CS 6110 Lecture 36 A Monadic Approach to Abstract Interpretation 22 April 2013

Lecturer: Andrew Myers

1 Monads

In an earlier lecture we introduced the idea of *monad* that transforms one domain D into another, $M(D)$. Monads are very useful as a way of characterizing the effects of computation. The original domain represents a value or state, and the result domain represents a computation over that value or state.

1.1 Monad operations

The monad is characterized by two key operations: unit and bind.

$$\begin{aligned} \textit{unit} &: D \to M(D) \\ \textit{bind} &: (D \to M(E)) \to (M(D) \to M(E)) \end{aligned}$$

It is also possible to define monads in terms of two other operations, *map* : $(D \to E) \to M(D) \to M(E)$ and *join* : $M(M(D)) \to M(D)$. This is often the approach taken in category theory, but the definitions are equivalent.

We use the notation $[\sigma]$ to denote *unit*(σ), and the notation $f^*(m)$ to denote *bind*$(f)(m)$.

Monads figure heavily in the programming language Haskell, which uses monads as a way to introduce impure computation effects into what is otherwise a mostly pure functional language (modulo divergence). The unit operation is called `return` in Haskell, and the bind operation is written as >>=. Haskell also supports syntactic sugar that makes monads look like imperative code.

1.2 Monad laws

The unit and bind operations should satisfy the following laws:

- Initial unit law: $f(\sigma) = f^*([\sigma])$
- Final unit law: $[\cdot]^*(m) = m$
- Associativity: $(f_1^* \circ f_2)^* = f_1^* \circ f_2^*$

In practice, it is often useful to use operations that do not quite obey these laws, though this means that something is lost in the transformation.

1.3 A generic semantics

Recall from the earlier lecture on nondeterminism that we can write the semantics of IMP generically in terms of these monad operations:

$$\begin{aligned} \mathcal{C}[\![\textbf{skip}]\!]\sigma &= [\sigma] \\ \mathcal{C}[\![x := a]\!]\sigma &= [\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]] \\ \mathcal{C}[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]\sigma &= \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } \mathcal{C}[\![c_1]\!]\sigma \text{ else } \mathcal{C}[\![c_2]\!]\sigma \\ \mathcal{C}[\![c_1; c_2]\!]\sigma &= \mathcal{C}[\![c_2]\!]^*(\mathcal{C}[\![c_1]\!]\sigma) = (\mathcal{C}[\![c_2]\!]^* \circ \mathcal{C}[\![c_1]\!])(\sigma) \\ \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!] &= \text{fix} \lambda w \in \Sigma \to M(\Sigma). \\ &\qquad \lambda \sigma \in \Sigma.\, \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } w^*(\mathcal{C}[\![c]\!]\sigma) \text{ else } [\sigma] \end{aligned}$$

The semantics of $\mathcal{C}[\![c]\!]\sigma$ is a function in $\Sigma \to M(\Sigma)$, where for the usual deterministic semantics, we have $M(\Sigma) = \Sigma_\perp$, and the monad operations are defined as follows: $[\sigma] = \lfloor \sigma \rfloor, f^*(\perp) = \perp, f^*(\lfloor \sigma \rfloor) = f(\sigma)$. This monad is called the *strictness monads*: its job is to handle the computational effect of divergence, which it does by ensuring that two sequentially composed computations are strict in the first computation (since $f^*(\perp) = \perp$).

1.4 The CPS monad

We have seen two monads for the IMP semantics. Here is a third, the CPS monad. We define the monad as follows:

$$\begin{aligned}
M(\Sigma) &= (\Sigma \to \textbf{Answer}) \to \textbf{Answer} \\
[\sigma] &= \lambda k \in \Sigma \to \textbf{Answer}.\, k\ \sigma \\
f^*(m) &= \lambda k \in \textbf{Answer}.\, m\ (\lambda\sigma \in \Sigma.\, f\ \sigma\ k)
\end{aligned}$$

