Monotone Frameworks Project Documentation

Samuel Klumpers (6057314) Philipp Zander (7983034)

June 1, 2022

Contents

1 Readme

Compile the project by executing `stack build`. Run all implemented analyses on `examples/cp1.c` by executing `stack run -- cp1`. Uncomment lines 123 to 134 in `Compiler` for the LaTeX encoded table representation of analysis results.

The purpose of each module is as follows.

`AttributeGrammar.ag` provided attribute grammar extended by attributes for administrative information like init, the finals, the flow, etc.

`Analyses` helper functions imported by `AttributeGrammar.ag`

`ConstantProp.ag` attribute for the transfer functions for the constant propagation analysis

`ConstantProp` helper functions imported by `ConstantProp.ag`

`ConstantBranch.ag` attribute for the transfer functions for the branch aware constant propagation analysis

`ConstantBranch` helper functions imported by `ConstantBranch.ag`

`StrongLive.ag` attribute for the transfer functions for the branch aware constant propagation analysis

`StrongLive` helper functions imported by `StrongLive.ag`

`MonotoneFrameworks` monotone framework solver

`AnalysesConversion` convert administrative AST information and analysis information into a monotone framework instances to be passed to the solver

`ContextSensitive` make a monotone framework instance context sensitive

`Latex` LaTeX encoded table representation of analysis results

`Std` custom prelude

`Compiler` print analysis results

2 Design of the monotone framework solver

We will choose terms assuming forward propagation. But the code is of course oblivious to the propagation direction. The function `mfpSolution` in `MonotoneFrameworks` implements the algorithm for solving data flow equations[NNH05, table 2.8, p. 75].

We adjusted it to support binary transfer functions for returning from procedures as described in [NNH05, section 2.5.3]. As in [NNH05, table 2.8, p. 75], the work list is extended with the outgoing flow from the updated label in every iteration. Additionally, we extend the work list, if the updated label is a call label, with the outgoing flow from the corresponding return label. That is because the return labels transfer function is binary and one argument is the updated call label's property. Nielson, Nielson, and Hankin [NNH05] fail to mention that.

We define a type class `BoundedSemiLattice` in the same module to specify the operations for the property space required by `mfpSolution`. Requiring the *ascending chain condition* seemed to specific for this type class. So `mfpSolution` requires it itself.

2.1 Context Sensitive

`contextSensitize` in `ContextSensitive` turns a monotone framework instance $(L, \mathcal{F}, F, E, \iota, f)$ into a context sensitive instance $(\widehat{L}, \widehat{\mathcal{F}}, F, E, \widehat{\iota}, \widehat{f})$ given a call string limit k according to the equations in [NNH05, section 2.5.4, subsection *Call strings of bounded length*].

But an additional adjustment to the monotone framework instance is required first for the following reason. If the context sensitive property space is defined by `type ContextSensitive` L `= [Label] ->` L, there is no easy way to implement $\sqsubseteq$ without applying the function to a large number of `[Label]` call strings. So we would like to define `type ContextSensitive` L `= Map [Label]` L instead. But now there is no easy way to lift a transfer function defined on L to `ContextSensitive` L unless there is a $\bot_{\text{preserved}} \in L$ such that $f(\bot_{\text{preserved}}) = \bot_{\text{preserved}}$ for all transfer functions f. Then absence from the `Map` can be interpreted as this $\bot_{\text{preserved}}$ and transfer functions can be lifted to `ContextSensitive` L via `Map`'s `Functor` instance. To lift binary transfer functions we require $f(\bot_{\text{preserved}}, l) = \bot_{\text{preserved}}$ for all $l \in L$ and transfer functions.

