the χ-nodes in a recursive descent without creating new χ-terms. At the leafs, the context-insensitive value according to f_n are selected and created if necessary, respectively. The recursive ascent of the Shannon expansion may create new or updates existing χ-terms that are not used elsewhere, as described in the cases 1 and 2.

The fact that χ-terms are monotonously increasing their memory footprint throughout an analysis motivates our use of the final result of a points-to analysis to evaluate the memory efficiency of χ-terms in Section 3.3.

3. Evaluating memory efficiency

In this section we present the result of an experiment where we evaluate the memory efficiency of using χ-terms to represent context-sensitive information. That is, we compare the memory costs of using χ-terms with three other frequently used internal representations based on tables, trees, and hash-tables – details will be given in Section 3.2 – in a context-sensitive points-to analysis.

In Section 3.1 we give a brief presentation of context-sensitive points-to analysis, in Section 3.2 we present the memory metrics we used to compare the different internal representations, in Section 3.3 we describe a context-sensitive points-to analysis and the experiment setup, and in Sections 3.4 and 3.5 we present and discuss the results of our evaluation.

3.1. Context-sensitive points-to analysis

Points-to analysis is an inter-procedural static program analysis that computes precise object reference information by tracking the flow of abstract objects from one part of a program to another.

An *abstract object o* is an analysis abstraction that represents one or more run-time objects and in this section each *syntactic creation point s* corresponds to a unique abstract object $o_s \in O$ representing *all* run-time objects created at s in *any* execution of an analyzed program. During analysis, each variable and object field v in the analyzed program is associated with a points-to set $Pt(v) \subseteq O$, the set of abstract objects that may be referenced by v.

Data-flow based points-to analysis will in general track the flow of abstract objects from their creation points along all possible (may) paths in the data dependency graph. In a typical approach to points-to analysis, e.g. Lhoták and Hendren (2003), each method in the program is represented by a graph. The SSA ϕ-functions $r = \phi(r_1, r_2, \ldots)$ is modeled as a specific node type that merges the values ($Pt(r_i)$) transported to it along its in-edges.

In *context-insensitive* analysis, arguments of different calls to the same target method (graph) m get propagated and mixed there. The result of m's analysis is then the merger of *all* calls targeting m and the effects of m itself. Furthermore, the analysis results, i.e., over-approximated return values and heap updates, affect other callees of m.

A *context-sensitive* analysis aims to reduce this over-approximation by partitioning all calls targeting m into a finite number of *call contexts* (cloned method graphs). Context-sensitivity gives, in general, a more precise points-to analysis since the arguments of calls targeting the same method do not get mixed when the calls are analyzed in different contexts. This partitioning can be made even more fine-grained by taking more than one level (call-depth k) of call history into account.

As a concrete example, consider a control-flow analysis with call-depth 3 (3-CFA) that uses the three most recent call sites in the call history to define a context (Shivers, 1991). That is, each analysis value is represented by a quintuple like: $v,\ cs_1,\ cs_2,\ cs_3,\ Pt(v)$. That is, the value of variable v in method m is $Pt(v)$ when m is called from call site cs_1, which is called from call site cs_2, which is called from call site cs_3. Hence, the triple $[cs_1, cs_2, cs_3]$ identifies the call context, v identifies the variable, and $Pt(v)$ is the actual value stored in the variable in this particular context.

3.2. Memory size metrics

The major drawback of context-sensitivity is the increased memory costs that come with maintaining a large number of contexts, and that increasing the call-depth comes with an exponential increase in memory costs. The memory problem is not just a theoretical issue, several papers, e.g., Lhoták and Hendren (2008a), Milanova et al. (2005a), Lundberg and Löwe (2013), evaluating various context-sensitive points-to analysis approaches report severe memory problems when using call-depths $k \geq 2$.

In this section we present memory size metrics for five different data structures (context table, context tree in two variants, double hash map, and χ-terms) to represent context-sensitive analysis information. As our running example we will use call-depth 3[2] where each analysis value is specified by a quintuple

$$v,\ ctx_1,\ ctx_2,\ ctx_3,\ val.$$

That is, the variable v, in context $[ctx_1, ctx_2, ctx_3]$, has the value val. This is not only general enough to handle any type of points-to analysis with call-depth 3, it can handle any type of context-sensitive data-flow analysis having a context-depth 3. In an approximation of χ-terms ($k = 3$) each such quintuple would contribute with a single root-to-leaf path in the tree representation of a χ-term for variable v.

