Chapter 13
Static Single Information Form

Fernando Magno Quintão Pereira and Fabrice Rastello

The objective of a data-flow analysis is to discover facts that are true about a program. We call such facts *information*. Using the notation introduced in Chap. 8, information is an element in the data-flow lattice. For example, the information that concerns liveness analysis is the set of variables alive at a certain program point. Similarly to liveness analysis, many other classical data-flow approaches bind information to pairs formed by a variable and a program point. However, if an invariant occurs for a variable v at any program point where v is alive, then we can associate this invariant directly with v. If the intermediate representation of a program guarantees this correspondence between information and variable for every variable, then we say that the program representation provides the *Static Single Information* (SSI) property.

In Chap. 8 we have shown how the SSA form allows us to solve sparse forward data-flow problems such as constant propagation. In the particular case of constant propagation, the SSA form lets us assign to each variable the invariant—or information—of being constant or not. The SSA intermediate representation gives us this invariant because it splits the live ranges of variables in such a way that each variable name is defined only once. Now we will show that live range splitting can also provide the SSI property not only to forward but also to backward data-flow analyses.

Different data-flow analyses might extract information from different program facts. Therefore, a program representation may provide the SSI property to some data-flow analyses but not to all of them. For instance, the SSA form naturally

F. M. Q. Pereira (✉)
Federal University of Minas Gerais, Belo Horizonte, Brazil
e-mail: fernando@dcc.ufmg.br

F. Rastello
Inria, Grenoble, France
e-mail: fabrice.rastello@inria.fr

F. Rastello, F. Bouchez Tichadou (eds.), *SSA-based Compiler Design*,
https://doi.org/10.1007/978-3-030-80515-9_13

provides the SSI property to the reaching definition analysis. Indeed, the SSA form provides the static single information property to any data-flow analysis that obtains information at the definition sites of variables. These analyses and transformations include copy and constant propagation, as illustrated in Chap. 8. However, for a data-flow analysis that derives information from the use sites of variables, such as the class inference analysis that we will describe in Sect. 13.1.6, the information associated with a variable might not be unique along its entire live range even under SSA: In that case the SSA form does not provide the SSI property.

There are extensions of the SSA form that provide the SSI property to more data-flow analyses than the original SSA does. Two classic examples—detailed later—are the *Extended-SSA* (e-SSA) form and the *Static Single Information* (SSI) form. 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. The SSI form provides the static single information property to data-flow analyses that extract information from the definition sites of variables and from the last use sites (which we define later). These different intermediate representations rely on a common strategy to achieve the SSI property: *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 data-flow analyses.

13.1 Static Single Information

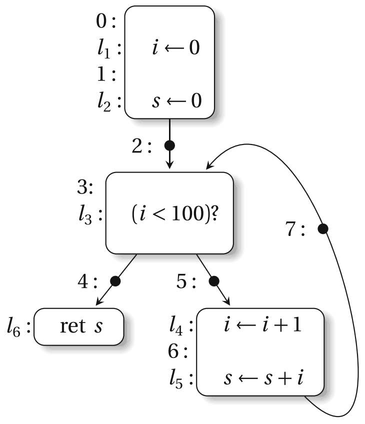
The goal of this section is to define the notion of *Static Single Information*, and to explain how it supports the sparse data-flow analyses discussed in Chap. 8. With this aim in mind, we revisit the concept of sparse analysis in Sect. 13.1.1. There is a special class of data-flow problems, which we call *Partitioned Lattice per Variable* (PLV), which fits into the sparse data-flow framework of this chapter very well. We will look more carefully into these problems in Sect. 13.1.2. The intermediate program representations discussed in this chapter provide the static single information property—formalized in Sect. 13.1.3—to any PLV problem. In Sect. 13.1.5 we give algorithms to solve sparsely any data-flow problem that contains the SSI property. This sparse framework is very broad: Many well-known data-flow problems are partitioned lattice, as we will see in the examples in Sect. 13.1.6.

13.1.1 Sparse Analysis

Traditionally, non-relational data-flow analyses bind information to pairs formed by a variable and a program point. Consider for example the problem of *range analysis*, i.e., estimating the interval of values that an integer variable may assume throughout

```
$i \leftarrow 0$
$s \leftarrow 0$
while $i < 100$ do
    $i \leftarrow i + 1$
    $s \leftarrow s + i$
return $s$
```

(a) original code



(b) CFG and program points

prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0,0]$	$\top$
2	$[0,0]$	$[0,0]$
3	$[0,100]$	$[0,+\infty[$
4	$[100,100]$	$[0,+\infty[$
5	$[0,99]$	$[0,+\infty[$
6	$[0,100]$	$[0,+\infty[$
7	$[0,100]$	$[0,+\infty[$

(c) result of a dense implementation of range analysis

Fig. 13.1 An example of a dense data-flow analysis that finds the range of possible values associated with each variable at each program point

the execution of a program. A traditional implementation of this analysis would find, for each pair (v, p) of a variable v and a program point p, the interval of possible values that v might assume at p (see the example in Fig. 13.1). In this case, we call a program point any region between two consecutive instructions and denote as $[v]$ the abstract information associated with variable v. Because this approach keeps information at each program point, we call it *dense*, in contrast to the sparse analyses seen in Chap. 8, Sect. 8.2.

The dense approach might result in a large quantity of redundant information during the data-flow analysis. For instance, if we denote $[v]^p$ the abstract state of variable v at program point p, we have for instance in our example $[i]^1 = [i]^2$, $[s]^5 = [s]^6$ and $[i]^6 = [i]^7$ (see Fig. 13.1). This redundancy happens because some transfer functions are identities: In range analysis, an instruction that neither defines nor uses any variable is associated with an identity transfer function. Similarly, the transfer function that updates the abstract state of i at program point 2 is an identity, because the instruction immediately before 2 does not add any new information to the abstract state of i, $[i]^2$ is updated with the information that flows directly from the direct predecessor point 1.

The goal of *sparse* data-flow analysis is to shortcut the identity transfer functions, a task that we accomplish by grouping contiguous program points bound to identities into larger regions. Solving a data-flow analysis sparsely has many advantages over doing it densely: Because we do not need to keep bitvectors associated with each program point, the sparse solution tends to be more economical in terms of space and time. Going back to our example, a given variable v may be mapped to the same interval along many consecutive program points. Furthermore, if the information associated with a variable is invariant along its entire live range, then we can bind this information to the variable itself. In other words, we can replace all the constraint variables $[v]^p$ by a single constraint variable $[v]$, for each variable v and every $p \in \text{live}(v)$.

Although not every data-flow problem can be easily solved sparsely, many of them can as they fit into the family of PLV problems described in the next section.

