Summarized Dimensions Revisited

Holger Siegel[1] and Axel Simon[1]

Technische Universität München
85748 Garching, Germany

Abstract

The task of summarizing memory regions becomes increasingly important in the analysis of data structures and the analysis of recursive functions. Calculating summaries of memory regions containing several abstract variables becomes quite subtle when using relational abstract domains. We address this challenge using two principal operations, *fold* and *expand*, that calculate such summaries for any numeric relational domain and argue optimality. We thereby generalize an earlier approach in the literature that was geared towards memory cells modelled by a single abstract variable.

Keywords: abstract interpretation, numeric domains, summarization

1 Introduction

Abstract interpretation is increasingly applied in the context of software verification. For this task, expressive abstract domains are required that are able to express relational information, that is, how the value of one variable is restricted by the value of another variable. For instance, when $x = y$ is known, restricting x by a test $x \leq 5$ will also restrict y such that $y \leq 5$. Also, as the number of memory cells allocated in a program cannot be bounded a priori, it is crucial to summarize the information contained in different memory cells into one abstract memory cell to ensure tractability.

Suppose that x_1 and x_2 represent the content of two memory cells that are to be summarized into x_s where $x_1 \in [0, 5]$ and $x_2 \in [5, 10]$ are kept in the standard interval domain [4]. Here, it seems obvious to merge the information on x_1 and x_2 by using the join operation of the interval domain, leading to $x_s \in [0, 10]$. Indeed, this approach was proposed by Gopan et al. [3] who

[1] Email: firstname.lastname@in.tum.de. Supported by DFG EN SI 1579/1.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.com/locate/entcs

addressed the task of summarizing the cells of arrays. However, the task of summarizing memory cells becomes more intricate when using a relational domain, for instance, linear inequalities [2]. Suppose now that x_1, y_1 represent the first memory cell and x_2, y_2 represent the second memory cell that are to be summarized into the cell x_s, y_s where $x_i = y_i + 5$ and $y_1 \in [0, 5], y_2 \in [5, 10]$. One way forward is to extract the intervals of x_1 and of x_2 and to set x_s to their join $[5, 15]$ (analogously for y_1, y_2 and y_s). However, the resulting values for x_s, y_s lack the relational property $x_s = y_s + 5$ that held for both, x_1, y_1 and x_2, y_2. In this paper we tackle the challenge of defining a summary operation that retains relational information between all components of a memory cell. Moreover, we discuss different options of summarizing information in memory cells and their precision trade-offs.

Overall, this paper makes the following three contributions:

- We present different ways of summarizing relational numeric content of memory cells and their impact on the precision of the analysis,
- we detail an operation *fold* for summarizing memory cells and its dual, *expand*, using standard numeric domain operations,
- we show optimality of these operations, assuming that the domain operations join, meet, projection and the swapping of two variables are optimal.

The remainder of the paper is organized as follows: After illustrating our techniques on a few examples, Sect. 2 formalizes the summarization of relational information using a generic concrete numeric domain. The herein developed operations are then lifted to abstract domains in Sect. 3 where their optimality is proven. We conclude the paper in Sect. 4.

2 Principles of Summarizing Memory Cells

Static analysis for finding faults in software amounts to a reachability analysis of numeric properties, which is done by calculating the set of possible values the program variables can take at each program point.

For tractability, these states are approximated by so-called abstract domains that map a set of variables to some abstract properties. While it seems plausible that the value of each program variable is represented by exactly one variable in the abstract domain, this no longer holds true for more sophisticated analyses. A C variable that holds a `struct` may be represented by several abstract variables, namely one per field. Also, the elements of an array or several dynamically allocated heap regions may be represented by the same abstract variable. In the latter case, the abstract variable over-approximates the content of each concrete memory cell. For this case, Gopan et al. [3] discuss how to implement transfer functions that manipulate these summarizing variables. In order to illustrate their approach, consider the following example.