In order to compare the memory size of different techniques we use abstract memory size metrics rather than actual memory measurements during an actual analysis. The reason for this is twofold: (1) We would like to avoid presenting implementation details that might affect the actual results. Hence, the memory size metrics is an abstraction indicating what memory costs can be expected in an ideal case where all other memory influencing parameters are kept constant, (2) Identifying what part of the actual memory used in an analysis that is due to handling context information is non-trivial.

Context table

The most obvious approach to store all the quintuples generated during an analysis is a table having one quintuple in each row. The memory size of such a table is:

$$TableSize = 5 \times N_r$$

where N_r is the number of rows in the table (the number of quintuples).

The left-hand side of Fig. 6 shows a small fraction of such a table related to a single variable v. In this artificial example each row corresponds to unique context-sensitive value. For example, row 1 states that variable v, in context $[A, P, X]$, has the points-to value V_1. A realistic program has thousands of such variables and each variable have 10–30 context-sensitive values, cf. Table 1. The memory size $TableSize$ related to the seven tuples we associate with variable v is $5 \times 7 = 35$.

Besides their memory footprint, the major drawback with context tables is the lookup speed. Accessing the current value for a given variable, in a given context, is $O(N_r)$. However, we will use $TableSize$ as a base case against which other representations are compared.

Context tree

Another possible approach to represent the context-sensitive values for a given variable v is a *tree* where each tuple $[v, ctx_1, ctx_2, ctx_3, val]$ forms a unique *path* starting in root node v and having the value val as its leaf. The right-hand side of Fig. 6 shows the context tree for the example variable v. Each context ctx_i can be seen as a selection over possible context values, e.g., A or B in ctx_1. The tree representation guarantees a speedy lookup. It can also be more memory efficient than a table, since many variable values might have partially the same call history, e.g., $[ctx_1, ctx_2, \ldots]$ or $[ctx_1, \ldots]$, and can therefore "share" the internal tree nodes. We have one tree for each variable and the size of each tree is given by the number of edges and nodes:

$$TreeSize = \sum_{v \in RV} (N_v + E_v)$$

where N_v, E_v are the number of nodes and edges in the tree for variable v. The memory size $TreeSize$ associated with example variable v in Fig. 6 is $13 + 12 = 25$.

A simple improvement of the tree representation is to avoid duplicate leaf nodes representing the same value. Thus, we merge all leaf nodes for a given value into one representative node (thus breaking the tree structure). The left-hand side of Fig. 7 shows the corresponding data structure for the example variable v. We use the same lookup mechanism, have the same number of edges, but reduce the total number of nodes.

$$MergeSize = N^{val} + \sum_{v \in RV} (N_v^{int} + E_v)$$

where N_v^{int} is the number of internal nodes and N^{val} is the number of unique values. The memory size $MergeSize$ associated with example variable v is $3 + 6 + 12 = 21$.

Double hash map

The representation currently used in our in-house points-to analysis is optimized for a speedy lookup and uses two hash maps (map to map). The analysis algorithm is analyzing one method at the time in a given context $[ctx_1, ctx_2, ctx_3]$, and for each such context it keeps a variable-to-value map. Hence, the technique can be seen as 2-step map context-to-(variable-to-value). Furthermore, it uses a similar avoid-duplicate-values approach as described in $MergeSize$. The memory size using this approach, using context depth 3, can then be computed as

$$DoubleHash = N^{val} + \sum_{c \in Ctx} (3 + 3N_c^v)$$

where Ctx is the set of all context's triples, 3 is the size to store such a triple, and N_c^v is the number of variables in context c, and N^{val} is the number of unique values. Notice also that the double hash map approach, in contrast to the tree representations and χ-terms, is not explicitly describing the possible values for a single variable v. Hence, we do not present any figure and size calculation for example variable v.

χ-terms

Each variable in an analysis with context depth 3 has their own χ-term (with maximum depth 3) representing their possible values in different contexts. The right-hand side of Fig. 7 shows

2 The metrics to be presented can easily be adjusted to any call-depth $k \geq 1$.

Var	ctx_1	ctx_2	ctx_3	**Pt(v)**
v	A	P	X	V1
v	A	P	Y	V1
v	A	P	Z	V1
v	A	Q	X	V1
v	A	Q	Y	V2
v	A	Q	Z	V3
v	B	P	X	V3

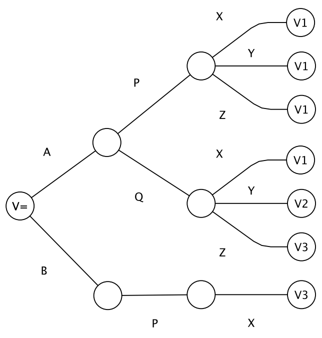
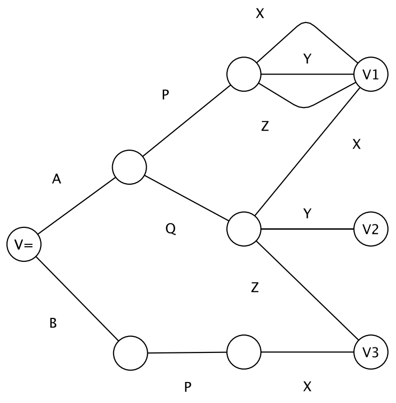


Fig. 6. Context descriptions for a single variable *v* and the used tree structure.