13.1.2 Partitioned Lattice per Variable (PLV) Problems

The non-relational data-flow analysis problems we are interested in are the ones that bind information to pairs of program variables and program points. We refer to this class of problems as *Partitioned Lattice per Variable* problems and formally describe them as follows.

Definition 13.1 (PLV) Let $\mathscr{V} = \{v_1, \ldots, v_n\}$ be the set of program variables. Let us consider, without loss of generality, a forward data-flow analysis that searches for a maximum. This data-flow analysis can be written as an equation system that associates each program point p, with n element of a lattice $\mathscr{L}$, given by the following equation:

$$x^p = \bigwedge_{s \in \text{directpreds}(p)} F^{s \to p}(x^s),$$

where x^p denotes the abstract state associated with program point p, and $F^{s \to p}$ is the transfer function from direct predecessor s to p. The analysis can alternatively be written as a constraint system that binds to each program point p and each $s \in \text{directpreds}(p)$ the equation $x^p = x^p \wedge F^{s \to p}(x^s)$ or, equivalently, the in equation

$$x^p \sqsubseteq F^{s \to p}(x^s).$$

The corresponding Maximum Fixed Point (MFP) problem is said to be a *Partitioned Lattice per Variable Problem* iff $\mathscr{L}$ can be decomposed into the product of $\mathscr{L}_{v_1} \times \cdots \times \mathscr{L}_{v_n}$ where each $\mathscr{L}_{v_i}$ is the lattice associated with program variable v_i. In other words x^s can be written as $([v_1]^s, \ldots, [v_n]^s)$ where $[v]^s$ denotes the abstract state associated with variable v and program point s. $F^{s \to p}$ can thus be decomposed into the product of $F_{v_1}^{s \to p} \times \cdots \times F_{v_n}^{s \to p}$ and the constraint system decomposed into the inequalities $[v_i]^p \sqsubseteq F_{v_i}^{s \to p}([v_1]^s, \ldots, [v_n]^s)$.

Going back to range analysis, if we denote as $\mathscr{I}$ the lattice of integer intervals, then the overall lattice can be written as $\mathscr{L} = \mathscr{I}^n$, where n is the number of variables. Note that the class of PLV problems includes a smaller class of problems called *Partitioned Variable Problems* (PVP). These analyses, which include live variables reaching definitions and forward/backward printing, can be decomposed into a set of sparse data-flow problems—usually one per variable—each independent of the others.

Note that not all data-flow analyses are PLV, for instance problems dealing with relational information, such as "$i < j$?", which needs to hold information on *pairs* of variables.

13.1.3 The Static Single Information Property

If the information associated with a variable is invariant along its entire live range, then we can bind this information to the variable itself. In other words, we can replace all the constraint variables $[v]^p$ by a single constraint variable $[v]$, for each variable v and every $p \in \text{live}(v)$. Consider the problem of range analysis again. There are two types of control-flow points associated with non-identity transfer functions: definitions and conditionals. (1) At the definition point of variable v, F_v simplifies to a function that depends only on some $[u]$ where each u is an argument of the instruction defining v; (2) At the conditional tests that use a variable v, F_v can be simplified to a function that uses $[v]$ and possibly other variables that appear in the test. The other program points are associated with an identity transfer function and can thus be ignored: $[v]^p = [v]^p \wedge F_v^{s \to p}([v_1]^s, \ldots, [v_n]^s)$ simplifies to $[v]^p = [v]^p \wedge [v]^p$ i.e., $[v]^p = [v]^p$. This gives the intuition on why a propagation engine along the def-use chains of an SSA form program can be used to solve the constant propagation problem in an equivalent, yet "sparser," manner.

A program representation that fulfils the Static Single Information (SSI) property allows us to attach the information to variables, instead of program points, and needs to fulfil the following four properties: *Split* forces the information related to a variable to be invariant along its entire live range; *Info* forces this information to be irrelevant outside the live range of the variable; *Link* forces the def-use chains to reach the points where information is available for a transfer function to be evaluated; finally, *Version* provides a one-to-one mapping between variable names and live ranges.

We now give a formal definition of the SSI and the four properties.

Property 1 (SSI) STATIC SINGLE INFORMATION: Consider a forward (resp. backward) monotone PLV problem E_{dense} stated as a set of constraints

$$[v]^p \sqsubseteq F_v^{s \to p}([v_1]^s, \ldots, [v_n]^s)$$

for every variable v, each program point p, and each $s \in \text{directpreds}(p)$ (resp. $s \in \text{directsuccs}(p)$). A program representation fulfils the Static Single Information property if and only if it fulfils the following four properties:

Split Let s be the unique direct predecessor (resp. direct successor) of a program point where a variable v is live and such that $F_v^{s \to p} \neq \lambda x.\bot$ is non-trivial, i.e., is not the simple projection on $\mathscr{L}_v$, then s should contain a definition (resp. last use) of v; for $(v, p) \in \textit{variables} \times \textit{progPoints}$, let (Y_v^p) be a maximum solution to E_{dense}. Each node p that has several direct predecessors (resp. direct successors), and for which $F_v^{s\rightarrow p}(Y_{v_1}^s, \ldots, Y_{v_n}^s)$ has different values on its incoming edges $(s \rightarrow p)$ (resp. outgoing edges $(p \rightarrow s)$), should have a ϕ-function at the entry of p (resp. σ-function at the exit of p) for v as defined in the next section.

Info Each program point p such that $v \notin$ live-out(p) (resp. $v \notin$ live-in(p))) should be bound to an undefined transfer function, i.e., $F_v^p = \lambda x.\bot$.

Link Each instruction *inst* for which F_v^{inst} depends on some $[u]^s$ should contain a use (resp. definition) of u live-in (resp. live-out) at *inst*.

Version For each variable v, live(v) is a connected component of the CFG.

We must split live ranges using special instructions to provide the SSI properties. A naive way would be to split them between each pair of consecutive instructions, then we would automatically provide these properties, as the newly created variables would be live at only one program point. However, this strategy would lead to the creation of many trivial program regions, and we would lose sparsity. We provide a sparser way to split live ranges that fit Property 1 in Sect. 13.2. We may also have to extend the live range of a variable to cover every program point where the information is relevant; we accomplish this last task by inserting pseudo-uses and pseudo-definitions of this variable.

13.1.4 Special Instructions Used to Split Live Ranges

We perform live range splitting via special instructions: the σ-functions and parallel copies that, together with ϕ-functions, create new definitions of variables. These notations are important elements of the propagation engine described in the section that follows. In short, a σ-function (for a branch point) is the dual of a ϕ-function (for a join point), and a parallel copy is a copy that *must* be done in parallel with another instruction. Each of these special instructions, ϕ-function, σ-functions, and parallel copies, split live ranges at different kinds of program points: interior nodes, branches, and joins.

