CHAPTER 1

Loop tree

S. Pop

Progress: 60%

Structural reordering in progress

This chapter presents an extension of the SSA under which the extraction of the reducible loop tree can be done only on the SSA graph itself. This extension of the SSA representation captures more than the scalar computations: the phi nodes and the scalar assignments encode the structure of the CFG and the strongly connected components of the CFG that are the reducible ~~natural~~ loops. This chapter will present two analysis algorithms: the extraction of the reducible loop tree and the analysis of induction variables based on the SSA representation.

English: that are reducible loops?

Here you do no encode the structure of the CFG but only loops.

You should express it differently as as it is presented here we think we can use your technique in place of classical loop forest detection algos. This is actually not the case as you need to have loop forest information to build this extension.

1.1 CFG and Loop Tree can be ~~discovered~~ from the SSA

You do not treat full CFG here

exposed?

During the construction of the SSA representation based on a CFG representation, a large part of the CFG information is translated into the SSA representation. As the construction of the SSA has precise rules to place the phi nodes in special points of the CFG (i.e., at the merge of control-flow branches), by identifying patterns of uses and definitions, it is possible to expose the CFG structure from the SSA representation.

Furthermore, it is possible to identify, based on the SSA definitions and uses patterns, higher level constructs inherent to the CFG representation, such as strongly connected components of basic blocks (or ~~natural~~ loops). The induction variable analysis, that we will see in this chapter, is based on the detection of self references in the SSA representation, and on the characterization of these cyclic definitions.

reducible

This first section shows that the classical SSA representation is not enough to represent the semantics of the original program. We will see the minimal amount of information that has to be added to the classical SSA representation in order to

represent the loop information in an elegant way: the loop closed SSA form adds an extra variable at the end of a loop for each variable defined in a loop and used after the loop. ~~This is similar to the Gated SSA form presented in Chapter ??~~.

A short discussion at the end of the chapter about different loop closed forms that exist and their purpose would be welcome.
1. for loop transformations and SSA update
2. its equivalent un functional programming (and its construction)
3. PDW and gated-SSA for data driven interpretation (I still do not understand how demand driven works for loops with gated-SSA without any control;
Your form is much more clear for this purpose).

1.1.1 An SSA representation without the CFG

We obtain what is sometimes named SSA graph

In the classic definition of the SSA, the CFG provides the skeleton of the program: basic blocks contain assignment statements defining SSA variable names, and the basic blocks with multiple predecessors contain phi nodes. Let's look at what happens when, starting from a classic SSA representation, we remove the CFG.

In order to remove the CFG, a pretty printer function writes the content of basic blocks by traversing the CFG structure (the CFG traversal could be performed in any order: random order, depth-first order, dominator order, etc.). Does the representation, obtained from this pretty printer, contain enough information to enable us to compute the same thing as the original program?

Let's see what happens with an example: supposing that the original program looks like this (in its CFG based SSA representation):

```
bb_1 (preds = {bb_0}, succs = {bb_2})
{
  a = #some computation independent of b
}
bb_2 (preds = {bb_1}, succs = {bb_3})
{
  b = #some computation independent of a
}
bb_3 (preds = {bb_2}, succs = {bb_4})
{
  c = a + b;
}
bb_4 (preds = {bb_3}, succs = {bb_5})
{
  return c;
}
```

after removing the CFG structure, using a random order traversal, we could obtain this:

```
  return c;
  b = #some computation independent of a
  c = a + b;
  a = #some computation independent of b
```

and this SSA code is enough, in the absence of side effects, to recover an order of computation that leads to the same result as in the original program. For example, the evaluation of this sequence of statements would produce the same result:

```
  b = #some computation independent of a
  a = #some computation independent of b
  c = a + b;
  return c;
```

1.1.2 ~~Discovering~~ natural loop structures on the SSA

We will now see how to represent the natural loops in the SSA form by systematically adding extra phi nodes at the end of loops, together with extra information about the loop exit predicate.