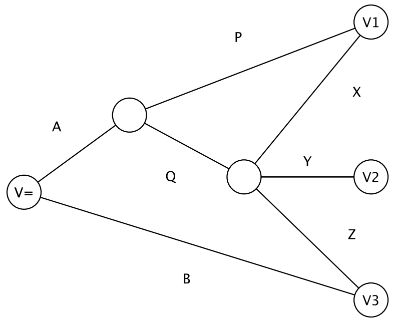


Fig. 7. Using Merge and χ-term structures for a single variable *v*.

the χ-term for the example variable *v*. Due to redundancy elimination, the χ-term only keeps the essential context information. In general, it is a directed acyclic graph and its size is therefore:

$$ChiSize = \sum_{v \in RV} (N_v + E_v)$$

where N_v, E_v are the number of nodes and edges in the χ-term for variable *v*. Consequently, the memory size for the example variable *v* is $6 + 7 = 13$.

Data structure time-complexity

The main reason for introducing χ-terms is memory efficiency since memory usage problems is the biggest issue related to context-sensitive analyses. However, time efficiency is also import when analyzing large programs. The time-complexity for looking up the current value for variable v in a context $[ctx_1, ctx_2, \ldots, ctx_k]$ is $O(N)$ in a context table with N contexts and $O(1)$ for the two trees and the χ-terms, since $O(\log(k)) = O(1)$, $k = const$. It is $O(1)$ for the double hash map. Hence, we do not expect the χ-terms to be a more time efficient data structure; its strength is the memory efficiency.

3.3. Experiment setup

The experiments to be presented in Section 3.4 are all based on context information taken from a points-to analysis. We run the analysis on ten different programs, extract the context-sensitive analysis results of points-to variables, and compute the five memory size metrics presented in Section 3.2. Using the final analysis results to construct the memory size metrics (rather than growing the metrics during the analysis) is motivated since: (1) It allows us to compare the memory size metrics of five different context-information data structures without having to implement each one of them, and (2) as shown in Section 2.7, the final results represents the maximum memory size for an analysis based on χ-terms.

The context-sensitive points-to analysis we use is presented in three papers (Lundberg and Löwe, 2007; Lundberg et al., 2009; Lundberg and Löwe, 2013) and forms the core of a PhD thesis (Lundberg, 2014). It uses a flow-sensitive data-flow algorithm called *SSA-Based Simulated Execution* and a context-sensitive approach to points-to analysis called *This-Sensitivity*. Simulated Execution is a data-flow algorithm simulating an abstract execution of the program where the processing of a method is interrupted when a call occurs, and later resumed when the processing of the called method is completed. This-sensitivity is a modified version of object-sensitivity presented by Milanova et al. (2005a) where the target context associated with a call site $a.m(\ldots)$ is determined by the pair $(m, Pt(a))$, where $Pt(a)$ is the points-to set of the target expression *a*. Hence, we distinguish analysis contexts of a method by its implicit variable *this* whereas the target contexts in object-sensitivity is given by a pair (m, o^i) for each $o^i \in Pt(a)$. Experiments in Lundberg et al. (2009), Lundberg and Löwe (2013) show that this-sensitivity gives almost as precise results as object-sensitivity (similar results in 3 out of 4 precision metrics). However, the slightly higher precision of object-sensitivity comes at the price of a much higher (a magnitude) time and memory costs.

The major reason for using this-sensitivity rather than object-sensitivity in this article is the (comparably) low memory costs for this-sensitivity. We show that χ-terms have a substantial effect on the memory costs even when using the already memory efficient this-sensitive points-to analysis instead of object-sensitivity that has a larger memory footprint.

In our experiments we use *k-this-sensitivity* to generate realistic context data (tuples). *k-this-sensitivity* makes use of the top *k* abstract this-sets on the call stack. This is similar to the way *k*-CFA is defined in terms of the top *k* call sites cs_i on the call stack (Shivers, 1991). Hence, a unique variable value in 3-this-sensitivity is given by a quintuple

$$v,\ Pt(a),\ Pt(b),\ Pt(c),\ Pt(v).$$

where v is a unique identifier of an SSA variable, $[Pt(a), Pt(b), Pt(c)]$ is the calling context, and $Pt(v)$ is the points-to set representing the analysis value of variable v in the given context.