Interior nodes are program points that have a unique direct predecessor and a unique direct successor. At these points we perform live range splitting via copies. If the program point already contains another instruction, then this copy *must* be done *in parallel* with the existing instruction. The notation,

$$\mathit{inst} \;\parallel\; v_1' = v_1 \;\parallel\; \ldots \;\parallel\; v_m' = v_m$$

denotes m copies $v_i' = v_i$ performed in parallel with instruction *inst*. This means that all the uses of *inst* plus all right-hand variables v_i are read simultaneously, then *inst* is computed, then all definitions of *inst* plus all left-hand variables v_i' are written simultaneously. For a usage example of parallel copies, we will see later in this chapter an example of null-pointer analysis: Fig. 13.4.

We call *joins* the program points that have one direct successor and multiple direct predecessors. For instance, two different definitions of the same variable v

might be associated with two different constants, hence providing two different pieces of information about v. To avoid these definitions reaching the same use of v, we merge them at the earliest program point where they meet. We do it via our well-known ϕ-functions.

In backward analyses the information that emerges from different uses of a variable may reach the same *branch point*, which is a program point with a unique direct predecessor and multiple direct successors. To ensure Property 1, the use that reaches the definition of a variable must be unique, in the same way that in an SSA form program the definition that reaches a use is unique. We ensure this property via special instructions called σ-functions. The σ-functions are the dual of ϕ-functions, performing a parallel assignment depending on the execution path taken. The assignment

$$(l^1 : v_1^1, \ldots, l^q : v_1^q) = \sigma(v_1) \;\parallel\; \ldots \;\parallel\; (l^1 : v_m^1, \ldots, l^q : v_m^q) = \sigma(v_m)$$

represents m σ-functions that assign to each variable v_i^j the value in v_i if control flows into block l^j. As with ϕ-functions, these assignments happen in parallel, i.e., the m σ-functions encapsulate m parallel copies. Also, note that variables live in different branch targets are given different names by the σ-function.

13.1.5 Propagating Information Forward and Backward

Let us consider a unidirectional forward (resp. backward) PLV problem E_{dense}^{ssi} stated as a set of equations $[v]^p \sqsubseteq F_v^{s \to p}([v_1]^s, \ldots, [v_n]^s)$ (or equivalently $[v]^p = [v]^p \wedge F_v^{s \to p}([v_1]^s, \ldots, [v_n]^s)$ for every variable v, each program point p, and each $s \in \text{directpreds}(p)$ (resp. $s \in \text{directsuccs}(p)$). To simplify the discussion, any ϕ-function (resp. σ-function) is seen as a set of copies, one per direct predecessor (resp. direct successor), which leads to many constraints. In other words, a ϕ-function such as $p : a = \phi(a_1 : l^1, \ldots, a_m : l^m)$ gives us n constraints such as

$$[a]^p \sqsubseteq F_a^{l^j \to p}([a_1]^{l^j}, \ldots, [a_n]^{l^j})$$

which usually simplifies into $[a]^p \sqsubseteq [a_j]^{l^j}$. This last can be written equivalently into the classical meet

$$[a]^p \sqsubseteq \bigwedge_{l^j \in \text{directpreds}(p)} [a_j]^{l^j}$$

Algorithm 13.1: Backward propagation engine under SSI

```
1  $worklist \leftarrow \emptyset$
2  foreach $v \in$ vars do $[v] \leftarrow \top$
3  foreach $i \in$ insts do push($worklist$, $i$)
4  while $worklist \neq \emptyset$ do
5      $i \leftarrow$ pop($worklist$)
6      foreach $v \in i.uses$ do
7          $[v]_{new} \leftarrow [v] \wedge G_v^i([i.defs])$
8          if $[v] \neq [v]_{new}$ then
9              $worklist \leftarrow worklist \cup v.defs$
10             $[v] \leftarrow [v]_{new}$
```

used in Chap. 8. Similarly, a σ-function $(l^1 : a_1, \ldots, l^m : a_m) = \sigma(p : a)$ after program point p yields n constraints such as

$$[a_j]^{l^j} \sqsubseteq F_v^{p \rightarrow l^j}([a_1]^p, \ldots, [a_n]^p)$$

which usually simplifies into $[a_j]^{l^j} \sqsubseteq [a]^p$. Given a program that fulfils the SSI property for E_{dense}^{ssi} and the set of transfer functions F_v^s, we show here how to build an equivalent sparse constrained system.

Definition 13.2 (SSI Constrained System) Consider that a program in SSI form gives us a constraint system that associates with each variable v the constraints $[v]^p \sqsubseteq F_v^{s \rightarrow p}([v_1]^s, \ldots, [v_n]^s)$. We define a system of sparse equations E_{sparse}^{ssi} as follows:

- For each instruction i that defines (resp. uses) a variable v, let $a \ldots z$ be the set of used (resp. defined) variables. Because of the *Link* property, $F_v^{s \rightarrow p}$ (that we will denote F_v^i from now) depends only on some $[a]^s \ldots [z]^s$. Thus, there exists a function G_v^i defined as the restriction of F_v^i on $\mathcal{L}_a \times \cdots \times \mathcal{L}_z$, i.e., informally, "$G_v^i([a], \ldots, [z]) = F_v^i([v_1], \ldots, [v_n])$."
- The sparse constrained system associates the constraint $[v] \sqsubseteq G_v^i([a], \ldots, [z])$ with each variable v, for each definition (resp. use) point i of v, where $a, \ldots, z$ are used (resp. defined) at i.

The propagation engine discussed in Chap. 8 sends information forwards along the def-use chains naturally formed by the SSA form program. If a given program fulfils the SSI property for a backward analysis, we can use a very similar propagation algorithm to communicate information backwards, such as the worklist Algorithm 13.1. A slightly modified version, presented in Algorithm 13.2, propagates information forwards. If necessary, these algorithms can be made control-flow sensitive, like Algorithm 8.1 in Chap. 8.

