CHAPTER 1

Static Single Information Form

F. Pereira

Progress: 20%

Material gathering in progress

1.1 Introduction

Excellent!

why we call that SSI

The objective of a dataflow analysis is to discover facts that are true about a program. We call such facts *informations*. Using the notation introduced in Section **??**, an information is a point in the dataflow lattice. For example, liveness analysis is interested in finding out the set of variables alive at a certain program point. Similarly to liveness analysis, many other classic dataflow approaches bind information to pairs formed by a variable and a program point. However, if the information is true for a variable v at any program point where v is alive, then we can associate this information directly to v. If a program's intermediate representation guarantees this correspondence between information and variable for every variable, then we say that the program representation provides the *Single Static Information* (SSI) property.

Different dataflow analysis might extract information from different program facts. Therefore, a program representation may afford the SSI property to some dataflow analysis, but not to all of them. For instance, the SSA form naturally provides the SSI property to the reaching definition analysis. Indeed, the SSA form provides the static single information property to any forward ~~dataflow~~ analysis that obtains information at the definition sites of variables. (see Chapter ...). ~~These analyses and transformations include classic constant propagation, as seen in Chapter **??**, and simple versions of type inference, for instance~~. However, the SSA form does not provide the SSI property to a dataflow analysis such as ... that derives information from the use site of variables. In

some existing "SSI" forms

Here you should outline that this chapter is intended to revisit the SSI of Ananian and provide the formalism to generalize it. To be clear SSI here does not refer specificaly to SSI of Ananian & Singer.

I love it! :-)

other words, because the same variable v in a SSA form program might be used at different program points, the informations associated to v might not be unique.

con't

There exist extensions of the SSA form that provide the SSI property to more dataflow analyses than the original SSA does. Two classic examples are the *Extended-SSA* (e-SSA) form ~~introduced by Bodik *et al.* [3]~~, and the *Static Single Information* (SSI) form ~~introduced by Ananian [1]~~. The e-SSA form provides the SSI property to analyses that take information from the definition site of variables, and also from conditional tests where these variables are used. Ananian's SSI form gives the static single information property to dataflow analyses that extract information from particular use sites of variables, such as *Busy Expressions Analysis*. A common strategy that any of these intermediate representations use to achieve the SSI property is *live range splitting*. In this chapter we show how to use live range splitting to build program representations that provide the static single information property to different types of dataflow analysis.

No references.
Say that we will come back to those extension later and described them further.

=> live-range splitting

1.2 Live Range Splitting

A program representation that splits the live range of each variable between each pair of consecutive program points naturally provides the SSI property to any dataflow analysis that discovers facts about variables. Actually, such a program representation exists: it is called *elementary form*, ~~and it has even seen use in optimizations such as register allocation [2, 7]~~. However, this strategy lends to the dataflow analysis many identity transfer functions. For instance, in liveness analysis, an instruction that does not use nor define any variable is bound to an identity transfer function: it simply propagates the values that come from its successors to its predecessors. Therefore, we would like to split the live ranges of variables only at program points where information about that variable is produced.

split everywhere:elementary form

please no refs in the main part. Only in the last section

Maybe we should remove this sentence as it is redundant with const_prop chapter

split at id transfert functions only

We call a *generator* a program point that produce information about a variable. A generator is usually an instruction, such as a use or a definition of a variable in our liveness analysis example. However, generators can also be conditional branches or even the entry/exit nodes of the program's control flow graph (CFG). Some program points are considered *meet nodes*, because they combine the information that comes from two or more regions. If a program point represents a meet node, then we call it a *combinator*. Ideally, we would like to split the live ranges of variables only at program points which are either generators or combinators.

generator, meet nodes, combinators

1.2.1 Special instructions to split live ranges

We split the live range of a variable v at program point p in a ~~two~~ three steps process. Firstly, we insert a copy ~~$v' = v$~~ v=v at p. ~~Secondly, we rename to v' every use of v at~~

As we want to enforce the resulting code to be under SSA, we then insert appropriate phi functions to enforce every use of v to be reachable and dominated by only one definition of v.
Finally, as for SSA construction, we rename (version) each new definition of v and and its corresponding uses accordingly.