Given a monotone framework instance $I = (L, \mathcal{F}, F, E, \iota, f)$, we extend it to $I' = (L \cup \{\bot_{\text{new}}\}, \mathcal{F}', F, E, \iota, f')$ where $\bot_{\text{new}}$ is the new bottom of L and functions g from $\mathcal{F}$ and f are extended as follows to yield $\mathcal{F}'$ and f'.

$$
\begin{aligned}
g(\bot_{\text{new}}) &\coloneqq \bot_{\text{new}} && \text{for unary functions} \\
g_{\ell_c,\ell_r}(\bot_{\text{new}}, x) &\coloneqq \bot_{\text{new}} && \text{for all } x \in L \cup \{\bot_{\text{new}}\} \text{ for binary functions} \\
g_{\ell_c,\ell_r}(x, \bot_{\text{new}}) &\coloneqq g_{\ell_c,\ell_r}(x, \bot_L) && \text{for all } x \in L \text{ for binary functions}
\end{aligned}
$$

Now define $\widehat{I'} = (\widehat{L'}, \widehat{\mathcal{F}'}, F, E, \widehat{\iota'}, \widehat{f'})$ to be I' made context sensitive as described before and $\widehat{I} = (\widehat{L}, \widehat{\mathcal{F}}, F, E, \widehat{\iota}, \widehat{f})$ to be I made context sensitive as described before.

If a transfer function's inputs are not $\bot_{\text{new}}$, their result is not either. ι does not contain any $\bot_{\text{new}}$. So $\widehat{\iota'}$ does not assign $\bot_{\text{new}}$ to any combinations of extremal labels and the empty call string. Therefore, a solution to $\widehat{I'}$ cannot assign $\bot_{\text{new}}$ to any reachable combinations of labels and call strings. Therefore, every solution to $\widehat{I'}$ is a solution to $\widehat{I}$ after $\bot_{\text{new}}$s were replaced by arbitrary elements of L. Arbitrary elements of L can be chosen freely because $\bot_{\text{new}}$ only occurs for unreachable combinations of labels and call strings, that is places unconstrained any data flow equations.

On the other hand, every solution to $\widehat{I}$ is also clearly a solution to $\widehat{I'}$. Therefore, a least solution to $\widehat{I'}$ is a least solution to $\widehat{I}$ after $\bot_{\text{new}}$s were replaced by $\bot_L$ as the following lemma demonstrates in more detail.

Note that we can describe a solution as a map $P \to L$ from all possible program points and contexts to the lattice.

Lemma 1. *If v is the least solution to $\widehat{I'}$, then $p \circ v$ is the least solution to $\widehat{I}$, where p send $\bot_{\text{new}} \to \bot_L$, and acts as the identity otherwise.*

Proof. Suppose that $v : P \to L \cup \{\bot_{\text{new}}\}$ is the least solution to $\widehat{I'}$. Then we can partition $P = B \cup B'$ such that $v(b) = \bot_{\text{new}}$ exactly when $b \in B'$. That is, B' is the set of unreachable points. We claim that $p \circ v$ then is the least solution to $\widehat{I}$. Suppose not, and there is a smaller solution $w \sqsubseteq p \circ v$. We see that $p(v(b)) = \bot_L$ when $b \in B'$, thus $w(b) = \bot_L$ for all $b \in B'$. Remark that $p(v(b)) = v(b)$ when $b \in B$. It follows that restricting to B, it holds that $w|_B < p \circ v|_B = v|_B$. Furthermore, we observe that $v|_B$ and $v|_{B'}$ do not depend on each other, as B' is unreachable, hence there are no equations relating values of v at B to values of v at B'. This lets us define $v'|_B = w|_B$ to get a solution $v' : P \to L \cup \{\bot_{\text{new}}\}$ for $\widehat{I'}$. But then we see that $v' < v$, which contradicts the assumption that v is the least solution to $\widehat{I'}$, so we conclude that $p \circ v$ is the least solution to $\widehat{I}$. □

With this described adjustment of an artificial new preserved bottom it is not necessary to instantiate the work list to all flow edges anymore. Instantiating

the work list to the outgoing flow from the extremal labels would suffice. We decided against this optimization to leave no doubt that our implementation is correct.