We have collected context information (tuples) using a benchmark containing 10 different programs. Since we analyze Java bytecode, we characterize the size of a program in terms of number of classes, methods, and SSA variables rather than lines of code. The benchmark programs range from 691 to 1294 classes. All programs are presented in Table 1. The programs in the upper part of the table are taken from well-known test suites[3]; we have picked programs that were (i) larger than 600 classes, and (ii) freely available on the Internet. In the lower half, we have our own set of "more recent" benchmark programs, which are also freely available. All programs are analyzed using version 1.6.1 of the Java standard library.

The right-hand side in Table 1 shows the relative tuple count for each analysis. We use a relative measure indicating the average number of tuples for each variable. Hence, 33.2 for chart 3-TS indicates that each variable (on average) comes with 33.2 quintuples when analyzed using 3-this-sensitivity.

The total tuple count for each case can be obtained by multiplying the number of SSA Variables and the Relative Tuple Count. For example, the total tuple count for chart 3-TS is $33.2 \times 41,876 = 1,390,283$. As expected, the number of tuples used for each variable increases with context depth k.

We use the k-this-sensitivity benchmark results with variable tuples

$[v, Pt(a), Pt(b), Pt(c), Pt(v)]$ to construct the five different memory data structures presented in Section 3.3. All constructed structures have a correct value semantic and are immutable. The main purpose of these structures is just to allow us to compute the corresponding memory size metrics.

3.4. Memory sizes for 3-this-sensitivity

Fig. 8 shows the memory sizes for each program in the benchmark when analyzed using 3-this-sensitivity (3-TS). The bars in the chart show the relative metrics (*MetricX*/*TableSize*) where we have used the result of the table metric *TableSize* as baseline. Hence, the value 1.10 for *TreeSize* in antlr means that *TreeSize* is 10% percent larger than *TableSize* in this case. Similarly, 0.11 for *ChiSize* in antlr means that *ChiSize* only need 11% percent of the memory used by *TableSize*.

The results are rather clear. According to the memory metrics we have a ranking of their memory usage as

$$\mathit{TreeSize}(1.06) > \mathit{MergeSize}(0.86) > \mathit{DoubleHash}(0.67) \gg \mathit{ChiSize}(0.09)$$

where the numbers in the parenthesis are the average values compared to our baseline *TableSize*. *TreeSize* requires on average 6% more memory space than *TableSize*. This can be explained by the fact that the tree metric also takes the number of edges into account, and that this additional memory cannot be compensated by quintuples having the same call history that can "share" the internal tree nodes.

MergeSize is by definition always as good and in most cases better than *TreeSize*. This is no surprise and the 19% reduction in memory size indicates that reusing leaf nodes is a simple, yet effective, way to reduce the memory costs. *DoubleHash* was initially designed to provide a speedy lookup. However, the 22% reduction in memory size compared to *MergeSize* shows that it is also rather memory efficient.

The representation with the smallest memory footprint (without any loss of precision), according to our memory metrics, is *ChiSize*. χ-terms use on average only 13% of the memory used by second best approach *DoubleHash*. In order to understand this result, we must take a closer look at what typical context-sensitive values look like.

The Single Path bars in Fig. 9 shows the percentage of *single path* values in the different benchmarks. A single path value is a context-sensitive value that in a tree representation only consists of a single tree path connecting the root and the leaf. Put in another way, a variable value is defined by a single quintuple $[v, ctx_1, ctx_2, ctx_3, val]$. It turns out that, on average, about 40% of all variable values are single path values. This type of value takes the memory size 7 (4 nodes + 3 edges) according to *TreeSize* and *MergeSize*, but only 1 according to *ChiSize*. The major difference is that χ-terms due to its redundancy elimination removes all internal nodes and edges and stores only the variable value.

The Single Value bars in Fig. 9 are even more important if we want to understand the memory efficiency of χ-terms. They show the percentage of variable values consisting of quintuples that, independent of what contexts are used, all result in the same value *val*. It turns out that (on average) 68% of all variable values in a 3-this-sensitive analysis are of type single value. And then we should still have in mind (see Table 1) that the context-sensitive value for each variable on average is defined by 23.4 quintuples.

The fact that 68% of all variables do not benefit from a points-to analysis using a call-depth $k = 3$ might be used as an argument to say that adding context-sensitivity does not pay off as suggested in Lhoták and Hendren (2008a). However, Lundberg and Löwe (2013) found that being able to increase the call-depth beyond $k = 1$ is important in certain types of applications that require very fine-grained context information.