SSI construction

~~any program point p' dominated by p. After performing the renaming phase~~, we ~~might have to insert ϕ-functions in the source program in order to avoid having two different denotations of the same variable alive together~~.

~~In principle we could split the live ranges of many variables at the same program point using simple copies, however, this notation brings semantic complications: which variable was created first? Which variables are alive together? In order to avoid such questions, we use special instructions to perform live range splitting.~~ There exists three basic types of program points where we might split live ranges: interior nodes, branches and joins. At each place we use a different notation to represent live range splitting.

In practice, we do not use simple copies to perform live-range splitting.

copies not "enough". Different places where to split.

nodes

~~We call *joins*~~ the program points that have one successor and multiple predecessors. Notice that because each program point exists before or after a program instruction, there exist no program point with multiple predecessors and successors. We split live ranges at join nodes via ϕ-functions. Therefore, the SSA form already suits dataflow analysis that combine information at these nodes.

phi

When we split at an interior node, we split on an instruction, not between 2 instructions (eg forward prop of of a use).

Interior nodes are program points that have a unique predecessor and a unique successor. At these points we perform live range splitting via parallel copies. A parallel copy such as:

$$(v_1, \ldots, v_n) = (v'_1, \ldots, v'_n)$$

// copy

performs ~~n~~ assignments like $v_i = v'_i$; ~~all in parallel~~. These copies are used mostly by dataflow analysis that generate information from the uses of variables. As a first example, lets consider a ~~the~~ backward analysis ~~used by An *et al.*~~ to type Ruby programs ~~[5]~~. The set of infered types of a variable v might be different before and after a use point p where v is used. For instance, if we have a use of v such as $v.m()$, then we know, from this point backwards, that v must have the method m. We should split the live range of v at p by inserting a copy ~~$(v') = (v)$~~ at that point, ~~where v' is a fresh name~~.

semantically

simultaneously

Could you avoid restricting to a specific language your example.

You already talked about renaming etc.

split or

~~A~~ *branch point*, is a program point with a unique predecessor and multiple successors. Going back ~~to An's type inference engine~~ ~~[5]~~, we may have that the definition of a variable v feeds two uses of this variable, e.g, $v.m_1()$ and $v.m_2()$, through different program paths. We would like to infer that v contains both methods, $m_1()$ and $m_2()$. Put in another way, the use that reaches ~~the definition of v must be unique~~, in the same way that in a SSA-form program the definition ~~that reaches a use is~~ unique. We ensure this uniqueness via another type of special instructions: the σ-functions. Consider, for instance, the assignment below:

our type inference engine example

the way it is writen use-def chains seems to be sufficient.

(forward)

(backward)

$$[(v_{11}, \ldots, v_{n1}) : l_1, \ldots (v_{1m}, \ldots, v_{nm}) : l_m] = \sigma(v_1, \ldots, v_n)$$

which represents n σ-nodes such as $(v_{i1} : l_1, \ldots, v_{im} : l_m) \leftarrow \sigma(v_i)$. This instruction assigns to each variable v_{ij} ~~: l_j~~ the value in v_i if control flows into block l_j. These assignments happen in parallel, i.e., these n σ-functions encapsulate m parallel copies. Also, notice that variables alive in different branches of a basic block are given different names by the σ-function that ends that basic block.

sigma

There is a beautiful symmetry between ϕ- and σ-functions. Whereas the latter has the functionality of a variable multiplexer, the former is analogous to a demultiplexer, that performs a parallel assignment depending on the execution path taken. ~~To see this duality more clearly, we can represent a ϕ-function as a set of m parallel copies:~~

$$(v_1, \ldots, v_n) = \phi[(v_{11}, \ldots, v_{n1}) : l_1, \ldots (v_{1m}, \ldots, v_{nm}) : l_m]$$

The assignment above contains n ϕ-functions such as $v_i \leftarrow \phi(v_{i1} : l_1, \ldots, v_{im} : l_m)$. It will assign to each v_i the value in v_{ij}, where j is determined by l_j, the basic block last visited before reaching the ϕ assignment. ~~Notice that, just like in the case of σ-functions, these assignments happen in parallel~~.