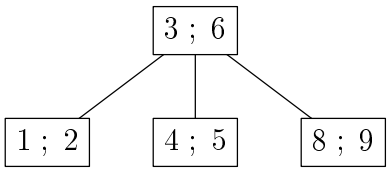
Example 2.1 The following C code fills the array a with increasing integers:

```
int a[100];
for(int i=0;i<100;i++) a[i]=i;
```

In the first iteration, the write access to the array will create a memory cell represented by the variable x in the abstract domain. This cell is initialized to zero, since i, the right hand side, is zero. In the next iteration, the write access to a[1] creates a second memory cell whose value 1 is represented by, say, x'. For tractability, the new memory cell is merged with the existing one by an operation called *fold*. The net effect is that afterwards x' no longer exists while x contains a summary of the previous values of x and x', for example, the interval $[0, 1]$.

The work of Gopan et al. addresses the analysis of arrays over single scalar values which is why their *expand* and *fold* operations only manipulate a single variable at a time. In particular, their method can lead to a significant loss of precision when summarizing compound memory cells. While arrays occasionally range over compound memory cells, they are ubiquitous in the analysis of heap allocated structures. This is illustrated in the next example.

Example 2.2 Consider the nodes of a B-Tree in memory. Each node contains an ordered list of n indices together with $n + 1$ pointers to subtrees. For the sake of presentation, we ignore the pointers between the different nodes and concentrate on the numeric properties of each node. A simple B-Tree can thus be represented as follows:



A crucial invariant of every node i is that the right index r_i is always greater than the left index l_i. On aspect of analyzing operations on B-Trees is to verify that this invariant is maintained.

The *fold* operation of Gopan et al. is too imprecise for this purpose, as it would summarize the left indices into $l_s \in \{1, 3, 4, 8\}$ and right indices into $r_s \in \{2, 5, 6, 9\}$, so that $l_s < r_s$ does not necessarily hold. We now present a way to summarize these nodes without losing relational information between their components.

2.1 Preliminaries

Before we consider actual abstract domains, we illustrate the idea of summarizing numeric information by operations on sets of concrete values. Specifically,

let $\mathbb{V}$ denote a fixed set of numeric values, such as $\mathbb{R}$, $\mathbb{Z}$ or some finite interval, that represents the values a single memory cell can hold. Let $C_n := \wp(\mathbb{V}^n)$ denote the numeric domain of dimension n, ordered by set inclusion. A vector $c \in C_n$ then represents the state of the n variables that are in scope at a program point.

For every dimension n we define functions $drop_i^n : C_n \to C_{n-1}$, functions $add_i^n : C_n \to C_{n+1}$ and functions $swap_{i,j}^n : C_n \to C_n$, such that

$$\begin{aligned} drop_i^n(c) &= \{(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \mid (x_1, \ldots, x_n) \in c\} \\ add_i^n(c) &= \{(x_1, \ldots, x_{i-1}, y, x_i, \ldots, x_n) \mid (x_1, \ldots, x_n) \in c,\ y \in \mathbb{V}\} \\ swap_{i,j}^n(c) &= \{(\ldots, x_j, \ldots, x_i, \ldots) \mid (\ldots, x_i, \ldots, x_j, \ldots) \in c\} \end{aligned}$$

for every state $c \in C_n$. In the following, we omit the superscript n when it is clear from the context. Intuitively, function $drop_i$ removes dimension i thereby discarding all information pertaining to variable x_i. Function add_i adds a new dimension i that corresponds to a new, unrestricted variable x_i. Function $swap_{i,j}$ exchanges the information stored in dimensions i and j.

In addition, we abbreviate sequences of operations as follows, assuming that $ab\ldots$ as well as $a'b'\ldots$ are ascending lists of dimensions that have no common elements:

$$\begin{aligned} drop_{ab\ldots} &= drop_a \circ drop_{b\ldots} \\ add_{ab\ldots} &= add_{b\ldots} \circ add_a \\ swap_{ab\ldots,a'b'\ldots} &= swap_{a,a'} \circ swap_{b\ldots,b'\ldots} \end{aligned}$$

These operations form the basis of summarizing the information of several variables which is the topic of the next section.