3 Constant propagation

The lattice is

$$L = \{\bot\} \cup (\text{Vars} \to \{\top\} \cup \mathbb{Z})$$

where the join is defined by the equations

$$\begin{aligned} \sigma \sqcup \bot &= \bot \\ \bot \sqcup \sigma &= \bot \\ \sigma(x) = \tau(x) &\implies (\sigma \sqcup \tau)(x) = \sigma(x) \\ \sigma(x) \neq \tau(x) &\implies (\sigma \sqcup \tau)(x) = \top \end{aligned}$$

Here Vars is the (finite) set of all variable names in the program[1]. Remark that if $\sigma(x) = \top$ or $\tau(x) = \top$ then $(\sigma \sqcup \tau)(x) = \top$.

The transfer functions are given as

$$\begin{aligned} f_\ell(\sigma) &\coloneqq \sigma && \text{for all } [\texttt{skip}]^\ell \in \text{blocks}(S_\star) \\ f_\ell(\sigma) &\coloneqq \sigma && \text{for all } [b]^\ell \in \text{blocks}(S_\star) \\ f_\ell(\sigma) &\coloneqq \begin{cases} \bot & \text{if } \sigma = \bot \\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] & \text{otherwise} \end{cases} && \text{for all } [x \coloneqq a]^\ell \in \text{blocks}(S_\star) \end{aligned}$$

and for all $[\texttt{call } p(a_0, \ldots, a_n, z)]^{l_c}_{l_r}$ and $\texttt{proc } p(\texttt{val } x_0, \ldots, x_n, \texttt{res y})$

$$\begin{aligned} f_{\ell_c}(\sigma) &\coloneqq \sigma[x_0 \mapsto \mathcal{A}[\![a_0]\!]\sigma, \ldots, x_n \mapsto \mathcal{A}[\![a_n]\!]\sigma, y \mapsto \top] \\ f_{\ell_c,\ell_r}(\sigma_0, \sigma_1) &\coloneqq \sigma_1[x_0 \mapsto \sigma_0(x_0), \ldots, x_n \mapsto \sigma_0(x_n), y \mapsto \sigma_0(y), z \mapsto \sigma_1(y)] \end{aligned}$$

where

$$\begin{aligned} \mathcal{A}[\![n]\!]\sigma &\coloneqq n \\ \mathcal{A}[\![x]\!]\sigma &\coloneqq \sigma(x) \\ \mathcal{A}[\![a_0 \,\text{op}_a\, a_1]\!]\sigma &\coloneqq \begin{cases} \mathcal{A}[\![a_0]\!]\sigma \,\text{op}_a\, \mathcal{A}[\![a_1]\!]\sigma & \text{if } A[\![a_0]\!]\sigma \neq \bot \text{ and } A[\![a_1]\!]\sigma \neq \bot \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

[1] In our implementation the target set is $\mathbb{Z} \cup \text{Bool}$, but this complicates the description without yielding insight.

3.1 Example

Let us motivate the correctness of the analysis by an example. Consider the following program

```
begin
  proc id(val x, out y) is1
    [y := x]2
  end3;
  [call id(x, y)]4−5;
  [call id(0, z)]6−7;
  if [z != 0]8 then
    [w := z + w]9
  else
    [w := 0]10;
  print(w)11
end
```

`const-poison.c` (pretty-printed)

The output of the MFP solver is as follows, note that absence of a variable in an entry is equivalent to the variable being non-constant.