1.2.2 *Examples of Live Range Splitting*

In order to ensure the static single information property, different dataflow analysis do live range splitting in different ways. In this section we will see some meaningful examples.

i would remove this example as this is redundant with chapter prop engine somehow. We should keep this chapter on extensions.

SSA for points-t-analysis (forward from defs)

Points-to analysis

Points-to analysis computes an approximation of the set of memory locations that can be pointed-to by a given variable. In this case, information is extracted from the definition sites of variables. Thus, the SSA form already gives to alias analysis the static single property, as we illustrate in Figure 1.1. Figure 1.1(a) shows the control flow graph of an example program, and Figure 1.1(b) shows the same CFG, this time in SSA form. Notice that we have augmented instructions with labels, which we will use later to explain where to split live ranges. The ~~SSA graph~~ of this program~~, as discussed in Chapter **??**~~, is given in Figure 1.1(c). If we represent by I the information bound to each variable, then from our program representation we derive the constraint system shown Figure 1.1(d) A fix-point to this constraint system is a solution to our dataflow problem. Notice that the solution of this dataflow problems depends only on the inter-relations between the program variables. As we will see in the rest of this section, this is a common characteristics of dataflow analyses that bind information to the live ranges of variables.

to keep if you remove this example

def-use chains

I would prefer not to represent any graph. Program povided in an equational way (keep all the instructions, remove the CFG) seems sufficient. If you really want a graph then SSA graph is use-def chains and not the reverse. Moreover, this is much more useful to put the full instruction as a label of nodes.

Type Inference

A type inference engine learns information from the uses of a variable, and tries to assign a principal type to this variable's definition, in such a way that the program would be semantically correct. Lets illustrates an instance of type inference using the Python program in Figure 1.2(a). Our objective is to infer the correct suite of methods for each object bound to variable v. Figure 1.2(b) shows the control flow graph of the program, and Figure 1.2(c) shows the methods inferred to variable v

type inference backward from uses

Notation I(...). You need to use the same both in chapter array SSA, propagation engine and this one.

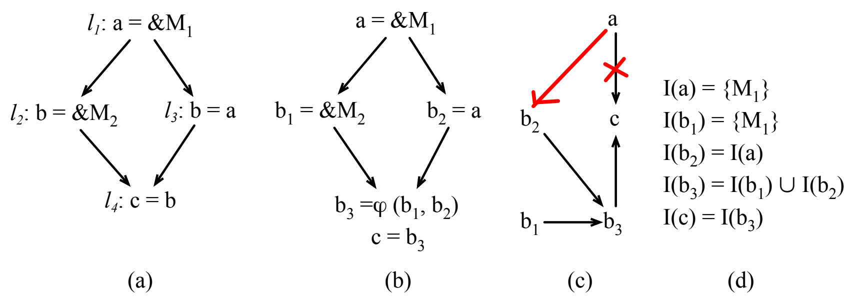


Fig. 1.1 Alias analysis as an example of forward dataflow analysis that takes information from the definition of variables.

didn't check. Have to be rewriten using a different example.

at each program point. Some instructions, such as that in label l_3, have no influence on these sets. On the other hand, instructions like those at l_4, l_6 or l_7 let us infer methods that belong into v. Similarly, the instructions at l_1 and l_5 kill the information that comes from their predecessors. All these instructions generate some sort of information to the type inference engine. Finally, the instruction at l_2 combines the possibly different informations that flow back from the two program paths that originate at it. These six program points are bound to non-identity transfer functions, in the context of the type inference analysis. If we split the live range of v at these program points, then we get the program representation given in Figure 1.2(d). We do not need to split v's live range neither at l_1 nor at l_5, because v is not alive before these points. Notice that the ϕ-function at l_7 only exists because we want to stay in SSA-form; hence, it joins the names created due to live range splitting. The constraints that characterize this dataflow program are given in Figure 1.2(e).

con't (on the example)

Taint analysis

(located at branch points)

(located at use points)

The objective of taint analysis ~~[8]~~ is to find program vulnerabilities. In this case, a harmful attack is possible when input data reaches sensitive program sites without going through special functions called sanitizers. Figure 1.3 illustrates this type of analysis. We have used ϕ and σ-functions to split the live ranges of the variables in Figure 1.3(a) producing the program in Figure 1.3(b). This intermediate representation, called *Extended Static Single Assignment* (e-SSA) form, is the same introduced by Bodik *et al.* [3] to remove redundant array bound checks.

