Capturing and manipulating context-sensitive program information

Martin Trapp

Mathias Hedenborg

Jonas Lundberg

Welf Löwe

Senacor Technologies AG Martin.Trapp@senacor.com Linnaeus Universitet, Software Technology Lab, {Mathias.Hedenborg|Jonas.Lundberg|Welf.Lowe}@lnu.se

Abstract

Designers of context-sensitive program analyses need to take special care of the memory consumption of the analysis results. In general, they need to sacrifice accuracy to cope with restricted memory resources. We introduce χ terms as a general data structure to capture and manipulate context-sensitive analysis results. A χ -term is a compact representation of arbitrary forward program analysis distinguishing the effects of different control-flow paths. While χ -terms can be represented by trees, we propose a memory efficient representation generalizing ordered binary decision diagrams (OBDDs).

1 Introduction

Program analysis is an important part of optimizing compilers and software engineering tools. 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., 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. For languages with conditional execution leads, in general, to countably (infinitely) many contexts. Hence, merging the analysis values of different contexts is needed for analyzing relevant programs of any realistic programming language. Having said this, we ignore this merging throughout the remainder of the present paper. Instead, we focus on capturing context-sensitive analysis information, i.e. contexts and analysis values for each program point, in a memory efficient way. In other words, we strive to delay merging the analysis values of different context for increasing analysis precision.

Our approach is based on so-called χ -terms that capture the analysis results of different contexts. We introduce a χ -operation over analysis values as the analysis semantics of ϕ -nodes in a Static Single Assignment (SSA) [CFR⁺91] representation of a program. These χ -terms can be implemented without any redundancy by exploiting a generalization of the idea behind Binary Decision Diagrams (BDDs) [Ake78]. This paper is very much inspired by the ideas first presented by Martin Trapp in his dissertation [Tra99].

The remainder of the paper is structured as follows: In Section 2, we give an informal introduction to χ -terms. In Section 3, we define χ -terms and introduce a number of basic concepts and notations. In Section 4, we discuss operations on χ -terms and introduce the concept of χ -induced operators. In Section 5, we conclude the paper.

2 Informal overview over χ -terms

In this section, we give an informal overview over χ -terms. The aim here is merely to introduce and motivate the idea. A more formal presentation will be given in the next section. We use a SSA-based program representation and we will refer to the used program representation as the SSA graph.

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

Submission to: 8. Arbeitstagung Programmiersprachen, Dresden, Germany, 18-Mar-2015, to appear at http://ceur-ws.org

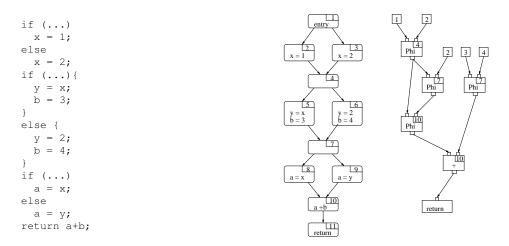


Figure 1: A source code example with corresponding basic block and SSA graph structures.

In Figure 1, we show a simple piece of code containing three if-statements and the corresponding basic block structure and SSA graph. In the SSA graph, we have annotated each ϕ -function Phi^b with their basic block number b. The leaves 1, 2, 3 and 4 (in boxes) in the rightmost part of the figure represent integer values.

The ϕ -functions in the SSA graph forbid static data-flow analysis to select a unique definition of a given variable. For example, the value of variable x in Figure 1 depends on which branch of the first *if*-statement that is chosen. This uncertainty, which we in general can not resolve, is in the SSA graph symbolized by the node Phi^4 .

In a context-insensitive data-flow analysis, the interpretation of the ϕ -functions assume that any possible definition is a value of that variable. This approach results in a set approximation of the run-time values. For example, the value of the variables x, y, and a in Figure 1 is $\{1, 2\}$, and the value of b is $\{3, 4\}$. Notice that in the context-insensitive interpretation, the variable value sets resulting from a join point in the control-flow graph do not contain any information about the different flow-path options that generated all the different values. It simply contains the possible values.

Context-sensitive data-flow analyses are more precise and map control-flow path(s) to possible value(s). As the number of possible paths grows exponentially with the number of if-statements (and is even unbounded if loops are involved) we should aim for compact representations of this mapping. Such a compact context-sensitive description of a given variable value can be achieved with χ -functions.