In any case, these numbers show that context-sensitive variable values have in general a rather complex structure when presented as a tree (*TreeSize*, *MergeSize*) or a double hash map (*DoubleHash*), 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.

The Single Value result in Fig. 9 shows that χ-terms can identify and reduce the size of analysis values that are context independent (i.e. same value in all contexts). The two remaining bars (Depth Reduction, Redundancy Elimination) show that the redundancy elimination in χ-terms also reduces the size of truly context-sensitive values. Depth Reduction (a superset of Single Value) shows that χ-terms manage to reduce the χ-term depth in no less than (on average) 97.6% of all variable values, and Redundancy Elimination (a superset of Depth Reduction) shows that the χ-term construction of every single variable value, in all benchmarks (average 100%), involves at least one redundancy elimination. Hence, according to our memory size metrics, and for 3-this-sensitivity, χ-term are not only very good at reducing the memory size for context independent values, they manage also to reduce the size of more complex context-sensitive analysis values.

3.5. Memory sizes for different this-sensitivity

We repeated our experiments using the same memory metrics and the same benchmark programs, but now for 1- and 2-this-sensitivity. A summary of the results for $k = 1, 2, 3$ is presented in Table 2.

We find that their mutual rankings based on their memory footprints are maintained. *ChiSize* still requires significantly less

[3] Ashes Suite: http://www.sable.mcgill.ca/ashes, SPEC JVM98: http://www.spec.org/osg/jvm98, DaCapo suite: http://www.dacapobench.org.

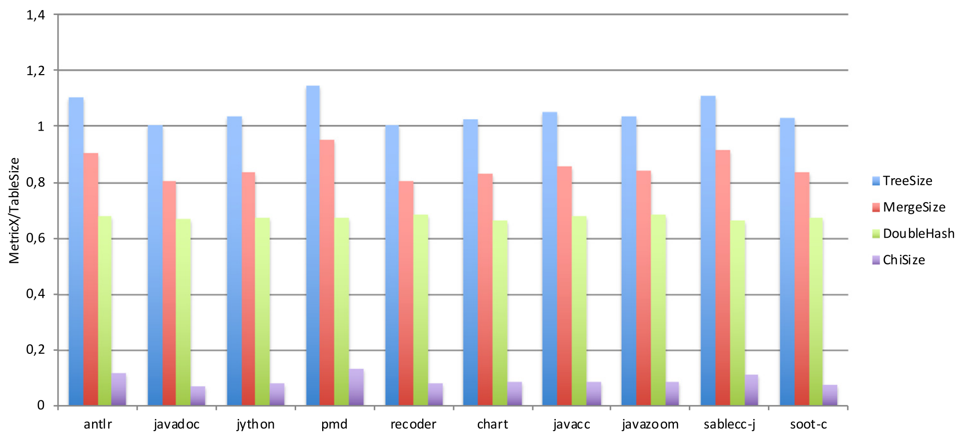


Fig. 8. Relative memory size for 3-this-sensitivity.

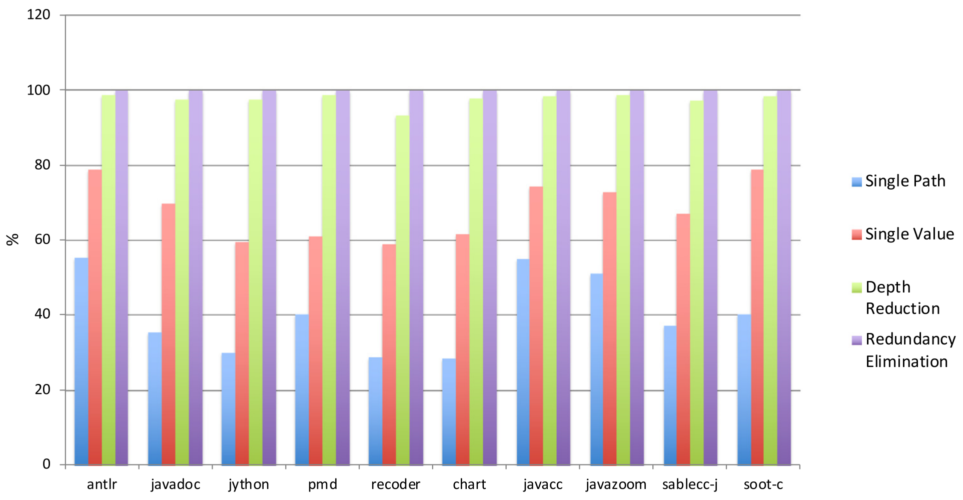


Fig. 9. Various metrics showing how redundancy elimination reduces the χ-term memory footprint.

