not much changes:
- mostly the layout of the first section and be more clear of what you want to show
- some references/explanation to how we build this SSA extension + references to the SSA graph chapter (Gated SSA)
- introduce all the notions you use about chain of recurrences etc. The less notations the better.
- more examples for the second section to understand the restrictions and to manipulate the formalism.
- change the style of the last section (currently conclusion) with more references, further reading

Chapter 1

Loop tree —(*S. Pop*)

No: 1. you present an extension of SSA under which the extraction of (reducible) loop tree can be done only on the SSA graph itself. 2. use this representation to recognize a characterize IV using polynoms

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 natural loops. This chapter will present two analysis algorithms: the extraction of the natural loop tree and the analysis of induction variables based on the SSA representation.

loop tree & IV detection on SSA

reducible loop (natural loop not well defined)?

1 CFG and Loop Tree can be discovered from the SSA

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.

Intro: identifying loops...

1.1 An SSA representation without the CFG

say here the goal of this section:
1. show that SSA alone is not enough to represent the program
2. give here the different things that your gonna add to SSA to represent loop info in an elegant way:
- loop close form & canonical SSA (even if SSA already exposes the connected components of the CFG
- conditions on phi (similar to gated SSA)
Otherwise we do not know where we are going and we think that SSA is enough to represent the program.

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?

Intro: does SSA represents the complete prog semantic?

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;
}
```

On an example:
for code without branches
use-def chains are
"sufficent".

- /!\ without side effect

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 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;
```

say that this obviously works because you do not have side effects

1.2 Discovering natural loop structures on the SSA

Now supposing that the original program contains a loop:

recall here what extensions you are going to present

```
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)
```

•with some control: a loop example
with a phi at the exit

- not minimal SSA

```
}
bb_5 (preds = {bb_4}, succs = {bb_6})
{
  return k;
}
```

(con't)

Awkward

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;
```

say that you are going to tackle this pb further

some of the information is lost.
"loop carried" variables can be detected

We can remark that some information is lost in this pretty printing: the exit condition of the loop disappeared. However, 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;
```

don't undertsand what you want to show her
I would simplify all this.
This is more misleading than really helpful ·

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 would be able to find an execution order that satisfies the scalar dependences with the help of a loop construct:

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

requires a loop construct

We can remark that there are two kinds of phi nodes used in this example:

cite the example:
i=phi(x,j)
x
j
j=i+1
k=phi(i)

- loop-ϕ nodes have an invariant argument (i.e., the definition does not depend on the values that the phi node takes) and an argument that contains a self reference (i.e., the defining expression contains a reference to the same loop-ϕ definition). It is possible to define a canonical SSA form by limiting the number of arguments of loop-ϕ nodes to two.
- close-ϕ nodes 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.

loop phi-nodes:
i=phi(inv, f(i))

close phi-nodes
k=phi(i)

non minimal SSA form
with close phi-nodes

say somewhere how you build this SSA form

1.3 Improving the SSA pretty printer for loops

refer to the chapter of SSA graph and Gated-SSA somewhere

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. With this extension, the SSA pretty printing of the above example would be:

adding conditions on phi

```
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.

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.

2 Analysis of Induction Variables

IV characterization

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 polynomial scalar sequences. The domain of polynomial scalar sequences will be represented by the polynomial chains of recurrences [1,2,3].

??
I know what is a chain of recurrence, but what is a polynomial...

2.1 Stride detection

IV detection:
-circuit of use-def chains = 1 iteration
- that my depend on another IV

The first step of the induction variables analysis is the detection of the strongly connected components of the SSA ~~(i.e., self-references or cyclic definitions)~~. 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 the cyclic 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.

graph cannot have self-references in SSA.

illustrate here what you want to do:
- give an example
- give the result in term of chains of recurrence (which semantic is given further)
- give the corresponding close function

illustrate here that you do not treat this case:

```
d3=phi(d1,d2)
a3=phi(a1,a2)
d2=a3+1
a2=d3+2
```

not treated (need 2-unrolling)

put all the use-def chains and highlight (eg in red) your actual traversal. Outline that it is nothing else than finding circuits in SSA graph.

Say that you stop at phi. And explain why

look for circuits: follow use-def chains. stop on phi.

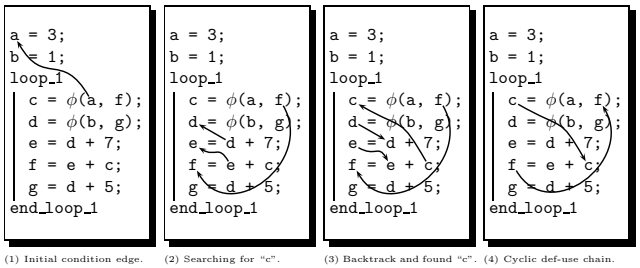

Fig. 1. Detection of the cyclic definition.

Let's look at an example, presented in Figure 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 backtrack search starts, exploring the uses that have not yet been analyzed. When the original phi node is found, as in (3), the cyclic def-use 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 defined in the varying loop to precisely characterize the induction variable.

find a circuit deduce the symb. expression of the step.

talking about backtracking is misleading. You just have to represent the full use-def chains (that stop @ phi)

e in our example here

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 [1,2,3]. 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. Note that when "base" or "step" translates to sequences variating in outer loops, the resulting sequence is represented by a multivariate chain of recurrences. When "step" translates into a sequence variating in the same loop "x", the chain of recurrence represents a polynomial of a higher degree.

polynom represented as a multivariate chain of recurrences with +&-

provide the definition of a chain of recurrence or at least the restricted notion that you need.
say {base,+,step}= base+Sum...
Say that we denote by {c0,+,c1...} {c0,+,{c1...}}
That if you have only + (a minus can be represented with a negative ci) this is a polynome and you give the expression.

The semantics of the chains of recurrences is defined, using the binomial coefficients $\binom{n}{p}$, by the equation:

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

recursive notation

semantic?

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.

you should restrict here to a notation with only one index as I do not see how you could represent a variable that variate in nexted loops.
See the example here that need to be tackled using some simplifications as you explain later on.


```
a=0
for i=1:N
  for j=1:M
    a=a+1
```

L=(i,j)
l_k==j
... not handled by the notations (need to be solved in 2 steps)

those are interesting rewriting rules
- the first when you rely on 2 other IVs
- the second for an IV on nested loops

rewriting rules

- {0,+,{0,+,1}+{0,+,2}}={0,+{0,+,3}}
- {ai0,+,1}(M)=ai0+{0,+,1}(M)=ai0+M

this is a definition of the notation introduced above

This semantics allows the rewriting rule:

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

what is useful is not the notation. It is the fact that you can express using a polynom a "nested" chain of recurrence.

that is very useful in the analysis of induction variables, as it allows the partial analysis of the outer induction variable:

?

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

2.3 Instantiation of symbols and region parameters

instantiation of expressions. Possibly left as parameter

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.

running example

Let's look again at the example of Figure 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:

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

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

application to #loop_iteration

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.

you should illustrate this point by running the example I gave page 5

IV on two nested loops

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.

also enable loop transformation when IV used after the loop

also WCET computation

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.

3 Conclusion

not paper style. I would prefer a last section that provides additional references for extensions etc. Also, the body of the chapter should be self content so no reference to any paper (to other chapters are possible). So you should explain what is a chain of recurrence etc. Then in the last section you can refer the papers on chain of recurrences, on abstract interpretation etc.

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 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.