Still, we should highlight a quite important subtlety that appears in line 7 of Algorithms 13.1 and 13.2: $[v]$ appears on the right-hand side of the assignment for

Algorithm 13.2: Forward propagation engine under SSI

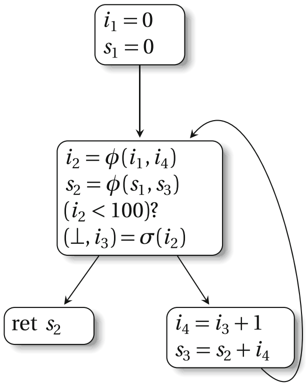
```
worklist ← $\emptyset$
foreach $v \in$ vars do $[v] \leftarrow \top$
foreach $i \in$ insts do push(worklist, $i$)
while worklist $\neq \emptyset$ do
    $i \leftarrow$ pop(worklist)
    foreach $v \in i.defs$ do
        $[v]_{new} \leftarrow G_v^i([i.uses])$
        if $[v] \neq [v]_{new}$ then
            worklist ← worklist $\cup$ v.uses
            $[v] \leftarrow [v]_{new}$
```

Algorithm 13.1 while it does not for Algorithm 13.2. This stems from the asymmetry of our SSI form that ensures (for practical purposes only, as we will explain soon) the Static Single Assignment property but not the Static Single Use (SSU) property. If we have several uses of the same variable, then the sparse backward constraint system will have several inequations—one per variable use—with the same left-hand side. Technically this is the reason why we manipulate a constraint system (system with inequations) and not an equation system as in Chap. 8. Both systems can be solved[1] using a scheme known as *chaotic iteration* such as the worklist algorithm we provide here. The slight and important difference for a constraint system as opposed to an equation system is that one needs to meet $G_v^i(\ldots)$ with the old value of $[v]$ to ensure the monotonicity of the consecutive values taken by $[v]$. It would still be possible to enforce the SSU property, in addition to the SSA property, of our intermediate representation, at the expense of adding more ϕ-functions and σ-functions. However, this guarantee is not necessary to every sparse analysis. The dead-code elimination problem illustrates this point well: For a program under SSA form, replacing G_v^i in Algorithm 13.1 by the property "i is a useful instruction or one of the variables it defines is marked as useful" leads to the standard SSA-based dead-code elimination algorithm. The sparse constraint system does have several equations (one per variable use) for the same left-hand side (one for each variable). It is not necessary to enforce the SSU property in this instance of dead-code elimination, and doing so would lead to a less efficient solution in terms of compilation time and memory consumption. In other words, a code under SSA form fulfils the SSI property for dead-code elimination.

13.1.6 Examples of Sparse Data-Flow Analyses

As we have mentioned before, many data-flow analyses can be classified as PLV problems. In this section we present some meaningful examples.

[1] In an ideal world, with monotone framework and lattice of finite height.



$$
\begin{array}{ll}
[i_1] = [i_1] \cup [0,0] & = [0,0] \\
[s_1] = [s_1] \cup [0,0] & = [0,0] \\
[i_2] = [i_2] \cup [i_1] \cup [i_4] & = [0,100] \\
[s_2] = [s_2] \cup [s_1] \cup [s_3] & = [0,+\infty[\\
[i_3] = [i_3] \cup ([i_2] \cap]-\infty,99]) & = [0,99] \\
[i_4] = [i_4] \cup ([i_3] + 1) & = [1,100] \\
[s_3] = [s_3] \cup ([s_2] + [i_4]) & = [1,+\infty[
\end{array}
$$

(a) Live range splitting used by a sparse implementation of range analysis

(b) sparse constraint system & solution

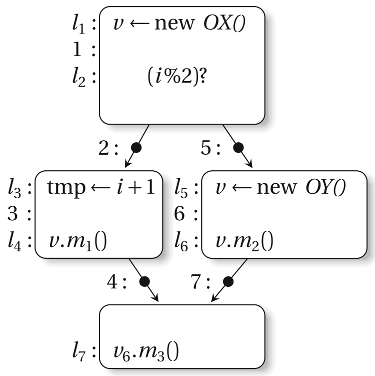
Fig. 13.2 Live range splitting on Fig. 13.1 and a solution to this instance of the range analysis problem

Range Analysis Revisited

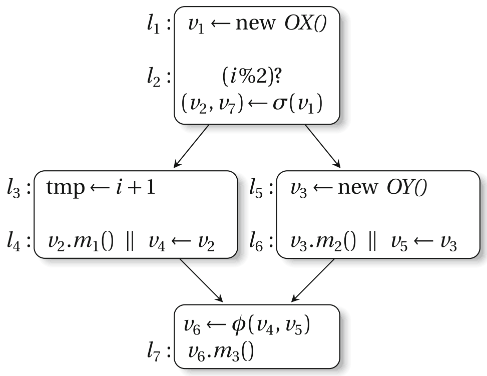
We start this section by revisiting the initial example of data-flow analysis of this chapter, given in Fig. 13.1. A range analysis acquires information from either the points where variables are defined, or from the points where variables are tested. In the original figure we know that i must be bound to the interval [0, 0] immediately after instruction l_1. Similarly, we know that this variable is upper bounded by 100 when arriving at l_4, due to the conditional test that happens before. Therefore, in order to achieve the SSI property, we should split the live ranges of variables at their definition points, or at the conditionals where they are used. Figure 13.2 shows on the left the original example after live range splitting. In order to ensure the SSI property in this example, the live range of variable i must be split at its definition, and at the conditional test. The live range of s, on the other hand, must be split only at its definition point, as it is not used in the conditional. Splitting at conditionals is done via σ-functions. The representation that we obtain by splitting live ranges at definitions and conditionals is called the Extended Static Single Assignment (e-SSA) form. Figure 13.2 also shows on the right the result of the range analysis on this intermediate representation. This solution assigns to each variable a unique range interval.

Class Inference

Some dynamically typed languages, such as Python, JavaScript, Ruby, or Lua, represent objects as tables containing methods and fields. It is possible to improve the execution of programs written in these languages if we can replace these simple tables by actual classes with virtual tables. A class inference engine tries to assign a class to a variable v based on the ways that v is defined and used. Figure 13.3 illustrates this optimization on a Python program (a). Our objective is to infer the correct suite of methods for each object bound to variable v. Figure 13.3b shows the results of a dense implementation of this analysis. Because type inference is


l_1 : $v \leftarrow$ new $OX()$
1 :
l_2 : $(i\%2)?$
2 :
5 :
l_3 : tmp $\leftarrow i+1$
3 :
l_4 : $v.m_1()$
l_5 : $v \leftarrow$ new $OY()$
6 :
l_6 : $v.m_2()$
4 :
7 :
l_7 : $v_6.m_3()$

(a) Example of a program written in a dynamically typed language


l_1 : $v_1 \leftarrow$ new $OX()$
l_2 : $(i\%2)?$
$(v_2, v_7) \leftarrow \sigma(v_1)$
l_3 : tmp $\leftarrow i+1$
l_4 : $v_2.m_1() \parallel v_4 \leftarrow v_2$
l_5 : $v_3 \leftarrow$ new $OY()$
l_6 : $v_3.m_2() \parallel v_5 \leftarrow v_3$
$v_6 \leftarrow \phi(v_4, v_5)$
l_7 : $v_6.m_3()$