Entry	[]	[4]	[6]
1			$x \mapsto 0$
2			$x \mapsto 0$
3			$x \mapsto 0, y \mapsto 0$
4			
5			$x \mapsto 0, y \mapsto 0$
6			
7			$x \mapsto 0, y \mapsto 0$
8	$z \mapsto 0$		
9	$z \mapsto 0$		
10	$z \mapsto 0$		
11	$z \mapsto 0$		

Exit	[]	[4]	[6]
1			$x \mapsto 0$
2			$x \mapsto 0, y \mapsto 0$
3			$x \mapsto 0, y \mapsto 0$
4			
5			
6			$x \mapsto 0$
7	$z \mapsto 0$		
8	$z \mapsto 0$		
9	$z \mapsto 0$		
10	$w \mapsto 0, z \mapsto 0$		
11	$z \mapsto 0$		

The initial label of the program is 4, so we walk through the output from 4 in context []. We see that `x` is not constant exiting 4, hence is also not constant entering 1 in [4]. Logically `y` also becomes non-constant at 2 and 3, and so does

the return value `y` when returning to 5 in []. Then `x` and `y` remain untouched and non-constant for the rest of the program.

In the call at 6 in [], the value of the first parameter is 0, so we see that `x` is constantly 0 at 1 in [6]. This makes `y` constantly 0 at 2 in [6], so that `z` also becomes 0 when returning to 7 in [], which it remains for the rest of the program.

We see that at `w` is not constant at 9, hence the assignment `w := z + w` does not fix a constant value for `w`. Evidently, `w` is constantly 0 at 10 in the `else` branch.

Remark that at 11, `w` is not constantly 0 at 11, despite `z != 0` being constantly `false`, making the `then` branch unreachable. This motivates us to extend the standard constant propagation analysis to a (more) branch aware analysis in the next section.

3.1.1 Poisoning

Note that this example also exhibits poisoning. Consider the exit values when the call strings are limited to length 0,

Exit	[]
1	
2	
3	
4	
5	
6	$x \mapsto 0$
7	
8	
9	
10	$w \mapsto 0$
11	

Like before `x` is not constant at 4, so it is not constant at 1 either, but now in [] since the call strings are truncated to length 0. Similarly `y` at 2 and 5 become non-constant. However, in the call at 6, the value of the parameter `x` is 0, while `x` is already non-constant at 1. The new value at 1 is computed by joining $\emptyset \wedge (x \mapsto 0) = \emptyset$. It follows that `x` and `y` remain non-constant. As a result, `z` also becomes non-constant at 7, unlike the analysis with sufficiently large call strings.

4 Branch aware constant propagation

The lattice is

$$L = (\{\bot\} \sqcup (\text{Vars} \to \{\top\} \cup \mathbb{Z})) \times \mathcal{P}(\text{Labels}),$$

where the join is defined by

$$(\sigma, d) \sqcup (\tau, d') = (\sigma \sqcup \tau, d \cap d')$$

and $\sigma \sqcup \tau$ is the join of the normal constant propagation lattice.

Here Labels is the (finite) set of all labels in the program. Occurrences of labels in such a subset $d \in \mathcal{P}(\text{Labels})$ indicate that they are constantly unreachable.

$$f_\ell(\sigma, d) \coloneqq \text{die}^\ell(\sigma, d) \qquad \text{for all } [\texttt{skip}]^\ell \in \text{blocks}(S_\star)$$

$$f_\ell(\sigma, d) \coloneqq \begin{cases} (\bot, d) & \text{if } \sigma = \bot \\ (\sigma, d \cup \{f\}) & \text{if } \mathcal{A}[\![b]\!]\sigma = \text{True} \\ (\sigma, d \cup \{t\}) & \text{if } \mathcal{A}[\![b]\!]\sigma = \text{False} \\ (\sigma, d) & \text{otherwise} \end{cases} \qquad \text{for all } \texttt{if } [b]^\ell \texttt{ then } [\texttt{t}]^t \texttt{ else } [\texttt{f}]^f$$

$$f_\ell(\sigma, d) \coloneqq \begin{cases} (\bot, d) & \text{if } \sigma = \bot \\ (\sigma, d \cup \{s\}) & \text{if } \mathcal{A}[\![b]\!]\sigma = \text{False} \\ (\sigma, d) & \text{otherwise} \end{cases} \qquad \text{for all } \texttt{while } [b]^\ell \texttt{ do } [\texttt{s}]^s \texttt{ end}$$