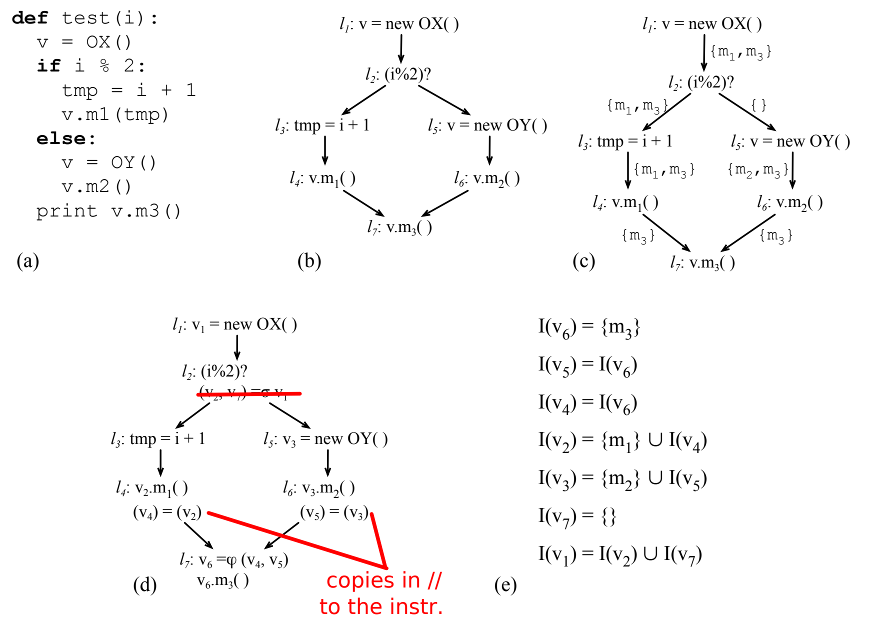
related work, discussions in the last section

Lets assume that *echo* is a sensitive function, because it is used to generate web pages. For instance, if the data passed to *echo* is a JavaScript program, then we could have an instance of cross-site scripting attack. Therefore, from this last figure we know that the instruction *echo* v_1 is a source of vulnerabilities, as it outputs data that comes directly from the program input. On the other hand, we know that *echo* v_2 is always safe, for variable v_2 is initialized with a constant value. We can

as

e-SSA for taint analysis (forward from defs and conds)

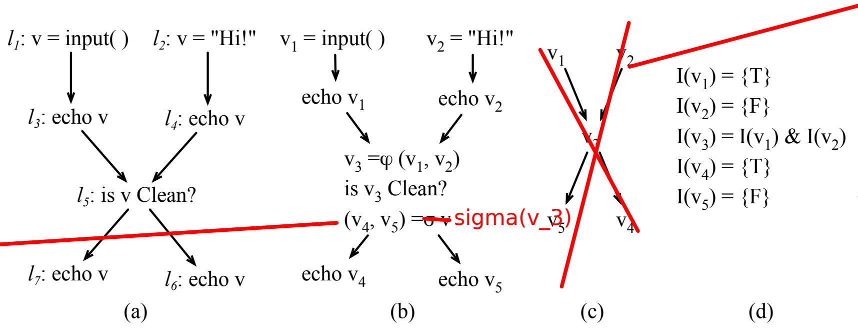
I do not like this example as v is not live on the right branch => there is no sigma



$I(v_6) = \{m_3\}$
$I(v_5) = I(v_6)$
$I(v_4) = I(v_6)$
$I(v_2) = \{m_1\} \cup I(v_4)$
$I(v_3) = \{m_2\} \cup I(v_5)$
$I(v_7) = \{\}$
$I(v_1) = I(v_2) \cup I(v_7)$

Fig. 1.2 Type inference analysis as an example of backward dataflow analysis that takes information from the uses of variables.

