Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rule

Technical Report UW-CSE-2004-07-04

Sorin Lerner
Univ. of Washington
lerns@cs.washington.edu

Todd Millstein
UCLA
todd@cs.ucla.edu

Erika Rice
Univ. of Washington
erice@cs.washington.edu

Craig Chambers
Univ. of Washington
chambers@cs.washington.edu

This document contains a formal description of the Rhodium language. Section 1 presents the Rhodium syntax. Section 2 presents our composing framework, which is used to formalize Rhodium analyses. Section 3 presents forward analyses and transformations, whereas section 4 presents backward analyses and transformations. Sections 5 and 6 present the frameworks for flow-insensitive and interprocedural analyses, respectively. Finally, appendix A is a transcript of all of our Rhodium code.

Chapter 1

Rhodium Syntax

In the following syntax, we use $(E)*$ to represent zero or more repetitions of E, E/SEP to represent zero or more repetitions of E separated by SEP, and $\{E\}$ to represent zero or one repetition of E. Non-terminals are in written in *italics* font, and terminals are written either in **boldface** or `typewriter` font.

The syntax for a Rhodium program is as follows:

$$
\begin{array}{lll}
\textit{RhodiumProg} & ::= & (\textit{Decl})* \\
\\
\textit{Decl} & ::= & \textit{VarDecls} \\
 & \mid & \textit{EdgeFactDecl} \\
 & \mid & \textit{VirtualEdgeFactDecl} \\
 & \mid & \textit{NodeFactDecl} \\
 & \mid & \textit{PropagateRule} \\
 & \mid & \textit{TransformRule} \\
\\
\textit{VarDecls} & ::= & \textbf{decl}\ \textit{VarDecl}/,\ \textbf{in}\ \textit{Decl}\ \textbf{end} \\
\\
\textit{EdgeFactDecl} & ::= & \textbf{define edge fact}\ id(\textit{VarDecl}/,)\ \textbf{with meaning}\ \textit{Meaning} \\
\\
\textit{VirtualEdgeFactDecl} & ::= & \textbf{define virtual edge fact}\ id(\textit{VarDecl}/,)@id = \psi \\
\\
\textit{NodeFactDecl} & ::= & \textbf{define node fact}\ id(\textit{VarDecl}/,) = \psi \\
\\
\textit{VarDecl} & ::= & id : \textit{Type}
\end{array}
$$

```
Meaning                ::=  MeaningTerm == MeaningTerm
                         |  Meaning && Meaning
                         |  Meaning || Meaning
                         |  Meaning => Meaning
                         |  ! Meaning
                         |  forall VarDecl/, . Meaning
                         |  exists VarDecl/, . Meaning
                         |  MeaningPrimPredSymbol{(MeaningTerm/,)}

MeaningTerm            ::=  id
                         |  η
                         |  MeaningPrimFunSymbol{(MeaningTerm/,)}

MeaningPrimPredSymbol  ::=  isStmtNotStuck
                         |  isExprNotStuck
                         |  isLoc
                         |  isConst
                         |  equalUpTo

MeaningPrimFunSymbol   ::=  evalExpr
                         |  evalLocDeref
                         |  newConst
                         |  getConst
                         |  applyBinaryOp
                         |  min
                         |  max

PropagateRule          ::=  if ψ then EdgePred

TransformRule          ::=  if ψ then transform Stmt

ψ                      ::=  NodePred
                         |  EdgePred
                         |  Term == Term
                         |  case Term of (Term => ψ) * endcase
                         |  ψ && ψ
                         |  ψ || ψ
                         |  ψ => ψ
                         |  ! ψ
                         |  forall VarDecl/, . ψ
                         |  exists VarDecl/, . ψ
```

NodePred	::=	*id*(*Term*/,)
EdgePred	::=	*id*(*Term*/,)@*Edge*
Term	::=	*id*
	|	[*Expr*]
	|	[*Stmt*]
	|	`currStmt`
	|	`currNode`
	|	*PrimFunSymbol*(*Term*/,)
PrimFunSymbol	::=	`newConst`
	|	`getConst`
	|	`applyBinaryOp`
	|	`min`
	|	`max`
Edge	::=	`cfg_in` (syntactic sugar for `cfg_in[0]`)
	|	`cfg_out` (syntactic sugar for `cfg_out[0]`)
	|	`cfg_in`[*EdgeName*] (`cfg_in[`i`]` is the i^{th} cfg input edge)
	|	`cfg_out`[*EdgeName*] (`cfg_out[`i`]` is the i^{th} cfg input edge)
EdgeName	::=	*IntLiteral*
	|	`true` (syntactic sugar for 0, the true successor of a branch node)
	|	`false` (syntactic sugar for 1, the false successor of a branch node)
Stmt	::=	**decl** *VarExpr*
	|	**decl** *VarExpr*[*VarExpr*]
	|	**skip**
	|	*VarExpr* := **new**
	|	*VarExpr* := **newarray**[*VarExpr*]
	|	*LHS* := *Expr*
	|	*VarExpr* := *Proc*(*BaseExpr*)
	|	**if** *BaseExpr* **goto** *Label* **else** *Label*
	|	**return** *VarExpr*
	|	*RhodiumVar*
VarExpr	::=	*ILVar*
	|	*RhodiumVar*

$$
\begin{array}{lcll}
\mathit{Expr} & ::= & \mathit{BaseExpr} & \\
 & | & \texttt{*}\,\mathit{VarExpr} & \\
 & | & \texttt{\&}\,\mathit{VarExpr} & \\
 & | & \mathit{VarExpr}[\mathit{VarExpr}] & \\
 & | & \mathit{BaseExpr}\ \mathit{OP}\ \mathit{BaseExpr} & \\
 & | & \mathit{RhodiumVar} & \\
 & & & \\
\mathit{LHS} & ::= & \mathit{VarExpr} & \\
 & | & *\,\mathit{VarExpr} & \\
 & | & \mathit{RhodiumVar} & \\
 & & & \\
\mathit{BaseExpr} & ::= & \mathit{VarExpr} & \\
 & | & \mathit{Constant} & \\
 & | & \mathit{RhodiumVar} & \\
 & & & \\
\mathit{Constant} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots & \\
 & & & \\
\mathit{RhodiumVar} & ::= & \mathit{id} & \text{(Rhodium variable names)} \\
 & & & \\
\mathit{ILVar} & ::= & `\mathit{id} & \text{(IL variable names)} \\
 & & & \\
\mathit{Proc} & ::= & \mathit{id} & \text{(Procedure names)} \\
 & & & \\
\mathit{Label} & ::= & \mathit{id} & \text{(Node identifiers)} \\
 & & & \\
\mathit{OP} & ::= & + \mid - \mid * \mid / &
\end{array}
$$

We allow "**if** ψ **then** EF_1 `&&` ... `&&` EF_n" as sugar for:

> **if** ψ **then** EF_1
> ...
> **if** ψ **then** EF_n

For fresh variables `t1` and `t2`, we also allow the following sugar for array statements:

- `*(a[i]) := x` as short for

```
t1 := a[i];
*t1 := x;
```

- `x := *(a[i])` as short for

```
t1 := a[i];
x := *t1;
```

- `*((&a)[i]) := x` as short for

```
t1 := &a;
```

```
        t2 := t1[i];
        *t2 := x;
```

- `x := *((&a)[i])` as short for

```
        t1 := &a;
        t2 := t1[i];
        x := *t2;
```

- `x := (&a)[i]` as short for

```
        t1 := &a;
        x := t1[i];
```

Chapter 2

Composing Framework

This section outlines the composing framework that is used to formalize Rhodium analyses. The framework presented here is the one from [3], updated in two ways. First, we make function calls explicit in the semantics of the framework, so that we can reason about them when arguing the soundness of Rhodium analyses. Second, the condition for semantics preservation does not require non-termination to be preserved. As a result, if an instruction does not terminate (for example, if it runs forever or if it gets stuck), then the instruction can be replaced with any other instruction.

2.1 Preliminaries

In this section we define basic notation and the abstract intermediate representation that we assume throughout the rest of the paper. Section 2.3 reviews the well-know definition of a single analysis followed by transformations, and serves as a foundation for the formalization of the novel parts of our framework in sections 2.4 and 2.5.

2.1.1 Notation

If A is a set, then A^* is the set $\bigcup_{i\geq 0} A^i$, where $A^k = \{(a_1, \ldots, a_k) | a_i \in A\}$. We denote the i^{th} projection of a tuple $x = (x_1, \ldots, x_k)$ by $x[i] \triangleq x_i$. Given a function $f : A \to B$, we extend f to work over tuples by defining $\overrightarrow{f} : A^* \to B^*$ as $\overrightarrow{f}((x_1, \ldots, x_k)) \triangleq (f(x_1), \ldots, f(x_k))$. We also extend f to work over maps by defining $\widetilde{f} : (O \to A) \to (O \to B)$ as $\widetilde{f}(m) \triangleq \lambda o.f(m(o))$.

We extend a binary relation $R \subseteq 2^{D\times D}$ over D to tuples by defining the $\overrightarrow{R}$ relation by: $\overrightarrow{R}((x_1, \ldots, x_k), (y_1, \ldots, y_k))$ iff $R(x_1, y_1) \wedge \ldots \wedge R(x_k, y_k)$. Finally, we extend a binary relation $R \subseteq 2^{D\times D}$ to maps by defining the $\widetilde{R}$ relation as: $\widetilde{R}(m_1, m_2)$ iff for all elements o in the domain of both m_1 and m_2, it is the case that $R(m_1(o), m_2(o))$. To make the equations clearer, we drop the tilde and arrow annotations on binary relations when they are clear from context.