$$f_\ell(\sigma, d) \coloneqq \text{die}^\ell\left(f \begin{cases} \bot & \text{if } \sigma = \bot \\ \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma] & \text{otherwise} \end{cases}, d\right) \qquad \text{for all } [x \coloneqq a]^\ell \in \text{blocks}(S_\star)$$

and for all $[\texttt{call } p\texttt{(}a_0\texttt{,...,} \ a_n\texttt{,} \ z\texttt{)}]^{l_c}_{l_r}$ and $\texttt{proc } p\texttt{(val } x_0\texttt{,...,} \ x_n\texttt{, res y)}$

$$f_{\ell_c}(\sigma, d) \coloneqq \text{die}^{\ell_c}(\sigma[x_0 \mapsto \mathcal{A}[\![a_0]\!]\sigma, \ldots, x_n \mapsto \mathcal{A}[\![a_n]\!]\sigma, y \mapsto \top], d)$$
$$f_{\ell_c,\ell_r}((\sigma_0, d_0), (\sigma_1, d_1)) \coloneqq \sigma_1[x_0 \mapsto \sigma_0(x_0), \ldots, x_n \mapsto \sigma_0(x_n), y \mapsto \sigma_0(y), z \mapsto \sigma_1(y)]$$

where

$$\text{die}^\ell(\sigma, d) \coloneqq \begin{cases} (\bot, d) & \text{if } \ell \in d \\ (\sigma, d) & \text{otherwise} \end{cases}$$

Note that, like in standard constant propagation, $\bot$ is preserved. Here, this also ensures that if a label becomes unreachable, then the analysis will remain $(\bot, d)$ for all successors, until some successor becomes reachable from elsewhere.

4.1 Example

Let us walk through an example analysis.

```
begin
  [x := 0 * x]1;
  if [x == 0]2 then
    [x := 1]3
  else
    [x := 0]4;
  print(x)5;
  while [x == 1]6 do
    [x := x]7;
  print(x)8
end
```

`const-branch-sample.c` (pretty-printed)

The output of the above program is as follows.

Entry	[]
1	$;\emptyset$
2	$x \mapsto 0; \emptyset$
3	$x \mapsto 0; \{4\}$
4	$x \mapsto 0; \{4\}$
5	$x \mapsto 1; \{4\}$
6	$x \mapsto 1; \{4, 8\}$
7	$x \mapsto 1; \{4, 8\}$
8	$x \mapsto 1; \{4, 8\}$

Exit	[]
1	$x \mapsto 0; \emptyset$
2	$x \mapsto 0; \{4\}$
3	$x \mapsto 1; \{4\}$
4	$\bot; \{4\}$
5	$x \mapsto 1; \{4\}$
6	$x \mapsto 1; \{4, 8\}$
7	$x \mapsto 1; \{4, 8\}$
8	$\bot; \{4, 8\}$

At the exit of 1, x becomes constantly 0, since $0 \cdot x = 0$ for all x. We see that the condition `x == 0` is then always `true`, so the analysis marks 4, the label of the `else` case, as unreachable. As normally, x becomes constantly 1 at 3. However, since 4 is now unreachable, the value of x (and any other variable if it existed) exiting 4 becomes irrelevant, so the analysis returns $\bot$. Since x is now constantly 1 exiting 3 and irrelevant exiting 4, we enter 6 with x still constantly 1. This makes the condition `x == 1` of the while loop `true`, hence 7 becomes reachable. At 7, x remains 1, so when we flow back to 6 is is also constantly 1. It follows that `x == 1` is always `true`, thus 8 is never reached. Consequently, 8 is marked with $\bot$.

5 Strongly live variables

The lattice for strongly live variables is

$$L = \mathcal{P}(\text{Vars})$$

with join $u \sqcup v = u \cup v$.