Supposing that the original program contains a loop:

```
bb_1 (preds = {bb_0}, succs = {bb_2})
{
  x = 3;
}
bb_2 (preds = {bb_1, bb_3}, succs = {bb_3, bb_4})
{
  i = phi (x, j)
  if (i < N) goto bb_3 else goto bb_4;
}
bb_3 (preds = {bb_2}, succs = {bb_3})
{
  j = i + 1;
}
bb_4 (preds = {bb_2}, succs = {bb_5})
{
  k = phi (i)
}
bb_5 (preds = {bb_4}, succs = {bb_6})
{
  return k;
}
```

Pretty printing, with a random order traversal, we could obtain this SSA code:

```
  x = 3;
  return k;
  i = phi (x, j)
  k = phi (i)
  j = i + 1;
```

We can remark that some information is lost in this pretty printing: the exit condition of the loop disappeared: we will have to record this information in the extension of the SSA representation. The information about the ~~natural~~ loop is still available

under the form of a cyclic definition: by simple substitutions, we can rewrite this SSA code to expose the self reference to "i", as:

```
i = phi (3, i + 1)
k = phi (i)
return k;
```

Thus, we have the definition of the SSA name "i" defined in function of itself. This pattern is characteristic of the existence of a ~~natural~~ loop. We can remark that there are two kinds of phi nodes used in this example:

reducible

- loop-ϕ nodes "i = phi (x, j)" have an invariant argument "x" (i.e., the definition does not depend on the values that the phi node takes) and an argument that contains a self reference "j" (i.e., the defining expression "j = i + 1" contains a reference to the same loop-ϕ definition "i"). It is possible to define a canonical SSA form by limiting the number of arguments of loop-ϕ nodes to two.
- close-ϕ nodes "k = phi (i)" merge the values in the end of a loop. They are used to capture the last value of a name defined in a loop. Names defined in a loop can only be used within that loop or in the arguments of a close-ϕ node (that is "closing" the set of uses of the names defined in that loop). In a canonical SSA form it is possible to limit the number of arguments of close-ϕ nodes to one.

1.1.3 *Improving the SSA pretty printer for loops*

As we have seen in the above example, the exit condition of the loop disappears during the basic pretty printing of the SSA. To capture the semantics of the computation of the loop, we have to specify in the close-ϕ node, which value will be available in the end of the loop. And thus we have to slightly modify the syntax of the close-ϕ nodes to also contain the loop exit condition. This representation is also known under the name of Gated SSA, as presented in Chapter **??**. With this extension, the SSA pretty printing of the above example would be:

Actually this is not strictly gated SSA as here:
- you focus only on loops (handle of loops is very fuzzy in GSSA)
- GSSA would have a predicate (not an expression)

```
x = 3;
i = loop-phi (x, j)
j = i + 1;
k = close-phi (i >= N, i)
return k;
```

So "k" is defined as the first value of "i" satisfying the loop exit condition, "i >= N". This is a well defined value (in the case of finite loops), as the sequence of values that "i" takes, is defined by the loop-ϕ node and taking the first element of that sequence satisfying the loop exit condition.

for this example yes. But what about the case where you would have another computation inside this loop?
What about your loop construct. I am still confused about what you are aiming to show in this section. It seems that you still need what you call a loop construct if you want to keep the full semantic of your program.

In the next section, we will look at an algorithm that translates the SSA representation into a representation of polynomial functions, describing the sequence of values that SSA names take during the execution of a loop.

1.2 Analysis of Induction Variables

The purpose of the induction variables analysis is to provide a characterization of the sequences of values taken by a variable during the execution of a loop. This characterization can be an exact function of the iteration counter of the loop (i.e., the canonical induction variable of a loop starts at zero with a step of one for each iteration of the loop) or an approximation of the values taken during the execution of the loop represented by values in an abstract domain. In this section, we will see a possible characterization of induction variables in terms of scalar sequences. The domain of scalar sequences will be represented by the chains of recurrences: ~~as we will see in more detail in this section, the canonical induction variable will be~~ syntactically represented by the chain of recurrence $\{0, +, 1\}_x$ the initial value of the canonical induction variable is 0, the stride is 1~~, and the evolution happens in loop number x~~.