Table 1
Benchmark information and relative tuple counts.

	General			*Relative tuple count*		
Program	Classes	Methods	SSA Variables	1-TS	2-TS	3-TS
Antlr	691	4,122	21,435	3.1	6.2	13.5
Chart	1,002	7,192	41,876	4.0	13.1	33.2
Javadoc	897	5,648	27,962	3.3	9.6	28.0
Jython	969	7,154	33,446	2.8	8.7	22.5
Sablecc-j	995	6,476	34,339	3.1	8.4	16.8
Soot-c	1,294	6.731	34,094	3.0	9.7	24.3
Javacc	691	4,710	20,021	5.0	12.3	30.0
Javazoom	765	4,917	22,319	3.1	6.7	18.7
Recoder	1,256	10,052	58,913	3.1	12.1	34.4
Pmd	987	6,503	33,363	3.2	6.8	12.6
Average				**3.4**	**9.4**	**23.4**
Median				**3.2**	**9.7**	**26.2**

Table 2
Summary of ranking results for memory usage (*MetricX/TableSize*).

Analysis	TreeSize	MergeSize	DoubleHash	ChiSize
1-TS	1.43	1.13	1.04	0.77
2-TS	1.22	0.98	0.80	0.24
3-TS	1.06	0.86	0.67	0.09

memory than the other ones. However, since the maximum depth is lower and number of tuples per variable is much reduced (see Table 1), the effect of the redundancy elimination is not as pronounced for the lower values of *k*.

Finally, Fig. 10 shows a summary of all memory related experiments. In this case we have chosen memory size as an indicator for memory cost. The *y*-axis shows the average relative memory costs for the different memory size metrics using the context-insensitive analysis (Ins, size 1 by definition) as our baseline. *TableSize*, *TreeSize*, *MergeSize*, and *DoubleHash* show a fast growth rate in memory costs when the call-depth (and the number of contexts) is increased, indicating that a further increase of the call-depth would be very costly when using any of these data

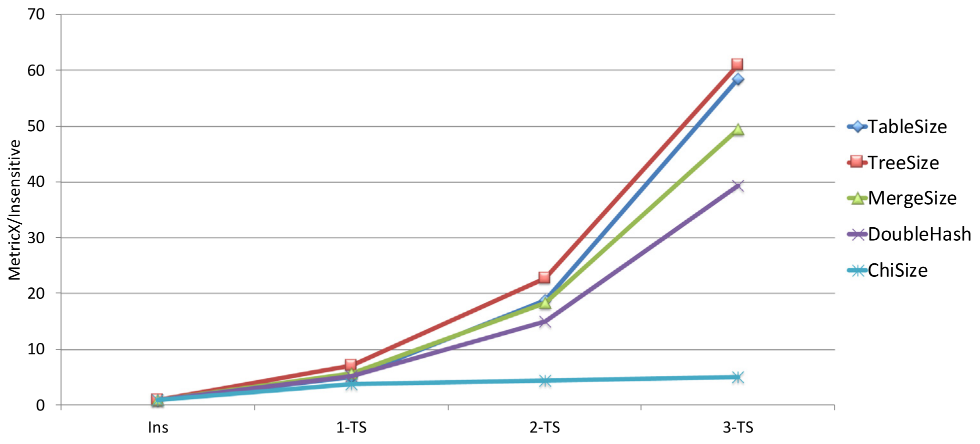


Fig. 10. Relative memory sizes for various analyses (Ins, 1-TS, 2-TS, 3-TS).

structures to handle the context-sensitive program information. χ-terms (*ChiSize*) on the other side shows (due to redundancy elimination) a very moderate growth rate when we increase the call-depth, indicating that a further increase in the call-depth ($k \geq 4$) could be possible.

4. Relation to previous work

In this paper we reuse the informal description and definitions of χ-terms from our conference paper (Trapp et al., 2015). A formal presentation of χ-terms and a framework for context-sensitive program analyses based on χ-terms, are presented in Hedenborg et al. (2021).

Rüthing and Steffen (1999), Knoop and Rüthing (2000) 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. (2010) 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.[4]

Heinze and Amme (2016) 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 (2002), Zhu and Calman (2004) 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 modeled 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. (2003) 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 (2004) 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, Lhoták and Hendren (2006) conclude that the best method for points-to analysis is object-sensitive (Milanova et al., 2005b). The usage of BDDs in capturing the 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 (Lhoták and Hendren, 2008b). 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 (Ball and Rajamani, 2001), a path-sensitive inter-procedural data-flow analysis tool for C programs. It adds data-flow facts to each vertex

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

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 papers above 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. Applying it to points-to analysis as one of many possible examples.