The transfer functions are

$$\begin{aligned}
f_\ell(\sigma) &\coloneqq \sigma && \text{for all } [\texttt{skip}]^\ell \in \text{blocks}(S_\star)\\
f_\ell(\sigma) &\coloneqq \sigma \cup \text{FV}(b) && \text{for all } [b]^\ell \in \text{blocks}(S_\star)\\
f_\ell(\sigma) &\coloneqq (\sigma - \{x\}) \cup \text{keep}_x(\text{FV(a)}, \sigma) && \text{for all } [x \coloneqq a]^\ell \in \text{blocks}(S_\star)
\end{aligned}$$

and for all $[\texttt{call } p(a_0, \ldots, a_n, z)]^{l_c}_{l_r}$ and $\texttt{proc } p(\texttt{val } x_0, \ldots, x_n, \texttt{res y})$:

$$\begin{aligned}
f_{\ell_r}(\sigma) &\coloneqq \begin{cases} \sigma \cup \{y\} - \{z\} - \vec{x} & \text{if } z \in \sigma \\ \sigma - \{z\} - \vec{x} & \text{otherwise} \end{cases}\\
f_{\ell_c,\ell_r}(\sigma_0, \sigma_1) &\coloneqq (\sigma_0 - \{z\}) \cup (\sigma_1 - \vec{x}) \cup \bigcup_{x \in \vec{x}} \text{keep}_x(\text{FV}(a), \sigma_1),
\end{aligned}$$

where

$$\begin{aligned}
\vec{x} &\coloneqq \{x_0, \ldots, x_n\}\\
\text{keep}_x(\tau, \sigma) &\coloneqq \begin{cases} \tau & \text{if } x \in \sigma \\ \emptyset & \text{otherwise} \end{cases}\\
\text{FV}(n) &\coloneqq \emptyset\\
\text{FV}(x) &\coloneqq x\\
\text{FV}(a_0 \,\text{op}_a\, a_1) &\coloneqq \text{FV}(a_0) \cup \text{FV}(a_1).
\end{aligned}$$

and move is extended to sets by repeated application.

5.1 Example

Let us walk through an example analysis. Consider the program

```
begin
  [x := z]1;
  while [x > 0]2 do
    skip3;
  [z := y * z]4;
  [y := y - w]5;
  print(z)6
end
```

`live-yes.c` (pretty-printed)

and its analysis

Entry	[]
1	$\{x, y, z\}$
2	$\{x, y, z\}$
3	$\{x, y, z\}$
4	$\{z\}$
5	$\{z\}$
6	$\emptyset$

Exit	[]
1	$\{y, z\}$
2	$\{x, y, z\}$
3	$\{x, y, z\}$
4	$\{y, z\}$
5	$\{z\}$
6	$\{z\}$

As this is a backwards analysis, so we read the analysis in reverse. Note that this also flips the entry and exit tables. Printing `z` at 6 must clearly make it (strongly) live. Since `y` was not live when entering 5, `w` does not become live due to the assignment. On the other hand, as `z` is live when entering 4, `y` also becomes live because of the assignment. In this assignment the LHS `z` is not killed as it also occurs in the RHS. Because `x` occurs in the condition of a statement which controls flow, it is also made live exiting 2. Note that this exit then flows back to 3. Finally, assigning `z` to `x` at 1 kills `x` at the exit.

6 Extensions

We made the following extensions to the required implementation:

- Embellished instance for the strongly live variables analysis
- Add the branch aware constant propagation analysis of *our own design*
- Make some operations create constants even if one operand is not constant (e.g. `&& false`)

- Implement `print` statement (and alter lexer and parser)
- Implement `break` and `continue` language constructs
- Hack lexer and parser to handle boolean assignments.

References

[NNH05] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Berlin, Heidelberg: Springer-Verlag, 2005. ISBN: 3540654100.