??

as an exple, a canonical induction variable (initial value 0 and stride 1 that would occur in loop "x" will be ...

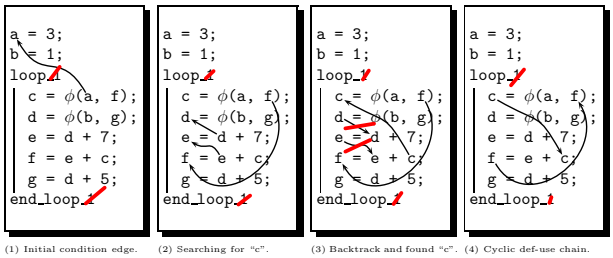
1.2.1 *Stride detection*

The first step of the induction variables analysis is the detection of the strongly connected components of the SSA. This can be performed by traversing the SSA chains (from a use to its unique definition) and detecting that some definitions are visited twice. From a self referring use-def chain, it is possible to compute the overall effect of one iteration of the loop on the cyclic definition: this is the step of the induction variable. When the step of an induction variable depends on another cyclic definition, one has to further analyze the inner cycle. The analysis of the induction variable ends when all the inner cyclic definitions used for the computation of the step are analyzed. Note that it is possible to construct SSA graphs with strongly connected components that are impossible to characterize with the chains of recurrences. For example, in the following example, "a" does not follow a linear progression:

```
a = loop-phi (0, b)
c = loop-phi (1, d)
b = c + 2
d = a + 3
```

Let's look at an example, presented in Figure 1.1, to see how this algorithm works. The arguments of a phi node are analyzed to determine whether they contain self references or if they are pointing towards the initial value of the induction variable. In this example, (1) represents the edge that points towards the invariant definition. When the argument to be analyzed points towards a longer use-def chain, the full chain is traversed, as shown in (2), until a phi node is reached. In this example, the phi node that is reached in (2) is different to the phi node from which the analysis started, and so in (3) a search starts over the uses that have not yet been analyzed. When the original phi node is found, as in (3), the cyclic def-use

Please remove the edges from d to d and e to e in (3). They are confusing as it does not corresponds to actual arcs.



loop_1 => loop_x

Fig. 1.1 Detection of the cyclic definition. using a depth first search traversal of the use-def chains

chain provides the step of the induction variable: in this example, the step is "+ e". Knowing the symbolic expression for the step of the induction variable may not be enough, as we will see next, one has to instantiate all the symbols ("e" in the current example) defined in the varying loop to precisely characterize the induction variable.

1.2.2 *Translation to chains of recurrences*

Once the cyclic def-use chain has been outlined and the overall loop update expression has been identified, it is possible to translate the sequence of values of the induction variable to a chain of recurrence. The syntax of a polynomial chain of recurrence is: $\{base, +, step\}_x$, where "base" and "step" may be arbitrary expressions or constants, and "x" is the loop in which the sequence is generated. As a chain of recurrence represents the sequence of values taken by a variable during the execution of a loop, the semantics of a chain of recurrence is given by $\{base, +, step\}_x(\ell_x) = base + step * \ell_x$, that is a function of ℓ_x that represents the number of times the body of loop number x has been executed.

Do we need to number loops in this chapter? I would just say loop "x" and replace 3 by x further. Indeed this is not clear how you handle non perfect nested loop with your loop numbering and corr. iteration vector.

?? English??

When "base" or "step" translates to sequences variating in outer loops, the resulting sequence is represented by a multivariate chain of recurrences. For example $\{\{0, +, 1\}_3, +, 2\}_4$ defines a multivariate chain of recurrence with a step of 1 in loop number 3 and a step of 2 in loop number 4.