2.1.2 Intermediate Representation

A program is a tuple $\pi = (p_1, \ldots, p_n)$, where each p_i is a procedure. We denote by $Prog$ the set of all programs, and by $Proc$ the set of all procedures. We denote by $proc(name, formal, cfg)$ the procedure with name $name$, formal name $formal$, and control flow graph cfg. We denote by $name(p)$, $formal(p)$ and $cfg(p)$ the name, formal name, and CFG of procedure p. We assume that all the procedures in a program have distinct names.

A control flow graph is a tuple $g = (N, E, In, Out, InEdges, OutEdges)$ where $N \subseteq Nodes$ is a set of nodes (with $Nodes$ being a predefined infinite set), $E \subseteq Edges$ is a set of edges (with $Edges$ being a predefined infinite set), $In : N \to E^*$ specifies the input edges for a node, $Out : N \to E^*$ specifies the output edges for a node, $InEdges \in E^*$ specifies the input edges of the graph, and $OutEdges \in E^*$ specifies the output edges of the graph. A CFG has one input edge and out output edge. We let $Graph$ be the set of all control flow graphs. When necessary, we use subscripts to extract the components of a graph. For example, if g is a graph, then its nodes are N_g, its edges are E_g, and so on.

2.2 Semantics of the intermediate representation

Each node n has a statement associated with it, which we denote by $stmtAt(n)$. The statement forms are given in the section on the Rhodium syntax.

Given a node n for which $stmtAt(n) = (x := f(b))$ we assume that there is a procedure p in π for which $name(p) = f$ and we use $cfg(n)$ to denote $cfg(p)$.

The arity of an operator op is denoted $arity(op)$. We assume a fixed interpretation function for each n-ary operator symbol op: $[\![op]\!] : Const^n \to Const$.

We assume an infinite set $Location$ of memory locations, with metavariable l ranging over the set. We assume that the set $Const$ is disjoint from $Location$ and contains the distinguished elements $true$ and $uninit$. We denote by $Natural \subseteq Const$ the set of natural numbers. We also assume a set $Array$ of array values, disjoint from $Const$ and $Location$. An array value is a pair $(len, locs)$, where $len \in Natural$ is the length of the array and $locs \in (Natural \rightharpoonup Location)$ is a map from indices to locations. The locations mapped to are the locations of the array elements in the store. Given an array a, we use $len(a)$ to denotes its length, and $locs(a)$ to denote the addresses of its contents. Assuming $0 \leq i < len(a)$, the notation $a[i]$ denotes $locs(a)(i)$. We use $newInitArray(\langle l_0, \ldots, l_j \rangle)$ to denote a newly initialized array $(j + 1, \lambda i.l_i)$.

The set of values is defined as $Value = (Location \cup Const \cup Array)$

An *environment* is a partial function $\rho : Vars \rightharpoonup Location$; we denote by $Environment$ the set of all environments. A *store* is a partial function $\sigma : Location \rightharpoonup Value$; we denote by $Store$ the set of all stores. The domain of an environment ρ is denoted $dom(\rho)$, and similarly for the domain of a store. If $s = \langle x_1, \ldots x_n \rangle$, $s \in dom(\rho)$ denotes that each element of s is in $dom(\rho)$; similar notation is defined for a store σ. The notation $\rho[x \mapsto l]$ denotes the environment identical to ρ but with variable x mapping to location l; if $x \in dom(\rho)$, the old mapping for x is shadowed by the new one. The notation $\sigma[l \mapsto v]$ is define similarly. The notation $\sigma[l_1 \mapsto v_1, \ldots, l_n \mapsto v_n]$ denotes the store identical to σ but with each location l_i mapping to value v_i. If l_i and l_j are the same and $j > i$, then the mapping for l_j shadows the mapping for l_i. We use $\langle l_1, \ldots, l_n \rangle \mapsto v$ to stand for $l_1 \mapsto v, \ldots, l_n \mapsto v$. Finely, the notation $\sigma / \{l_1, \ldots, l_i\}$ denotes the store identical to σ except that all pairs $(l, v) \in \sigma$ such that $l \in \{l_1, \ldots, l_i\}$ are removed.

The current dynamic call chain is represented by a stack. A *stack frame* is a triple $f = (n, l, \rho) : Node \times$

Location $\times$ *Environment*. Here n is the CFG node that made the call to the function currently being executed, l is the location in which to put the return value from the call, and ρ is the current lexical environment at the point of the call. We denote by *Frame* the set of all stack frames. A *stack* $\xi = \langle f_1 \ldots f_n \rangle : \mathit{Frame}^*$ is a sequence of stack frames. The set of all stacks is denoted *Stack*. Stacks support two operations defined as follows:

$$\begin{array}{l} \mathit{push} : (\mathit{Frame} \times \mathit{Stack}) \rightarrow \mathit{Stack} \\ \mathit{push}(f, \langle f_1 \ldots f_n \rangle) = \langle f\ f_1 \ldots f_n \rangle \\ \\ \mathit{pop} : \mathit{Stack} \rightharpoonup (\mathit{Frame} \times \mathit{Stack}) \\ \mathit{pop}(\langle f_1\ f_2 \ldots f_n \rangle) = (f_1, \langle f_2 \ldots f_n \rangle), \text{where } n > 0 \end{array}$$

Finally, a memory allocator $\mathcal{M}$ is an infinite stream $\langle l_1, l_2, \ldots \rangle$ of locations. We denote the set of all memory allocators as *MemAlloc*.

A *state* of execution of a program π is a four-tuple $\eta = (\rho, \sigma, \xi, \mathcal{M})$ where $\rho \in \mathit{Environment}$, $\sigma \in \mathit{Store}$, $\xi \in \mathit{Stack}$, and $\mathcal{M} \in \mathit{MemAlloc}$. We denote the set of program states by *State*. We refer to the corresponding environment of a state η as $\mathit{env}(\eta)$, and we similarly define accessors *store*, *stack*, and *mem*.

Definition 1 *The evaluation of an expression e in a program state η, where $\mathit{env}(\eta) = \rho$ and $\mathit{store}(\eta) = \sigma$, is given by the partial function $\eta(e) : (\mathit{State} \times \mathit{Expr}) \rightharpoonup \mathit{Value}$ defined by:*

$$\begin{array}{rcl} \eta(c) & = & c \\ \eta(\&x) & = & \rho(x) \\ & & \textit{where } x \in \mathit{dom}(\rho) \\ \eta(x) & = & \sigma(\rho(x)) \\ & & \textit{where } x \in \mathit{dom}(\rho), \rho(x) \in \mathit{dom}(\sigma) \\ \eta(*x) & = & \sigma(\sigma(\rho(x))) \\ & & \textit{where } x \in \mathit{dom}(\rho), \rho(x) \in \mathit{dom}(\sigma), \sigma(\rho(x)) \in \mathit{dom}(\sigma) \\ \eta(op\ b_1 \ldots b_n) & = & [\![op]\!](\eta(b_1), \ldots, \eta(b_n)) \\ & & \textit{where } \mathit{arity}(op) = n \textit{ and } \forall\ 1 \leq j \leq n\ .\ (\eta(b_j) \in \mathit{Const}) \\ \eta(x[i]) & = & \eta(*x)[\eta(i)] \\ & & \textit{where } \eta(*x) \in \mathit{Array}, \eta(i) \in \mathit{Natural}, 0 \leq \eta(i) < \mathit{len}(\eta(*x)) \end{array}$$

Note that $\eta(e)$ is partial because of the side conditions in the above definition.

Each node n is the CFG has a set of input edges and a set of output edges. The vectors $\mathit{in}(n)$ and $\mathit{out}(n)$ refer to the incoming and outgoing edges of n. All statements have one incoming edge and one outgoing edge except for the following exceptions:

- If $\mathit{stmtAt}(n) =$ `merge` then $\mathit{len}(\mathit{in}(n)) = 2$ where $\mathit{in}(n)[0]$ and $\mathit{in}(n)[1]$ are the two inputs to the merge.
- If $\mathit{stmtAt}(n) = ($`if` b `goto` L_1 `else` $L_2)$ then $\mathit{len}(\mathit{out}(n)) = 2$ where $\mathit{out}(n)[0]$ is the false successor and $\mathit{out}(n)[1]$ is the true successor
- If $\mathit{stmtAt}(n) = (x := f(b))$ then $\mathit{len}(\mathit{out}(n)) = 2$, where $\mathit{out}(n)[0]$ is the intraprocedural edge from the call site n to the return site (the immediate successor of n in the intraprocedural CFG), and $\mathit{out}(n)[1]$ is the interprocedural edge from the call site n to the entry node of f.

- If $stmtAt(n) = (\texttt{return}\ x)$ then $len(out(n)) = k$ where k is the number of call sites to the function that n belongs to. We define $callSiteIndex : Node \rightharpoonup Natural$ so that $out(n)[callSiteIndex(n')]$ is the edge from the return statement n to the return site of call site n'.