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 Figure 1 using χ -functions as $b = \chi^7(3, 4)$. Interpretation: variable *b* has the value 3 if 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. Using this approach we can write down the χ -term values for the variables in Figure 1 as:

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

Notice that values of variables depending on several control-flow options (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 -functions representing the different options.

2.1 χ -term Construction

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.

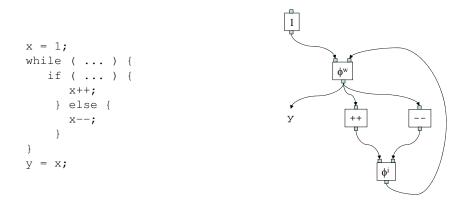


Figure 2: A simple code fragment with a loop that encloses an *if*-statement and the corresponding SSA graph.

Loops in the control-flow cause some problems as they would lead to ever growing χ -term, i.e., analysis would not reach a fixed point. We will ignore the termination problem for now, and discuss and solve this problem later. For a first try, we may have a (possibly countable) number of χ -terms with the same block number *b* since every ϕ -node in *b* generates a new χ -term value when the analysis reaches that block.

2.2 Switching Behavior

As discussed in the introduction to Section 2, χ -term construction is tightly coupled to a context-sensitive analysis. A new set of χ -terms is generated every time t the analysis reaches a block b containing ϕ -nodes. We denote this set with χ_t^b .

All χ -terms generated of block *b* at time *t* have the same *switching behavior*. That is, for any two χ -terms $\chi_i^b(x_1, \ldots, x_n)$ and $\chi_j^b(y_1, \ldots, y_n)$, and for any execution of the program, it holds that if i = j and branch x_k in $\chi_i^b(x_1, \ldots, x_n)$ is selected, then also branch y_k in $\chi_j^b(y_1, \ldots, y_n)$ is selected. This property will be important later on when we introduce operations on χ -terms. Anyhow, two χ -terms generated from the same block but at different times of the analysis will not necessarily have the same switching behavior.

This situation is illustrated in Figure 2 where we show a while-loop that encloses an if-statement that assigns a new value to a variable x. At first entry of a block w containing the while loop header, we assign x the value $\chi_0^w(1, \bot)$ where the bottom element \bot symbolizes "value undefined". This value is later propagated inside the loop body where the if-statement is handled. After the if statement (in block i) we assign x the value $\chi_0^i(\chi_0^w(2, \bot), \chi_0^w(0, \bot))$, a value that is later propagated to the loop header block w. Using this approach we can compute all possible values for x after an arbitrary number of loop iterations. The first three x values that might escape the loop are:

There are a few things to notice. First, x_n^w is the value of x after the *n*:th loop iteration. Secondly, the χ -term values x_n^w can be interpreted as: Either x has the value 1, or the value after $1, \ldots, n$ loop iterations, i.e., $x \in [-(n-1), n+1]$ after n loop iterations.

Finally, χ -terms generated at different iterations of the loop do not have the same switching behavior. For example, the two χ -functions χ_0^i and χ_1^i will have different switching behavior since we can not say that if the control flows through one branch of the if-statement in one iteration, it will flow through the same branch in the following iteration. Thus, only two χ -terms $\chi_i^b(x_1, \ldots, x_n)$ and $\chi_j^b(y_1, \ldots, y_n)$ from the same block b, with equal iteration index (i.e. i = j), will have a similar switching behavior.

2.3 χ -term Representation

Every term can be naturally viewed as a tree and χ -terms are no exception. This is illustrated in Figure 3 (left) where we show the tree representation of the χ -term values for the variable *a* in Figure 1. 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.

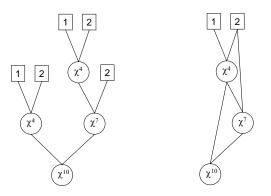


Figure 3: Term for variable $a = \chi^{10}(\chi^4(1,2),\chi^7(\chi^4(1,2),2))$ is illustrated by its tree and directed acyclic graph representations (edge direction from bottom to top).