(c) Live range splitting used by a sparse implementation of class inference

prog. point	$[v]$
1	$\{m_1, m_3\}$
2	$\{m_1, m_3\}$
3	$\{m_1, m_3\}$
4	$\{m_3\}$
5	$\top$
6	$\{m_2, m_3\}$
7	$\{m_3\}$

(b) Result of a dense implementation of class inference

$$\begin{aligned}
[v_6] &= [v_6] \wedge \{m_3\} = \{m_3\}\\
[v_5] &= [v_5] \wedge [v_6] = \{m_3\}\\
[v_4] &= [v_4] \wedge [v_6] = \{m_3\}\\
[v_2] &= [v_2] \wedge (\{m_1\} \wedge [v_4]) = \{m_1, m_3\}\\
[v_3] &= [v_3] \wedge (\{m_2\} \wedge [v_4]) = \{m_2, m_3\}\\
[v_1] &= [v_1] \wedge [v_2] = \{m_1, m_3\}\\
[v_1] &= [v_1] \wedge [v_7] = \{m_1, m_3\}
\end{aligned}$$

(d) Constraints that determine a solution for the sparse version of class inference

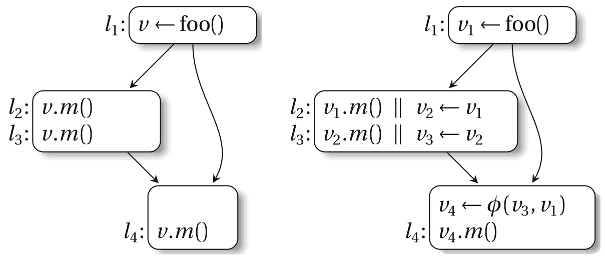
Fig. 13.3 Class inference analysis as an example of backward data-flow analysis that takes information from the uses of variables

a backward analysis that extracts information from use sites, we split live ranges using parallel copies at these program points and rely on σ-functions to merge them back, as shown in Fig. 13.3c. The use-def chains that we derive from the program representation lead naturally to a constraint system, shown in Fig. 13.3d, where $[v_j]$ denotes the set of methods associated with variable v_j. A fixed point to this constraint system is a solution to our data-flow problem. This instance of class inference is a Partitioned Variable Problem (PVP),[2] because the data-flow information associated with a variable v can be computed independently from the other variables.

Null-Pointer Analysis

The objective of null-pointer analysis is to determine which references may hold null values. This analysis allows compilers to remove redundant null-exception

[2] Actually, class inference is no longer a PVP as soon as we want to propagate the information through copies.


l_1: $v \leftarrow \text{foo}()$
l_2: $v.m()$
l_3: $v.m()$
l_4: $v.m()$
l_1: $v_1 \leftarrow \text{foo}()$
l_2: $v_1.m() \parallel v_2 \leftarrow v_1$
l_3: $v_2.m() \parallel v_3 \leftarrow v_2$
$v_4 \leftarrow \phi(v_3, v_1)$
l_4: $v_4.m()$

$$[v_1] = [v_1] \wedge 0 = 0$$
$$[v_2] = [v_2] \wedge \cancel{0} = \cancel{0}$$
$$[v_3] = [v_3] \wedge \cancel{0} = \cancel{0}$$
$$[v_4] = [v_4] \wedge ([v_3] \wedge [v_1]) = 0$$

(a) Object oriented program that might invoke methods of null objects

(b) Live range splitting strategy used by a sparse implementation of null-pointer analysis

(c) Constraints that determine a solution for the sparse version of null-pointer analysis

Fig. 13.4 Null-pointer analysis as an example of forward data-flow analysis that takes information from the definitions and uses of variables (0 represents the fact that the pointer is possibly null, $\cancel{0}$ if it cannot be) (**a**) Object oriented program that might invoke methods of null objects (**b**) Live range splitting strategy used by a sparse implementation of null-pointer analysis (**c**) Constraints that determine a solution for the sparse version of null-pointer analysis

tests and helps developers find null-pointer dereferences. Figure 13.4 illustrates this analysis. Because information is produced not only at definition but also at use sites, we split live ranges after each variable is used, as shown in Fig. 13.4b. For instance, we know that v_2 cannot be null, otherwise an exception would have been thrown during the invocation $v_1.m()$; hence the call $v_2.m()$ cannot result in a null-pointer dereference exception. On the other hand, we notice in Fig. 13.4a 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.

13.2 Construction and Destruction of the Intermediate Program Representation

In the previous section we have seen how the static single information property gives the compiler the opportunity to solve a data-flow problem sparsely. However, we have not yet seen how to convert a program to a format that provides the SSI property. This is a task that we address in this section, via the three-step algorithm from Sect. 13.2.2.

13.2.1 Splitting Strategy

A *live range splitting strategy* $\mathscr{P}_v = I_\uparrow \cup I_\downarrow$ over a variable v consists of a set of "oriented" program points. We let $I_\downarrow$ denote a set of points i with forward direction.

Client	**Splitting strategy $\mathscr{P}$**
Alias analysis, reaching defs., cond. constant propagation	$\mathit{Defs}_\downarrow$
Partial Redundancy Elimination	$\mathit{Defs}_\downarrow \bigcup \mathit{LastUses}_\uparrow$
ABCD, taint analysis, range analysis	$\mathit{Defs}_\downarrow \bigcup \mathrm{Out}\mathit{(Conds)}_\downarrow$
Stephenson's bitwidth analysis	$\mathit{Defs}_\downarrow \bigcup \mathrm{Out}\mathit{(Conds)}_\downarrow \bigcup \mathit{Uses}_\uparrow$
Mahlke's bitwidth analysis	$\mathit{Defs}_\downarrow \bigcup \mathit{Uses}_\uparrow$
An's type inference, Class inference	$\mathit{Uses}_\uparrow$
Hochstadt's type inference	$\mathit{Uses}_\uparrow \bigcup \mathrm{Out}\mathit{(Conds)}_\uparrow$
Null-pointer analysis	$\mathit{Defs}_\downarrow \bigcup \mathit{Uses}_\downarrow$

Fig. 13.5 Live range splitting strategies for different data-flow analyses. *Defs* (resp. *Uses*) denotes the set of instructions that define (resp. use) the variable; *Conds* denotes the set of instructions that apply a conditional test on a variable; Out(*Conds*) denotes the exits of the corresponding basic blocks; *LastUses* denotes the set of instructions where a variable is used, and after which it is no longer live

Similarly, we let $I_\uparrow$ denote a set of points i with backward direction. The live range of v must be split at least at every point in $\mathscr{P}_v$. Going back to the examples from Sect. 13.1.6, we have the live range splitting strategies enumerated below. The list in Fig. 13.5 gives further examples of live range splitting strategies. Corresponding references are given in the last section of this chapter.