Definition 2 *We use $i, \eta \xrightarrow{n} j, \eta'$ to say that program state η coming along the i^{th} input edge of n steps to η' on the j^{th} output edge of n. The state transition function $\cdot,\cdot \overset{\cdot}{\rightarrow} \cdot,\cdot \subseteq Natural \times State \times Node \times Natural \times State$ is defined by:*

- *If $stmtAt(n) = \texttt{decl}\ x$ then $0, (\rho, \sigma, \xi, \langle l, l_1, l_2, \ldots\rangle) \xrightarrow{n} 0, (\rho[x \mapsto l], \sigma[l \mapsto uninit], \xi, \langle l_1, l_2, \ldots\rangle)$ where $l \notin dom(\sigma)$*

- *If $stmtAt(n) = (\texttt{decl}\ x[i])$ then $0, (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots, \rangle) \xrightarrow{n} 0, (\rho[x \mapsto l], \sigma[l \mapsto newInitArray(s), s \mapsto uninit], \xi, \langle l_{\eta(i)}, l_{\eta(i)+1}, \ldots\rangle)$ where $\eta = (\rho, \sigma, \xi, \mathcal{M})$, $l \notin dom(\sigma)$, $s \notin dom(\sigma)$, $\eta(i) \in Natural$, $\eta(i) > 0$, $s = \langle l_0, \ldots, l_{\eta(i)-1}\rangle$*

- *If $stmtAt(n) = \texttt{skip}$ then $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma, \xi, \mathcal{M})$*

- *If $stmtAt(n) = \texttt{merge}$ then $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma, \xi, \mathcal{M})$ and $1, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma, \xi, \mathcal{M})$*

- *If $stmtAt(n) = (x := e)$ then $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma[\eta(\&x) \mapsto \eta(e)], \xi, \mathcal{M})$ where $\eta = (\rho, \sigma, \xi, \mathcal{M})$, $\eta(x) \in Location \cup Const$[1], $\eta(e) \in Location \cup Const$[2]*

- *If $stmtAt(n) = (*x := e)$ then $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma[\eta(x) \mapsto \eta(e)], \xi, \mathcal{M})$ where $\eta = (\rho, \sigma, \xi, \mathcal{M})$, $\eta(x) \in Location$, $\eta(*x) \in Location \cup Const$, $\eta(e) \in Location \cup Const$*

- *If $stmtAt(n) = (x := \texttt{new})$ then $0, (\rho, \sigma, \xi, \langle l, l_1, l_2, \ldots\rangle) \xrightarrow{n} 0, (\rho, \sigma[\eta(\&x) \mapsto l, l \mapsto uninit], \xi, \langle l_1, l_2, \ldots\rangle)$ where $\eta = (\rho, \sigma, \xi, \mathcal{M})$, $\eta(x) \in Location \cup Const$, $l \notin dom(\sigma)$*

- *If $stmtAt(n) = (x := \texttt{new array[i]})$ then $0, (\rho, \sigma, \xi, \langle l, l_0, l_1, \ldots\rangle) \xrightarrow{n}$ $0, (\rho, \sigma[\eta(\&x) \mapsto l, l \mapsto newInitArray(s), s \mapsto uninit], \xi, \langle l_{\eta(i)}, l_{\eta(i)+1}, \ldots\rangle)$ where $\eta = (\rho, \sigma, \xi, \mathcal{M})$, $l \notin dom(\sigma)$, $s \notin dom(\sigma)$, $\eta(x) \in Location \cup Const$, $\eta(i) \in Natural$, $\eta(i) > 0$, $s = \langle l_0, \ldots, l_{\eta(i)-1}\rangle$*

- *If $stmtAt(n) = (x := f(b))$ then $0, (\rho, \sigma, \xi, \langle l, l_1, l_2, \ldots\rangle) \xrightarrow{n} 1, (\{(y, l)\}, \sigma[l \mapsto \eta(b)], push(f, \xi), \langle l_1, l_2, \ldots\rangle)$ where $\eta = (\rho, \sigma, \xi, \langle l, l_1, l_2, \ldots\rangle)$, $y = formal_f$, $l \notin dom(\sigma)$, $x \in dom(\rho)$, $f = (n, \rho(x), \rho)$, $\eta(x) \in Location \cup Const$*

- *If $stmtAt(n) = (\texttt{if}\ b\ \texttt{goto}\ L_1\ \texttt{else}\ L_2)$ then $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 1, (\rho, \sigma, \xi, \mathcal{M})$ where $(\rho, \sigma, \xi, \mathcal{M})(b) = true$*

- *If $stmtAt(n) = (\texttt{if}\ b\ \texttt{goto}\ L_1\ \texttt{else}\ L_2)$ then $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} 0, (\rho, \sigma, \xi, \mathcal{M})$ where $(\rho, \sigma, \xi, \mathcal{M})(b) = false$*

- *If $stmtAt(n) = (\texttt{return}\ x)$ then $0, (\rho, \sigma, \xi, \mathcal{M}) \xrightarrow{n} j, (\rho_0, \sigma_0, \xi_0, \mathcal{M})$ where $pop(\xi) = ((n_0, l_0, \rho_0), \xi_0)$, $j = callSiteIndex(n_0)$, $dom(\rho) = \{x_1, \ldots, x_i\}$, $\sigma_0 = (\sigma / \{\rho(x_1), \ldots, \rho(x_i)\})[l_0 \mapsto (\rho, \sigma, \xi, \mathcal{M})(x)]$ where $(\rho, \sigma, \xi, \mathcal{M})(x) \in Location \cup Const$*

[1] This check, and the ones like it for other statement types, prevents assignment to array values.
[2] This check, and the ones like it, prevents array values from being copied.

Because the $\eta(e)$ function is partial, and because of the side conditions in the above definition, the $\cdot, \cdot \overset{\cdot}{\rightarrow} \cdot, \cdot$ function is partial, in that there are some i, η that cannot step. For example, if $x \notin \mathit{dom}(\rho)$ then $0, (\rho, \sigma, \xi, \mathcal{M})$ cannot step through the statement $x := e$.

A *machine configuration* is a pair $\delta = (e, \eta)$ where $e \in \mathit{Edge}$ and $\eta \in \mathit{State}$. Here e indicates where control has reached, and η represents the program state. We denote by *MachineConfig* be the set of all machine configurations. We use $\mathit{edge}(\delta)$ to denote the edge component of δ. We use $\mathit{src}(e)$ and $\mathit{dst}(e)$ to denote the source and destination nodes of an edge e. We use $\mathit{node}(\delta)$ to denote $\mathit{dst}(\mathit{edge}(\delta))$, which means that $\mathit{node}(\delta)$ is the node about to be executed by δ.

We define *inIndex* and *outIndex*:

$$\mathit{inIndex}(e) = \begin{cases} i & \text{if } \exists i. \mathit{in}(\mathit{dst}(e))[i] = e \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$$\mathit{outIndex}(e) = \begin{cases} i & \text{if } \exists i. \mathit{out}(\mathit{src}(e))[i] = e \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Definition 3 *The machine configuration transition function* $\rightarrow \;\subseteq \mathit{MachineConfig} \times \mathit{MachineConfig}$ *is defined by:*

$$(e, \eta) \rightarrow (e', \eta') \Leftrightarrow \left[\begin{array}{l} \mathit{inIndex}(e) \text{ is defined} \wedge \\ \mathit{outIndex}(e') \text{ is defined } \wedge \\ \mathit{inIndex}(e), \eta \overset{\mathit{dst}(e)}{\rightarrow} \mathit{outIndex}(e'), \eta' \end{array} \right]$$

The $\rightarrow^*$ *relation is the reflexive transitive closure of the* $\rightarrow$ *relation.*

Definition 4 *The intraprocedural state transition function* $\cdot, \cdot \overset{\cdot}{\hookrightarrow} \cdot, \cdot \;\subseteq \mathit{Natural} \times \mathit{State} \times \mathit{Node} \times \mathit{Natural} \times \mathit{State}$ *is defined by:*

- *If* $\mathit{stmtAt}(n)$ *is not a procedure call, then* $i, \eta \overset{n}{\hookrightarrow} j, \eta'$
  where $i, \eta \overset{n}{\rightarrow} j, \eta'$

- *If* $\mathit{stmtAt}(n) = (x := f(b))$ *then* $0, \eta \overset{n}{\hookrightarrow} 0, \eta'$
  where $0, \eta \overset{n}{\rightarrow} 1, \eta_p$ *and* $(\mathit{InEdges}_{\mathit{cfg}(n)}[0], \eta_p) \rightarrow^* (e, \eta')$ *and* η' *is the first state on the trace between* η_p *and* η' *such that* $\mathit{stack}(\eta') = \mathit{stack}(\eta)$

2.3 A Single Analysis Followed by Transformations

This section reviews the well-known lattice-theoretic formulation of dataflow analysis frameworks using abstract interpretation [1]. It shows how we use this formulation to define analyses and transformations over the IR defined in the previous section, and provides the foundation for describing our approach in sections 2.4 and 2.5.

2.3.1 Definition

An *analysis* is a tuple $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$ where $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ is a complete lattice, $\alpha : D_c \to D$ is the abstraction function, and $F : Node \times D^* \to D^*$ is the flow function for nodes. The elements of D, the domain of the analysis, are dataflow facts about edges in the IR (which would correspond to program points in a CFG representation). The flow function F provides the interpretation of nodes: given a node and a tuple of input dataflow values, one per incoming edge to the node, F produces a tuple of output dataflow values, one per outgoing edge from the node. D_c is the domain of a distinguished analysis, the concrete analysis $\mathcal{C} = (D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c, id, F_c)$, which specifies the concrete semantics of the program.