2.2 Folding Memory Cells Element-Wise

With the help of the previously defined functions, we can now define functions $fold_{i,j}^n : C_n \to C_{n-1}$ that summarize two dimensions i and j into dimension i:

$$fold_{i,j}^n(c) := drop_j(c \cup swap_{i,j}(c))$$

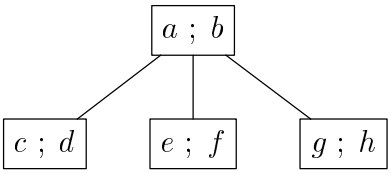
and their counterparts $expand_{i,j}^n : C_n \to C_{n+1}$ by

$$expand_{i,j}^n(c) := add_j(c) \cap swap_{i,j}(add_j(c))$$

These functions are equivalent to the functions $fold_{i,j}$ and $expand_{i,j}$ presented by Gopan et al. [3]. Function $fold_{i,j}$ summarizes the information contained in the variables x_i and x_j into the variable x_i. Function $expand_{i,j}$ restores dimensions i and j from the summarized dimension i, so that x_i and x_j contain an approximation of their previous content. For instance, in Example 2.1

we have variables $(x, x', i) \in c = \{(0, 1, 1)\}$ after two loop iterations. The dimensions x and x' are summarized by calculating $fold_{x,x'}(c) = \{(0, 1), (1, 1)\}$.

The element-wise folding of memory cells has limitations when it comes to summarizing compound memory cells, like nodes of the B-Tree in Example 2.2. Consider the task of summarizing the second and third child node of the B-Tree into one summarized node. We first introduce variables $a, \ldots, h$ to represent the contents of the nodes which store the information that $B := \{(3, 6, 1, 2, 4, 5, 8, 9)\} \ni (a, b, c, d, e, f, g, h)$:



The 8-dimensional numeric domain now has to be summarized into a 6-dimensional numeric domain by merging dimensions g and h into dimensions e and f, respectively. This is done by calculating $fold_{e,g}(fold_{f,h}(c))$:

$$fold_{e,g}(fold_{f,h}(B)) = fold_{e,g}\left(\left\{\begin{array}{l}(3,6,1,2,4,5,8),\\(3,6,1,2,4,9,8)\end{array}\right\}\right) = \left\{\begin{array}{l}(3,6,1,2,4,5),\\(3,6,1,2,8,5),\\(3,6,1,2,4,9),\\(3,6,1,2,8,9)\end{array}\right\}$$

In order to regain information about the concrete cells, we re-expand this set by applying $expand_{f,h} \circ expand_{e,g}$ to it, obtaining the following 16-element set:

$$\{(3, 6, 1, 2, e, f, g, h) \mid e, g \in \{4, 8\} \wedge f, h \in \{5, 9\}\}$$

Notice that for $e = 8$ and $f = 9$ we have lost the information that $e < f$, which makes this summarization unsuitable to verify the B-Tree invariant of increasing indices. Indeed, this summarization method is rather weak, in particular when relational information is of interest. The next section investigates a novel summarization method that honors relational information within a node.

2.3 Folding Compound Memory Cells en bloc

We now illustrate how to summarize complete nodes at once. To this end, define the following generalisations of the *fold* and *expand* operations:

$$\begin{aligned} fold_{ab\ldots,a'b'\ldots}(c) &= drop_{a'b'\ldots}(c \cup swap_{ab\ldots,a'b'\ldots}(c)) \\ expand_{ab\ldots,a'b'\ldots}(c) &= add_{a'b'\ldots}(c) \cap swap_{ab\ldots,a'b'\ldots}(add_{a'b'\ldots}(c)) \end{aligned}$$

Here, the intention is that $ab\ldots$ represent all variables of a compound memory cell, and $a'b'\ldots$ those of the memory cell to be merged into the former.