Kim et al. (2018) 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.

5. 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. The memory problem related to context-sensitive analyses is not just a theoretical issue, several papers (including, e.g., Lhoták and Hendren (2008a), Lundberg and Löwe (2013)) evaluating various inter-procedural context-sensitive data-flow approaches report severe memory problems when using call-depths $k \geq 2$, and the path-explosion problem, cf. (Cadar and Sen, 2013; Boonstoppel et al., 2008), has for a long time been a major issue in program verification and model checking.

This paper presents χ-terms as a mean 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 (Bryant, 1986, 1992). This leads to a very memory efficient representation of context-sensitive analysis information. Furthermore, operations over context-sensitive analysis values, needed to implement transfer functions of the analysis, are defined using restriction and Shannon expansion, which are also similar to the corresponding OBDD operations.

For languages with conditional execution, the number of different contexts grows, in general, exponentially with the program size. Adding iterations lead, in general, to countably (infinitely) many contexts.

In order to show the memory efficiency of χ-terms, we set up an experiment, using context data from an existing points-to analysis, to compare the memory usage of χ-terms with four other data structures for saving context-sensitive information. Our experiments show that χ-terms outperform all of the presented alternatives in terms of memory efficiency. In the case of a points-to analysis with call-depth 3, χ-terms used only 13% of the memory of the second-best data structure without any loss of precision. In a context-sensitive data-flow analysis, this memory gain can be used to increase the number of contexts that will be used in the analysis phase, e.g. by increasing the call-depth, thus allowing a more precise analysis.

Evaluating the memory efficiency using an implementation of χ-terms, in concrete analysis scenarios is future work. Experiments for future work would apply the methodology for some other types of inter-procedural analysis (e.g., for static garbage collection or synchronization barrier removal) and to intra-procedural value analysis.

CRediT authorship contribution statement

Mathias Hedenborg: Contributed with definition and evaluation of the experiments, Writing the original draft and the revision, Internal validation. **Jonas Lundberg:** Contributed with definition, Implementation, Conduction, Evaluation of the experiments, Writing the original draft and the revision, Internal validation. **Welf Löwe:** Contributed with definition and evaluation of the experiments, Writing the original draft and discussions for the revision, Internal validation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Akers, S.B., 1978. Binary decision diagrams. IEEE Trans. Comput. 27, 509–516.

Alpern, B., Wegman, M.N., Zadeck, F.K., 1988. Detecting equality of variables in programs. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 1–11.

Ball, T., Rajamani, S.K., 2001. Bebop: A path-sensitive interprocedural dataflow engine. In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, New York, NY, USA, pp. 97–103.

Berndl, M., Lhotak, O., Qian, F., Hendren, L., Umanee, N., 2003. Points-to analysis using BDDs. In: Proceedings of the Conference on Programmimg Language Design and Implementation (PLDI'03), pp. 103–114.

Boonstoppel, P., Cadar, C., Engler, D., 2008. Rwset: Attacking path explosion in constraint-based test generation. In: Ramakrishnan, C.R., Rehof, J. (Eds.), Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings, 14th International Conference. TACAS 2008, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 351–366.

Bryant, R.E., 1986. Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. C-35, 677–691.

Bryant, R.E., 1992. Symbolic boolean manipulation with ordered binary decision diagrams. ACM Comput. Surv. 24, 293–318.

Cadar, C., Sen, K., 2013. Symbolic execution for software testing: Three decades later. Commun. ACM 56, 82–90.

Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, K., 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13, 451–490.

Harris, W.R., Sankaranarayanan, S., Ivančić, F., Gupta, A., 2010. Program analysis via satisfiability modulo path programs. In: Hermenegildo, M.V., Palsberg, J. (Eds.), Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17–23, 2010. ACM, pp. 71–82.

Hasti, R., Horwitz, S., 1998. Using single static assignment form to improve flow-insensitive pointer analysis. In: Procedings of the Conference on Programmimg Language Design and Implementation (PLDI'98), pp. 97–105.

Hedenborg, M., Lundberg, J., Löwe, W., 2021. A framework for memory efficient context-sensitive program analysis. J. Theory Comput. Sci. (reviewing is ongoing).

Heinze, T.S., Amme, W., 2016. Sparse analysis of variable path predicates based upon ssa-form. In: Margaria, T., Steffen, B. (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10–14, 2016, Proceedings, Part I, 7th International Symposium. ISoLA 2016, Springer International Publishing, Cham, pp. 227–242.