We use an intraprocedural collecting semantics (section 6 will define the interprocedural concrete semantics):

Definition 5 *The concrete analysis is:*

$$(D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c) = (2^{State}, \cup, \cap, \subseteq, State, \emptyset)$$

where the concrete flow function F_c is:

$$F_c(n, cs)[k] = \{\eta \mid \quad \exists\ \eta' \in State, i \in Natural\ .\ [\eta' \in cs[i]\ \wedge\ i, \eta' \overset{n}{\hookrightarrow} k, \eta]\} \tag{2.1}$$

Because of the predicate $\eta' \in cs[i]$ in equation (2.1), F_c is monotonic. The concrete flow function can also be expressed in the following equivalent formulation:

Definition 6 *The concrete flow function can be equivalently defined as:*

- *if $stmtAt(n)$ is not a procedure call, then $F_c(n, cs)[k] = \{\eta \mid \exists\ \eta' \in State, i \in Natural\ .\ [\eta' \in cs[i]\ \wedge\ i, \eta' \overset{n}{\hookrightarrow} k, \eta]\}$*
- *if $stmtAt(n) = (x := f(b))$ then $F_c(n, cs)[0] = \overrightarrow{S_{\mathcal{C}}(cfg(n), (X))}(OutEdges_{cfg(n)})$ where $X = \{\eta \mid \exists \eta' \in State\ .\ [\eta' \in cs[0] \wedge 0, \eta' \overset{n}{\to} 1, \eta\}$*

The solution of an analysis $\mathcal{A}$ over a domain D is provided by the function $S_{\mathcal{A}} : Graph \times D^* \to (Edges \to D)$. Given a graph g and a tuple of abstract values for the input edges of g, $S_{\mathcal{A}}$ returns the final abstract value for each edge in g. This is done by initializing all edges in g to bottom, and then applying the flow functions of $\mathcal{A}$ until a fixed point is reached. The definition of $S_{\mathcal{A}}$ is given in the following definition:[3]

Definition 7 *Given an analysis $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$, a graph g and a tuple of dataflow values $\iota \in D^*$ for the input edges of g, $S_{\mathcal{A}}(g, \iota)$ is defined as follows.*

First, we define the interpretation function $Int : E_g \times (E_g \to D) \to D$ as in Cousot and Cousot [1]: given an edge e and the current dataflow solution m, Int computes the dataflow value for e at the next iteration. Int is defined as:

$$Int(e, m) = \begin{cases} \iota[k] & \textit{if } \exists k.e = InEdges_g[k] \\ F(n, \overrightarrow{m}(In_g(n)))[k] & \textit{where } e = Out_g(n)[k] \end{cases}$$

[3]Although the concrete solution function $S_{\mathcal{C}}$ is usually not computable, the mathematical definition of $S_{\mathcal{C}}$ is still perfectly valid. Our framework does not evaluate $S_{\mathcal{C}}$; we only use $S_{\mathcal{C}}$ to formalize the soundness of analyses.

The global flow function $FG : (E_g \to D) \to (E_g \to D)$ takes a map representing the current dataflow solution, and computes the dataflow solution at the next iteration. FG is defined as:

$$FG(m) = \lambda e.Int(e, m)$$

The global ascending flow function FGA is the same as FG, except that it joins the result of the next iteration with the current solution before returning. This ensures that the solution monotonically increases as iteration proceeds, even if F is not monotonic. FGA is defined as:

$$FGA(m) = FG(m) \mathbin{\widetilde{\sqcup}} m$$

Finally, the result of $S_{\mathcal{A}}$ is a fixed point of FGA (the least fixed point if F is monotonic):

$$S_{\mathcal{A}}(g, \iota) = \bigsqcup_{n=0}^{\infty} FGA^n(\widetilde{\bot})$$

where $\widetilde{\bot} \triangleq \lambda e.\bot$, $FGA^0 = \lambda x.x$ *and* $FGA^k = FGA \circ FGA^{k-1}$ *for* $k > 0$.

An *Analysis followed by Transformations*, or an *AT-analysis* for short, is a pair $(\mathcal{A}, R)$ where $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$ is an analysis, and $R : Node \times D^* \to Graph \cup \{\epsilon\}$ is a *local replacement function.* The local replacement function R specifies how a node should be transformed after the analysis has been solved. Given a node n and a tuple of elements of D representing the final dataflow analysis solution for the input edges of n, R either returns a graph with which to replace n, or ϵ to indicate that no transformation should be applied to this node. To be syntactically valid, a replacement graph must have the same number of input and output edges as the node it replaces, and its nodes and edges must be unique (so that splicing a replacement graph into the enclosing graph does not cause conflicts). We also assume that replacement graphs do not contain function call nodes. We denote by RF_D the set of all replacement functions over the domain D, or in other words $RF_D = Node \times D^* \to Graph \cup \{\epsilon\}$.

After analysis completes, the intermediate representation is transformed in a separate pass by a transformation function $T : RF_D \times Graph \times (Edges \to D) \to Graph$. Given a replacement function R, a graph g, and the final dataflow analysis solution, T replaces each node in g with the graph returned by R for that node, thus producing a new graph. The definition of T is given below:

Definition 8 *Given a replacement function R, a graph g and some analysis results m, $T(R, g, m)$ is defined as follows. First, we introduce the update function $Update : Graph \times Node \times Graph \to Graph$, which is used to replace a single node in a graph. Given an original graph old, a node n and a replacement graph repl for this node, Update returns the result of replacing the node n with repl in old. Update is defined as follows:*

$$\begin{aligned} Update(old, node, repl) = (&N_{new}, E_{new}, In_{new}, Out_{new},\\ &InEdges_{new}, OutEdges_{new}) \end{aligned}$$

where

$$\begin{aligned}
&N_{new} = (N_{old} - \{n\}) \cup N_{repl}\\
&E_{new} = (E_{old} \cup E_{repl}) -\\
&\qquad (Elmts(InEdges_{repl}) \cup Elmts(OutEdges_{repl}))\\
&\qquad\quad \textit{with } Elmts(tuple) = \{d \mid \exists i.tuple[i] = d\}\\
&InEdges_{new} = InEdges_{old}\\
&OutEdges_{new} = OutEdges_{old}\\
&In_{new}(s) = \begin{cases} In_{old}(s) & \textit{if } s \in N_{old} - \{n\} \\ \overrightarrow{ReplIn}(In_{repl}(s)) & \textit{if } s \in N_{repl} \end{cases}\\
&Out_{new}(s) = \begin{cases} Out_{old}(s) & \textit{if } s \in N_{old} - \{n\} \\ \overrightarrow{ReplOut}(Out_{repl}(s)) & \textit{if } s \in N_{repl} \end{cases}
\end{aligned}$$

and

$$ReplIn(e) = \begin{cases} In_{old}(n)[k] & \textit{if } \exists k.e = InEdges_{repl}[k] \\ e & \textit{otherwise} \end{cases}$$

$$ReplOut(e) = \begin{cases} Out_{old}(n)[k] & \textit{if } \exists k.e = OutEdges_{repl}[k] \\ e & \textit{otherwise} \end{cases}$$

We now define $Update_\epsilon$, a simple extension to Update that works correctly if the replacement graph is ϵ:

$$Update_\epsilon(g, n, r) = \begin{cases} g & \textit{if } r = \epsilon \\ Update(g, n, r) & \textit{otherwise} \end{cases}$$

The graph returned by $T(R, g, m)$ is then simply the iterated application of $Update_\epsilon$ on all the nodes of g. Thus, $T(R, g, m)$ is defined by:

$$T(R, g, m) = IT(R, g, m, N_g)$$

where IT (which stands for $IteratedT$) is:

$$IT(R, g, m, N) = \begin{cases} IT(R, g_{new}, m, N - \{n\}) & \textit{if } \exists n \in N \\ g & \textit{if } N = \emptyset \end{cases}$$

with

$$g_{new} = Update_\epsilon(g, n, R(n, \overrightarrow{m}(In_g(n))))$$

The effect of an AT-analysis $(\mathcal{A}, R)$ is given by the function $[\![\mathcal{A}, R]\!] : Prog \to Prog$ defined below:

Definition 9 *If $\pi = (p_1, \ldots, p_n)$ then $[\![\mathcal{A}, R]\!](\pi) = (p'_1, \ldots, p'_n)$ where:*

$$\begin{aligned}
&p'_i = proc(name(p_i), formal(p_i), r_i)\\
&r_i = T(R', g_i, S_{\mathcal{A}}(g_i, \top_a))\\
&g_i = cfg(p_i)
\end{aligned}$$

and $R'(n, ds)$ is defined as follows:

- *if* $stmtAt(n) = (x := f(b))$ *and* $R(n, ds) = \epsilon$, *then* $R'(n, ds) = singleNodeGraph(n')$ *where* $stmtAt(n') = (x := f(b))$, $cfg(n') = r_k$, *and* $name(p_k) = f$
- *otherwise* $R'(n, ds) = R(n, ds)$

2.3.2 Soundness

We want each CFG produced by $(\mathcal{A}, R)$ to have the same concrete semantics as the corresponding original CFG. This is formalized in the following definition of soundness of $(\mathcal{A}, R)$:

Definition 10 *Let* $(\mathcal{A}, R)$ *be an AT-analysis, let* $\pi = (p_1, \ldots, p_n)$ *be a program where* $g_i = cfg(p_i)$, *and let* $[\![\mathcal{A}, R]\!](\pi) = \pi'$ *where* $\pi' = (p'_1, \ldots, p'_n)$ *and* $r_i = cfg(p'_i)$. *We say that* $(\mathcal{A}, R)$ *is sound iff:*

$$\forall i \in [1..n] \;.\; \forall \iota_c \in D_c \;.\; \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(OutEdges_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(OutEdges_{r_i})$$

We define here two conditions that together are sufficient to show that an AT-analysis is sound. First, the analysis $\mathcal{A}$ in $(\mathcal{A}, R)$ must be *locally sound* according to the following definition:

Definition 11 *We say that an analysis* $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$ *is locally sound iff it satisfies the following local soundness property:*

$$\begin{aligned} &\forall (n, cs, ds) \in Node \times D_c^* \times D_a^*. \\ &\quad \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_a(n, ds) \end{aligned} \tag{2.2}$$

If $\mathcal{A}$ is *locally sound*, then it is possible to show that $\mathcal{A}$ is *sound*, meaning that its solution correctly approximates the solution of the concrete analysis $\mathcal{C}$. This is formalized by the following definition and theorem.

