1 Review on Abstract Algebra Definitions

• **Set**: a collection of things, with union, intersection and membership

• **Relation**: a collection of pair of things; we write $R(a, b)$ to indicate that the pair $(a, b)$ is in the relation $R$.

  A relation can be:
  
  – **reflexive**: for all $a$, $R(a, a)$
  – **symmetric**: if $R(a, b)$ then $R(b, a)$
  – **antisymmetric**: if $R(a, b)$ and $a \neq b$ then not $R(b, a)$, or alternatively if $R(a, b)$ and $R(b, a)$ then $a = b$
  – **transitive**: if $R(a, b)$ and $R(b, c)$ then $R(a, c)$

• **(Loose, partial) Order Relation**: a transitive, reflexive, antisymmetric relation; usually indicated with $\leq$ or $\preceq$.

• **Total Order Relation**: an order relation such that $a \preceq b$ or $b \preceq a$ for all $a, b$.

• **(Internal) Binary Operation**: a function from pair of things to things ($X \times X \to X$).

  An operation can be:
  
  – **associative**: $f(f(a, b), c) = f(a, f(b, c))$
  – **commutative**: $f(a, b) = f(b, a)$
  – **admit an identity element**: exists $1$ such that $f(a, 1) = f(1, a) = a$
  – **invertible**: for all $a$ exists $a^{-1}$ such that $f(a, a^{-1}) = 1$
  – **admit a zero element**: exists $0$ such that $f(a, 0) = f(0, a) = 0$
  – **idempotent**: $f(a, a) = a$

A set with an associative operation is called a **semi-group**. If the operation has an identity the structure is called a **monoid**. If the operation is also invertible, it becomes a **group**. If the operation is also commutative, it is called an **abelian group**.
• **Meet operation**: an associative, commutative, idempotent operation; usually meet operations are indicated with $\wedge$.

• **Join operation**: the same as a meet operation; if a set has two meet operations, and they are distributive w.r.t. to each other, one is called a meet and the other a join.

• **Monotonic operation** (w.r.t. an order relation): if $a \preceq b$ then $f(a) \preceq f(b)$.

## 2 Meet Operations and Order Relations

Meet operations and partial orders are almost interchangeable.

Given a meet operation $\wedge$, we can define the partial order $\preceq$. If $c = a \wedge b$, then $c \preceq a$ and $c \preceq b$. The rest of the relation is defined by *transitive closure*. In particular, observe that because $\wedge$ is idempotent, $\preceq$ is reflexive.

Given a partial order $\preceq$, we can define the meet operation $\wedge$ as the greatest lower bound. In other words, $c = a \wedge b$ if and only if $c \preceq a$ and $c \preceq b$ and there exists no $c' \neq c$ such that $c \preceq c' \preceq a$ and $c \preceq c' \preceq b$. This works only if $c$ exists: if two distinct elements are incomparable, and there is no element smaller than both, then we cannot define $\wedge$.

A meet operation on a set immediately defines a semi-lattice on that set. A partial order on set defines a semi-lattice, provided that the greatest lower bound of any pair of elements exists.

## 3 Available and Anticipated Expressions

An *anticipated expression* at a point $p$ is one that will be computed on all paths after that point, without a subsequent overwrite of one of the operands of the expression. Similarly, an *available expression* at a point $p$ is one which has been computed on all paths up to that point.

This can be used to eliminate some repeated computations by writing the computed value to a common temporary variable.

For example, at the indicated program point $b + c$ is anticipated, because it is computed immediately after with no overwrite of $b$ or $c$. It is available, because it is computed on both basic blocks leading to the program point. Therefore we can rewrite to the equivalent program on the right:

```
\begin{align*}
\text{a = } b + c \\
\text{i = i + 1} \\
\text{z = b + c} \\
\text{x = b + c}
\end{align*} \quad \Rightarrow \quad
\begin{align*}
\text{t = b + c} \\
\text{a = t} \\
\text{i = i + 1} \\
\text{t = b + c} \\
\text{z = t} \\
\text{x = t}
\end{align*}
```

Note that if we change $i = i + 1$ to $b = i + 1$ in the left block, the expression is no longer available at the marked point because one of the operands has been overwritten on that path.
4 Static Single Assignment

4.1 Basics of SSA

Consider the following program:

\[
\begin{align*}
&b_0: x = 1; \\
&b_1: x = -1; \\
&b_2: y = x; \\
&\text{exit;}
\end{align*}
\]

We might wonder if \(x\) is live in this program. Unfortunately, in this form, the question is ill-formed: we need to ask: “is \(x\) live at a particular program point?” There are two definitions of \(x\), and indeed \(x\) is live at the beginning of \(b_2\), but it is dead at the beginning of \(b_1\).

