Induction Variable Analysis with Delayed Abstractions

Sebastian Pop [1], Albert Cohen [2], and Georges-André Silber [1]

[1] CRI, Mines Paris, Fontainebleau, France
[2] ALCHEMY group, INRIA Futurs, Orsay, France

Abstract. This paper presents the design of an induction variable analyzer suitable for the analysis of typed, low-level, three address representations in SSA form. At the heart of our analyzer is a new algorithm recognizing scalar evolutions. We define a representation called trees of recurrences that is able to capture different levels of abstractions: from the finer level that is a subset of the SSA representation restricted to arithmetic operations on scalar variables, to the coarser levels such as the evolution envelopes that abstract sets of possible evolutions in loops. Unlike previous work, our algorithm tracks induction variables without prior classification of a few evolution patterns: different levels of abstraction can be obtained on demand. The low complexity of the algorithm fits the constraints of a production compiler as illustrated by the evaluation of our implementation on standard benchmark programs.

1 Introduction and Motivation

Supercomputing research has produced a wealth of techniques to optimize a program and tune code generation for a target architecture, both for uniprocessor and multiprocessor performance [35, 2]. But the context is different when dealing with common retargetable compilers for general-purpose and/or embedded architectures: the automatic exploitation of parallelism (fine-grain or thread-level) and the tuning for dynamic hardware components become far more challenging. Modern compilers implement some of the sophisticated optimizations introduced for supercomputing applications [2], provide performance models and transformations to improve fine-grain parallelism and exploit the memory hierarchy. Most of these optimizations are loop-oriented and assume a high-level code representation with rich control and data structures: `do` loops with regular control, constant bounds and strides, typed arrays with linear subscripts. Good optimizations require manual efforts in the syntactic presentation of loops and array subscripts (avoiding, e.g., `while` loops, pointers, exceptions, or goto statements). Programs written for embedded systems often make use of such low-level constructs, and this programming style is not suited to traditional source-to-source loop nest optimizers. Because any rewrite of the existing code involves an important investment, we see the need of a compiler that could optimize low-level constructs. Several works demonstrated the interest of enriched low-level representations [15, 23]: they build on the normalization of three address code, adding data types, *Static Single-Assignment* form (SSA) [6, 19] to ease data-flow analysis and scalar optimizations. Starting from version 4.0, GCC uses such a representation: GIMPLE [21, 17], a three-address code derived from SIMPLE [12]. In a three-address representation, subscript expressions, loop bounds and strides are spread across several instructions and basic blocks. The most popular techniques to retrieve scalar evolutions [34, 8, 27] are not well suited to work on loosely structured loops because they rely on classification schemes into a set of predefined forms based on pattern-matching rules. Such rules are sometimes sufficient at the source level, but too restrictive to cover the wide variability of inductive schemes on a low-level representation. To address the challenges of induction variable recognition on a low-level representation, we designed a general and flexible algorithm to build closed form expressions for scalar evolutions. This algorithm can retrieve array subscripts, loop bounds and strides lost in the lowering to three-address code, as well as other properties that do not appear in the source code. We demonstrate that induction-variable recognition and dependence analysis can be effectively implemented at such a low level. We also show that our method is more flexible and robust than comparable solutions on high-level code [8, 32, 31], since our method captures affine and polynomial closed forms without restrictions on the complexity of the flow of control, the recursive scalar definitions, and the intricateness of ϕ nodes. Finally, speed, robustness and language-independence are natural benefits of a low-level SSA representation.

Intermediate representations. We recall some SSA terminology, see [6, 19] for details: the *SSA graph* is the graph of def-use chains; ϕ nodes occur at merge points and restore the flow of values from the renamed variables; ϕ nodes are split into the *loop-*ϕ and *condition-*ϕ nodes. In this paper, we use a typed three-address code in SSA form. Control-flow primitives are a conditional expression `if`, a goto expression, and a loop annotation discovered from the control-flow graph: *loop* (ℓ_k) stands for loop number k, and ℓ_k denotes its associated implicit counter. The number of iterations is computed from the evolutions of scalars in the loop exit conditions, providing informations lost in the translation to a low-level, or not exposed at source level, as in `while` or `goto` loops.

Introductory examples. To illustrate the main issues and concepts, we consider the examples in Figure 1. A closed-form for `f` in the first example is a second-degree polynomial. In the second example, `d` has a *multivariate* evolution: it depends on several loop counters. To compute the evolution of `c`, `x` and `d` in the second example, one must know the *trip count* of the inner loop, here 10 iterations. Yet, to statically evaluate the trip count of ℓ_2 one must already understand the evolutions of `c` and `d`. In the third example, `c` is a typical case of *wrap-around* variable [34]. In the fourth example `a` and `b` have linear closed forms. Unlike our algorithm, previous works could not compute this closed form due to the intricateness of the SSA graph. The fifth example illustrates a data dependence problem: when language standards define modulo arithmetics for a type, the compiler has to respect effects of overflows, and otherwise, as in the sixth example, the compiler can deduce constraints from undefined behavior.

```
a = 3;
b = 1;
loop (ℓ1)
  c = φ(a, f);
  d = φ(b, g);
  if (d>=123) goto end;
  e = d + 7;
  f = e + c;
  g = d + 5;
end:
```