3=>x ; 4=>y

x y where loop x is enclosed in loop y

When "step" translates into a sequence variating in the same loop, the chain of recurrence represents a polynomial of a higher degree. For example, $\{0, +, \{4, +, 5\}_3\}_3$ represents a polynomial evolution of degree 2 in loop number 3. In this case, the chain of recurrence is also written omitting the extra braces: $\{0, +, 4, +, 5\}_3$. The semantics of the chains of recurrences is defined, using the binomial coefficients $\binom{n}{p}$, by the equation:

the corresponding code as in fig 1.1 would be welcome

x

take the example of fig 1.1 instead: {3,+,{1,+,12}_x}_x

$$\{c_0, +, c_1, +, c_2, +, \ldots, +, c_n\}_x(\ell) = \sum_{p=0}^{n} c_p \binom{\ell_x}{p}$$

l_x

x

Please recall how to compute C^n_p

Please refer to remark end of page 5 in previous review

with $\boldsymbol{\ell}$ the iteration domain vector (the iteration loop counters for all the loops in which the chain of recurrence variates), and ℓ_k the iteration counter for loop number k. This semantics allows the rewriting rule:

$$\{c_0, +, \{c_1, +, c_2\}_x\}_x = \{c_0, +, c_1, +, c_2\}_x$$

that is very useful in the analysis of induction variables, as it makes it possible to split the analysis into two phases, with a symbolic representation as a partial intermediate result:

- first, the analysis leads to a symbolic form, where the step part "s" is left in a symbolic form, i.e., $\{c_0, +, s\}_x$;
- then, by instantiating the step, i.e., $s = \{c_1, +, c_2\}_x$, the chain of recurrence is that of a higher degree polynomial, i.e., $\{c_0, +, \{c_1, +, c_2\}_x\}_x = \{c_0, +, c_1, +, c_2\}_x$.

cf my last review
Maybe you should provide the formula using binomial coefficients first.
Then list the different rewriting rules (including the ones I mentioned in previous review), then apply it to the cases where base or step translate to sequences variating in outer/same loop

1.2.3 Instantiation of symbols and region parameters

what about end value of inner loops?

The last step of the induction variable analysis consists in the instantiation (or further analysis) of symbolic expressions left from the first step. This includes the analysis of induction variables in outer loops, the analysis of the end value of a loop preceding the analyzed loop, and the replacement of any definitions occurring in sequence before the loop with their defined expression. In some cases, it becomes necessary to leave in a symbolic form every definition outside a given region, and these symbols are then called parameters of the region.

Let's look again at the example of Figure 1.1 to see how the sequence of values of the induction variable "c" is characterized with the chains of recurrences notation. The first step, after the cyclic definition is detected, is the translation of this information into a chain of recurrence: in this example, the initial value (or base of the induction variable) is "a" and the step is "e", and so "c" is represented by a chain of recurrence $\{a, +, e\}_1$ that is variating in loop number 1. The symbols are then instantiated: "a" is trivially replaced by its definition leading to $\{3, +, e\}_1$. The analysis of "e" leads to this chain of recurrence: $\{8, +, 5\}_1$ that is then used in the chain of recurrence of "c", $\{3, +, \{8, +, 5\}_1\}_1$ and that is equivalent to $\{3, +, 8, +, 5\}_1$, a polynomial of degree two:

$$\begin{aligned} F(\ell) &= 3\binom{\ell}{0} + 8\binom{\ell}{1} + 5\binom{\ell}{2} \\ &= \frac{5}{2}\ell^2 + \frac{11}{2}\ell + 3. \end{aligned}$$

1.2.4 Number of iterations and computation of the end of loop value

One of the important properties of loops is their trip count (i.e., the number of times the loop body is executed before the exit condition becomes true). In simple loops, the exit condition of the loop is a comparison of an induction variable against some constant, parameter, or another induction variable. The number of iterations is then computed as the solution of an equation with integer coefficients and integer solutions, also called a Diophantine equation. When one or more coefficients of the Diophantine equation are parameters, the solution is left under a parametric form. The number of iterations can also be an expression variating in an outer loop, in which case, it can be characterized using a chain of recurrence expression.