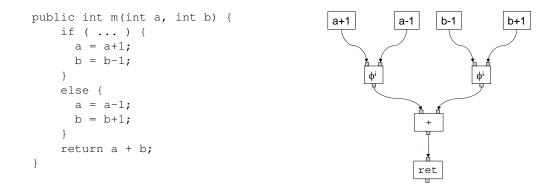


Figure 4: A method with the corresponding SSA graph of the basic block containing the return-statement.

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. More cost efficient representations are available [Bry86, Bry92]. They avoid redundant subtrees, cf. Figure 3 (right).

2.4 Operations on χ -terms

The code fragment on the left-hand side of Figure 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:

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

This situation is illustrated in Figure 5 where we now consider the addition as an operator

$$+: X_{int} \times X_{int} \mapsto X_{int}$$

working on χ -terms rather than integers.

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:

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

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

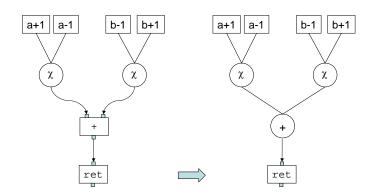


Figure 5: Symbolic input of the "+"- and the "ret"-SSA-nodes in Figure 4.

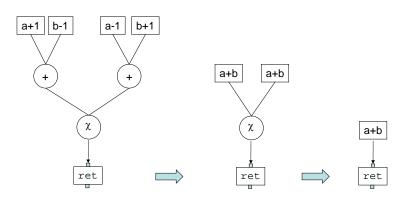


Figure 6: The Shannon expansion can be used to "push" operators toward the leaf values where the context-insensitive operator definitions can be applied.

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 Figure 6. 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 -function χ_j^b . This rewrite rule for χ -term expressions is called a *Shannon expansion*¹.

Finally, once we applied the Shannon expansion of the operator + in Figure 4, we are 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. The resulting simplifications are shown in Figure 6 where we also used the redundancy rule $\chi(t,t) = t$. This manipulation can symbolically be written as:

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

= $\chi(a+b,a+b)$
= $a+b$

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.

Finally, it should be noted that our presentation of χ -terms as a tree, and all operations on χ -terms as tree manipulations, is very much inspired by *Orded Binary Decision Diagrams*² (OBDD) as presented in [Bry86, Bry92]. Furthermore, the

¹The word "Shannon expansion" is taken from the OBDD literature where a similar procedure is used to rewrite boolean functions represented as OBDDs.

²An OBDD is a clever (and slim) representation of Boolean functions as a directed acyclic graph that allows an efficient computation of frequently used operations such as disjunction, conjunction and function composition.

algorithms describing various operations on χ -terms presented in this chapter are intended to be easy to understand, more efficient algorithms can be found in the above cited papers.

3 Definition and Notations

In this section, we present a number of definitions and notations related to χ -terms that are of general interest and that will be used in the remainder of this paper later on. Many of the ideas and notations are taken from [Tra99] and [Bry86, Bry92] and adjusted to fit our needs.

3.1 χ -functions

The set of χ -functions depends on the set of ϕ -functions which depends on the program we are analyzing. Furthermore, the exact number of χ -functions at use depends on how we handle loops and other approximations we do in the analysis (e.g. finite k approximations). To simplify what follows, we assume that each program p has a (possibly infinite) set of χ -functions $\mathcal{X}(p)$.

Each χ -function $\chi_i^b \in \mathcal{X}(p)$ is identified by a pair (b, j) where:

- 1. The *block number* $b = block(\chi_j^b), b \in [1, B_p]$ indicates in what basic block its generating ϕ -node is contained. Here B_p is the number of basic blocks in a program p.
- 2. The *iteration index* $j = index(\chi_j^b)$ indicates on what analysis iteration over block b the χ -function was generated. The χ -functions generated from the first analysis iteration over b always have iteration index 0.

Two χ -terms $\chi_i^b(x_1, \ldots, x_n)$ and $\chi_j^b(y_1, \ldots, y_n)$ 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

branch x_k is selected \Leftrightarrow branch y_k 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.

Finally, the *arity* of a χ -function, denoted $arity(\chi_i^b)$, with n arguments is n.³

3.2 *χ***-terms**

A programming language type is a set of concrete values. In program analysis, concrete values are abstracted to abstract values requiring a Galois connection between concrete and abstract types [CC77]. Associated with each abstract type V, we have a set of χ -terms. These χ -terms form the context-sensitive counterpart to the abstract values V as used in a context-insensitive analysis.