First example: polynomial functions. At each step of the loop, an integer value following the sequence $1, 6, 11, \ldots, 126$ is assigned to d, that is the affine function $5\ell_1 + 1$; a value in the sequence $3, 11, 24, \ldots, 1703$ is assigned to f, that is a polynomial of degree 2: $\frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3$.

```
a = 3;
loop (ℓ1)
  c = φ(a, x);
  loop (ℓ2)
    d = φ(c, e);
    e = d + 1;
    t = d - c;
    if (t>=9) goto end2;
  end2:
  x = e + 3;
  if (x>=123) goto end1;
end1:
```

Second example: multivariate functions. The successive values of c are $3, 17, 31, \ldots, 115$, that is the affine univariate function $14\ell_1 + 3$. The successive values of x in the loop are $17, 31, \ldots, 129$ that is $14\ell_1 + 17$. The evolution of variable d, $3, 4, 5, \ldots, 13, 17, 18, 19, \ldots, 129$ depends on the iteration number of both loops: that is the multivariate affine function $14\ell_1 + \ell_2 + 3$.

```
loop (ℓ1)
  a = φ(1, b);
  if (a>=100) goto end1;
  b = a + 4;
  loop (ℓ2)
    c = φ(a, e);
    e = φ(b, f);
    if (e>=100) goto end2;
    f = e + 6;
  end2:
end1:
```

Third example: wrap-around. The sequence of values taken by a is $1, 5, 9, \ldots, 101$ that can be written in a condensed form as $4\ell_1 + 1$. The values taken by variable e are $5, 11, 17, \ldots, 95, 101, 9, 15, 21, \ldots, 95, 101$ and generated by the multivariate function $6\ell_2 + 4\ell_1 + 5$. These two variables are used to define the variable c, that will contain the successive values $1, 5, 11, \ldots, 89, 95, 5, 9, 15, \ldots, 89, 95$: the first value of c in the loop ℓ_2 is the value coming from a, while the subsequent values are those of variable e.

```
loop (ℓ1)
  a = φ(0, d);
  b = φ(0, c);
  if (a>=100) goto end;
  c = a + 1;
  d = b + 1;
end:
```

Fourth example : periodic functions. Both a and b have affine evolutions: $0, 1, 2, \ldots, 100$, because they both have the same initial value. However, if their initial value is different, their evolution can only be described by a periodic affine function.

```
loop (ℓ1)
  (unsigned char) a = φ(0, c);
  (int) b = φ(0, d);
  (unsigned char) c = a + 1
  (int) d = b + 1
  if (d >= 1000) goto end;
  T[b] = U[a];
end:
```

Fifth example: effects of types on the evolution of scalar variables. The C programming language defines modulo arithmetics for unsigned typed variables. In this example, the successive values of variable a are periodic: $0, 1, 2, \ldots, 255, 0, 1, \ldots$, or in a condensed notation $\ell_1 \; mod \; 256$.

```
loop (ℓ1)
  (char) a = φ(0, c);
  (int) b = φ(0, d);
  (char) c = a + 1
  (int) d = b + 1
  if (d > N) goto end;
end:
```

Sixth example: inferring properties from undefined behavior. Signed types overflow are not defined in C. The behavior is only defined for the values of a in $0, 1, 2, \ldots, 126$, consequently d is only defined for $1, 2, 3, \ldots, 127$, and the loop is defined only for the first 127 iterations.

Fig. 1. Examples

Overview of the paper. In the following, we expose a set of techniques to extract and to represent evolutions of scalar variables in the presence of complex control flow and intricate inductive definitions. We focus on designing low-complexity algorithms that do not sacrifice on the effectiveness of retrieving precise scalar evolutions, using a typed, low-level, SSA-form representation of the program. Section 2 introduces the algebraic structure that we use to capture a wide spectrum of scalar evolution functions. Section 3 presents the analysis algorithm to extract closed form expressions for scalar evolutions. Section 4 compares our method to other existing approaches. Finally, section 5 concludes and sketches future work. For space constraints, we have shortened this presentation. A longer version of the paper is available as a technical report [26].

2 Trees of Recurrences

In this section, we introduce the notion of *Tree of Recurrences* (TREC), a closed-form that captures the evolution of induction variables as a function of iteration indices and allows an efficient computation of values at given iteration points. This formalism extends the expressive power of *Multivariate Chains of Recurrences* (MCR) [3, 14, 36, 32] by symbolic references. MCR are obtained by an abstraction operation that instantiate all the varying symbols: some evolutions are mapped to a "don't know" symbol $\top$. Arithmetic operations on MCR are defined as rewriting rules [32]. Let $F(\ell_1, \ell_2, \ldots, \ell_m)$, or $F(\boldsymbol{\ell})$, represent the evolution of a variable inside a loop of depth m as a function of $\ell_1, \ell_2, \ldots, \ell_m$. F can be written as a closed form Θ, called TREC, that can be statically processed by further analyzes and efficiently evaluated at compile-time. The syntax of a TREC is derived from MCR and inductively defined as: $\Theta = \{\Theta_a, +, \Theta_b\}_k$ or $\Theta = c$, where Θ_a and Θ_b are trees of recurrences and c is a constant or a variable name, and subscript k indexes the dimension. As a form of syntactic sugar, $\{\Theta_a, +, \{\Theta_b, +, \Theta_c\}_k\}_k = \{\Theta_a, +, \Theta_b, +, \Theta_c\}_k$.