Here the domain $\Sigma \to$ **Answer** describes continuations k, so the meaning of a command in a particular state, $M(\Sigma)$, is a function that will supply an answer if given a continuation to send it to. If we use these definitions in the generic semantics above, we obtain a CPS semantics for IMP:

$$\begin{aligned}
\mathcal{C}[\![\textbf{skip}]\!]\sigma &= [\sigma] = \lambda k \in \Sigma \to \textbf{Answer}.\, k\ \sigma \\
\mathcal{C}[\![x := a]\!]\sigma &= \lambda k \in \Sigma \to \textbf{Answer}.\, k\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] \\
\mathcal{C}[\![\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2]\!]\sigma &= \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } \mathcal{C}[\![c_1]\!]\sigma \text{ else } \mathcal{C}[\![c_2]\!]\sigma \\
\mathcal{C}[\![c_1; c_2]\!]\sigma &= \mathcal{C}[\![c_2]\!]^*(\mathcal{C}[\![c_1]\!]\sigma) \\
&= \lambda k \in \Sigma \to \textbf{Answer}.\, \mathcal{C}[\![c_1]\!]\sigma(\lambda\sigma'.\, \mathcal{C}[\![c_2]\!]\sigma') \\
\mathcal{C}[\![\textbf{while}\ b\ \textbf{do}\ c]\!] &= \text{fix } \lambda w \in \Sigma \to M(\Sigma). \\
&\qquad \lambda\sigma \in \Sigma.\, \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } \lambda k.\, \mathcal{C}[\![c]\!]\sigma(\lambda\sigma'.\, w\ \sigma'\ k) \text{ else } \lambda k.\, k\ \sigma \\
&\approx \text{fix } \lambda w \in \Sigma \to M(\Sigma). \\
&\qquad \lambda\sigma \in \Sigma.\, \lambda k.\, \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } \mathcal{C}[\![c]\!]\sigma(\lambda\sigma'.\, w\ \sigma'\ k) \text{ else } k\ \sigma
\end{aligned}$$

Clearly, the CPS monad is capturing the essence of CPS conversion, which is to make the sequence of control transfers during execution explicit as function applications.

2 Abstract interpretation

We have spent several lectures talking about type systems, which are a form of static program analysis in which we analyze each term to assign it a type. Assuming the type system is sound, the type is then a characterization of the possible values that can occur at run time. It does not describe exactly the possible values (usually), but *overapproximates* the possible values in that the type corresponds to a larger set of values than can actually occur in the program at that point.

Abstract interpretation is the other major way to obtain such approximate characterizations of program behavior. Abstract interpretation is the evaluation of programs using using an alternate, abstract semantics. Whereas ordinary evaluation of programs computes particular, concrete values or states, the abstract interpretation computes an abstract value or state that may correspond to many concrete values or states. If the analysis is sound, this abstract result is a conservative overapproximation of the actual possible concrete result.

Unlike type systems, abstract interpretation is usually done on programs without the help of programmer-supplied annotations. Abstract interpretation also differs from (most) type systems in that the results it computes are *flow-sensitive*: the static characterization of the program may differ at different points in the program.

Dataflow analysis, which is used in compilers, is an instance of abstract interpretation, in which the transfer function associated with a control-flow graph node expresses the abstract semantics of computation at that node.

2.1 An example abstraction

Suppose that we want to ensure that all array indices are in bounds in a language like C or Java, to avoid checking indices at run time. For every expression $a[i]$, this requires that $0 \leq i < \text{length}(a)$. We don't need to know the exact value of i at run time; we only need to know that it is in bounds. Therefore, we might perform an abstract interpretation in which we keep track of the possible values of all integer variables, as an interval $[x, y]$ where the possible values of i at run time must lie in this interval.

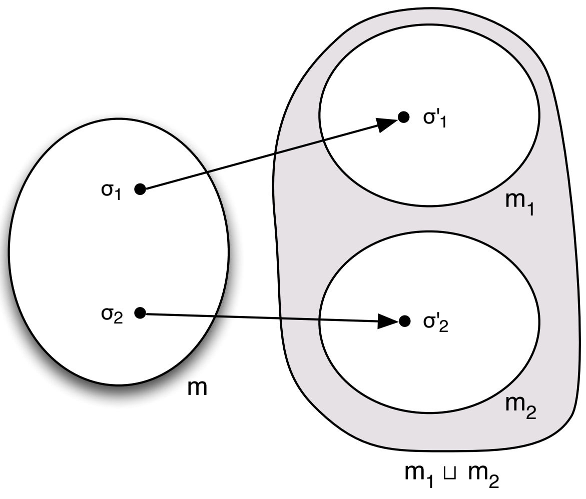

Figure 1: Joining abstract states

Given a statement like $i := i + 1$, if its possible values before the statement lie in $[x, y]$, the possible values after the statement lie in $[x + 1, y + 1]$. An *abstract interpreter* executes the program while operating on these abstract values, by lifting the semantics of concrete executions up to the abstract domain.