Definition 1 (χ -terms) Let $\mathcal{L}_V = \{V, \Box, \sqcup, \top, \bot\}$ be a lattice⁴ of abstract values of type V and let $\mathcal{X}(p)$ be the set of χ -functions associated with a given program p. The set of all χ -terms $X_V(p)$ over the abstract values V is then defined recursively as:

$$v \in V \quad \Rightarrow \quad v \in X_V(p)$$

$$t_1, \dots, t_n \in X_V(p) \land \chi_j^b \in \mathcal{X}(p) \quad \Rightarrow \quad \chi_j^b(t_1, \dots, t_n) \in X_V(p), n \in arity(\chi_j^b)$$

Notice, even if the set of abstract values V is finite, $X_V(p)$ of a program with loops is infinite since there is no upper limit on the composition depth.

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

Definition 2 (Block number) The block number of a χ -term $t \in X_V$, denoted block(t), is defined as:

$$block(t) = \begin{cases} 0 & \text{if } t \in V, \\ b & \text{if } t = \chi^b(t_1, \dots, t_n). \end{cases}$$

³The arity is independent of the loop iteration j. However, we index χ_j^b with j to be consistent in our notations.

⁴The lattice property is introduced because it is required to widen the abstract values. This, in turn, is required for an analysis to terminate (in general). However, widening and termination of analysis are deliberately omitted from the present discussion.

Notice that we have associated all the values $v \in V$ with a (default) block number 0 not corresponding to any existing block in the program.

Definition 3 (Function set) The function set of a χ -term $t \in X_V(p)$, denoted func(t), is the set of all χ -functions $\chi_j^b \in \mathcal{X}(p)$ used to construct t.

For example:

$$t = \chi^3(\chi^1(1,2),\chi^2(\chi^1(3,4),2)) \Rightarrow func(t) = \{\chi^3,\chi^2,\chi^1\}$$

From now on we will drop the reference to a specific program p in $\mathcal{X}(p)$ and $X_V(p)$ and assume that it is implicit. It should by now be clear that all these entities are directly related to a given analysis instantiation.

3.3 The Tree Representation of χ -terms

Each χ -term $t \in X_V$ has a unique tree representation $G_t = \{N, E, r\}$ where $N \subseteq X_V$ is a set of nodes, $E \subseteq X_V \times X_V$ is a set of edges, and $r \in N$ is the root of the tree. The two sets N and E can be defined recursively as;

$$r = t,$$

$$n \in N \implies n_i \in N \land (n, n_i) \in E.$$

where n_i denotes the *i*:th argument of $n = \chi_i^b(n_1, \ldots, n_n)$.

Our reason for introducing the tree representation of a χ -term is that many notions and χ -term operations are easiest to understand in terms of tree manipulations. Basic operations include

- The *leaves* of a χ -term $t \in X_V$, denoted $leaves(t) \subseteq V$, is the set of all leaf values in the tree representation of t.
- The subterms of a χ -term $t \in X_V$, denoted subterms $(t) \subseteq X_V$, is the set of all subtrees of t in the tree representation of t. The leaves and t are not included.
- The *children* of χ -term $t \in X_V$, denoted *children* $(t) \subseteq X_V$, is defined as:

$$children(t) = \begin{cases} \emptyset & \text{if } t \in V \\ \{t_1, \dots, t_n\} & \text{if } t = \chi_j^b(t_1, \dots, t_n) \end{cases}$$

- The *depth* of a χ -term $t \in X_V$, denoted *depth*(t), is the depth of the tree representation of t.
- Let $n \in X_V$ be a node in the tree representation of a χ -term $t \in X_V$. The *depth* of n in t, denoted *depth*(t, n) = depth(t) depth(n), is the depth of the node n in the tree representation of t.

For example,

$$t = \chi^{10}(\chi^4(1,2),\chi^7(\chi^4(3,4),2)) \Rightarrow \begin{cases} leaves(t) = \{1,2,3,4\} \\ subterms(t) = \{\chi^4(1,2),\chi^7(\chi^4(3,4),2),\chi^4(3,4)\} \\ children(t) = \{\chi^4(1,2),\chi^7(\chi^4(3,4),2)\} \\ depth(t) = 3, \\ depth(t,\chi^4(3,4)) = 2 \end{cases}$$

3.4 Basic χ -term operations

In this section we present further basic operations on χ -terms. Most important is the *Shannon expansion* which can be used to manipulate χ -term expressions without affecting their value.