Evaluation of TREC. The value $\Theta(\ell_1, \ell_2, \ldots, \ell_m)$ of a TREC Θ is defined as follows: if Θ is a constant c then $\Theta(\boldsymbol{\ell}) = c$, else Θ is $\{\Theta_a, +, \Theta_b\}_k$ and

$$\Theta(\boldsymbol{\ell}) = \Theta_a(\boldsymbol{\ell}) + \sum_{l=0}^{\ell_k - 1} \Theta_b(\ell_1, \ldots, \ell_{k-1}, l, \ell_{k+1}, \ldots, \ell_m) \ .$$

The evaluation of $\{\Theta_a, +, \Theta_b\}_k$ for a given $\boldsymbol{\ell}$ matches the inductive updates across ℓ_k iterations of loop k: Θ_a is the initial value, and Θ_b the increment in loop k. This is an exponential algorithm to evaluate a TREC, but [3] gives a linear time and space algorithm based on Newton interpolation series. Given a univariate MCR with $c_0, c_1, \ldots, c_n$, constant parameters (either scalar constants, or symbolic names defined outside loop k):

$$\{c_0, +, c_1, +, c_2, +, \ldots, +, c_n\}_k(\boldsymbol{\ell}) = \sum_{p=0}^{n} c_p \binom{\ell_k}{p} \ . \tag{1}$$

This result comes from the following observation: a sum of multiples of binomial coefficients — called *Newton series* — can represent any polynomial. The closed form for `f` in the first example of Figure 1 is the second order polynomial $F(\ell_1) = \frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3 = 3\binom{\ell_1}{0} + 8\binom{\ell_1}{1} + 5\binom{\ell_1}{2}$, and is written $\{3, +, 8, +, 5\}_1$. The coefficients of a TREC derive from a finite differentiation table: for example, the coefficients for the TREC associated with $\frac{5}{2}{\ell_1}^2 + \frac{11}{2}\ell_1 + 3$ can be computed either by differencing the successive values [11]:

ℓ_1	0	1	2	3	4
c_0	**3**	11	24	42	65
c_1	**8**	13	18	23	
c_2	**5**	5	5		
c_3	**0**	0			

or, by directly extracting the coefficients from the code [32]. We present our algorithm for extracting TREC from a SSA representation in Section 3. We illustrate the fast evaluation of a TREC from the second introductory example Figure 1, where the evolution of `d` is the affine equation $F(\ell_1, \ell_2) = 14\ell_1 + \ell_2 + 3$. A TREC for `d` is $\Theta(\ell_1, \ell_2) = \{\{3, +, 14\}_1, +, 1\}_2$, that can be evaluated for $\ell_1 = 10$ and $\ell_2 = 15$ as follows:

$$\Theta(10, 15) = \{\{3, +, 14\}_1, +, 1\}_2(10, 15) = 3 + 14 \cdot \binom{10}{1} + \binom{15}{1} = 158 \ .$$

Instantiation of TREC and abstract envelopes. In order be able to use the efficient evaluation scheme presented above, symbolic coefficients of a TREC have to be analyzed: the role of the instantiation pass is to limit the expressive power of TREC to MCR. Difficult TREC constructs such as exponential self referring evolutions (as the Fibonacci sequence that defines the simplest case of the class of mixers: $fib \rightarrow \{0, +, 1, +, fib\}_k$) are either translated to some appropriate representation, or discarded. Optimizers such as symbolic propagation could handle such difficult constructs, however they lead to problems that are difficult to solve (e.g. determining the number of iterations of a loop whose exit edge is guarded by a Fibonacci sequence). Because a large class of optimizers and analyzers are expecting simpler cases, TREC information is filtered using an instantiation pass. Several abstract views can be defined by different instantiation passes, such as mapping every non polynomial scalar evolution to $\top$, or even more practically, mapping non affine functions to $\top$. In appropriate cases, it is natural to map uncertain values to an abstract value: we have experimented instantiations of TREC with intervals, in which case we obtain a set of possible evolutions that we call an envelope. Allowing the coefficients of TREC to contain abstract scalar values is a more natural extension than the use of maximum and minimum functions over MCR as proposed by van Engelen in [31] because it is then possible to define other kinds of envelopes using classic scalar abstract domains, such as polyhedra, octagons [18], or congruences [10].