- Range analysis is a forward analysis that takes information from points where variables are defined and conditional tests that use these variables. For instance, in Fig. 13.1, we have $\mathscr{P}_i = \{l_1, \mathrm{Out}(l_3), l_4\}_\downarrow$ where $\mathrm{Out}(l_i)$ is the exit of l_i (i.e., the program point immediately after l_i), and $\mathscr{P}_s = \{l_2, l_5\}_\downarrow$.
- Class inference is a backward analysis that takes information from the uses of variables; thus, for each variable, the live range splitting strategy is characterized by the set $\mathit{Uses}_\uparrow$ where *Uses* is the set of use points. For instance, in Fig. 13.3, we have $\mathscr{P}_v = \{l_4, l_6, l_7\}_\uparrow$.
- Null-pointer analysis takes information from definitions and uses and propagates this information forwardly. For instance, in Fig. 13.4, we have $\mathscr{P}_v = \{l_1, l_2, l_3, l_4\}_\downarrow$.

The algorithm `SSIfy` in Fig. 13.6 implements a live range splitting strategy in three steps. Firstly, it splits live ranges, inserting new definitions of variables into the program code. Secondly, it renames these newly created definitions; hence, ensuring that the live ranges of two different re-definitions of the same variable do not overlap. Finally, it removes dead and non-initialized definitions from the program code. We describe each of these phases in the rest of this section.

```
Function SSIfy($var\ v, Splitting\_Strategy\ \mathscr{P}_v$)
    split($v, \mathscr{P}_v$)
    rename($v$)
    clean($v$)
```

Fig. 13.6 Split the live ranges of v to convert it to SSI form

13.2.2 Splitting Live Ranges

In order to implement $\mathscr{P}_v$ we must split the live ranges of v at each program point listed by $\mathscr{P}_v$. However, these points are not the only ones where splitting might be necessary. As we have pointed out in Sect. 13.1.4, we might have, for the same original variable, many different sources of information reaching a common program point. For instance, in Fig. 13.1, there exist two definitions of variable i, l_1 and l_4, which reach the use of i at l_3. The information that flows forward from l_1 and l_4 collides at l_3, the loop entry. Hence the live range of i has to be split immediately before l_3—at In(l_3)—leading, in our example, to a new definition, i_1. In general, the set of program points where information collides can be easily characterized by the notion of join sets and iterated dominance frontier (DF^+) seen in Chap. 4. Similarly, split sets created by the backward propagation of information can be over-approximated by the notion of *iterated post-dominance frontier* (pDF^+) , which is the dual of DF^+. That is, the post-dominance frontier is the dominance frontier in a CFG where the directions of edges have been reversed. Note that, just as the notion of dominance requires the existence of a unique entry node that can reach every CFG node, the notion of post-dominance requires the existence of a unique exit node reachable by any CFG node. For control-flow graphs that contain several exit nodes or loops with no exit, we can ensure the single-exit property by creating a dummy common exit node and inserting some never-taken exit edges into the program.

Figure 13.7 shows the algorithm that we use to create new definitions of variables. This algorithm has three main phases. First, in lines 2–7 we create new definitions to split the live ranges of variables due to backward collisions of information. These new definitions are created at the iterated post-dominance frontier of points at which information originates. If a program point is a join node, then each of its direct predecessors will contain the live range of a different definition of v, as we ensure in lines 5–6 of our algorithm. Note that these new definitions are not placed parallel to an instruction, but in the region immediately after it, which we denote as "Out(. . .)." In lines 8–13 we perform the inverse operation: We create new definitions of variables due to the forward collision of information. Our starting points $S_\downarrow$, in this case, also include the original definitions of v, as we see in line 9, because we want to stay in SSA form in order to have access to a fast liveness check as described in Chap. 9. Finally, in lines 14–20 we

```
1  Function split(var $v$, Splitting_Strategy $\mathscr{P}_v = I_\downarrow \cup I_\uparrow$)
       ▷ compute the set of split nodes
2      $S_\uparrow \leftarrow \emptyset$
3      foreach $i \in I_\uparrow$ do
4          if $i$.is_join then
5              foreach $e \in incoming\_edges(i)$ do
6                  $S_\uparrow \leftarrow S_\uparrow \cup \mathrm{Out}(pDF^+(e))$
7          else $S_\uparrow \leftarrow S_\uparrow \cup \mathrm{Out}(pDF^+(i))$
8      $S_\downarrow \leftarrow \emptyset$
9      foreach $i \in S_\uparrow \cup \mathrm{Defs}(v) \cup I_\downarrow$ do
10         if $i$.is_branch then
11             foreach $e \in outgoing\_edges(i)$ do
12                 $S_\downarrow \leftarrow S_\downarrow \cup \mathrm{In}(DF^+(e))$
13         else $S_\downarrow \leftarrow S_\downarrow \cup \mathrm{In}(DF^+(i))$
14     $S \leftarrow \mathscr{P}_v \cup S_\uparrow \cup S_\downarrow$
       ▷ Split live-range of $v$ by inserting $\phi$, $\sigma$, and copies
15     foreach $i \in S$ do
16         if $i$ does not already contain any definition of $v$ then
17             if $i$.is_join then insert "$v \leftarrow \phi(v, ..., v)$" at $i$
18             else
19                 if $i$.is_branch then insert "$(v, ..., v) \leftarrow \sigma(v)$" at $i$
20                 else insert a copy "$v \leftarrow v$" at $i$
```

Fig. 13.7 Live range splitting. In(l) denotes a program point immediately before l, and Out(l) a program point immediately after l

actually insert the new definitions of v. These new definitions might be created by σ functions (due to $\mathscr{P}_v$ or to the splitting in lines 2–7); by ϕ-functions (due to $\mathscr{P}_v$ or to the splitting in lines 8–13); or by parallel copies.