Definition 12 *We say that an analysis* $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$ *is sound iff:*

$$\begin{aligned} &\forall (g, \iota_c, \iota_a) \in Graph \times D_c^* \times D_a^*. \\ &\quad \overrightarrow{\alpha}(\iota_c) \sqsubseteq \iota_a \Rightarrow \widetilde{\alpha}(S_{\mathcal{C}}(g, \iota_c)) \sqsubseteq S_{\mathcal{A}}(g, \iota_a) \end{aligned}$$

Theorem 1 *If an analysis* $\mathcal{A}$ *is locally sound then* $\mathcal{A}$ *is sound.*

Proof

Let $\mathcal{A} = (D_a, \sqcup_a, \sqcap_a, \sqsubseteq_a, \top_a, \bot_a, \alpha, F_a)$ be an analysis that is locally sound. Let $(g, \iota_c, \iota_a) \in Graph \times D_c^* \times D_a^*$ such that $\overrightarrow{\alpha}(\iota_c) \sqsubseteq_a \iota_a$. Also, suppose that FG_c and FG_a are the global flow functions in the definitions of $S_{\mathcal{C}}$ and $S_{\mathcal{A}}$ respectively (see definition 7), and similarly for the global ascending flow functions FGA_a and FGA_c. We need to show that:

$$\widetilde{\alpha}\left(\bigsqcup_{n=0}^{\infty} FGA_c^n(\widetilde{\bot_c})\right) \sqsubseteq_a \bigsqcup_{n=0}^{\infty} FGA_a^n(\widetilde{\bot_a})$$

To do this, we show $\forall n \geq 0 \,.\, \widetilde{\alpha}(FGA_c^n(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^n(\widetilde{\bot_a})$, and then the continuity of α implies the above equation. We first establish a few facts.

- Since F_c is continuous, it is monotonic, and therefore $FG_c^n(\widetilde{\bot_c})$ is an ascending chain. Thus, we get:

$$\forall n \geq 0 . FG_c^n(\widetilde{\bot_c}) = FGA_c^n(\widetilde{\bot_c}) \tag{2.3}$$

- Let $M_c = Edges_g \to D_c$, and $M_a = Edges_g \to D_a$. Since $\mathcal{A}$ is locally sound, it satisfies property (2.2), which combined with $\overrightarrow{\alpha}(\iota_c) \sqsubseteq_a \iota_a$ can be used to get:

$$\begin{aligned} &\forall (m_c, m_a) \in M_c \times M_a . \\ &\quad \widetilde{\alpha}(m_c) \sqsubseteq_a m_a \Rightarrow \widetilde{\alpha}(FG_c(m_c)) \sqsubseteq_a FG_a(m_a) \end{aligned} \tag{2.4}$$

- Since $FGA_a(m) = FG_a(m) \sqcup_a m$, we have:

$$\forall m \in M_a . FG_a(m) \sqsubseteq_a FGA_a(m) \tag{2.5}$$

Now we can show $\forall n \geq 0 \,.\, \widetilde{\alpha}(FGA_c^n(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^n(\widetilde{\bot_a})$. We do this by induction on n.

- *Base case.* When $n = 0$, $FGA_c^0(\widetilde{\bot_c}) = \widetilde{\bot_c}$, and $FGA_a^0(\widetilde{\bot_a}) = \widetilde{\bot_a}$. Since $\widetilde{\alpha}(\widetilde{\bot_c}) = \widetilde{\bot_a}$, we then get $\widetilde{\alpha}(FGA_c^0(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^0(\widetilde{\bot_a})$

- *Inductive case.* Assume $\widetilde{\alpha}(FGA_c^k(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^k(\widetilde{\bot_a})$ for some $k \geq 0$. We need to show $\widetilde{\alpha}(FGA_c^{k+1}(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^{k+1}(\widetilde{\bot_a})$. The proof is as follows:

$$\begin{array}{clr} & \widetilde{\alpha}(FGA_c^k(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^k(\widetilde{\bot_a}) & \\ \Leftrightarrow & \widetilde{\alpha}(FG_c^k(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^k(\widetilde{\bot_a}) & \text{using (2.3)} \\ \Leftrightarrow & \widetilde{\alpha}(FG_c(FG_c^k(\widetilde{\bot_c}))) \sqsubseteq_a FG_a(FGA_a^k(\widetilde{\bot_a})) & \text{using (2.4)} \\ \Leftrightarrow & \widetilde{\alpha}(FG_c(FG_c^k(\widetilde{\bot_c}))) \sqsubseteq_a FGA_a(FGA_a^k(\widetilde{\bot_a})) & \text{using (2.5)} \\ \Leftrightarrow & \widetilde{\alpha}(FG_c^{k+1}(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^{k+1}(\widetilde{\bot_a}) & \\ \Leftrightarrow & \widetilde{\alpha}(FGA_c^{k+1}(\widetilde{\bot_c})) \sqsubseteq_a FGA_a^{k+1}(\widetilde{\bot_a}) & \text{using (2.3)} \end{array}$$

■

Property (2.2) is sufficient for proving Theorem 1. Moreover it is weaker than the local consistency property of Cousot and Cousot (property 6.5 in [1]), which is:

$$\begin{aligned} &\forall (n, cs, ds) \in Node \times D_c^* \times D_a^* . \\ &\qquad\qquad \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F_a(n, \overrightarrow{\alpha}(cs)) \end{aligned}$$

Indeed, the above property and the monotonicity of F_a imply property (2.2). We use the weaker condition (2.2) because in this way our formalization of soundness does not depend on the monotonicity of F_a. As shown in sections 2.4 and 2.5, the flow function F_a is usually generated by our framework and reasoning about its monotonicity requires additional effort on the part of the analysis writer. By decoupling our soundness result from the monotonicity of F_a, we can guarantee soundness even if F_a has not been shown to be monotonic.[4]

In addition to the analysis having to be locally sound, R must produce graph replacements that are semantics-preserving. This is formalized by requiring that the replacement function R be *locally sound* according to the following definition:

[4]Termination in the face of a non-monotonic flow function is discussed in section 2.3.3.

Definition 13 *We say that a replacement function R in $(\mathcal{A}, R)$ is locally sound iff it satisfies the following local soundness property, where $\mathcal{A} = (D_a, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_a)$:*

$$\begin{aligned} &\forall (n, ds, g) \in Node \times D_a^* \times Graph. \\ &\quad R(n, ds) = g \Rightarrow \\ &\quad\quad [\forall cs \in D_c^*. \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \\ &\quad\quad\quad F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)] \end{aligned} \tag{2.6}$$

Property (2.6) requires that if R decides to replace a node n with a graph g on the basis of some analysis result ds, then for all possible input tuples of concrete values consistent with ds, it must be the case that n and g compute exactly the same output tuple of concrete values. It is not required that n and g produce the same output for all possible inputs, just those consistent with ds. For example, if $\mathcal{A}$ determines that all stores coming into a node n will always assign some variable `x` a value between 1 and 100 then n and g are not required to produce the same output for any store in which `x` is assigned a value outside of this range.

We say that $(\mathcal{A}, R)$ is *locally sound* iff both $\mathcal{A}$ and R are *locally sound*. If $(\mathcal{A}, R)$ is *locally sound*, then it is possible to show that $(\mathcal{A}, R)$ is *sound* according to definition 10, which means that the final graph produced by $(\mathcal{A}, R)$ has the same concrete behavior as the original graph. This is stated in the following theorem:

Theorem 2 *If an AT-analysis $(\mathcal{A}, R)$ is locally sound, then $(\mathcal{A}, R)$ is sound.*

Before proving theorem 2, we establish two helper lemmas. Throughout the following proofs, we use the notation $Eqs(g, m)$ to represent the set of dataflow equations of the concrete analysis $\mathcal{C}$ over the graph g with input dataflow information m.

Lemma 1 *Let g and r be graphs such that r is a subgraph of g, let $cs \in D_c^*$, and let $lfp_g = S_{\mathcal{C}}(g, cs)$. Then:*

$$lfp_g \setminus Edges_r = S_{\mathcal{C}}(r, lfp_g(InEdges_r))$$

Proof

Let $lfp_r = S_{\mathcal{C}}(r, lfp_g(InEdges_r))$, so that we need to show:

$$lfp_g \setminus Edges_r = lfp_r$$

Let FG_g be the global flow function FG from the definition of $S_{\mathcal{C}}(g, cs)$ (definition 7).

Let FG_r be the global flow function FG from the definition of $S_{\mathcal{C}}(r, lfp_g(InEdges_r))$ (definition 7).

Because F_c is monotonic, we have that $\forall n\ .\ FGA_g^n(\bot_c) = FG_g^n(\bot_c)$ and $\forall n\ .\ FGA_r^n(\bot_c) = FG_r^n(\bot_c)$.

Therefore:

$$lfp_g = \bigsqcup_{n=0}^{\infty} FG_g^n(\widetilde{\bot_c})$$

$$lfp_r = \bigsqcup_{n=0}^{\infty} FG_r^n(\widetilde{\bot_c})$$

If we can show that $\forall\, n \,.\, FG_g^n(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^n(\widetilde{\bot_c})$, then we are done, for then $lfp_g \setminus Edges_r \sqsubseteq lfp_r$, and furthermore, since $lfp_g \setminus Edges_r$ is a fixed point of $Eqs(r, lfp_g(InEdges_r))$, we also have $lfp_r \sqsubseteq lfp_g \setminus Edges_r$, which implies $lfp_g \setminus Edges_r \sqsubseteq lfp_r$ (and this is what we had to show).

All we need to show now is that $\forall\, n \,.\, FG_g^n(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^n(\widetilde{\bot_c})$. We do this by induction on n.