Peeled trees of recurrences. A frequent occurring pattern consists in variables that are initialized to a value during the first iteration of a loop, and then is replaced by the values of an induction variable for the rest of iterations. We have chosen to represent these variables by explicitly listing the first value that they contain, and then the evolution function that they follow. The *peeled* TREC are described by the syntax $(a, b)_k$ whose semantics is given by:

$$(a, b)_k(x) = \begin{cases} \text{a} & \text{if } x = 0, \\ \text{b (x - 1)} & \text{for } x \geq 1, \end{cases}$$

where a is a TREC with no evolution in loop k, b is a TREC that can have an evolution in loop k, and x is indexing the iterations in loop k. Most closed forms for wrap-around variables [34] are peeled TREC. Indeed, back to the third introductory example (see Figure 1), the closed form for `c` can be represented by a peeled multivariate affine TREC: $(\{1, +, 4\}_1, \{\{5, +, 4\}_1, +, 6\}_2)_2$. A peeled TREC describes the first values of a closed form chain of recurrence. In some cases it is interesting to replace it by a simpler MCR, and vice versa, to peel some iterations out of a MCR. For example, the peeled TREC $(0, \{1, +, 1\}_1)_1$ describes the same function as $\{0, +, 1\}_1$. This last form is a unique representative of a class of TREC that can be generated by peeling one or more elements from the beginning. Simplifying a peeled TREC amounts to the unification of its first element with the function represented in the right-hand side of the peeled TREC. A simple unification algorithm tries to add a new column to the differentiation table without changing the last element in that column. Since this first column contains the coefficients of the TREC, the transformation is possible if it does not modify the last coefficient of the column, as illustrated in Figure 2. This technique allows

ℓ_1	0	1	2	3	4
c_0	3	11	24	42	65
c_1	8	13	18	23	
c_2	5	5	5		
c_3	0	0			

ℓ_1	0	1	2	3	4	5
c_0	**0**	3	11	24	42	65
c_1	**3**	8	13	18	23	
c_2	**5**	5	5	5		
c_3	**0**	0	0			

Fig. 2. Adding a new column to the differentiation table of the chain of recurrence $\{3, +, 8, +, 5\}_1$ leads to the chain of recurrence $\{0, +, 3, +, 5\}_1$.

to unify 29 wrap around loop-ϕ in the SPEC CPU2000, 337 on the GCC code itself, and 5 on the JavaGrande. Finally, we formalize the notion of peeled TREC equivalence class: given integers v, $a_1, \ldots, a_n$, a TREC $c = \{a_1, +, \ldots, +, a_n\}_1$, a peeled TREC $p = (v, c)_1$, and a TREC $r = \{b_1, +, \ldots, +, b_{n-1}, +, a_n\}_1$, with the integer coefficients $b_1, \ldots, b_{n-1}$ computed as follows: $b_{n-1} = a_{n-1} - a_n$, $b_{n-2} = a_{n-2} - b_{n-1}$, …, $b_1 = a_1 - b_2$, we say that r is equivalent to p if and only if $b_1 = v$.

Typed and periodic trees of recurrences. Induction variable analysis in the context of typed operations is not new: all the compilers that have loop optimizers based on typed intermediate representations have solved this problem. However there is little literature that describes the problems and solutions [33]: these details are often considered too low level, and language dependent. As illustrated in the fifth introductory example, in Figure 1, the analysis of data dependences has to correctly handle the effects of overflowing on variables that are indexing the data. One of the solutions for preserving the semantics of wrapping types on TREC operations is to type the TREC and to map the effects of types from the SSA representation to the TREC representation. For example, the conversion from *unsigned char* to *unsigned int* of TREC $\{(uchar)100, +, (uchar)240\}_1$ is $\{(uint)100, +, (uint)\texttt{0xfffffff0}\}_1$, such that the original sequence remains unchanged $(100, 84, 68, \ldots)$. The first step of a TREC conversion proves that the sequence does not wrap. In the previous example, if the number of iterations in loop 1 is greater than 6, the converted TREC should also contain a wrap modulo 256, as illustrated by the first values of the sequence: $100, 84, 68, 52, 36, 20, 4, 244, 228, \ldots$. When it is impossible to prove that an evolution cannot wrap, it is safe to assume that it wraps, and keep the cast: $(uint)(\{(uchar)100, +, (uchar)240\}_1)$. Another solution is to use a periodic TREC, that lists all the values in a period: in the previous example we would have to store 15 values. Using periodic TREC for sequences wrapping over narrow types can seem practical, but this method is not practical for arbitrary sequences over wider types. Periodic sequences may also be generated by flip-flop operations, that are special cases of self referenced peeled TREC. Variables in a flip-flop exchange their initial values over the iterations, for example:

$$flip \rightarrow (3, 5, flip)_k(x) = [3, 5]_k(x) = \begin{cases} 3 & \text{if } x = 0 \bmod 2, \\ 5 & \text{if } x = 1 \bmod 2. \end{cases}$$

Exponential trees of recurrences. The exponential MCR [3] used by [32] and then extended to handle sums or products of polynomial and exponential evolutions [31] are useless in compiler technology for typed integer sequences, as integer typed arithmetic has limited domains of definition. Any overflowing operation either has defined modulo semantics, or is not defined by the language standard. The longer exponential integer sequence that can exist for an integer type of size 2^n is $n-1$: left shifting the first bit $n-2$ times. Storing exponential evolutions as peeled TREC seems efficient, because in general $n \leq 64$. We acknowledge that exponential MCR can have applications in compiler technology for floating point evolutions, but we have intentionally excluded floating point evolutions for simplifying this presentation. The next section will present our efficient algorithm that translates a part of the SSA dealing with scalar variables to TREC.