The intuition is that relational information that holds between $ab\ldots$ and $a'b'\ldots$ is retained during the fold operation. We illustrate this behavior in the B-Tree example by, again, folding the second and third child node into one summary node:

$$fold_{ef,gh}(\{(3,6,1,2,4,5,8,9)\}) = \left\{ \begin{array}{l} (3,6,1,2,4,5), \\ (3,6,1,2,8,9) \end{array} \right\}$$

Note that all vectors in the resulting set obey the invariant of the B-Tree in that $e < f$. This invariant is maintained even when expanding the summarized node into two again:

$$expand_{ef,gh}\left(\left\{ \begin{array}{l} (3,6,1,2,4,5), \\ (3,6,1,2,8,9) \end{array} \right\}\right) = \left\{ \begin{array}{l} (3,6,1,2,4,5,4,5), \\ (3,6,1,2,4,5,8,9), \\ (3,6,1,2,8,9,4,5), \\ (3,6,1,2,8,9,8,9) \end{array} \right\}$$

Although the two nodes are summarized, the important relational information between the *components* of each node has been preserved. Indeed, not only does our summarization method yield more precise results, it is also cheaper, as only one join and meet operation is required to calculate *fold* and *expand*, respectively. In contrast, folding element-wise requires one such operation for each dimension that is summarized.

2.4 Discussion

One question that arises is whether folding memory *en bloc* is always more precise than folding element-wise. Surprisingly, this is not the case. This is illustrated in the following example that exhibits a 4-dimensional state on which folding and expanding two dimensions is exact using the element-wise method but approximate when using the *en bloc* method.

Example 2.3 Consider two compound memory cells, consisting of two elements each, labelled a, b and c, d. Let their values be given by:

$$(a,b,c,d) \in \{(0,0,0,0),(1,1,1,1))\}$$

First, consider summarizing dimensions element-wise, starting with indices b

and d, followed by indices a and c:

$$fold_{a,c}\left(fold_{b,d}\left(\left\{\begin{array}{l}(0,0,0,0),\\(1,1,1,1)\end{array}\right\}\right)\right) = fold_{a,c}\left(\left\{\begin{array}{l}(0,0,0),\\(1,1,1)\end{array}\right\}\right) = \left\{\begin{array}{l}(0,0),\\(1,1)\end{array}\right\}$$

Unfolding the summarized dimensions by the reverse operation $expand_{b,d} \circ expand_{a,c}$ reconstructs the original 4-tuple $\{(0,0,0,0),(1,1,1,1))\}$.

Now, consider summarizing dimensions *en bloc*:

$$fold_{ab,cd}\left(\left\{\begin{array}{l}(0,0,0,0),\\(1,1,1,1)\end{array}\right\}\right) = \left\{\begin{array}{l}(0,0),\\(1,1)\end{array}\right\}$$

Seemingly, we have obtained the same abstraction. But when it is expanded by the corresponding operation $expand_{ab,cd}$, the original set is not restored. Instead, it is approximated by a greater set:

$$expand_{ac,bd}\{(0,0,1,1)\}) = B := \left\{\begin{array}{l}(0,0,0,0),\ (0,0,1,1),\\(1,1,0,0),\ (1,1,1,1)\end{array}\right\}.$$

Thus, in this example, summarizing element-wise turns out to be more precise. In general, a $fold_{a,c}$ operation retains the relations that a as well as c has with other dimensions. In the context above, both a and c are equal to b so that this information is retained in the summary.