- *Base case.* We have $FG_g^0(\widetilde{\bot_c}) = \widetilde{\bot_c}$, $FG_r^0(\widetilde{\bot_c}) = \widetilde{\bot_c}$, and so $FG_g^0(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^0(\widetilde{\bot_c})$
- *Inductive case.* We assume $FG_g^n(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^n(\widetilde{\bot_c})$, and we need to show:

$$FG_g^{(n+1)}(\widetilde{\bot_c}) \setminus Edges_r \sqsubseteq FG_r^{n+1}(\widetilde{\bot_c})$$

Let $m_g = FG_g^n(\widetilde{\bot_c})$ and $m_r = FG_r^n(\widetilde{\bot_c})$. We therefore need to show:

$$FG_g(m_g) \setminus Edges_r \sqsubseteq FG_r(m_r)$$

which is:

$$\forall e \in Edges_r \,.\, FG_g(m_g)(e) \sqsubseteq FG_r(m_r)(e)$$

We pick $e \in Edges_r$, and show $FG_g(m_g)(e) \sqsubseteq FG_r(m_r)(e)$.

There are two cases:

  - $\exists k \,.\, e = InEdges_r[k]$. Then from the definition of FG_r (definition 7), we have:

$$\begin{array}{rcl} FG_r(m_r)(e) & = & \iota[k] \quad \text{where } \iota \text{ is the tuple used to initialize the fixed point computation of } lfp_r \\ & = & lfp_g(InEdges_r)[k] \quad \text{since } \iota = lfp_g(InEdges_r) \\ & = & lfp_g(e) \quad \text{since } e = InEdges_r[k] \end{array}$$

  Because F_c is monotonic, $FG_g^0(\widetilde{\bot_c}), FG_g^1(\widetilde{\bot_c}), FG_g^2(\widetilde{\bot_c}), \ldots$ is an ascending chain, and so we have $\forall n \,.\, FG_g^n(\widetilde{\bot_c}) \sqsubseteq lfp_g$.
  Therefore, $FG_g(m_g)(e) \sqsubseteq lfp_g(e)$, and so $FG_g(m_g)(e) \sqsubseteq FG_r(m_r)(e)$.
  - $\exists n, k \,.\, e = out_r(n)[k]$. From the definition of FG (definition 7), we therefore have:

$$FG_r(m_r)(e) = F_c(n, \overrightarrow{m_r}(in_r(n)))[k] \tag{2.7}$$

  Because r is a subgraph of g, we have $in_g(n) = in_r(n)$, and so we also have:

$$FG_g(m_g)(e) = F_c(n, \overrightarrow{m_g}(in_r(n)))[k] \tag{2.8}$$

  Since n is a node of r, the edges in the tuple $in_r(n)$ are all elements of $Edges_r$, and so from the inductive hypothesis:

$$\begin{array}{rl} & \overrightarrow{m_g}(in_r(n)) \sqsubseteq \overrightarrow{m_r}(in_r(n)) \\ \Rightarrow & F_c(n, \overrightarrow{m_g}(in_r(n)))[k] \sqsubseteq F_c(n, \overrightarrow{m_r}(in_r(n)))[k] \quad \text{(monotonicity of } F_c) \\ \Rightarrow & FG_g(m_g)(e) \sqsubseteq FG_r(m_r)(e) \quad \text{(using (2.7) and (2.8))} \end{array}$$

■

Lemma 2 *Let g be a graph, and let $n \in Node_g$ be a node in g. Let r be a replacement graph for n such that n and r have the same concrete semantics, or formally $\forall\ cs \in D_c^*\ .\ F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r)$. Let g' be the graph resulting from replacing n by r in g, or $g' = Update(g, n, r)$. Then g and g' have the same concrete semantics, or:*

$$\forall cs \in D_c^*\ .\ \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g) = \overrightarrow{S_{\mathcal{C}}(g', cs)}(OutEdges_{g'})$$

Proof

Pick $cs \in D_c^*$, and show $\overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g) = \overrightarrow{S_{\mathcal{C}}(g', cs)}(OutEdges_{g'})$.

Let r' be the graph that is exactly the same as r, except that it's input and output edges are $in_g(n)$ and $out_g(n)$. The graph r' is the subgraph of g' that resulted from the substitution of n.

Let $\overrightarrow{x} = in_g(n)$.

Let $lfp_g = S_{\mathcal{C}}(g, cs)$.

Let $cs_n = \overrightarrow{lfp_g}(\overrightarrow{x})$.

Let $lfp_{r'} = S_{\mathcal{C}}(r', cs_n)$.

Let $E = Edges_{g'} - Edges_g$. Intuitively, E is the set of edges in the replacement graph r, but without its input and output edges.

Let $m : Edge_{g'} \to D_c$ be defined as follows:

$$m(e) = \begin{cases} lfp_g(e) & \text{if } e \in Edges_g \\ lfp_{r'}(e) & \text{if } e \in E \end{cases} \tag{2.9}$$

m is a fixed point of $Eqs(g', cs)$, since by construction it satisfies all the dataflow equations in $Eqs(g', cs)$.

The claim is that m is the least fixed point of $Eqs(g', cs)$, which means that $m = S_{\mathcal{C}}(g', cs)$.

If this is the case, then because $OutEdges_g = OutEdges_{g'}$ and because all edges in $OutEdges_g$ are in $Edges_g$, we would have from equation (2.9):

$$\overrightarrow{m}(OutEdges_{g'}) = \overrightarrow{lfp_g}(OutEdges_g)$$

which, since $m = S_{\mathcal{C}}(g', cs)$ and $lfp_g = S_{\mathcal{C}}(g, cs)$, becomes:

$$\overrightarrow{S_{\mathcal{C}}(g', cs)}(OutEdges_{g'}) = \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)$$

which is what we had to show.

All we need to show now is that m is indeed the least fixed point of $Eqs(g', cs)$.

Let m' be the least fixed point of $Eqs(g', cs)$, so that $m' = S_{\mathcal{C}}(g', cs)$.

Since m' is the least fixed point of $Eqs(g', cs)$, and since m is a fixed point of $Eqs(g', cs)$, we must have $m' \sqsubseteq m$. We will now show $m \sqsubseteq m'$, which will establish that $m = m'$ and that m is indeed the least fixed point.

We will prove $m \setminus Edges_g \sqsubseteq m' \setminus Edges_g$, and $m \setminus E \sqsubseteq m' \setminus E$, which will imply $m \sqsubseteq m'$ because the domain of m and m' is $Edges_g \cup E$.

- Proof of $m \setminus Edges_g \sqsubseteq m' \setminus Edges_g$

  By lemma 1, instantiated with $g = g'$, $r = r'$, we get that:

$$m' \setminus Edges_{r'} = S_{\mathcal{C}}(r', \overrightarrow{m'}(InEdges_{r'})) \tag{2.10}$$

  By the construction of how n gets replaced with r in g, we have $\overrightarrow{x} = InEdges_{r'}$ and so we get:

$$m' \setminus Edges_{r'} = S_{\mathcal{C}}(r', \overrightarrow{m'}(\overrightarrow{x})) \tag{2.11}$$

  From our assumptions, we know:

$$\forall\ cs \in D_c^*\ .\ F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r)$$

  which, because r and r' only differ in their incoming and outgoing edges, gives us:

$$\forall\ cs \in D_c^*\ .\ F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r', cs)}(OutEdges_{r'}) \tag{2.12}$$

  Equations (2.10) and (2.12) tell us that if we replace r' with n, the local constraint for n is satisfied by m'. As a result:

$$\begin{array}{rrl} & m' \setminus Edges_g \text{ is a fixed point of } Eqs(g, cs) & \\ \Rightarrow & \mathit{lfp}_g \sqsubseteq m' \setminus Edges_g & \text{(by the defn of least fixed point)} \\ \Rightarrow & m \setminus Edges_g \sqsubseteq m' \setminus Edges_g & \text{(since for } \forall e \in Edges_g\ .\ m(e) = \mathit{lfp}_g(e)) \end{array} \tag{2.13}$$

- Proof of $m \setminus E \sqsubseteq m' \setminus E$

  Let $cs'_n = \overrightarrow{m'}(\overrightarrow{x})$. Since all the edges in the $\overrightarrow{x}$ tuple are in $Edges_g$, we get from (2.13):

$$\begin{array}{rcl} & \overrightarrow{m}(\overrightarrow{x}) \sqsubseteq \overrightarrow{m'}(\overrightarrow{x}) & \\ \Rightarrow & cs_n \sqsubseteq cs'_n & \\ \Rightarrow & S_{\mathcal{C}}(r', cs_n) \sqsubseteq S_{\mathcal{C}}(r', cs'_n) & \text{(by the monotonicity of } S_{\mathcal{C}}) \\ \Rightarrow & \mathit{lfp}_{r'} \sqsubseteq S_{\mathcal{C}}(r', cs'_n) & \text{(by defn of } \mathit{lfp}_{r'}) \\ \Rightarrow & \mathit{lfp}_{r'} \sqsubseteq m' \setminus Edges_{r'} & \text{(by (2.11))} \\ \Rightarrow & \mathit{lfp}_{r'} \setminus E \sqsubseteq m' \setminus E & \text{(since } E \subseteq Edges_{r'}) \\ \Rightarrow & m \setminus E \sqsubseteq m' \setminus E & \text{(by (2.9))} \end{array} \tag{2.14}$$

$\blacksquare$

Proof of theorem 2

Without loss of generality, we can assume that the replacement function R always returns graph replacements with only one node in them.