3 Analysis of Scalar Evolutions

We will now present an algorithm to compute closed-form expressions for inductive variables. Our algorithm translates a subset of the SSA to the TREC representation, interprets a part of the expressions and enriches the available information with properties that it computes, as the number of iterations, or the value of a variable at the end of a loop. It extends the applicability of classic optimizations, and allows the extraction of precise high level informations. We have designed our analyzer such that it does not assume a particular control-flow structure and makes no restriction on the recursive intricate variable definitions. It however fails to analyze irreducible control flow graphs [1], for which an uncomputable evolution $\top$ is returned. Our analysis does not use the syntactic information, making no distinction between names defined in the code or introduced by the compiler. The algorithm is also able to delay a part of the analysis until more information is known by leaving symbolic names in the target representation. The last constraint for inclusion in a production compiler is that the analyzer should be linear in time and space: even if the structure of our algorithm is complex, composed of a double recursion as sketched in Figure3, it presents similarities with the algorithm for linear unification by Paterson and

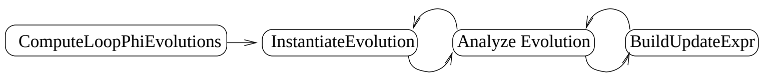

Fig. 3. Bird's eye view of the analyzer

Wegman [24], where the double recursion is hidden behind a single recursion with a stack.

3.1 Algorithm

Figures 4 and 5 present our algorithm to compute the scalar evolutions of all variables defined by loop-ϕ nodes: ComputeLoopPhiEvolutions is a driver that illustrates the use of the analyzer and instantiation. In general, AnalyzeEvolution is called for a given loop number and a variable name. The evolution functions are stored in a database that is visible only to AnalyzeEvolution, and that is accessed using Evolution$[n]$, for an SSA name n. The initial value for a not yet analyzed name is $\perp$. The cornerstone of the algorithm is the

Algorithm: ComputeLoopPhiEvolutions
Input: SSA representation of the procedure
Output: a TREC for every variable defined by loop-ϕ nodes
  For each loop l in a depth-first traversal of the loop nest
    For each loop-ϕ node n in loop l
      InstantiateEvolution(AnalyzeEvolution(l, n), l)

Algorithm: AnalyzeEvolution(l, n)
Input: l the current loop, n the definition of an SSA name
Output: TREC for the variable defined by n within l
  `v` $\leftarrow$ variable defined by n
  $ln \leftarrow$ loop of n
  If Evolution$[n] \neq \perp$ Then $res \leftarrow$ Evolution$[n]$
  Else If n matches "`v = constant`" Then $res \leftarrow$ `constant`
  Else If n matches "`v = a`" Then $res \leftarrow$ AnalyzeEvolution(l, `a`)
  Else If n matches "`v = a` $\odot$ `b`" (with $\odot \in \{+, -, *\}$) Then
    $res \leftarrow$ AnalyzeEvolution(l, `a`) $\odot$ AnalyzeEvolution(l, `b`)
  Else If n matches "`v = loop-`ϕ`(a, b)`" Then
    (notice `a` is defined outside loop ln and `b` is defined in ln)
    Search in depth-first order a path from `b` to `v`:
    $(exist, update) \leftarrow$ BuildUpdateExpr(n, definition of `b`)
    If not *exist* (if such a path does not exist) Then $res \leftarrow (\mathsf{a}, \mathsf{b})_l$: a peeled TREC
    Else If *update* is $\top$ Then $res \leftarrow \top$
    Else $res \leftarrow \{\mathsf{a}, +, update\}_l$: a TREC
  Else If n matches "`v = condition-`ϕ`(a, b)`" Then
    $eva \leftarrow$ InstantiateEvolution(AnalyzeEvolution(l, `a`), ln)
    $evb \leftarrow$ InstantiateEvolution(AnalyzeEvolution(l, `b`), ln)
    If $eva = evb$ Then $res \leftarrow eva$ Else $res \leftarrow \top$
  Else $res \leftarrow \top$
  Evolution$[n] \leftarrow res$
  Return Eval(res, l)

Fig. 4. ComputeLoopPhiEvolutions and AnalyzeEvolution.

search and reconstruction of the symbolic update expression on a path of the

Algorithm: BUILDUPDATEEXPR(h, n)
Input: h the halting loop-ϕ, n the definition of an SSA name
Output: (*exist*, *update*), *exist* is true if h has been reached,
update is the reconstructed expression for the overall effect in the loop of h

```
If (n is h) Then Return (true, 0)
Else If n is a statement in an outer loop Then Return (false, ⊥),
Else If n matches "v = a" Then Return BuildUpdateExpr(h, definition of a)
Else If n matches "v = a + b" Then
  (exist, update) ← BuildUpdateExpr(h, a)
  If exist Then Return (true, update + b),
  (exist, update) ← BuildUpdateExpr(h, b)
  If exist Then Return (true, update + a)
Else If n matches "v = loop-φ(a, b)" Then ln ←loop of n
  (notice a is defined outside ln and b is defined in ln)
  If a is defined outside the loop of h Then Return (false, ⊥)
  s ← Apply(ln, AnalyzeEvolution(ln, n), NumberOfIterations(ln))
  If s matches "a + t" Then (exist, update) ← BuildUpdateExpr(h, a)
    If exist Then Return (exist, update + t)
Else If n matches "v = condition-φ(a, b)" Then
  (exist, update) ← BuildUpdateExpr(h, a)
  If exist Then Return (true, ⊤)
  (exist, update) ← BuildUpdateExpr(h, b)
  If exist Then Return (true, ⊤)
Else Return (false, ⊥)
```