Intuitively, though, the two definitions of \(x\) seem unrelated, given that \(x\) is overwritten entirely in \(b_1\). So alternatively, we might wonder: “is a particular definition of \(x\) live in this program?” Definitions appear only once in a program, so this question is well-formed.

Static Single Assignment (SSA for short) is a formalism that allows to distinguish between multiple definitions of a variable, by renaming each variable such that it is only assigned once in text of the program. In SSA, we would rewrite the program as:

\[
\begin{align*}
&b_0: x_1 = 1; \\
&b_1: x_2 = 2; \\
&b_2: y_1 = x_1; \\
&\text{exit;}
\end{align*}
\]

Given this transformation, the question “is \(x_1\) live in the program?” becomes well-formed, because \(x_1\) is guaranteed to be assigned only once. Static Single Assignment therefore allows us to rewrite many common optimization algorithms (liveness analysis, constant propagation, copy propagation) so that they compute a value for each variable once for the entire program (or procedure), rather than once at each program point. This incurs a great reduction in memory use and compilation speed.
4.2 Coping with Control Flow

Consider the following program:

\[
\begin{align*}
&b_1: x = 1; \\
&b_2: x = 2; \\
&b_3: y = x; \\
&\text{exit}
\end{align*}
\]

Which definition of \( x \) should we use in the assignment to \( y \) in block \( b_3 \)? We need a definition of \( x \) that incorporates the properties of both \( x_1 \) (in block \( b_1 \)) and \( x_2 \) (in block \( b_2 \)).

To address this problem, the SSA formalism introduces the concept of phi-nodes (denoted \( \varphi \)). Informally, a phi-node is a special instruction with many operands, each coming from a different basic block. The phi-node acts as an oracle and “chooses” which value to propagate to the new definition.

With phi-nodes, we rewrite the previous program as:

\[
\begin{align*}
&b_1: x_1 = 1; \\
&b_2: x_1 = 2; \\
&b_3: x_3 = \varphi(x_1 \text{ from } b_1, x_2 \text{ from } b_2); \\
&b_4: y_1 = x_3; \\
&\text{exit}
\end{align*}
\]

Again, in the new program each variable \( x_1, x_2, x_3, y_1 \) is defined only once, so this is a valid Static Single Assignment form. Each operand in a phi-node is annotated with the basic block where that operand comes from, which allows us to reason about the phi-node, even though the phi-node is just an abstraction and not a real machine instruction.

\( \phi \) nodes generalize to control flow with loops as well, for example:

\[
\begin{align*}
&b_1: x_1 = 0; \\
&b_2: x_2 = \varphi(x_1 \text{ from } b_1, x_3 \text{ from } b_3); \\
&b_3: x_3 = x_2 + 1; \\
&\text{exit}
\end{align*}
\]
In that case, it is important to observe that a variable can still be set multiple times in the course of a program, and can assume multiple values. All Static Single Assignment cares about is that there is one assignment to the variable in the text of the program (hence “static”).

4.3 Using SSA in the dataflow framework

To use SSA efficiently in the dataflow framework, we will define a lattice as a product of lattices for each variable. At each program point, the lattice will represent the state of the variables (e.g. whether they are constant, or whether they are live) at that point. But because variables are assigned only once, we do not need to store this value at every point: at the end of the computation, all program points will converge to the same value. Therefore we keep only one value for the whole program (for each variable) and we continuously update it until convergence.

Initialization and boundary condition for an analysis in SSA form are the same as the regular dataflow analysis. The transfer function for individual regular instructions is also the same. We are left to define the transfer function for phi-nodes.

For backward passes, the transfer function simply copies the value from target of the phi-node to each operand. For example, if the target of a phi-node is live, all its operands are live. For forward passes, the transfer function is the meet of the values of each operand. For example, in constant propagation, the value of a phi-node is the meet of the constant values of each operand. If both phi-node operands are the same constant, the phi-node is also a constant; if they are different constants, the value of the phi-node is bottom (not a constant).

4.4 Converting to and from Static Single Assignment

Multiple algorithms exist to convert any program into static single assignment. These algorithms number the variable based on the definitions and insert the appropriate phi-nodes. These algorithms differ in how efficient they are to compute, and how many redundant phi-nodes are inserted.

Conversely, we will want to go from SSA back to the normal program, right before machine dependent optimizations are performed (register allocation and instruction scheduling). This involves removing phi-nodes, because they do not correspond to any real instruction. A simple algorithm to remove phi-node replaces them with copies: given a phi node assigning to $x_i$, for every operand of the form “$x_j$ from $b_k$”, we insert a copy $x_i = x_j$ at the end of $b_k$. For example, in the loop program we would write:
This program is no longer in static single assignment form, because there are now two assignments to \( x_2 \), and the subsequent code (in particular register allocation) will have to deal with that.