2.2 Structure of the abstract domain

In the case of IMP, we want to evaluate programs using abstract states drawn from a space M rather than concrete states from Σ. We assume that each concrete state σ has a best representative $\beta(\sigma)$ in M, where $\beta : \Sigma \to M$ is the *representation function* that maps concrete states to their most precise representations as abstract states. Two different concrete states may share the same best representative, such as the states σ_1 and σ_2 in Figure 1.

In general, during evaluation of the program, two states that have the same abstract representative may step to states that do not. In the figure, the states σ_1 and σ_2 lead to states σ_1' and σ_2' that are represented by different abstract states m_1 and m_2. In an abstract interpretation, we don't know which concrete state we started from, so both m_1 and m_2 are possible abstract states. To solve this problem, we require that any two abstract states m_1 and m_2 have a least upper bound (or *join*) $m_1 \sqcup m_2$. Since we can take arbitary joins, M is a *complete lattice* with both a bottom element $\bot$ and a top element $\top$.

The ordering of elements in M corresponds to *decreasing* precision: if $m \sqsubseteq m'$, then m' represents at least as many states as m. The most precise information that the analysis can produce is $\bot$, meaning that it represents no states (for example, the computation does not terminate), and the least precise information is $\top$, representing all possible states.

In a complete lattice, just as in a CPO, every chain has a LUB, but that LUB need not be in the chain. If we are building a series of approximations to the desired abstract result, as will be necessary for **while**, this is unfortunate because perhaps the analysis will never arrive at the desired result. To ensure analyses terminate, it is important to have a *widening operator* ∇. The widening of two elements produces an element that is an upper bound of both: $m_1 \sqsubseteq m_1 \;\nabla\; m_2 \sqsupseteq m_2$, so therefore it must also be an upper bound of their LUB: $m_1 \sqcup m_2 \sqsubseteq m_1 \;\nabla\; m_2$. Further, any chain constructed using the widening operator is guaranteed to *stabilize* in the sense that it contains its LUB. Such a chain has this form:

$$m_1 \sqsubseteq m_1 \;\nabla\; m_2 \sqsubseteq m_1 \;\nabla\; m_2 \;\nabla\; m_3 \sqsubseteq \ldots \sqsubseteq \text{LUB} \sqsubseteq \text{LUB} \sqsubseteq \ldots$$

Note that if the lattice has finite height, as is often the case for program analyses, *all* chains are guaranteed to stabilize. In that case, the join operator $\sqcup$ may be used as the widening operator.

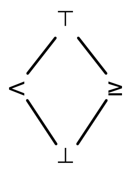

Figure 2: A lattice that keeps track of whether x is negative.

2.3 Abstract interpretation as a monad

To construct an abstract interpretation, it is helpful to think of it as a monadic computation. We define the monad as follows:

$$\begin{aligned} M(\Sigma) &= M \\ [\sigma] &= \beta(\sigma) \\ f^*(m) &= \bigsqcup \{f(\sigma) \mid \beta(\sigma) \sqsubseteq m\} \end{aligned}$$

Instantiated on this monad, the generic semantics of the program $\mathcal{C}[\![c]\!]\sigma$ becomes a mapping from concrete states to abstract states: $\mathcal{C}[\![c]\!] : \Sigma \to M(\Sigma)$. Therefore we can use the bind operation to convert this semantics into an evaluation purely on abstract states, which is what we want. The meaning of a command becomes a transformer on abstract states:

$$\mathcal{C}[\![c]\!]^* \ \in \ M(\Sigma) \to M(\Sigma) \ = \ M \to M$$

For some IMP expressions, we can evaluate $\mathcal{C}[\![c]\!]^*$ directly. For example, since $\mathcal{C}[\![\textbf{skip}]\!] = [\sigma]$, then by the final unit law, $\mathcal{C}[\![\textbf{skip}]\!]^*(m) = [\cdot]^*(m) = m$. Of course, we ought to validate that this monad really obeys the final unit law. Because $\mathcal{C}[\![\textbf{skip}]\!]\sigma = [\sigma] = \beta(\sigma)$, therefore, $\mathcal{C}[\![\textbf{skip}]\!]^*(m) = \bigsqcup \{\beta(\sigma) \mid \beta(\sigma) \sqsubseteq m\} = m$.