Algorithm: INSTANTIATEEVOLUTION(*trec*, l)
Input: *trec* a symbolic TREC, l the instantiation loop
Output: an instantiation of *trec*

```
If trec is a constant c Then Return c
Else If trec is a variable v Then
  If v has not been instantiated
    Mark v as instantiated and Return AnalyzeEvolution(l, v)
  Else v is in a mixer structure, Return ⊤
Else If trec is of the form {e1, +, e2}_x Then
  Return {InstantiateEvolution(e1, l), +, InstantiateEvolution(e2, l)}_x
Else If trec is of the form (e1, e2)_x Then
  Return Unify((InstantiateEvolution(e1, l), InstantiateEvolution(e2, l))_x)
Else Return ⊤
```

Fig. 5. BUILDUPDATEEXPR and INSTANTIATEEVOLUTION algorithms.

SSA graph: BUILDUPDATEEXPR. This corresponds to a depth-first search algorithm in the SSA graph with a pattern matching rule at each step, halting either with a success on the starting loop-ϕ node, or with a fail on any other loop-ϕ node of the same loop. Based on these results, ANALYZEEVOLUTION constructs either a TREC or a peeled TREC. INSTANTIATEEVOLUTION substitutes symbolic parameters in a TREC. It computes their statically known value, i.e., a constant, a periodic function, or an approximation with intervals, possibly triggering other computations of TREC in the process. The call to InstantiateEvolution is postponed until the end of the depth-first search, avoiding early approximations in the computation of update expressions. Combined with the introduction of symbolic parameters in the TREC, postponing the instantiation alleviates the need for a specific ordering of the computation steps. The correctness and complexity of this algorithm are established by structural induction [25].

3.2 Application of the Analyzer to an Example

We illustrate the analysis of scalar evolutions algorithm on the first introductory example in Figure 1, with the analysis of `c = φ(a, f)`. The SSA edge exiting the loop, Figure 6.(1), is left symbolic. The analyzer starts a depth-first search,

```
a = 3;
b = 1;
loop (ℓ1)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123)
| | goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(1) Initial condition edge

```
a = 3;
b = 1;
loop (ℓ1)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123)
| | goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(2) Searching for "c"

```
a = 3;
b = 1;
loop (ℓ1)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123)
| | goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(3) Found the halting phi