I would remove this as (b) is sufficient



$I(v_1) = \{T\}$
$I(v_2) = \{F\}$
$I(v_3) = I(v_1)$ & $I(v_2)$
$I(v_4) = \{T\}$
$I(v_5) = \{F\}$

in // to the instr.

/!\ Insert prog points for the phi and for the sigma in fig. (a). In every figures /!\

Fig. 1.3 Taint analysis as an example of forward dataflow analysis that takes information from the definitions of variables and conditional tests on these variables.

use specific functions, the *sanitizers*, to clean data, or to check that data is safe. The call *echo* v_5 is always safe, because the variable v_5 has been sanitized; however, the call *echo* v_4 might be tainted, as variable v_4 results from a failed attempt to sanitize v. ~~The SSA graph is given in Figure 1.3(c)~~, and the corresponding constraints are shown in Figure 1.3(d).

In case you remove (c)

con't

in // with the instr.

idem

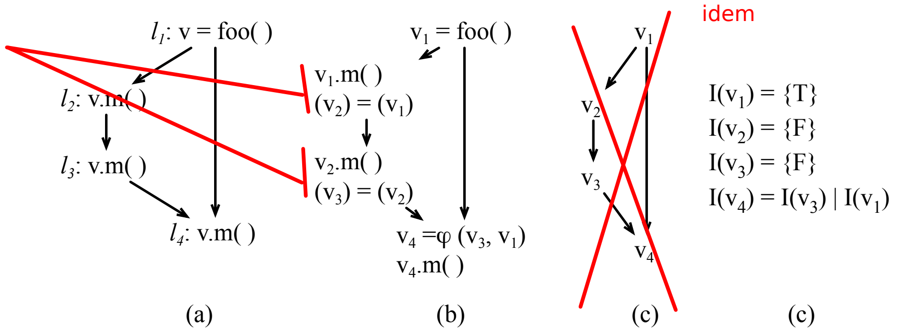


Fig. 1.4 Null pointer analysis as an example of forward dataflow analysis that takes information from the definitions and uses of variables.

Null pointer analysis

no refs

null pointer analysis
foward from uses and defs

The objective of null pointer analysis is to determine which references may hold null values. Nanda and Sinha have used a variant of this analysis to find which method dereferences may throw exceptions, and which may not [**?**]. This analysis allows compilers to remove redundant null-exception tests and helps developers to find null pointer dereferences. Figure 1.4 illustrates an example of this analysis. Because we take information from use sites, in Figure 1.4(b) we have split the live range of each variable right after it its used. Thus, we know, for instance, that the call $v_2.m()$ cannot be null, otherwise an exception would have been thrown during the invocation $v_1.m()$. On the other hand, ~~in Figure 1.4(c) we notice that the state of~~ v_4 is the meet of the state of v_3, definitely not-null, and the state of v_1, possibly null, and we must conservatively assume that v_4 may be null. Notice that the program representation that we use for a given dataflow problem depends on the level of detail that we obtain from the source code. Null pointer checks are implemented as exceptions, which are guarded by conditional tests. If we had access to the code of these tests, then we would implement the null pointer analysis using the same representation that we use for the tainted flow problem.

1.3 Live Range Splitting Strategies

Notation S_dir(points)

A *live range splitting strategy* over a variable v, which we denote by S_v, is a pair (d, P). The first element, $d \subseteq \{\uparrow, \downarrow\}$, denotes a direction along which the information flows: it can be forward ($\downarrow$) or backward ($\uparrow$). The second element, P, denotes the set of programs points where the information is acquired. Going back to the examples from Section 1.2.2, we have the following live range splitting directives:

what about using a function notation? such as S_down(...)
Remove the brackets around defs, uses etc.