Definition 4 (Restriction) 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 $\chi_i^b(t_1, \ldots, t_n)$ with switching behavior (b, j) in t has been replaced by its k:th child t_k .

For example:

$$t = \chi_0^3(\chi_0^1(1,2),\chi_0^2(\chi_0^1(1,2),2)) \Rightarrow \begin{cases} t|_{(3,0):2} = \chi_0^2(\chi_0^1(1,2),2) \\ t|_{(1,0):1} = \chi_0^3(1,\chi_0^2(1,2)) \end{cases}$$

A restriction $f|_{x_i=true}$ in the OBDD description of a Boolean formula $f(x_1, \ldots, x_n)$ is a new formula where the *i*:th variable x_i has been assigned the constant value true. The χ -term counterpart is that we define a new χ -term $t|_{(b,j):k}$ by only considering the value from the k:th predecessor of block b (in iteration j) in the construction of t. It should also be noticed that if we restrict us to the k:th branch of χ_j^b , when $\chi_j^b \notin func(t)$, t is left unaffected.

Definition 5 (Equivalence) Two χ -terms are equivalent (\equiv) if they are (syntactically) the same or represent the same χ -term value, i.e. if they can be transformed into the same χ -term by sequences of Shannon expansions or redundancy eliminations.

Definition 6 (Shannon expansion) The Shannon expansion of a χ -term $t \in X_V$ over χ_i^b is a new χ -term defined as:

$$t = \chi_j^b(t|_{(b,j):1}, t|_{(b,j):2}, \dots, t|_{(b,j):n})$$
 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. For example,

$$t = \chi_1^3(\chi_1^1(1,2),\chi_1^2(\chi_1^1(1,2),2))$$

$$\equiv \chi_1^1(\chi_1^3(\chi_1^2(1,1),\chi_1^2(1,2)),\chi_1^3(\chi_1^2(2,2),\chi_1^2(2,2)))$$
(expansion over χ_1^1)

$$\equiv \chi_1^2(\chi_1^3(\chi_1^1(1,2),\chi_1^1(1,2)),\chi_1^3(\chi_1^1(1,2),\chi_1^1(2,2)))$$
(expansion over χ_1^2)

Definition 7 (Redundancy and redundancy elimination) A χ -term is redundant if all its sub-terms are equivalent. That is, $\chi_j^b(t_1, \ldots, t_n) \equiv t$ if $t_1 \equiv \ldots \equiv t_n$. The process of removing redundant χ -terms is called redundancy elimination.

Obviously, a χ -term t containing a redundant subterm χ^r can, without any loss of information, be reduced according to the pattern

$$t = \dots \chi^i(\dots, \chi^r(t, \dots, t), \dots) \dots \equiv \dots \chi^i(\dots, t, \dots) \dots$$

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

$$t = \chi_1^1(\chi_1^3(1,\chi_1^2(1,2)),\chi_1^3(2,\chi_1^2(2,2)))$$

$$\equiv \chi_1^1(\chi_1^3(1,\chi_1^2(1,2)),\chi_1^3(2,2)) \quad (\text{since } \chi_1^2(2,2) \equiv 2)$$

$$\equiv \chi_1^1(\chi_1^3(1,\chi_1^2(1,2)),2) \quad (\text{since } \chi_1^3(2,2) \equiv 2)$$

4 Operations on χ -terms

When doing a program analysis there is a need of defining an abstract operator related to an existing concrete operation. The aim of this section is to say that every context-insensitive operator τ has a χ -induced context-sensitive counterpart $\tilde{\tau}$ with a semantics completely determined by the semantics of τ . We will define $\tilde{\tau}$ in terms of an algorithm apply that recursively performs Shannon expansions of the operator $\tilde{\tau}$ over the χ -functions in the input χ -term values. We think of this process as if we "push" the operator towards the leafs of the input χ -term where the ordinary context-insensitive semantics of the operation can be applied. This idea is inspired by the ideas of evaluating logical formulas represented as OBDDs [Bry86, Bry92].