For other language forms, we can't evaluate $\mathcal{C}[\![c]\!]^*$ without knowing what the actual abstraction is. And in general, even if we do know the abstraction, it will be infeasible to compute $\mathcal{C}[\![c]\!]^*$ precisely. Instead, we will settle for defining a overapproximation to this abstraction, which we will denote by $\mathcal{C}^*[\![c]\!]$. For any element m, we require $\mathcal{C}[\![c]\!]^* \sqsubseteq \mathcal{C}^*[\![c]\!]$ so that the analysis is conservative. For precision, we define $\mathcal{C}^*[\![c]\!]$ to be $\mathcal{C}[\![c]\!]^*$ whenever the latter is tractable.

2.3.1 Example

Let us consider a simple abstraction of program states in which the only thing we care about whether a particular program variable x is negative or nonnegative. This is similar to but simpler than the example given earlier, because we only care about one variable, not all variables, and we only care about its negativity.

The resulting lattice of abstract states contains just four elements, depicted in Figure 2. We can define $\mathcal{C}^*[\![c]\!]$, and even $\mathcal{C}[\![c]\!]^*$, in a mostly straightforward way. For example, consider the assignment command:

$$\begin{aligned} \mathcal{C}[\![x := a]\!]\sigma &= [\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]] = \beta(\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]) \\ &= \text{“}<\text{”} && (\text{if } \mathcal{A}[\![a]\!]\sigma < 0) \\ &= \text{“}\geq\text{”} && (\text{if } \mathcal{A}[\![a]\!]\sigma \geq 0) \end{aligned}$$

By the monad definition, we have:

$$\mathcal{C}[\![x := a]\!]^*(m) = \bigsqcup \{\beta(\sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]) \mid \beta(\sigma) \sqsubseteq m\}$$

Expanding this for each of the possible values of m, we obtain the following abstract interpretation of assignment, in which we no longer need to use concrete states:

$$\mathcal{C}[\![x := a]\!]^*\bot = \bot \qquad \text{(In fact, this is true for \textit{every} command)}$$

$$\mathcal{C}[\![x := a]\!]^*\text{“}{<}\text{”} = \begin{cases} \text{“}{<}\text{”} & \text{if } x < 0 \Rightarrow a < 0 \\ \text{“}{\geq}\text{”} & \text{if } x < 0 \Rightarrow a \geq 0 \\ \top & \text{otherwise} \end{cases}$$

$$\mathcal{C}[\![x := a]\!]^*\text{“}{\geq}\text{”} = \begin{cases} \text{“}{<}\text{”} & \text{if } x \geq 0 \Rightarrow a < 0 \\ \text{“}{\geq}\text{”} & \text{if } x \geq 0 \Rightarrow a \geq 0 \\ \top & \text{otherwise} \end{cases}$$

$$\mathcal{C}[\![x := a]\!]^*\top = \begin{cases} \text{“}{<}\text{”} & \text{if } a < 0\text{, for all } \sigma \\ \text{“}{\geq}\text{”} & \text{if } a \geq 0\text{, for all } \sigma \\ \top & \text{otherwise} \end{cases}$$

In this case we can compute $\mathcal{C}[\![c]\!]^*$ precisely, so we can define $\mathcal{C}^*[\![x := a]\!] = \mathcal{C}[\![x := a]\!]^*$.