defs U entry for reaching defs

uses U conds

I would prefer the list of clients instead

Client	Splitting strategy
Alias analysis, reaching definitions	$(\downarrow, \{defs\})$
SSI Clients [1, 9]	$(\downarrow, \{defs\}), (\uparrow, \{uses_{last}\})$
Cond. constant propagation [13], ABCD [3], Taint analysis [8], range analysis [11, 4]	$(\downarrow, \{defs, conds\})$
Stephenson's Bitwidth Analysis [10]	$(\downarrow, \{defs, conds\}), (\uparrow, \{uses\})$
Mahlke's Bitwidth Analysis [6]	$(\downarrow, \{defs\}), (\uparrow, \{uses\})$
An's type inference [5]	$(\uparrow, \{uses\})$
Hochstadt's type inference [12]	$(\uparrow, \{uses, conds\})$
Null-pointer analysis [?]	$(\downarrow, \{defs, uses\})$

Restrict this table to the examples provided previously.
You can put the full "table" in the text of the last section. 'cause in case you did not understand I do not want any references before :-)

Fig. 1.5 Live range splitting strategies for different dataflow analyses. We use *def* (*use*) to denote the set of instructions that define (use) the variable; *cond* to denote the set of instructions that apply a conditional test on a variable; and use_{last} to denote the set of instructions where a variable is used, and after which it is no longer alive.

- Alias analysis is a forward analysis that takes information from the definition site of variables; thus, for each variable v, N is the set of definition points of v. For instance, in Figure 1.4, we have that $S_v = (\downarrow, \{l_2, l_3\})$.
- Type inference is a backward analysis that takes information from the uses of variables; thus, for each variable, N is its set of use sites. For instance, in Figure 1.2(e), we have that $S_v = (\uparrow, \{l_4, l_6\})$. Notice that we do not split live ranges at the last use of a variable, because this variable is not alive henceforth.
- Taint analysis is a forward analysis that takes information from definition points, and conditional tests that use variables; thus, we split live ranges at conditionals. For instance, in Figure 1.3, we have that $S_v = (\downarrow, \{l_1, l_2, l_5\})$.
- The null pointer check is a forward analysis that takes information from definitions and uses. For instance, in Figure 1.4, we have that $S_v = (\downarrow, \{l_1, l_2, l_3\})$.

I think you can summaries and just give the result in one sentence.
On the other end you should discuss the cases where you add entry to the set of definitions and exit to the set of uses.

The list in Figure 1.5 gives further examples of live range splitting strategies.

Splitting algorithm for forward analyses

where to split for forward:
infos + SSA

Given a splitting strategy $S_v = (\downarrow, N)$, we split the live range of v at every program point in N. Some of these program points, like l_5 in Figure 1.3, ~~have two or more successors~~ are branch/split points. We split live ranges at these program points via σ-functions. After live range splitting, we might have many sources of information reaching a common program point. For instance, in Figure 1.1(a), the information that flows forward from l_2 and l_3 collide at l_4. We use a ϕ-function to merge these two flows. In our example, this ϕ-function defines b_3. The set of program points where the information collide is easily characterized if we use the notion of join sets introduced in Chap-

You should be more explicit. Add in the table the formula S_down(defs) = defs U J(defs) =~ defs U iDF(defs)
S_down(defs U {entry}) = defs U {entry} U iDF(defs) ...
You should add a short discussion about iDF for approximating J and ipDF for approximating S.

ter **??**. Therefore, the very SSA construction algorithm suffices to split live ranges to forward dataflow analysis.

Splitting algorithm for backward analyses

where to split for backward: split or pDF+

A live range splitting strategy $S_v = (\uparrow, N)$ requires us to merge information that is propagated backwardly. For instance, lets assume, in Figure 1.2(a), that we did not have the instruction at l_5. Thus, we would know from the use of O's instance at l_4 that the class OY contains m_1. Similarly, we know, due to l_6, that it also contains m_2. These two informations are equally valid at l_1; thus, leading us to conclude that both methods are defined in the object v. In Figure 1.2(b) we represent this backward merge via the σ-function $(v_2, v_7) = \sigma v_1$. use parenthesis for sigma