In order to address the question which method is more suitable for the purpose of program analysis, we consider a second example in which summarization *en bloc* is more precise: Consider the states $(a,b,c,d) \in B$ from above. Here, applying the element-wise summary first on b,d then on a,c yields:

$$fold_{a,c}(fold_{b,d}(B)) = B' := \{(0,0),(0,1),(1,0),(1,1)\}$$

The element-wise expansion of this most-general state results in a state that is greater than B, since $expand_{b,d}(expand_{a,c}(B')) = \{0,1\}^4$. In contrast, the summarization *en bloc* is able to re-construct the original set B:

$$expand_{ab,cd}(fold_{ab,cd}(B)) = expand_{ab,cd}(\{(0,0),(1,1)\}) = B$$

The *en bloc* summarization is more precise in this example since it can express that the dimensions that are not summarized are equal, that is, the fact that $a = b$ and $c = d$ is retained. Intuitively, by summarizing two sets of dimensions, our *en bloc* method will retain all relational information that exists amongst the dimensions in each set. In contrast, the element-wise method looses this relation and rather sporadically recovers relational information with

variables in the environment. Thus, the *en bloc* summarization reliably retains information that holds within a compound memory cell while it is difficult to find use cases for the relational information the element-wise method infers.

The existence of two summarization approaches begs the question whether there are others and, in particular, if there is a best method. Indeed, given two lists of indices $ab\ldots$ and $a'b'\ldots$, any partitioning P of these lists can be used to apply *fold* and *expand* $|P|$ times, once for each partition. Each choice of partitioning leads to a different abstraction and, thus, precision.

In the following, it is shown that there exists no best partitioning; indeed, while each choice of a partitioning leads to a different abstraction, each of these abstractions has the same precision.

We commence by pointing out that a Galois insertion consists of two functions, of which one is injective and one is surjective. Then the following proposition helps in reasoning about the cardinality of summarizations:

Proposition 2.4 *Each pair* $fold_{ab\ldots,a'b'\ldots}$ *and* $expand_{ab\ldots,a'b'\ldots}$ *is a Galois insertion.*

Proof. Being composed of monotone functions, $fold_{ab\ldots,a'b'\ldots}$ and $expand_{ab\ldots,a'b'\ldots}$ are also monotone. Also, there is

$$\begin{aligned}
&fold_{ab\ldots,a'b'\ldots}(expand_{ab\ldots,a'b'\ldots}(c))\\
&\quad = drop_{a'b'\ldots}(add_{a'b'\ldots}(c) \cap swap_{ab\ldots,a'b'\ldots}(add_{a'b'\ldots}(c))) = c
\end{aligned}$$

$$\begin{aligned}
&expand_{ab\ldots,a'b'\ldots}(fold_{ab\ldots,a'b'\ldots}(c))\\
&\quad = add_{a'b'\ldots}(drop_{a'b'\ldots}(h)) \cap swap_{ab\ldots,a'b'\ldots}(add_{a'b'\ldots}(drop_{a'b'\ldots}(h)))\\
&\quad \supseteq h \cap swap_{ab\ldots,a'b'\ldots}(h) = h \supseteq c
\end{aligned}$$

with $h = c \cup swap_{ab\ldots,a'b'\ldots}(c)$. The first inequality holds because $add_{a'b'\ldots} \circ drop_{a'b'\ldots}$ is extensive and the second holds because $h = swap_{ab\ldots,a'b'\ldots}(h)$. Thus, pair $fold_{ab\ldots,a'b'\ldots}$ and $expand_{ab\ldots,a'b'\ldots}$ form a Galois insertion. □

The next proposition states the result for finite value sets $\mathbb{V}$. It will then be generalized to arbitrary value sets.

Proposition 2.5 *If* $\mathbb{V}$ *is finite, then no Galois insertion between* C_n *and* C_{n-k} *is better than another.*

Proof. Let α, γ and α', γ' be Galois insertions between C_n and C_{n-k}. Then

$$|\mathrm{img}(\gamma \circ \alpha)| = |\mathrm{img}(\gamma)| = |\mathbb{V}|^{n-k} = |\mathrm{img}(\gamma')| = |\mathrm{img}(\gamma' \circ \alpha')|.$$

Thus, while every choice of α, γ may lead to different abstractions, each abstraction is equal in terms of precision. □

The next proposition states that the result of applying *fold* and *expand* consists of the same elements of $\mathbb{V}$ as its argument.