We can perform a similar analysis for the other language constructs. For assignments to variables other than x, the abstract state does not change at all:

$$\mathcal{C}[\![y := a]\!]^*(m) = m \qquad \text{(where } y \neq x\text{)}$$

For sequential composition, we can make use of the monad associativity law:

$$\begin{aligned} \mathcal{C}[\![c_1; c_2]\!]^* &= (\mathcal{C}[\![c_2]\!]^* \circ \mathcal{C}[\![c_1]\!])^* \\ &= \mathcal{C}[\![c_2]\!]^* \circ \mathcal{C}[\![c_1]\!]^* \end{aligned}$$

In general we cannot compute $\mathcal{C}[\![c_1]\!]^*$ or $\mathcal{C}[\![c_2]\!]^*$, but since (inductively) we know $\mathcal{C}[\![c_i]\!]^* \sqsubseteq \mathcal{C}^*[\![c_i]\!]$, and the functions are monotonic, we have $\mathcal{C}[\![c_2]\!]^* \circ \mathcal{C}[\![c_1]\!]^* \sqsubseteq \mathcal{C}^*[\![c_2]\!] \circ \mathcal{C}^*[\![c_1]\!]$. Therefore the right-hand side is a conservative approximation that we can use to define $\mathcal{C}^*$:

$$\mathcal{C}^*[\![c_1; c_2]\!] = \mathcal{C}^*[\![c_2]\!] \circ \mathcal{C}^*[\![c_1]\!]$$

This is exactly how we would expect the analysis of sequential composition to behave, independent of the analysis.

Applying a similar analysis to **if**, we obtain:

$$\mathcal{C}^*[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]\text{“}{<}\text{”} = \begin{cases} \mathcal{C}^*[\![c_1]\!]\text{“}{<}\text{”} & \text{if } x < 0 \Rightarrow b \\ \mathcal{C}^*[\![c_2]\!]\text{“}{<}\text{”} & \text{if } x < 0 \Rightarrow \neg b \\ \mathcal{C}^*[\![c_1]\!]\text{“}{<}\text{”} \sqcup \mathcal{C}^*[\![c_2]\!]\text{“}{<}\text{”} & \text{otherwise} \end{cases}$$

$$\mathcal{C}^*[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]\text{“}{\geq}\text{”} = \begin{cases} \mathcal{C}^*[\![c_1]\!]\text{“}{\geq}\text{”} & \text{if } x \geq 0 \Rightarrow b \\ \mathcal{C}^*[\![c_2]\!]\text{“}{\geq}\text{”} & \text{if } x \geq 0 \Rightarrow \neg b \\ \mathcal{C}^*[\![c_1]\!]\text{“}{\geq}\text{”} \sqcup \mathcal{C}^*[\![c_2]\!]\text{“}{\geq}\text{”} & \text{otherwise} \end{cases}$$

$$\mathcal{C}^*[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!]\top = \mathcal{C}^*[\![c_1]\!]m_1 \sqcup \mathcal{C}^*[\![c_2]\!]m_2$$

$$\text{where} \quad m_1 = \begin{cases} \text{“}{<}\text{”} & \text{if } b \Rightarrow x < 0 \\ \text{“}{\geq}\text{”} & \text{if } b \Rightarrow x \geq 0 \\ \top & \text{otherwise} \end{cases} \quad \text{and} \quad m_2 = \begin{cases} \text{“}{<}\text{”} & \text{if } \neg b \Rightarrow x < 0 \\ \text{“}{\geq}\text{”} & \text{if } \neg b \Rightarrow x \geq 0 \\ \top & \text{otherwise} \end{cases}$$

Using the notation $\sigma \models \phi$ to mean that boolean expression ϕ is true in state σ, and the notation $\sigma \models m$ to mean

$\beta(\sigma) = m$, we can collapse all of these cases as follows:

$$\mathcal{C}^*[\![\textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2]\!](m) = \mathcal{C}^*[\![c_1]\!]m_1 \;\sqcup\; \mathcal{C}^*[\![c_2]\!]m_2$$

$$\text{where} \quad m_1 = \begin{cases} \bot & \text{if } \sigma \models m \Rightarrow \sigma \models \neg b \\ \text{“}<\text{”} & \text{otherwise, if } \sigma \models m \wedge \sigma \models b \Rightarrow \sigma \models x < 0 \\ \text{“}\geq\text{”} & \text{otherwise, if } \sigma \models m \wedge \sigma \models b \Rightarrow \sigma \models x \geq 0 \\ \top & \text{in all other cases} \end{cases}$$

$$\text{and} \quad m_2 = \begin{cases} \bot & \text{if } \sigma \models m \Rightarrow \sigma \models b \\ \text{“}<\text{”} & \text{otherwise, if } \sigma \models m \wedge \sigma \models \neg b \Rightarrow \sigma \models x < 0 \\ \text{“}\geq\text{”} & \text{otherwise, if } \sigma \models m \wedge \sigma \models \neg b \Rightarrow \sigma \models x \geq 0 \\ m & \text{in all other cases} \end{cases}$$