13.2.3 Variable Renaming

The `rename` algorithm in Fig. 13.8 builds def-use and use-def chains for a program after live range splitting. This algorithm is similar to the classical algorithm used to rename variables during the SSA construction that we saw in Chap. 3. To rename a variable v we traverse the program's dominance tree, from top to bottom, stacking each new definition of v that we find. The definition currently on the top of the stack is used to replace all the uses of v that we find during the traversal. If the stack is empty, this means that the variable is not defined at this point. The renaming process replaces the uses of undefined variables by $\perp$ (see comment of function `stack.set_use`). We have two methods, `stack.set_use` and

```
1  Function rename(var $v$)
       ▷ Compute use-def & def-use chains.
2      stack ← $\emptyset$
3      foreach CFG node $n$ in dominance order do
4          if $\exists v \leftarrow \phi(v:l^1,\dots,v:l^q)$ in In($n$) then
5              stack.set_def($v \leftarrow \phi(v:l^1,\dots,v:l^q)$)
6          foreach instruction $u$ in $n$ that uses $v$ do
7              stack.set_use($u$)
8          if $\exists$ instruction $d$ in $n$ that defines $v$ then
9              stack.set_def($d$)
10         foreach instruction $(\dots) \leftarrow \sigma(v)$ in Out($n$) do
11             stack.set_use($(\dots) \leftarrow \sigma(v)$)
12         if $\exists (v:l^1,\dots,v:l^q) \leftarrow \sigma(v)$ in Out($n$) then
13             foreach $v:l^i \leftarrow v$ in $(v:l^1,\dots,v:l^q) \leftarrow \sigma(v)$ do
14                 stack.set_def($v:l^i \leftarrow v$)
15         foreach $m$ in direct-successors($n$) do
16             if $\exists v \leftarrow \phi(\dots,v:l^n,\dots)$ in In($m$) then
17                 stack.set_use($v \leftarrow v:l^n$)
```

```
1  Function stack.set_use(instruction inst)
       ▷ We consider here that stack.peek() = $\perp$ if stack.isempty(), and that
         Def($\perp$) = entry
2      while Def(stack.peek()) does not dominate inst do
3          stack.pop()
4      $v_i$ ← stack.peek()
5      replace the uses of $v$ by $v_i$ in inst
6      if $v_i \neq \perp$ then set Uses($v_i$) = Uses($v_i$)$\cup$ inst
```

```
1  Function stack.set_def(instruction inst)
2      let $v_i$ be a fresh version of $v$
3      replace the defs of $v$ by $v_i$ in inst
4      set Def($v_i$) = inst
5      stack.push($v_i$)
```

Fig. 13.8 Versioning

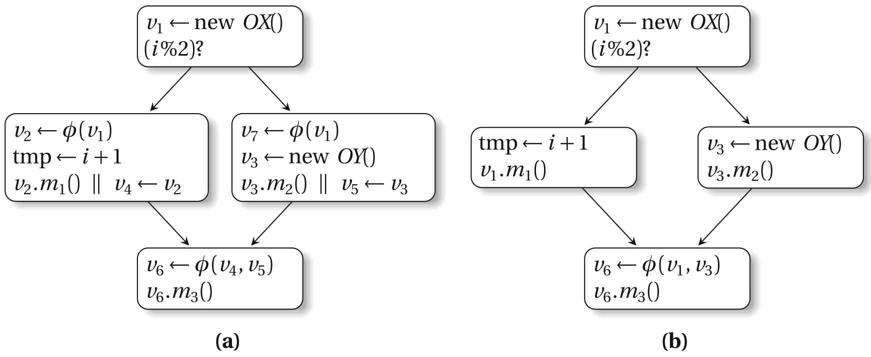
`stack.set_def`, that build the chains of relations between variables. Note that sometimes we must rename a single use inside a ϕ-function, as in lines 16–17 of the algorithm. For simplicity we consider this single use as a simple assignment when calling `stack.set_use`, as can be seen in line 17. Similarly, if we must rename a single definition inside a σ-function, then we treat it as a simple assignment, like we do in lines 12–14 of the algorithm.

```
1  Function clean(var v)
2      let web = $\{v_i | v_i$ is a version of $v\}$
3      defined $\leftarrow \emptyset$
4      active $\leftarrow \{inst | inst$ actual instruction and web $\cap$ inst.defs $\neq \emptyset\}$
5      while $\exists inst \in$ active | web $\cap$ inst.defs\defined $\neq \emptyset$ do
6          foreach $v_i \in$ web $\cap$ inst.defs\defined do
7              active $\leftarrow$ active $\cup$ Uses($v_i$)
8              defined $\leftarrow$ defined $\cup \{v_i\}$
9      used $\leftarrow \emptyset$
10     active $\leftarrow \{inst | inst$ actual instruction and web $\cap$ inst.uses $\neq \emptyset\}$
11     while $\exists inst \in$ active | inst.uses\used $\neq \emptyset$ do
12         foreach $v_i \in$ web $\cap$ inst.uses\used do
13             active $\leftarrow$ active $\cup$ Def($v_i$)
14             used $\leftarrow$ used $\cup \{v_i\}$
15     let live = defined $\cap$ used
16     foreach non actual $inst \in$ Def(web) do
17         foreach $v_i$ operand of inst | $v_i \notin$ live do
18             replace $v_i$ by $\perp$
19         if inst.defs = $\{\perp\}$ or inst.uses = $\{\perp\}$ then
20             remove inst
```

Fig. 13.9 Dead and undefined code elimination. Original instructions not inserted by `split` are called *actual* instructions. *inst*.defs denotes the (set) of variable(s) defined by *inst*, and *inst*.uses denotes the set of variables used by *inst*

13.2.4 Dead and Undefined Code Elimination

Just like Algorithm 3.7, the algorithm in Fig. 13.9 eliminates ϕ-functions and parallel copies that define variables not actually used in the code. By way of symmetry, it also eliminates σ-functions and parallel copies that use variables not actually defined in the code. We mean by "actual" instructions those that already existed in the program before we transformed it with `split`. In line 2, "web" is fixed to the set of versions of v, so as to restrict the cleaning process to variable v, as we see in the first two loops. The "active" set is initialized to actual instructions, line 4. Then, during the first loop in lines 5–8, we augment it with ϕ-functions, σ-functions, and copies that can reach actual definitions through use-def chains. The corresponding version of v is hence marked as *defined* (line 8). The next loop, lines 11–14, performs a similar process, this time to add to the active set instructions that can reach actual uses through def-use chains. The corresponding version of v is then marked as *used* (line 14). Each non-live variable, i.e., either undefined or dead (non-used), hence not in the "live" set (line 15) is replaced by $\perp$ in all ϕ, σ, or copy functions where it appears by the loop, lines 15–18. Finally, all useless ϕ, σ, or copy functions are removed by lines 19–20.