Proposition 2.6 *Let $S \subseteq \mathbb{V}$. Then for every $a \in C_{n-k}$ there is*

$$A(c) := expand_{ab\ldots,a'b'\ldots}(c \cap S) \subseteq S$$
$$B(c) := expand_{ab\ldots,a'b'\ldots}(c \setminus S) \subseteq C_n \setminus S$$

and $expand_{ab\ldots,a'b'\ldots}(c) = A(c) \uplus B(c)$.

We now formally conclude that the partitioning should be chosen based on the kind of information that the analysis aims to infer.

Proposition 2.7 *No partitioning $P_1 \ldots P_k$ of $ab\ldots$ leads to a better summarization $fold_{P_1,P_1'} \cdots fold_{P_k,P_k'}$, $expand_{P_k,P_k'} \cdots expand_{P_1,P_1'}$ than another.*

Proof. From Prop. 2.4 it follows that such a summarization forms a Galois insertion. Thus, from Prop. 2.6 it follows that two such summarizations are incomparable if their restriction to any finite $S \subseteq \mathbb{V}$ is incomparable. But then from Prop. 2.5 it follows that all summarizations are equal or incomparable. □

From this proposition it follows that there is no best abstraction: For every choice of partitioning there are program states where one summarization delivers a better approximation than another. As in the example, where the choice of summarizing all indices *en bloc* is motivated by the invariant that has to be verified, the appropriate summarization method must be chosen depending on the application.

3 Applying Summarization in Abstract Domains

Up to now, we have only considered operations on sets of states C_n that are, in general, not computer representable. Indeed, in the abstract interpretation framework [1] the partial order $\langle C_n, \subseteq \rangle$ can be seen as a concrete domain over which the semantics of the program is expressed. This section shows how summarization can be applied to abstract domains that approximate a set of concrete states $c \in C_n$ with a machine representable abstract state. After some preliminary definitions, we discuss the *fold* and *expand* operations in the context of abstract domains.

3.1 Preliminaries

For the sake of the presentation, we use the following, slightly restrictive definition of an abstract domain.

Definition 3.1 An *abstract domain* of dimension n is a complete lattice $\langle A_n, \sqsubseteq_n, \sqcup_n, \sqcap_n \rangle$ with join $\sqcup_n$ and meet $\sqcap_n$ together with monotone func-

tions $\alpha_n : C_n \to A_n$ and $\gamma_n : A_n \to C_n$ such that for each $c \in C_n$ there is $c \subseteq \gamma_n(\alpha_n(c))$ and for each $a \in A_n$ there is $\alpha_n(\gamma_n(a)) = a$.

We will omit the dimension of each operation when it is clear from the context. With this definition, every pair α and γ is a Galois insertion, for which the following laws are known to hold:

Proposition 3.2 *For all $x, y \in C$ and $p, q \in A$ there is*

$$\alpha(x) \sqcup \alpha(y) = \alpha(x \cup y) \tag{1}$$
$$\gamma(p) \cap \gamma(q) = \gamma(p \sqcap q) \tag{2}$$

The next section lifts *fold* and *expand* to their abstract counterparts. The algebraic laws above are then used to argue optimality in Sect. 3.3.

3.2 Domain Operations

We have defined operations *fold*, *expand* in terms of three operations *drop*, *add* and *swap* that work on sets of concrete values. In order to define the corresponding abstract operations $expand^\#$ and $fold^\#$ on numeric abstract domains, we observe that their constituent abstract operations $drop^\#$, $add^\#$ and $swap^\#$ are already present in common abstract domains:

- A new dimension has to be inserted whenever a, say, local variable or a new memory cell is introduced. Hence, a numeric domain must provide an operation $add^\#$ that introduces a new, unbounded variable.
- When a variable goes out of scope, the information for that variable becomes obsolete: The dimension associated with this variable should be removed for tractability via an operation *drop*, thereby obtaining a more compact representation of the program state.
- All numeric domains provide operations for assigning numeric values, such that swapping two variables x and y can be expressed by assignments to and from an auxiliary variable [6].