Finally, we come to the case of **while**, where we need to iterate to a fixed point, relying on the widening operator to achieve convergence. For this lattice we can choose a widening operator $\nabla = \sqcup$. Recall that by the definition of *fix*, we have:

$$\begin{aligned} \mathcal{C}[\![\textbf{while } b \textbf{ do } c]\!]^* m &= (\text{fix } F)^*(m) \\ &\quad \text{where } F(w\!:\!\Sigma \to M) = \lambda\sigma\!\in\!\Sigma.\, \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } w^*(\mathcal{C}[\![c]\!]\sigma) \text{ else } [\sigma] \\ (\text{fix } F)^* &= (F(\text{fix}F))^* \qquad \text{(because fix finds a fixed point)} \\ &= (\lambda\sigma.\, \text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } (\text{fix } F)^*(\mathcal{C}[\![c]\!]\sigma) \text{ else } [\sigma])^* \\ &= \lambda m.\, \bigsqcup\{\text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } (\text{fix } F)^*(\mathcal{C}[\![c]\!]\sigma) \text{ else } [\sigma] \mid m = \beta(\sigma)\} \end{aligned}$$

This is a recursive definition of $(\text{fix } F)^*$, meaning that we can find it by taking a fixed point in $M \to M$ directly:

$$\begin{aligned} (\text{fix } F)^* &= \text{fix } G \\ &\quad \text{where } G(w'\!:\!M \to M) = \lambda m.\, \bigsqcup\{\text{if } \mathcal{B}[\![b]\!]\sigma \text{ then } w'(\mathcal{C}[\![c]\!]\sigma) \text{ else } [\sigma] \mid m = \beta(\sigma)\} \end{aligned}$$

This can be expanded on the possible arguments, conservatively using $\mathcal{C}^*[\![c]\!]\sigma$ as an overapproximation to $\mathcal{C}[\![c]\!]^*\sigma$:

$$G(w')(\bot) = \bot$$

$$G(w')(\text{“}<\text{”}) = \begin{cases} \text{“}<\text{”} & \text{if } x < 0 \Rightarrow \neg b \\ (w' \circ \mathcal{C}^*[\![c]\!])\text{“}<\text{”} & \text{if } x < 0 \Rightarrow b \\ \text{“}<\text{”} \sqcup (w' \circ \mathcal{C}^*[\![c]\!])\text{“}<\text{”} & \text{otherwise} \end{cases}$$

$$G(w')(\text{“}\geq\text{”}) = \begin{cases} \text{“}\geq\text{”} & \text{if } x \geq 0 \Rightarrow \neg b \\ (w' \circ \mathcal{C}^*[\![c]\!])\text{“}\geq\text{”} & \text{if } x \geq 0 \Rightarrow b \\ \text{“}\geq\text{”} \sqcup (w' \circ \mathcal{C}^*[\![c]\!])\text{“}\geq\text{”} & \text{otherwise} \end{cases}$$

$$G(w')(\top) = (w' \circ \mathcal{C}^*[\![c]\!])(m_1) \sqcup m_2$$

$$\text{where } m_1 = \begin{cases} \bot & \text{if } b = \mathit{false} \\ \text{“}<\text{”} & \text{otherwise, if } b \Rightarrow x < 0 \\ \text{“}\geq\text{”} & \text{otherwise, if } b \Rightarrow x \geq 0 \\ \top & \text{otherwise} \end{cases} \quad \text{and } m_2 = \begin{cases} \bot & \text{if } b = \mathit{true} \\ \text{“}<\text{”} & \text{otherwise, if } \neg b \Rightarrow x < 0 \\ \text{“}\geq\text{”} & \text{otherwise, if } \neg b \Rightarrow x \geq 0 \\ \top & \text{otherwise} \end{cases}$$

Similarly to **if**, these cases can be combined:

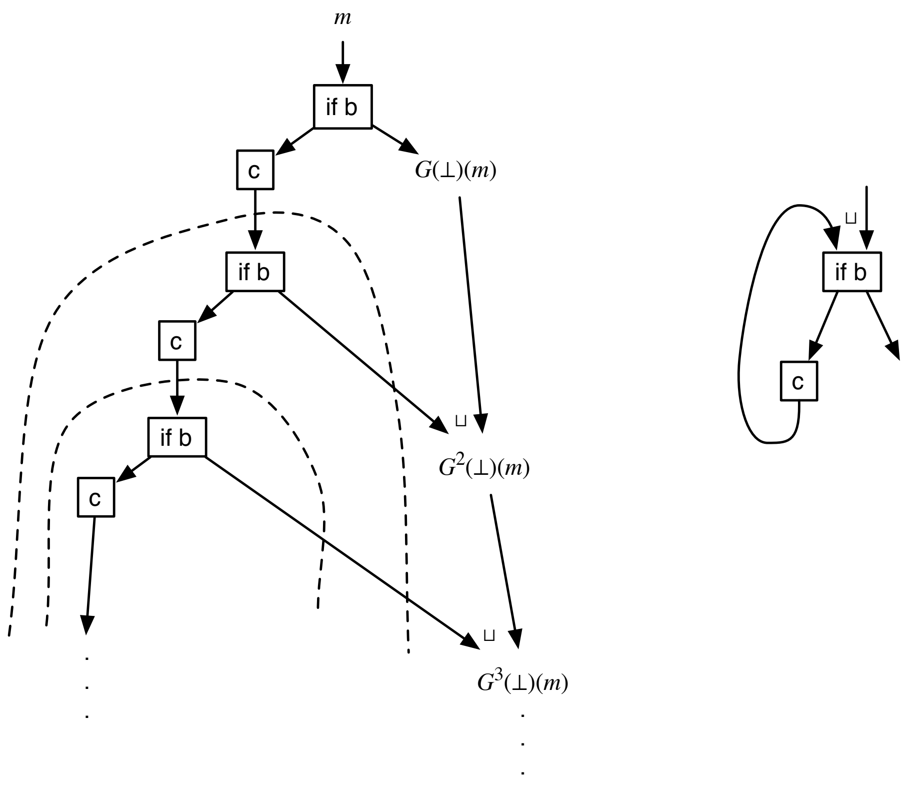

Figure 3: Left: unrolling a while loop for analysis. Right: a control-flow graph for a while loop

$$G(w')(m) = (w' \circ \mathcal{C}^*[\![c]\!])(m_1) \sqcup m_2$$

$$\text{where} \quad m_1 = \begin{cases} \bot & \text{if } \sigma \models m \Rightarrow \sigma \models \neg b \\ \text{“}<\text{”} & \text{otherwise, if } \sigma \models m \wedge \sigma \models b \Rightarrow \sigma \models x < 0 \\ \text{“}\geq\text{”} & \text{otherwise, if } \sigma \models m \wedge \sigma \models b \Rightarrow \sigma \models x \geq 0 \\ m & \text{in all other cases} \end{cases}$$

$$\text{and} \quad m_2 = \begin{cases} \bot & \text{if } \sigma \models m \Rightarrow \sigma \models b \\ \text{“}<\text{”} & \text{otherwise, if } \sigma \models m \wedge \sigma \models \neg b \Rightarrow \sigma \models x < 0 \\ \text{“}\geq\text{”} & \text{otherwise, if } \sigma \models m \wedge \sigma \models \neg b \Rightarrow \sigma \models x \geq 0 \\ m & \text{in all other cases} \end{cases}$$

We can then find an overapproximation to (fix $F)^*$ as an upper bound of this sequence

$$G(\bot)(m), G^2(\bot)(m), G^3(\bot)(m), \ldots$$

To ensure that this sequence converges, we can in general use the widening operator instead of the join operator:

$$G(\bot)(m) \;\nabla\; G^2(\bot)(m) \;\nabla\; G^3(\bot)(m) \;\nabla\; \ldots$$

This widening must eventually stabilize, at the point where adding one more term does not increase the result of the widening.