Kim, S.W., Rival, X., Ryu, S., 2018. A theoretical foundation of sensitivity in an abstract interpretation framework. ACM Trans. Program. Lang. Syst. 40, 13:1–13:44.
Knoop, J., Rüthing, O., 2000. Constant propagation on the value graph: Simple constants and beyond. In: Watt, D. (Ed.), Compiler Construction. In: Lecture Notes in Computer Science, vol. 1781, Springer Berlin Heidelberg, pp. 94–110.
Lhoták, O., Hendren, L., 2003. Scaling Java points-to analysis using Spark. In: Proceedings of the International Conference on Compiler Construction, CC'03, pp. 153–169.
Lhoták, O., Hendren, L., 2006. Context-sensitive points-to analysis: Is it worth it? In: Proceedings of the 15th International Conference on Compiler Construction. Springer-Verlag, Berlin, Heidelberg, pp. 47–64.
Lhoták, O., Hendren, L., 2008a. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Methodol. 18, 1–53.
Lhoták, O., Hendren, L., 2008b. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. ACM Trans. Softw. Eng. Methodol. 18, 3:1–3:53.
Lundberg, J., 2004. Points-To Analysis for Software Engineering (Licentiate thesis). Växjö University.
Lundberg, J., 2014. Fast and Precise Points-to Analysis (Ph.D. thesis). Linnaeus University.
Lundberg, J., Gutzmann, T., Edvinsson, M., Löwe, W., 2009. Fast and precise points-to analysis. J. Inf. Softw. Technol. 51, 1428–1439.
Lundberg, J., Löwe, W., 2007. A Scalable Flow-Sensitive Points-To Analysis. Technical Report, Matematiska och systemtekniska institutionen, Växjö Universitet, URL: http://w3.msi.vxu.se/~wlo/files/goos07.pdf.
Lundberg, J., Löwe, W., 2013. Points-to analysis: A fine-grained evaluation. J. UCS 18, 2851–2878.
Marlowe, T., Ryder, B., 1990. Properties of data flow frameworks: A unified model. Acta Inform. 28, 121–163.
Milanova, A., Rountev, A., Ryder, B.G., 2005a. Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. 14, 1–41.
Milanova, A., Rountev, A., Ryder, B.G., 2005b. Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. 14, 1–41.
Muchnick, S.S., 1997. Advanced Compiler Design Implementation. Morgan Kaufmann Publishers, San Francisco, California.
Rival, X., Mauborgne, L., 2007. The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. 29.
Rüthing, J., Steffen, B., 1999. Detecting equalities of variables: Combining efficiency with precision. In: Cortesi, A., Filé, G. (Eds.), Static Analysis. In: Lecture Notes in Computer Science, vol. 1694, Springer Berlin Heidelberg, pp. 232–247.
Shivers, O., 1991. Control-Flow Analysis of Higher-Order Languages. Technical Report (Ph.D. thesis). Carnegie-Mellon University, CMU-CS-91-145.
Trapp, M., 1999. Optimerung Objektorientierter Programme (Ph.D. thesis). Universität Karlsruhe.
Trapp, M., Hedenborg, M., Lundberg, J., Löwe, W., 2015. Capturing and manipulating context-sensitive program information. In: Software Engineering Workshops 2015, vol. 1337. pp. 154–163, URL: http://ceur-ws.org/Vol-1337/.
Whaley, J., Lam, M.S., 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. SIGPLAN Not. 39, 131–144.
Zhu, J., 2002. Symbolic pointer analysis. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design. ACM, New York, NY, USA, pp. 150–157.
Zhu, J., Calman, S., 2004. Symbolic pointer analysis revisited. SIGPLAN Not. 39, 145–157.

MSc Mathias Hedenborg has been working as a Senior Lecturer at Linnaeus University in Växjö (Sweden) since 1980. He is now studying for a Ph.D. in Computer Science at Linnaeus University. Before, he studied and received his M.Sc. at Högskolan in Växjö, received a degree in Teacher Training from Gothenburg University. He is interested in operating systems, computer networks and software analysis.

Dr Jonas Lundberg is an Associate Professor at the Computer Science Department at Linnaeus University in Växjö (Sweden), Before, he studied at Umeå University where he received a Ph.D. in Theoretical Physics in 1995. In 2014 he had a second Ph.D. (Computer Science) at Linnaeus University. He is currently interested in research at the intersection of machine learning, program analysis, and software engineering.

Prof. Dr. Welf Löwe holds the chair in software technology at Linnaeus University in Växjö (Sweden) since 2002. Before, he studied at TU Dresden (Germany), received a Ph.D. from TH Karlsruhe (Germany), was postdoc at ICSE Berkeley (CA, USA), and assistant professor at TH Karlsruhe. He is interested in technology for software construction, analysis, optimization, and runtime support. He is also co-founder of Softwerk, Aimo, and DueDive.