Based on these functions, the definition of the abstract counterparts to *fold* and *expand* are simply their liftings to the abstract domain:

$$fold^\#_{ab\ldots,a'b'\ldots}(a) = drop^\#_{a'b'\ldots}(a \sqcup swap^\#_{ab\ldots,a'b'\ldots}(a))$$
$$expand^\#_{ab\ldots,a'b'\ldots}(a) = add^\#_{a'b'\ldots}(a) \sqcap swap^\#_{ab\ldots,a'b'\ldots}(add^\#_{a'b'\ldots}(a))$$

The merit of defining $expand^\#$ and $fold^\#$ in terms the three simple operations $drop^\#$, $add^\#$ and $swap^\#$ is that interesting theoretic results can be derived more easily. For instance, the following section derives the optimality of $fold^\#$ and $expand^\#$ for a range of existing abstract domains.

3.3 Completeness of $expand^{\#}$ and $fold^{\#}$

One of the attractive properties of the element-wise summarization of Gopan et al. [3] is the simplicity of implementing the $fold^{\#}$ operation. When summarizing several dimensions, one may wonder if another abstract fold operation exists that gives a better approximation of $fold_{ab\ldots,a'b'\ldots}$ than $fold^{\#}_{ab\ldots,a'b'\ldots}$. In this section, we show that this is not the case, that is, we show that $fold^{\#}$ and $expand^{\#}$ are optimal whenever the constituent functions $drop^{\#}$, $add^{\#}$ and $swap^{\#}$ are optimal. To this end, we introduce two notions of completeness and a notion of optimality [5]:

Definition 3.3 Let $f : C_n \to C_n$ be a concrete domain operation and $f^{\#} : A_n \to A_n$ its corresponding abstract domain operation. Then

- $f^{\#}$ is *forward complete* iff $f \circ \gamma = \gamma \circ f^{\#}$.
- $f^{\#}$ is *backwards complete* iff $\alpha \circ f = f^{\#} \circ \alpha$.
- $f^{\#}$ is a *best approximation* of f iff $f^{\#} = \alpha \circ f \circ \gamma$

Proposition 3.4 *If the abstract domain provides a backwards complete operation $drop^{\#}_{a'b'\ldots}$, a forward complete operation $add^{\#}_{a'b'\ldots}$ and a forward and backwards complete operation $swap^{\#}_{ab\ldots,a'b'\ldots}$, then $fold^{\#}_{ab\ldots,a'b'\ldots}$ and $expand^{\#}_{ab\ldots,a'b'\ldots}$ are the best approximations of $fold_{ab\ldots,a'b'\ldots}$ and $expand_{ab\ldots,a'b'\ldots}$.*

Proof. Completeness follows from Prop. 3.2. From that it follows directly that $fold^{\#}_{ab\ldots,a'b'\ldots}$ and $expand^{\#}_{ab\ldots,a'b'\ldots}$ are the best approximations of $fold_{ab\ldots,a'b'\ldots}$ and $expand_{ab\ldots,a'b'\ldots}$. □

The following diagram shows the completeness properties for functions *fold* and *expand*:

$$\begin{array}{ccc} C_n & \xrightarrow{fold_{ab\ldots,a'b'\ldots}} & C_{n-1} \\ \downarrow{\scriptstyle\alpha} & & \downarrow{\scriptstyle\alpha} \\ A_n & \xrightarrow{fold^{\#}_{ab\ldots,a'b'\ldots}} & A_{n-1} \end{array} \qquad \begin{array}{ccc} C_{n-1} & \xrightarrow{expand_{ab\ldots,a'b'\ldots}} & C_n \\ \uparrow{\scriptstyle\gamma} & & \uparrow{\scriptstyle\gamma} \\ A_{n-1} & \xrightarrow{expand^{\#}_{ab\ldots,a'b'\ldots}} & A_n \end{array}$$

The next proposition states that $fold^{\#}_{ab\ldots,a'b'\ldots}$ and $expand^{\#}_{ab\ldots,a'b'\ldots}$ form a Galois insertion. Namely, function $fold^{\#}_{ab\ldots,a'b'\ldots}$ is surjective, so that the whole expressivity of the summarized abstract numeric domain is exploited.