Definition 8 (Restriction) Let $\tilde{\tau} : X_1 \times \ldots \times X_n \mapsto X_v$ be an operator on χ -terms. The restriction of an operation $\tilde{\tau}(t_1, \ldots, t_n)$ to the k:th branch of χ_j^b , denoted $\tilde{\tau}|_{(b,j):k}$, is defined as

$$\widetilde{\tau}|_{(b,j):k}(t_1,\ldots,t_n) = \widetilde{\tau}(t_1|_{(b,j):k},\ldots,t_n|_{(b,j):k})$$

where $t_i|_{(b,j):k}$ is the restriction of the *i*:th operand t_i to the *k*:th branch of χ_j^b .

It simply means that we are restricting all input operands to the k:th branch of χ_j^b before we apply the operation $\tilde{\tau}$. Notice that if $\chi_j^b \notin func(t_1) \cup \ldots \cup func(t_n)$ we have $t_i|_{(b,j):k} = t_i$ and consequently that $\tilde{\tau}|_{(b,j):k} = \tilde{\tau}(t_1, \ldots, t_n)$. That is, the restriction has no effect in such cases.

Definition 9 (Shannon expansion) Let $\tilde{\tau} : X_1 \times \ldots \times X_n \mapsto X_v$ be an operator on χ -terms. The Shannon expansion of an operation $\tilde{\tau}(t_1, \ldots, t_n)$ over the χ -function χ_j^b is defined as

$$\widetilde{\tau}(t_1,\ldots,t_n) = \chi_j^b(\widetilde{\tau}|_{(b,j):1},\ldots,\widetilde{\tau}|_{(b,j):m})$$

where $m = arity(\chi_j^b)$ and $\tilde{\tau}|_{(b,j):k}$ is the restriction of the operation $\tilde{\tau}(t_1,\ldots,t_n)$ to the k:th branch of χ_j^b .

The Shannon expansion of an operation $\tilde{\tau}(t_1, \ldots, t_n)$ over χ_j^b can be seen as if we "push" the operator $\tilde{\tau}$ one step towards the leaf values of the input operands t_i by computing $\tilde{\tau}$ for each one of the branches of χ_j^b individually, and then merge these values using χ_j^b .

The Shannon expansion has the following properties:

- It is a rewrite rule for a χ -term expression that has no effect on the resulting χ -term value. It can be seen as a first step in the computation of the operation.
- It leaves the χ -expression unchanged if $\chi_j^b \notin func(t_1) \cup \ldots \cup func(t_n)$ since in that case $\tilde{\tau}|_{(b,j):k} = \tilde{\tau}(t_1, \ldots, t_n)$ and we can apply the redundancy rule

$$\chi_j^b(\widetilde{\tau}(t_1,\ldots,t_n),\ldots,\widetilde{\tau}(t_1,\ldots,t_n)) = \widetilde{\tau}(t_1,\ldots,t_n).$$

• If $\chi_i^b \in func(t_i)$ for any t_i in t_1, \ldots, t_n then

$$|func(\tilde{\tau}|_{(b,j):1}) \cup \ldots \cup func(\tilde{\tau}|_{(b,j):m})| < |func(t_1) \cup \ldots \cup func(t_n)|,$$

 $m = arity(\chi_j^b)$. That is, the "size" of the input-operands to the $\tilde{\tau}$ operator (in terms of the number of unique χ -functions contained) is always decreasing.

This last property indicates that repeated Shannon expansions over the χ -functions in the input operands will result in a situation where the input-operands to $\tilde{\tau}$ are all leaf values. Thus, the semantics of $\tilde{\tau}$ is completely defined in how $\tilde{\tau}$ operates on the leaf values. This observation leads to the concept of χ -induced operations to be presented shortly.

The algorithm apply (see Algorithm 1) is a recursive function that performs repeated Shannon expansion until it reaches the leaf values where the context-insensitive operator can be applied.