```
a = 3;
b = 1;
loop (ℓ1)
| c = φ(a, f);
| d = φ(b, g);
| if (d>=123)
| | goto end;
| e = d + 7;
| f = e + c;
| g = d + 5;
end:
```

(4) On the "return path"

Fig. 6. Application to the first example

illustrated in Figure 6.(2): the edge `c`$\rightarrow$`f` is followed to the definition `f = e + c`, then following the first edge `f`$\rightarrow$`e`, reaches the assignment `e = d + 7`, and finally `e`$\rightarrow$`d` leads to a loop-ϕ node of the same loop. Since this is not the starting loop-ϕ, the search continues on the other unexplored operands: back on `e = d + 7`, operand `7` is a scalar, then back on `f = e + c`, the edge `f`$\rightarrow$`c` is followed to the starting loop-ϕ node, as illustrated in Figure 6.(3). Following the path of iterative updates in execution order, as illustrated in Figure 6.(4), gives the update expression: `e`. Finally, the analyzer records the TREC $\mathtt{c} = \{\mathtt{a}, +, \mathtt{e}\}_1$. An instantiation of $\{\mathtt{a}, +, \mathtt{e}\}_1$ yields: $\mathtt{a} = 3$, $\mathtt{e} = \{8, +, 5\}_1$, and $\{3, +, 8, +, 5\}_1$.

3.3 Applications

Empirical study. To show the robustness and language-independence of our implementation, and to evaluate the accuracy of our algorithm, we determine a compact representation of *all* variables defined by loop-ϕ nodes in the SPEC CPU2000 [30] and JavaGrande [13] benchmarks. Figure 7 summarizes our experiments: affine univariate variables are very frequent because well structured loops are most of the time using simple constructs, affine multivariate less common, as

Benchmark	U.	M.	C.	$\top$	Loops	Trip	A.
CINT2000	12986	20	13526	52656	10593	1809	82
CFP2000	13139	52	12051	12797	6720	4137	68
JavaGrande	334	0	455	866	481	84	0

Fig. 7. Induction variables and loop trip count. Break-down of evolutions into: "U." affine univariate, "M." affine multivariate, "C." other compound expressions containing determined components, and "$\top$" undetermined evolutions. Last columns describe: "Loops" the number of *natural* loops, "Trip" the number of *single-exit* loops whose trip count is successfully analyzed, "A." the number of loops for which an upper bound approximation of the trip count is available.

they are used for indexing multi dimensional arrays. Difficult constructs such as polynomials of degree greater than one occur very rarely: we have detected only three occurrences in SPEC CPU2000, and none in JavaGrande. The last three columns in Figure 7 show the precision of the detector of the number of iterations: only the single-exit loops are exactly analyzed, excluding a big number of loops that contain irregular control flow (probably containing exception exits) as in the case of java programs. An approximation of the loop count can enable aggressive loop transformations as in the 171.swim SPEC CPU2000 benchmark, where the data accesses are used to determine a safe loop bound, allowing safe refinements of the data dependence relations.

Optimization passes. Based on our induction variable analysis, several scalar and high level loop optimizations have been contributed: Zdeněk Dvořák from SuSE has contributed strength reduction, induction variable canonicalization and elimination, and loop invariant code motion [1]. Dorit Naishlos from IBM Haifa has contributed a "simdization" pass [7, 20] that rewrites loops to use SIMD instructions such as Altivec, SSE, etc. Daniel Berlin from IBM Research and Sebastian Pop have contributed a linear loop transformation framework [5] that enables the loop interchange transformation: on the 171.swim benchmark a critical loop is interchanged, giving 1320 points compared to 796 points without interchange: a 65% benefit. Finally, Diego Novillo from RedHat has contributed a value range propagation pass [22]. The dependence-based transformations use uniform dependence vectors [4], but our method for identifying conflicting accesses between TREC can be applicable to the computation of more general dependence abstractions, tests for periodic, polynomial, exponential or envelope TREC. We implemented an extended Banerjee test [4], and we will integrate the Omega test [28] in the next GCC version 4.2. In order to show the effectiveness of the Banerjee data dependence analyzer as used in an optimizer, we have measured the compilation time of the vectorization pass: for SPEC CPU2000 benchmarks, the vectorization pass does not exceed 1 second, nor 5 percent of the compilation time per file. The experiments were performed on a Pentium4 2.40 GHz with 512 Kb of cache, 2 GB of RAM, on a Linux kernel 2.6.8. Figure 8 illustrates the scalability and accuracy of the analysis: we computed all de-

Benchmark	# tests	d	i	u	ZIV	SIV	MIV
CINT2000	303235	73180	105264	124791	168942	5301	5134
CFP2000	655055	47903	98682	508470	105429	17900	60543
JavaGrande v2.0	87139	13357	67366	6416	76254	2641	916

Fig. 8. Classification of data dependence tests in SPEC CPU2000 and JavaGrande. Columns "d", "i" and "u" represent the number of tests classified as dependent, independent, and undetermined. Last columns split the dependence tests into "ZIV", "SIV", "MIV": zero, single and multiple induction variable.

pendences between pairs of references — both accessing the same array — in every function. We have to stress that this evaluation is quite artificial because an optimizer would focus the data dependence analysis only on a few loop nests. The number of MIV dependence tests witness the stress on the analyzer: these tests involve arrays accessed in different loops, that could be separated by an important number of statements. Even with these extreme test conditions, our data dependence analyzer catches an important number of dependence relations, and the worst case is 15 seconds and 70 percent of the compilation time.

4 Comparison with Closely Related Works

Induction variable detection has been studied extensively in the past because of its central role in loop optimizations. Our target closed form expressions is an extension of the chains of recurrences [3, 32, 31]. The starting representation is close to the one used in the Open64 compiler [8, 16], but our algorithm avoids the syntactic case distinction made in [16] that has severe consequences in terms of generality (when analyzing intricate SSA graphs) and maintainability: as syntactic information is altered by several transformation passes, pattern matching at a low level may lead to an explosion of the number of cases to be recognized; e.g., if a simple recurrence is split across two variables, its evolution would be classified as wrap around if not handled correctly in a special case; in practice, [16] does not consider these cases. Path-sensitive approaches have been proposed [31, 29] to increase precision in the context of conditional variable updates. These techniques may lead to an exponential number of paths, and although interesting, seem not yet suitable for a production compiler, where even quadratic space complexity is unacceptable on benchmarks like GNU Go[9].