We use the notions of split sets and iterated post dominance frontier, introduced in Chapter **??**, to provide the live range splitting algorithm for backward analyses. However, if necessary to keep the SSA property, then forward and backward splitting algorithms are not symmetric, as the latter does not ensure static single definitions. Thus, splitting for $S_v = (\uparrow, N)$ demands two steps: first we insert σ-functions at $\mathcal{S}(\mathcal{N})$, and do variable renaming, walking up the program's post-dominator tree. This renaming step is symmetric to that used to build the SSA form. In the second phase we reconstruct the SSA form. The price to stay under SSA-form is that variables created by ϕ-functions might be bound to identity transfer functions. For instance, in Figure 1.2(b) we learn from the use of v_6 that the class O contains method m_3. Our transfer function passes this information from v_3 to v_4 and v_5 without any change.

+SSA if needed

renaming if no SSA

This renaming step is necessary only if you do not do the SSA splitting.

+ same remarks as for forward analysis

Add a section about mixing analysis, forward, backward etc. eg null pointer + Stephenson: SSA[S_down(defs U uses U conds) U S_up(uses)] = iDF(J(defs U uses U conds) defs U uses U conds U S(uses)) U uses)
= iDF(defs U uses U conds U S(uses))

Extending information beyond variable's live ranges

On the usefulness of pruning

Yes! Please develop this a little bit more (add a column in the table, an example...)

The minimal, non pruned SSA form might be useful to those analyses that require information outside the live range of the variable that originates this information, such as partial redundancy elimination. A ϕ-function might represent a value useful in value numbering, even if the original program did not refer to that value **??**. Hence, the decision of pruning ϕ functions depends on the nature of the analyses.

You do all the splitting, then only you rename

Recall that pruning is dead code elimination. We should discuss about pDF pruning also (this is a point we never talked about I discover is important). See (*) below.

1.3.1 ~~Representing σ-functions as ϕ-functions~~

Practical considerations

is sigma useful...

In the absence of a special notation, a compiler can represent σ-functions as single arity ϕ-functions. In this way, the compiler leaves to the SSA elimination module the task of replacing σ-functions with special instructions. As an example, Figure 1.6(a) shows how we would represent the σ-functions in Figure 1.2(b), and Figure 1.6(b) shows the same thing for Figure 1.3(b).

You can talk here also about copies that have to be placed in parallel with an instruction (when splitting at a use point)
Can we afford placing them after the instruction...?

It would be good to outline the interest of sigma:

- for backward data-flow analysis, sigma explicits where the merge is performed. This is more "clean" somehow. Same for forward as the predicate condition is explicit in the sigma itself. Actually, as soon as we are interested in using conds, using phis in place of sigma is a kind of hack.
- for data-driven (and possibly demand driven I have not checked) program interpretation (so this is particularly useful for circuit synthesis), sigma is an efficient representation to implement the switch of loop exit for loop closed form (better than the operator used in gated SSA). Actually this advantage of sigma leads to confusions from circuit synthesis people who wrote that their representation was SSI while it was not. This is clearly out of the scope of this chapter but this is important to outline to clarify that we do not say "sigma is useless": Sigma can indeed be replaced by phi as soon as we only care about representing live-range splitting.

(*) I just discussed with Florian about dead code elimination that is difficult to put in the propagation engine. The scope would be thi
chapter as it is backward propagation. It would be good to describe the algorithm here and take the opportunity to discuss why this is backward but we do not need to split at ipDF...

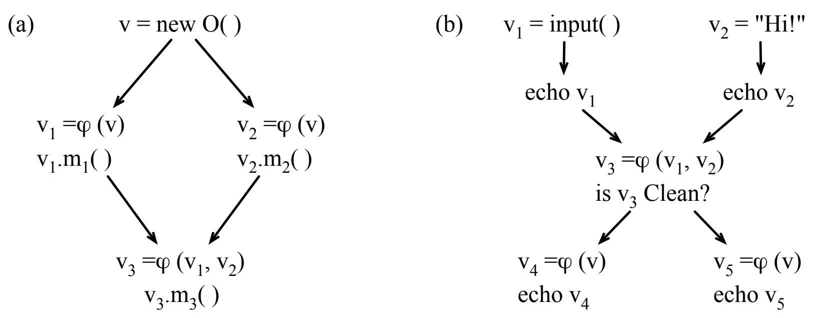


Fig. 1.6 Implementing σ functions via single arity ϕ-functions.