With an expression representing the number of iterations in a loop, it becomes possible to express the evolution functions of scalar variables variating in outer loops with strides dependent on the value computed in an inner loop: the overall effect of a loop on a scalar variable can be computed as an apply of the number of iterations on the evolution function (of the scalar variable) in the varying loop.

For example, the following code

```
x = 0;
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    x = x + 1;
```

Faut nommer les boucles ici genre avec un commentaire
// loop x
// loop y
Il faut utiliser les même notations partout, des fois il y a des boucles des fois pas...

would be written in loop closed SSA form as:

```
a = 0
b = loop_1_phi (a, c)
c = close_2_phi (j < M, d)
d = loop_2_phi (b, d + 1)
e = close_1_phi (i < N, b)
i = loop_1_phi (0, i + 1)
j = loop_2_phi (0, j + 1)
```

remplacer a,b,c,d,e par x_1, x_2...

"e" represents the value of variable "x" at the end of the original imperative program. The analysis of scalar evolutions for variable "e" would trigger the analysis of scalar evolutions for all the other variables defined in the loop closed SSA form as follows:

- first, the analysis of variable "e" would trigger the analysis of "i", "N" and "b"
  - the analysis of "i" leads to $i = \{0, +, 1\}_1$
  - "N" is a parameter and is left under its symbolic form
  - the analysis of "b" triggers the analysis of "a" and "c"
    - the analysis of "a" leads to $a = 0$
    - analyzing "c" triggers the analysis of "j", "M", and "d"
      - $j = \{0, +, 1\}_2$
      - "M" is a parameter
      - $d = loop\_2\_phi(b, d + 1) = \{b, +, 1\}_2$

  - $c = close\_2\_phi(j < M, d)$ is then computed as the last value of "d" after $loop_2$, i.e., it is the chain of recurrence of "d" applied to the first iteration that does not satisfy $j < M$ and because "j" counts the number of times the body of $loop_2$ has run, that is the number of iterations of $loop_2$, i.e., M. So, to finish the computation of the scalar evolution of "c" we apply "M" to the scalar evolution of "d", leading to $c = \{b, +, 1\}_2(M) = b + M$;
- the scalar evolution analysis of "b" then leads to $b = loop\_1\_phi(a, c) = loop\_1\_phi(a, b + M) = \{a, +, M\}_1 = \{0, +, M\}_1$

Non! En pratique, tu cherches le plus petit m>=0 qui ne satisfasse pas {0,+,1}_2(m)<M, ie min(m) st m>=M. Il faut dire comment dans le cas général tu résouds cette équation.

- and finally the analysis of "e" ends with $e = close\_1\_phi(i < N, b) = \{0, +, M\}_1(N) = M * N$.

The computation of the end of loop value can also be used for optimization purposes, for example in the case of the constant propagation or the value range propagation after loops. It enables more scalar transformations when the induction variable is used after its defining loop. Another interesting use of the end of loop value is the estimation of the worst case execution time (WCET) where one tries to obtain an upper bound approximation of the time necessary for a program to terminate.

This is also useful when performing loop transformations

1.3 Conclusion

See previous review

As we have seen in this chapter, the CFG representation is embedded in the SSA structure, and properties of the CFG like the execution order and the natural loops can be detected by only looking at the SSA definitions and uses. The detection of natural loops is the first step in the analysis of induction variables: after the detection of self referent definitions, it is practical to use the chains of recurrences [1, 2, 3] to characterize the sequence of values taken by a variable during the execution of a loop. The number of iterations of loops can be computed based on the characterization of induction variables. This paves the way to advanced loop optimizations that need both the number of iterations and induction variable characterizations.

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

References

1. Olaf Bachmann, Paul S. Wang, and Eugene 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*, pages 242–249. ACM Press, 1994.
2. 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.
3. Eugene 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.