To see what is happening intuitively, and to connect this to ordinary dataflow analysis that some readers may be familiar with, the value $G^n(\bot)(m)$ is the result of analyzing an n-fold unrolling of the original loop, as depicted on

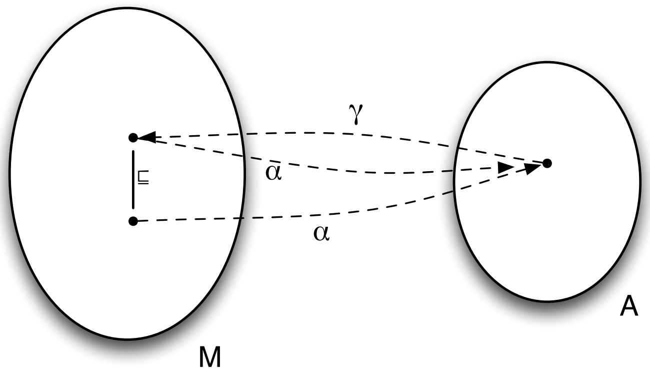

Figure 4: A Galois connection

the left side of Figure 3. The dotted contours represent the points at which various n-fold unrollings stop, replacing the result of the subtree below with $\bot$.

Compare this to the result of a conventional dataflow analysis performed over a control flow graph, as depicted on the right side of the figure. Here, the arrow leaving the node for c loops back to the **if** node. An analysis run on this graph will overapproximate the result of the tree on the left, because it superimposes the result of all loop iterations.

Thus, we can think of the abstract interpretation as a more precise version of the conventional dataflow analysis.

2.4 Galois connections

Suppose we have an analysis that gives a value from some lattice M to each program, but we want a result from a coarser lattice A. We can map an analysis over M into an analysis over A via a *Galois connection* between the two lattices.

A Galois connection between M and A is defined by a pair of monotonic functions:

$$\alpha : M \to A$$
$$\gamma : A \to M$$

such that for all $m \in M$, $m \sqsubseteq \gamma(\alpha(m))$ and for all $a \in A$, $\alpha(\gamma(a)) \sqsubseteq a$. Equivalently, $\mathit{id}_M \sqsubseteq \gamma \circ \alpha$ and $\alpha \circ \gamma \sqsubseteq \mathit{id}_A$. This relationship between M and A is depicted graphically in Figure 4.

Think of function α as producing the abstract version of its argument, and function γ as mapping an abstract element back to its most precise concrete representative.

Note that because the two functions are monotonic, $\alpha(\gamma(\alpha(m)) = m$. Therefore, any element a in A such that $\alpha(\gamma(a)) \neq a$ cannot be the image of any concrete element m. Removing all such "useless" elements from A leaves us with a *Galois insertion* meeting the stronger requirement that $\alpha(\gamma(a)) = a$.[1]

Suppose we have an abstract interpretation for M, with corresponding monadic operations $[\cdot]$ and $(\cdot)^*$, and a Galois insertion (α, γ) from M to A. Then we can define a coarser abstract interpretation for A via monadic operations $[\cdot]'$ and $(\cdot)^{*'}$:

$$[\sigma]' = \alpha([\sigma]) \qquad \text{that is, } [\cdot]' = \alpha \circ [\cdot]$$
$$(f)^{*'}(a) = \alpha(f^*(\gamma(a))) \qquad \text{that is, } (f)^{*'} = \alpha \circ f^* \circ \gamma$$

For example, we might choose as our original abstract interpretation the nondeterministic semantics for IMP (see the lecture 24 notes), which looks exactly like an abstract interpretation in which $M = \wp(\Sigma_\bot)$. This abstract interpretation is not very "abstract" since it coincides with the deterministic semantics, producing singleton sets for

[1]In this case (γ, α) also form an e–p pair for the lattices dual to A and M.

all program executions. However, the powerset domain has the necessary lattice structure so that any other abstract interpretation to some lattice A can be defined via a Galois insertion from $\wp(\Sigma_\perp)$ to A.

Another reason that we might want to define a Galois connection is as a principled way to obtain a widening operator. Suppose that the lattice M over which we are doing interpretation does not have finite height. We can obtain a widening operator in which chains are guaranteed to stabilize by performing a join in an more abstract lattice A that *does* have finite height:

$$m_1 \ \nabla \ m_2 \quad = \quad \gamma(\alpha(m_1) \sqcup \alpha(m_2))$$

In this way, the imprecision of using A to represent computation is incurred only when the widening operator is used, rather than throughout the abstract interpretation.