Indeed, if R returned a multiple node replacement graph r, then one can build a new replacement function R_{single} which, instead of returning r, returns a graph replacement containing a single node n equivalent to r (in that it satisfies $\forall\ cs \in D_c^*\ .\ F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r)$). Lemma 2 can then be used to argue that the soundness of $(\mathcal{A}, R_{single})$ implies the soundness of $(\mathcal{A}, R)$.

Let $(\mathcal{A}, R)$ be an AT-analysis that is locally sound, let $\pi = (p_1, \ldots, p_n)$ be a program where $g_i = cfg(p_i)$, and let $[\![\mathcal{A}, R]\!](\pi) = \pi'$ where $\pi' = (p'_1, \ldots, p'_n)$ and $r_i = cfg(p'_i)$. We want to show:

$$\forall i \in [1..n]\ .\ \forall \iota_c \in D_c^*\ .\ \overrightarrow{S_{\mathcal{C}}(g_i, \iota_c)}(OutEdges_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, \iota_c)}(OutEdges_{r_i})$$

The CFG has one input edge, and so ι_c will always be a tuple of length 1. Thus, we need to prove:

$$\forall i \in [1..n]\ .\ \forall c \in D_c\ .\ \overrightarrow{S_{\mathcal{C}}(g_i, (c))}(OutEdges_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, (c))}(OutEdges_{r_i})$$

We let FGA_{g_i} and FG_{g_i} be the global flow function and the global ascending flow function in the definition of $S_{\mathcal{C}}(g_i, (c))$.

We let FGA_{r_i} and FG_{r_i} be the global flow function and the global ascending flow function in the definition of $S_{\mathcal{C}}(r_i, (c))$.

Since F_c is monotonic, we have that $FG_{r_i}^n(\widetilde{\bot_c})$ and $FG_{g_i}^n(\widetilde{\bot_c})$ are ascending chains, and so:

$$\begin{aligned} \forall n \geq 0 . FG_{g_i}^n(\widetilde{\bot_c}) = FGA_{g_i}^n(\widetilde{\bot_c}) \\ \forall n \geq 0 . FG_{r_i}^n(\widetilde{\bot_c}) = FGA_{r_i}^n(\widetilde{\bot_c}) \end{aligned} \tag{2.15}$$

For $\eta \in State$, we let $stackDepth(\eta, i)$ be the maximum stack depth that occurs during execution of g_i starting in η, and we let $stackDepth^+(\eta, i)$ be $stackDepth(\eta, i)$ minus the stack depth of η. In other words, $stackDepth^+(\eta, i)$ is the number of *additional* stack frames required for execution of g_i starting in η.

For $c \in D_c$ (i.e. c is a set of states η), we let $maxStackDepth^+(c, i)$ be $max_{\eta \in c}(stackDepth^+(\eta, i))$.

For $l \geq 0$, let $P(l)$ be the following formula:

$$P(l) = \forall i \in [1..n]\ .\ \forall c \in D_c\ .\ maxStackDepth^+(c, i) \leq l \Rightarrow \overrightarrow{S_{\mathcal{C}}(g_i, (c))}(OutEdges_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, (c))}(OutEdges_{r_i})$$

Our goal is to prove $\forall l \geq 0\ .\ P(l)$. We do this by induction on l.

- *Base case.* We need to show $P(0)$.

  Pick $i \in [1..n]$, $d \in D_c$, assume:

  $$maxStackDepth^+(c, i) \leq 0 \tag{2.16}$$

  and show:

  $$\overrightarrow{S_{\mathcal{C}}(g_i, (c))}(OutEdges_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, (c))}(OutEdges_{r_i})$$

Using the definition of $S_{\mathcal{C}}$ this is:

$$\bigsqcup_{j=0}^{\infty} FGA_{g_i}^{j}(\widetilde{\bot_c}) \sqsubseteq_c \bigsqcup_{j=0}^{\infty} FGA_{r_i}^{j}(\widetilde{\bot_c})$$

Using (2.15), we need to show:

$$\bigsqcup_{j=0}^{\infty} FG_{g_i}^{j}(\widetilde{\bot_c}) \sqsubseteq_c \bigsqcup_{j=0}^{\infty} FG_{r_i}^{j}(\widetilde{\bot_c})$$

To do this, we show:

$$\forall j \geq 0 \ . \ FG_{g_i}^{j}(\widetilde{\bot_c}) \sqsubseteq_c FG_{r_i}^{j}(\widetilde{\bot_c})$$

We do this by induction on j.

- *Base case.* We need to show $FG_{g_i}^{0}(\widetilde{\bot_c}) \sqsubseteq_c FG_{r_i}^{0}(\widetilde{\bot_c})$. This follows immediately from the fact that $FG_{g_i}^{0}(\widetilde{\bot_c}) = \widetilde{\bot_c}$ and $FG_{r_i}^{0}(\widetilde{\bot_c}) = \widetilde{\bot_c}$.
- *Inductive case.* We assume $FG_{g_i}^{j}(\widetilde{\bot_c}) \sqsubseteq_c FG_{r_i}^{j}(\widetilde{\bot_c})$, and we need to show $FG_{g_i}^{j+1}(\widetilde{\bot_c}) \sqsubseteq_c FG_{r_i}^{j+1}(\widetilde{\bot_c})$.

  Let $a = FG_{g_i}^{j}(\widetilde{\bot_c})$ and $b = FG_{r_i}^{j}(\widetilde{\bot_c})$, so that we assume:

  $$a \sqsubseteq_c b \tag{2.17}$$

  We need to show $FG_{g_i}(a) \sqsubseteq_c FG_{r_i}(b)$, which, because $Edges_{g_i} = Edges_{r_i}$, is:

  $$\forall e \in Edges_{g_i} \ . \ FG_{g_i}(a)(e) \sqsubseteq_c FG_{r_i}(b)(e)$$

  Which is:

  $$\forall e \in Edges_{g_i} \ . \ Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b)$$

  Let $e \in Edges_{g_i}$, and we need to show:

  $$Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b) \tag{2.18}$$

  There are four cases, based on the definition of $Int(e, m)$, and on how n was transformed:

  * There is some integer k such that $e = InEdges[k]$
    In this case, $Int_{g_i}(e, a) = c$ and $Int_{r_i}(e, b) = c$, and so $Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b)$.
  * There is some integer k and some node n such that $e = out_{g_i}(n)[k]$, and n was not modified from g_i to r_i.
    Then by the definition of Int, we get $Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(in_{g_i}(n)))[k]$.
    Since n was not modified from g_i to r_i, we have that $e = out_{r_i}(n)[k]$, and so by the definition of Int, we get $Int_{r_i}(e, b) = F_c(n, \overrightarrow{b}(in_{r_i}(n)))[k]$.
    Since n was not modified from g_i to r_i, we have that $in_{g_i}(n) = in_{r_i}(n)$. Let $\overrightarrow{x} = in_{g_i}(n)$. Thus:

$$Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k] \tag{2.19}$$

$$Int_{r_i}(e, b) = F_c(n, \overrightarrow{b}(\overrightarrow{x}))[k] \tag{2.20}$$

From (2.17), we know that $a \sqsubseteq_c b$, and so $\overrightarrow{a}(\overrightarrow{x}) \sqsubseteq_c \overrightarrow{b}(\overrightarrow{x})$.
By the monotonicity of F_c, we then get:

$$F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k] \sqsubseteq_c F_c(n, \overrightarrow{b}(\overrightarrow{x}))[k]$$

And this is:

$$Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b)$$

which is what we had to show in (2.18).

* There is some integer k and some node n such that $e = out_{g_i}(n)[k]$, and n in g_i was modified to n' in r_i because R returned a single-node replacement graph r_n for n.
Then by the definition of Int, we get $Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(in_{g_i}(n)))[k]$.
Since n was modified to a single-node graph n', by the definition of T, we get that $e = out_{r_i}(n')[k]$, and so by the definition of Int, we get $Int_{r_i}(e, b) = F_c(n', \overrightarrow{b}(in_{r_i}(n')))[k]$.
Since n was modified to a single-node graph n', by the definition of T, we get that $in_{g_i}(n) = in_{r_i}(n')$. Let $\overrightarrow{x} = in_{g_i}(n)$. Thus:

$$Int_{g_i}(e, a) = F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k]$$
$$Int_{r_i}(e, b) = F_c(n', \overrightarrow{b}(\overrightarrow{x}))[k]$$

Let FGA_a be the global ascending function from the definition of $S_{\mathcal{A}}(g_i, \top_a)$.
Since $\alpha(\bot_c) \sqsubseteq_a \top_a$, we have from the proof of of theorem 1:

$$\forall j \geq 0 \,.\, \widetilde{\alpha}(FG^j_{g_i}(\widetilde{\bot_c})) \sqsubseteq_a FGA^j_a(\widetilde{\bot_a})$$

Instantiating this with the current j, we get::

$$\widetilde{\alpha}(FG^j_{g_i}(\widetilde{\bot_c})) \sqsubseteq_a FGA^j_a(\widetilde{\bot_a})$$

Or:

$$\widetilde{\alpha}(a) \sqsubseteq_a FGA^j_a(\widetilde{\bot_a})$$

Furthermore, from the definition of $S_{\mathcal{A}}$, we have that:

$$FGA^k_a(\widetilde{\bot_a}) \sqsubseteq_a S_{\mathcal{A}}(g_i, \top_a)$$

Thus, by transitivity:

$$\widetilde{\alpha}(a) \sqsubseteq_a S_{\mathcal{A}}(g_i, \top_a)$$

Let $m = S_{\mathcal{A}}(g_i, \top_a)$, so that:

$$\widetilde{\alpha}(a) \sqsubseteq_a m$$

Let $cs = \overrightarrow{a}(\overrightarrow{x})$ and let $ds = \overrightarrow{m}(\overrightarrow{x})$. Therefore $\overrightarrow{\alpha}(cs) \sqsubseteq_a ds$.