Our work is based on the previous research results presented in [32]. We have experimented with similar algorithms and dealt with several restrictions and difficulties that remained unsolved in later papers: for example, loop sequences are not correctly handled, unless inserting at the end of each loop an assignment for each variable modified in the loop and then used after the loop. Because they are using a representation that is not in SSA form, they have to deal with all the difficulties of building an "SSA-like" form. With some minor changes, their algorithm can be seen as a translation from an unstructured list of instructions to a weak SSA form restricted to operations on scalars. This weak SSA form could be of interest for representations that cannot be translated to classic SSA form, as the RTL representation of GCC. Another interesting result for their algorithm would be a proof that constructing a weak SSA representation is faster than building the classic SSA representation, however they have not presented experimental results on real codes or standard benchmarks for showing the effectiveness of their approach. In contrast, our algorithm is analyzing a classic SSA representation, and instead of worrying about enriching the expressiveness of the intermediate representation, we are concerned about the opposite question: how to limit the expressiveness of the SSA representation in order to provide the optimizers a level of abstraction that they can process. It might well be argued that a new representation is not necessary for concepts that can be expressed in the SSA representation: this point is well taken. We acknowledge that we could have presented the current algorithm as a transformer from SSA to an abstract SSA, containing abstract elements. However, we deliberately have chosen to present the analyzer producing trees of recurrences for highlighting the sources of our inspiration and for presenting the extensions that we proposed to the chains of recurrences. Finally, we wanted the algorithm presented in this paper to reflect the underlying implementation in GCC.

5 Conclusion and Perspectives

We introduced *trees of recurrences*, a formalism based on *multivariate chains of recurrences* [3, 14], with symbolic and algebraic extensions, such as the peeled chains of recurrences. These extensions increase the expressiveness of standard chains of recurrences and alleviate the need to resort to intractable exponential expressions to handle wrap-around and mixer induction variables. We extended this representation with the evolution envelopes that handle abstract elements as approximations of runtime values. We also presented a novel algorithm for the analysis of scalar evolutions. This algorithm is capable of traversing an arbitrary program in Static Single-Assignment (SSA) form, without prior classification of the induction variables. The algorithm is proven by induction on the structure of the SSA graph. Unlike prior works, our method does not attempt to retrieve more complex closed form expressions, but focuses on generality: starting from a low-level three-address code representation that has been seriously scrambled by complex phases of data- and control-flow optimizations, the goal is to recognize simple and tractable induction variables whose algebraic properties allow precise static analysis, including accurate dependence testing. We have implemented and integrated our algorithm in a production compiler, the GNU Compiler Collection (4.0), showing the scalability and robustness of an implementation that is the basis for several optimizations being developed, including vectorization, loop transformations and modulo-scheduling. We presented experimental results on the SPEC CPU2000 and JavaGrande benchmarks, with an application to dependence analysis. Our results show no degradations in compilation time. Independently of the algorithmic and formal contributions to induction variable recognition, this work is part of an effort to bring competitive loop transformations to the free production compiler GCC.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan and Kaufman, 2002.
3. O. Bachmann, P. S. Wang, and E. V. Zima. Chains of recurrences a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation.* ACM Press, 1994.
4. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Boston, 1992.
5. D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers Summit*, 2004. `http://www.gccsummit.org/2004`.
6. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
7. A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 82–93. ACM Press, 2004.
8. M. P. Gerlek, E. Stoltz, and M. J. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. on Programming Languages and Systems*, 17(1):85–122, Jan. 1995.
9. Gnu go. `http://www.gnu.org/software/gnugo/gnugo.html`.
10. P. Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
11. M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4), July 1996.
12. L. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in LNCS. Springer-Verlag, 1993.
13. Java grande forum. `http://www.javagrande.org`.
14. V. Kislenkov, V. Mitrofanov, and E. Zima. Multidimensional chains of recurrences. In *Proceedings of the 1998 international symposium on symbolic and algebraic computation*, pages 199–206. ACM Press, 1998.
15. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM Symp. on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.
16. S.-M. Liu, R. Lo, and F. Chow. Loop induction variable canonicalization in parallelizing compilers. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, page 228. IEEE Computer Society, 1996.

17. J. Merill. GENERIC and GIMPLE: a new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, 2003. `http://www.gccsummit.org/2003`.
18. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
19. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
20. D. Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004. `http://www.gccsummit.org/2004`.
21. D. Novillo. Tree SSA - a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers Summit*, 2003. `http://www.gccsummit.org/2003`.
22. D. Novillo. A propagation engine for gcc. In *Proceedings of the 2005 GCC Developers Summit*, 2005. `http://www.gccsummit.org/2005`.
23. K. O'Brien, K. M. O'Brien, M. Hopkins, A. Shepherd, and R. Unrau. Xil and yil: the intermediate languages of tobey. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, pages 71–82, New York, NY, USA, 1995. ACM Press.
24. M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM Press.
25. S. Pop, P. Clauss, A. Cohen, V. Loechner, and G.-A. Silber. Fast recognition of scalar evolutions on three-address ssa code. Technical Report A/354/CRI, Centre de Recherche en Informatique (CRI), École des mines de Paris, 2004. `http://www.cri.ensmp.fr/classement/doc/A-354.ps`.
26. S. Pop, A. Cohen, and G.-A. Silber. Induction variable analysis with delayed abstractions. Technical Report A/367/CRI, Centre de Recherche en Informatique (CRI), École des mines de Paris, 2005. `http://www.cri.ensmp.fr/classement/doc/A-367.ps`.
27. B. Pottenger and R. Eigenmann. Parallelization in the presence of generalized induction and reduction variables. In *ACM Int. Conf. on Supercomputing (ICS'95)*.
28. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, Aug. 1992.
29. S. Rus, D. Zhang, and L. Rauchwerger. The value evolution graph and its use in memory reference analysis. In *Proceedings of the 2004 Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.
30. Standard performance evaluation corporation. `http://www.spec.org`.
31. R. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 106–115, 2004.
32. R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'01)*, pages 118–132, 2001.
33. H. Warren. *Hacker's Delight*. Addison-Wesley, 2003.
34. M. J. Wolfe. Beyond induction variables. In *ACM Symp. on Programming Language Design and Implementation (PLDI'92)*, pages 162–174, San Francisco, California, June 1992.
35. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
36. E. V. Zima. On computational properties of chains of recurrences. In *Proceedings of the 2001 international symposium on symbolic and algebraic computation*, pages 345–352. ACM Press, 2001.