How to replace sigma by phis

If l is a branch point with n successors that would contain a σ-function $(v_1, \ldots, v_n) = \sigma v$, then, for each successor l_i of l, we insert at the beginning of l_i an instruction $v_i = \phi(v)$. Notice that it is possible that l_i already contains a ϕ-function for v. This case happens when the control flow edge $l \rightarrow l_i$ is *critical*. A critical edge links a basic block with many successors to a basic block with many predecessors. If l_i already contains a ϕ-function $v' = \phi(\ldots, \cancel{v}, \ldots,)$, then we rename ~~$v$ to v_i.~~

v_i v_i to v

1.3.2 Deriving dense information from sparse analyses

liveness info in place of the mapping of sparse sets

We can use our sparse dataflow analysis framework to solve even some dataflow problems that demand information at every program point. For instance, a well-known dataflow problem consists in determining the bit-width of each integer variable in the program code [6, 10, 4, **?**]. This is a typical sparse analysis, for we are interested in collecting the width, in bits, of each variable. However, there exist clients of bit-width analyses that need to know the bit sizes of the variables at particular program points. For instance, Barik *et al.* [**?**] have designed a bit-width aware register allocator. In this setting, the register pressure at a program point p is the sum of the bit sizes of all the variables alive at p. The SSA graph can support the register allocator of Barik *et al.* by coupling the result of the sparse bit-width analysis with live range information. If the live range splitting strategy preserves the single static assignment properties, then we can perform this coupling very efficiently.

liveness easy under SSA

Preserving the SSA properties is key due to two reasons. First, liveness analysis has a non-iterative implementation for SSA form programs that is linear on the program size [**?**, p.429]. Second, and more important, in case we only need liveness information for some specific variables, at some specific program points, there exist a fast liveness check for SSA-form programs. The problem of answering the question "is variable v alive at program point p" has an algorithm that is $O(U)$, where U is the number of times that v is used in the program code [$\cancel{?}$]. Given that the vast

refer to liveness chapter instead

Actually, as soon as we do not need "dense" information, splitting as aggressively as we do is useless. A good example is the deadcode elimination algo that is backward from uses and propagate to defs. This is an extremly important point to outline that many times use-def and def-use chains are sufficient. The merge is done on the def for backward and on the use for forward.
Not clear to me if you should talk about it in this section or in the last section (that is missing and would contain related work, open pbs, extensions etc) or merge those sections...

majority of variables in common benchmarks is used less than 5 times [**?**], we can consider this asymptotic complexity constant in practice.

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

References

1. Scott Ananian. The static single information form. Master's thesis, MIT, September 1999.
2. Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *PLDI*, pages 243–253. ACM, 2001.
3. Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.
4. Thomas Gawlitza, Jerome Leroux, Jan Reineke, Helmut Seidl, Gregoire Sutre, and Reinhard Wilhelm. Polynomial precise interval analysis revisited. *Efficient Algorithms*, 1:422 – 437, 2009.
5. Jong hoon An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In *POPL*, pages XX–XX. ACM, 2011.
6. S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371, 2001.
7. Fernando Magno Quintao Pereira and Jens Palsberg. Register allocation by puzzle solving. In *PLDI*, pages 216–226. ACM, 2008.
8. Andrei Alves Rimsa, Marcelo D'Amorim, and Fernando M. Q. Pereira. Tainted flow analysis on e-SSA-form programs. In *CC*, pages 124–143. Springer, 2011.
9. Jeremy Singer. *Static Program Analysis Based on Virtual Register Renaming*. PhD thesis, University of Cambridge, 2006.
10. Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM, 2000.
11. Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138, 2005.
12. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *POPL*, pages 395–406, 2008.
13. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2), 1991.