Proposition 3.5 *Given the preconditions of Prop. 3.4, function $fold^{\#}_{ab\ldots,a'b'\ldots}$ together with function $expand^{\#}_{ab\ldots,a'b'\ldots}$ forms a Galois insertion.*

Proof. From Prop. 3.4 it follows that $fold^{\#}_{ab\ldots,a'b'\ldots} = \alpha \circ fold_{ab\ldots,a'b'\ldots} \circ \gamma$, which is a composition of monotone functions. Therefore $fold^{\#}_{ab\ldots,a'b'\ldots}$ is also monotone. Similarly, $expand^{\#}_{ab\ldots,a'b'\ldots}$ is monotone. Also, for all $c \in C_n$ and

$a \in C_{n-k}$ the following holds:

$$\begin{aligned} fold^{\#}_{ab\ldots,a'b'\ldots}(expand^{\#}_{ab\ldots,a'b'\ldots}(a)) &= \alpha(fold_{ab\ldots,a'b'\ldots}(expand_{ab\ldots,a'b'\ldots}(\gamma(a)))) \\ &= \alpha(\gamma(a)) = a \\ expand^{\#}_{ab\ldots,a'b'\ldots}(fold^{\#}_{ab\ldots,a'b'\ldots}(c)) &= \alpha(expand_{ab\ldots,a'b'\ldots}(\gamma(\alpha(fold_{ab\ldots,a'b'\ldots}(\gamma(c)))))) \\ &\supseteq \alpha(expand_{ab\ldots,a'b'\ldots}(fold_{ab\ldots,a'b'\ldots}(\gamma(c)))) \supseteq c \end{aligned}$$

Thus, $fold^{\#}_{ab\ldots,a'b'\ldots}$ and $expand^{\#}_{ab\ldots,a'b'\ldots}$ form a Galois insertion. □

Prop. 3.4 and Prop. 3.5 show that, given appropriate functions $drop^{\#}$, $add^{\#}$ and $swap^{\#}$, it is not only easy to implement summarization for an abstract numeric domain, but this implementation is also the best approximation of the concrete operations for that domain.

4 Conclusion and Future Work

In this work we have observed that summarizing relational information is non-trivial. Using generic fold and expand operations we proposed a family of abstractions that feature the same precision but retain different relational information. It remains for future work to identify the right family member for a given application. No matter which member is chosen, the summarization is optimally precise given complete concrete and abstract lattices. An interesting open question is whether this optimality result can be generalized to incomplete abstract lattices. This question is particularly relevant as common relational numeric abstract domains such as convex polyhedra [2] do not form complete lattices.

References

[1] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In B. Robinet, editor, *International Symposium on Programming*, pages 106–130, Paris, France, April 1976.

[2] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Constraints among Variables of a Program. In *Principles of Programming Languages*, pages 84–97, Tucson, January 1978. ACM.

[3] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 512–529. Springer Berlin / Heidelberg, 2004.

[4] W. H. Harrison. Compiler Analysis of the Value Ranges for Variables. *Transactions on Software Engineering*, 3(3):243–250, May 1977.

[5] David Schmidt. Comparing completeness properties of static analyses and their logics. In Naoki Kobayashi, editor, *Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 183–199. Springer Berlin / Heidelberg, 2006.

[6] A. Simon and A. King. Exploiting Sparsity in Polyhedral Analysis. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium*, volume 3672 of *LNCS*, pages 336–351, London, September 2005. Springer.