 $\label{eq:algorithm1} \begin{array}{l} \texttt{Algorithm1} \texttt{apply}(\tau, t_1, \dots, t_n) \mapsto t \\ \hline \texttt{let} func_set = func(t_1) \cup \dots \cup func(t_n) \\ \texttt{if} func_set = \emptyset \texttt{then} \\ \texttt{return} \, \tau(t_1, \dots, t_n) \\ \texttt{else} \\ \texttt{pick} \, \chi_j^b \texttt{ from } func_set \\ \texttt{for all } i \in 1 \dots arity(\chi_j^b) \texttt{do} \\ \texttt{let} \, c_i = \texttt{apply}(\tau, t_1|_{j:i}, \dots, t_n|_{j:i}) \\ \texttt{end for} \\ \texttt{return} \, \chi_j^b(c_1, \dots, c_{arity}(\chi_j^b)) \\ \texttt{end if} \end{array}$

The test $func_set = \emptyset$ checks if t_1, \ldots, t_n are all leaf values. If so, we apply the context-insensitive operator τ . Otherwise, we push the operator towards the leafs by a Shannon expansion in one of the remaining χ -functions. Note also that:

apply $(\tau, v_1, \ldots, v_n) = \tau(v_1, \ldots, v_n)$ if v_1, \ldots, v_n are all leaf values.

This follows immediately from the base case in the algorithm apply.

Finally, Algorithm 1 is intended to be easy to understand, not efficient, since it will be used in the definition of χ -induced context-sensitive operators. More efficient algorithms are available [Bry86].

Definition 10 (χ -induced context-sensitive operator) For each context-insensitive operator $\tau : A \times B \times ... \times N \mapsto V$ their is a corresponding χ -induced operator $\tilde{\tau} : X_A \times X_B \times ... \times X_N \mapsto X_V$ defined as

$$\widetilde{\tau}(t_1,\ldots,t_n) = \operatorname{apply}(\tau,t_1,\ldots,t_n).$$

One important observation is that the algorithm apply in general increases the depth of the resulting χ -term expression. More precisely, a closer look at apply shows that

$$depth(\widetilde{\tau}(t_1,\ldots,t_n)) = |func(t_1) \cup \ldots \cup func(t_n)|.$$

As a consequence, $\tilde{\tau}$ applied on operands with a similar control-flow history (i.e. $|func(t_1) \cup \ldots \cup func(t_n)| \ll |func(t_1)| + \ldots + |func(t_n)|$) will generate less complex results than if applied on operands with very different control-flow histories (i.e. $|func(t_1) \cup \ldots \cup func(t_n)| \approx |func(t_1)| + \ldots + |func(t_n)|$).

In conclusion, every context-insensitive operator τ has a χ -induced counterpart $\tilde{\tau}$ and its semantics is completely determined by the context-insensitive semantics of τ . We compute $\tilde{\tau}$ by repeated Shannon expansions where each step means that we are "pushing" $\tilde{\tau}$ closer to the leaf values where the context-sensitive operator τ is defined and can be applied.

5 Conclusions

We defined χ -terms as a means to capture context-sensitive analysis values for programs represented as SSA-graphs. Each program point, i.e., each SSA-node, is mapped to a corresponding χ -term representing a context-sensitive analysis value for that point. Especially, each meet point of execution paths in the program, i.e., each *phi*-node, is mapped to a χ -term whose children represent the alternative analysis values of these paths. χ -terms are represented by graphs without any redundancy which generalizes the idea behind BDDs. This leads to a memory efficient representation of context-sensitive analysis information. Operations over context-insensitive analysis values, needed to implement transfer functions of the analysis, are defined using restriction and Shannon expansion which are also similar to the corresponding BDD operations. As a result, any context-insensitive analysis can automatically be transformed into a corresponding context-sensitive analysis.

We deliberately ignored the merging of analysis values of different contexts, which is needed for making contextsensitive analysis practical for large and iterative programs; otherwise even χ -terms grow beyond any memory bound. Merging, and, hence, the reduction memory requirements of χ -term at the price of analysis precision, can easily be achieved by applying the meet operation of the context-insensitive analysis values to leafs of χ -(sub-)terms. Controlling this merging of sub-terms based on available memory and precision of analysis values is matter of future work.

References

- [Ake78] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, June 1978.
- [Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys, 24(3):293–318, September 1992.
- [CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. In *Conference Record of the Fourth Annual ACM* SIGACT/SIGPLAN Symposium on Principles of Programming Languages, pages 238–252, January 1977.
- [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.
- [Tra99] Martin Trapp. Optimerung Objektorientierter Programme. PhD thesis, Universität Karlsruhe, December 1999.