$v_1 \leftarrow$ new $OX()$
$(i\%2)?$
$v_2 \leftarrow \phi(v_1)$
tmp $\leftarrow i+1$
$v_2.m_1() \parallel v_4 \leftarrow v_2$
$v_7 \leftarrow \phi(v_1)$
$v_3 \leftarrow$ new $OY()$
$v_3.m_2() \parallel v_5 \leftarrow v_3$
$v_6 \leftarrow \phi(v_4, v_5)$
$v_6.m_3()$
(a)
$v_1 \leftarrow$ new $OX()$
$(i\%2)?$
tmp $\leftarrow i+1$
$v_1.m_1()$
$v_3 \leftarrow$ new $OY()$
$v_3.m_2()$
$v_6 \leftarrow \phi(v_1, v_3)$
$v_6.m_3()$
(b)

Fig. 13.10 (**a**) Implementing σ-functions via single arity ϕ-functions; (**b**) getting rid of copies and σ-functions

13.2.5 Implementation Details

Implementing σ-Functions

The most straightforward way to implement σ-functions, in a compiler that already supports the SSA form, is to represent them by ϕ-functions. In this case, the σ-functions can be implemented as single arity ϕ-functions. As an example, Fig. 13.10a shows how we would represent the σ-functions of Fig. 13.3d. If l is a branch point with n direct successors that would contain a σ-function $(l^1 : v_1, \ldots, l^n : v_n) \leftarrow \sigma(v)$, then, for each direct successor l^j of l, we insert at the beginning of l^j an instruction $v_j \leftarrow \phi(l^j : v)$. Note that l^j may already contain a ϕ-function for v. This happens when the control-flow edge $l \rightarrow l^j$ is *critical*: A critical edge links a basic block with several direct successors to a basic block with several direct predecessors. If l^j already contains a ϕ-function $v' \leftarrow \phi(\ldots, v_j, \ldots)$, then we rename v_j to v.

SSI Destruction

Traditional instruction sets do not provide ϕ-functions or σ-functions. Thus, before producing an executable program, the compiler must implement these instructions. We have already seen in Chap. 3 how to replace ϕ-functions with actual assembly instructions; however, now we must also replace σ-functions and parallel copies. A simple way to eliminate all the σ-functions and parallel copies is via copy-propagation. In this case, we copy-propagate the variables that these special instructions define. As an example, Fig. 13.10b shows the result of copy folding applied on Fig. 13.10a.

13.3 Further Reading

The monotone data-flow framework is an old ally of compiler writers. Since the work of pioneers like Prosser [234], Allen [3, 4], Kildall [166], Kam [158], and Hecht [143], data-flow analyses such as reaching definitions, available expressions, and liveness analysis have made their way into the implementation of virtually every important compiler. Many compiler textbooks describe the theoretical basis of the notions of lattice, monotone data-flow framework, and fixed points. For a comprehensive overview of these concepts, including algorithms and formal proofs, we refer the interested reader to Nielson et al.'s book [208] on static program analysis.

The original description of the intermediate program representation known as Static Single Information form was given by Ananian in his Master's thesis [8]. The notation for σ-functions that we use in this chapter was borrowed from Ananian's work. The SSI program representation was subsequently revisited by Jeremy Singer in his PhD thesis [261]. Singer proposed new algorithms to convert programs to SSI form, and also showed how this program representation could be used to handle truly bidirectional data-flow analyses. We have not discussed bidirectional data-flow problems, but the interested reader can find examples of such analyses in Khedker et al.'s work [165]. Working on top of Ananian's and Singer's work, Boissinot et al. [37] have proposed a new algorithm to convert a program to SSI form. Boissinot et al. have also separated the SSI program representation into two flavours, which they call *weak* and *strong*. Tavares et al. [282] have extended the literature on SSI representations, defining building algorithms and giving formal proofs that these algorithms are correct. The presentation that we use in this chapter is mostly based on Tavares et al.'s work.

There exist other intermediate program representations that, like the SSI form, make it possible to solve some data-flow problems sparsely. Well-known among these representations is the *Extended Static Single Assignment* form, introduced by Bodik *et al.* to provide a fast algorithm to eliminate array bound checks in the context of a JIT compiler [32]. Another important representation, which supports data-flow analyses that acquire information at use sites, is the *Static Single Use* form (SSU). As uses and definitions are not fully symmetric (the live range can "traverse" a use while it cannot traverse a definition), there are different variants of SSU [125, 187, 228]. For instance, the "strict" SSU form enforces that each definition reaches a single use, whereas SSI and other variations of SSU allow two consecutive uses of a variable on the same path. All these program representations are very effective, having seen use in a number of implementations of flow analyses; however, they only fit specific data-flow problems.

The notion of *Partitioned Variable Problem* (PVP) was introduced by Zadeck, in his PhD dissertation [316]. Zadeck proposed fast ways to build data structures that allow one to solve these problems efficiently. He also discussed a number of data-flow analyses that are partitioned variable problems. There are data-flow analyses that do not meet the Partitioned Lattice per Variable property. Notable examples include abstract interpretation problems on relational domains, such as Polyhedrons [86], Octagons [199], and Pentagons [188].

In terms of data structures, the first, and best known method proposed to support sparse data-flow analyses is Choi et al.'s *Sparse Evaluation Graph* (SEG) [67]. The nodes of this graph represent program regions where information produced by the data-flow analysis might change. Choi et al.'s ideas have been further expanded, for example by Johnson et al.'s *Quick Propagation Graphs* [153], or Ramalingam's *Compact Evaluation Graphs* [237]. Nowadays we have efficient algorithms that build such data structures [154, 224, 225]. These data structures work best when applied on partitioned variable problems.

As opposed to those approaches, the solution promoted by this chapter consists in an intermediate representation (IR) based evaluation graph, and has advantages and disadvantages when compared to the data structure approach. The intermediate representation based approach has two disadvantages, which we have already discussed in the context of the standard SSA form. First it has to be maintained and at some point destructed. Second, because it increases the number of variables, it might add some overhead to analyses and transformations that do not require it. On the other hand, IR based solutions to sparse data-flow analyses have many advantages over data structure based approaches. For instance, an IR allows concrete or abstract interpretation. Solving any coupled data-flow analysis problem along with a SEG was mentioned by Choi et al. [67] as an open problem. However, as illustrated by the conditional constant propagation problem described in Chap. 8, coupled data-flow analysis can be solved naturally in IR based evaluation graphs. Last, SSI is compatible with SSA extensions such as gated SSA described in Chap. 14, which allows demand-driven interpretation.

The data-flow analyses discussed in this chapter are well-known in the literature. Class inference was used by Chambers et al. in order to compile Self programs more efficiently [63]. Nanda and Sinha have used a variant of null-pointer analysis to find which call sites may cause errors due to the dereference of null objects [207]. Ananian [8], and later Singer [261], have shown how to use the SSI representation to do partial redundancy elimination sparsely. In addition to being used to eliminate redundant array bound checks [32], the e-SSA form has been used to solve Taint Analysis [248], and range analysis [123, 277]. Stephenson et al. [273] described a bitwidth analysis that is both forward and backward, taking information from definitions, uses, and conditional tests. For another example of bidirectional bitwidth analysis, see Mahlke et al.'s algorithm [194]. The type inference analysis that we mentioned in Fig. 13.5 was taken from Hochstadt et al.'s work [284].