From the definition of $[\![\mathcal{A}, R]\!]$ (definition 9), we know that:

$$r_i = T(R', g_i, m)$$

For n to have been replaced with r_n, by the definition of T, it therefore must be the case that:

$$R'(n, \overrightarrow{m}(\overrightarrow{x})) = r_n$$

Since we assumed that r_n was returned by R, we must be in the second case of the definition of R' (definition 9), and thus:

$$R(n, \overrightarrow{m}(\overrightarrow{x})) = r_n$$

From the soundness of R, we know that (2.6) holds. Instantiating (2.6) with $n = n, ds = ds, g = r_n, cs = cs$, and using $\overrightarrow{\alpha}(cs) \sqsubseteq_a ds$, we get:

$$F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_n, cs)}(OutEdges_{r_n})$$

Since r_n is a single-node graph with node n', we have that

$$\overrightarrow{S_{\mathcal{C}}(r_n, cs)}(OutEdges_{r_n}) = F_c(n', cs)$$

Thus:

$$\begin{array}{rl} & F_c(n, cs) \sqsubseteq_c F_c(n', cs) \\ \Rightarrow & F_c(n, \overrightarrow{a}(\overrightarrow{x})) \sqsubseteq_c F_c(n', \overrightarrow{a}(\overrightarrow{x})) \end{array} \tag{2.21}$$

From(2.17), we have:

$$\begin{array}{rll} & a \sqsubseteq_c b & \\ \Rightarrow & \overrightarrow{a}(\overrightarrow{x}) \sqsubseteq_c \overrightarrow{b}(\overrightarrow{x}) & \\ \Rightarrow & F_c(n', \overrightarrow{a}(\overrightarrow{x})) \sqsubseteq_c F_c(n', \overrightarrow{b}(\overrightarrow{x})) & \text{monotonicity of } F_c \\ \Rightarrow & F_c(n, \overrightarrow{a}(\overrightarrow{x})) \sqsubseteq_c F_c(n', \overrightarrow{b}(\overrightarrow{x})) & \text{transitivity and (2.21)} \\ \Rightarrow & F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k] \sqsubseteq_c F_c(n', \overrightarrow{b}(\overrightarrow{x}))[k] & \\ \Rightarrow & Int_{g_i}(e, a) \sqsubseteq_c Int_{r_i}(e, b) & \text{using (2.19) and (2.20)} \end{array}$$

And this is what we had to show in (2.18).

* There is some integer k and some node n such that $e = out_{g_i}(n)[k]$, and n in g_i was modified to n' in r_i because R' return a single-node replacement graph r_n for n, but R returned ϵ as a replacement for n.
  We are in the first case of the definition of R' (from definition 9).
  We therefore know that $stmtAt(n) = (y := f(z))$.
  From (2.16), we know that $maxStackDepth^+(c, i) \leq 0$, which means that $maxStackDepth^+(c, i) = 0$. This implies that the execution of this CFG on input c does not cause any calls.
  Since n is a call (since $stmtAt(n) = (y := f(z))$), it follows that n cannot execute.
  As a result it must be the case that $\overrightarrow{a}(in_{g_i}(n)) = \overrightarrow{\bot_c}$, for otherwise n would execute in the current CFG on input c.

By the definition of Int, we have

$$\begin{aligned} Int_{g_i}(e,a) &= F_c(n, \overrightarrow{a}(in_{g_i}(n)))[k] \\ &= F_c(n, \overrightarrow{\bot_c})[k] \\ &= \bot_c \end{aligned}$$

It is then trivial that $Int_{g_i}(e,a) \sqsubseteq_c Int_{r_i}(e,b)$, which is what we had to show in (2.18).

- *Inductive case.* We assume $P(l)$ and show $P(l+1)$, which is:

$$\forall i \in [1..n] . \forall c \in D_c . \mathit{maxStackDepth}^+(c,i) \leq l{+}1 \Rightarrow \overrightarrow{S_{\mathcal{C}}(g_i,(c))}(OutEdges_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i,(c))}(OutEdges_{r_i})$$

Pick $i \in [1..n]$, $d \in D_c$, assume:

$$\mathit{maxStackDepth}^+(c,i) \leq l+1 \tag{2.22}$$

and show:

$$\overrightarrow{S_{\mathcal{C}}(g_i,(c))}(OutEdges_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i,(c))}(OutEdges_{r_i})$$

The proof proceeds as in the base case, except for the following subcase:

  – Inside the *inductive case* for the induction over j

    ∗ There is some integer k and some node n such that $e = out_{g_i}(n)[k]$, and n in g_i was modified to n' in r_i because R' return a single-node replacement graph r_n for n, but R returned ϵ as a replacement for n.
    Then by the definition of Int, we get $Int_{g_i}(e,a) = F_c(n, \overrightarrow{a}(in_{g_i}(n)))[k]$.
    Since n was modified to a single-node graph n', by the definition of T, we get that $e = out_{r_i}(n')[k]$, and so by the definition of Int, we get $Int_{r_i}(e,b) = F_c(n', \overrightarrow{b}(in_{r_i}(n')))[k]$.
    Since n was modified to a single-node graph n', by the definition of T, we get that $in_{g_i}(n) = in_{r_i}(n')$. Let $\overrightarrow{x} = in_{g_i}(n)$. Thus:

$$\begin{aligned} Int_{g_i}(e,a) = F_c(n, \overrightarrow{a}(\overrightarrow{x}))[k] \\ Int_{r_i}(e,b) = F_c(n', \overrightarrow{b}(\overrightarrow{x}))[k] \end{aligned}$$

    Because R returned ϵ, we are in the first case of the definition of R' (from definition 9).
    We therefore know that $\mathit{stmtAt}(n) = (y := f(z))$ and $\mathit{stmtAt}(n') = (y := f(z))$. Since both n and n' are call nodes, they only have one successor, and so it must be the case that $k = 0$.
    Thus:

$$\begin{aligned} Int_{g_i}(e,a) = F_c(n, \overrightarrow{a}(\overrightarrow{x}))[0] \\ Int_{r_i}(e,b) = F_c(n', \overrightarrow{b}(\overrightarrow{x}))[0] \end{aligned}$$

    From the definition of F_c in (6), we get:

$$\begin{aligned} &Int_{g_i}(e,a) = \overrightarrow{S_{\mathcal{C}}(cfg(n),(X))}(OutEdges_{cfg(n')}) \\ &\quad \text{where } X = \{\eta \mid \exists \eta' \in \mathit{State} \,.\, [\eta' \in a(x[0]) \wedge 0, \eta' \xrightarrow{n} 1, \eta\} \\ &Int_{r_i}(e,b) = \overrightarrow{S_{\mathcal{C}}(cfg(n'),(Y))}(OutEdges_{cfg(n')}) \\ &\quad \text{where } Y = \{\eta \mid \exists \eta' \in \mathit{State} \,.\, [\eta' \in b(x[0]) \wedge 0, \eta' \xrightarrow{n'} 1, \eta\} \end{aligned} \tag{2.23}$$

    From the definition of R', we know that there is some $u \in [1..n]$ such that $cfg(n') = r_u$ and $name(p_u) = f$.

Since $\mathit{name}(p_u) = f$, and since $\mathit{stmtAt}(n) = (y := f(z))$, it must be the case that $\mathit{cfg}(n) = g_u$. Because n is a call, we have:

$$\mathit{maxStackDepth}^+(c, i) \geq \mathit{maxStackDepth}^+(X, u) + 1$$

Combined with (2.22), this gives:

$$\begin{array}{rl} & \mathit{maxStackDepth}^+(X, u) + 1 \leq l + 1 \\ \Rightarrow & \mathit{maxStackDepth}^+(X, u) \leq l \end{array}$$

The induction hypothesis is $P(l)$, which is:

$$\forall i \in [1..n] \,.\, \forall c \in D_c \,.\, \mathit{maxStackDepth}^+(c, i) \leq l \Rightarrow \overrightarrow{S_{\mathcal{C}}(g_i, (c))}(\mathit{OutEdges}_{g_i}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_i, (c))}(\mathit{OutEdges}_{r_i})$$

We instantiate this with $i = u$ and $c = X$, and we get:

$$\overrightarrow{S_{\mathcal{C}}(g_u, (X))}(\mathit{OutEdges}_{g_u}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_u, (X))}(\mathit{OutEdges}_{r_u})$$

From (2.17), we know that $a \sqsubseteq_c b$, and so $a(x[0]) \sqsubseteq_c b(x[0])$. As a result, $X \sqsubseteq_c Y$, and by the monotonicity of $S_{\mathcal{C}}$:

$$\overrightarrow{S_{\mathcal{C}}(r_u, (X))}(\mathit{OutEdges}_{r_u}) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(r_u, (Y))}(\mathit{OutEdges}_{r_u})$$

Since $\mathit{cfg}(n) = r_u$ and $\mathit{cfg}(n') = r_u$, and using (2.23), we get:

$$\mathit{Int}_{g_i}(e, a) \sqsubseteq_c \mathit{Int}_{r_i}(e, b)$$

which is what we had to show in (2.18)

■

2.3.3 Termination

If the lattice has finite height, then the termination of an analysis $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$ is guaranteed from within $S_{\mathcal{A}}$, even if F is not monotonic: as iteration proceeds, $S_{\mathcal{A}}$ forces the dataflow values to monotonically increase by joining the next solution with the current solution at each step. If the lattice has infinite height, then the flow function for loop header nodes can include widening operators [1] to guarantee termination.

We chose to enforce termination from within $S_{\mathcal{A}}$, instead of requiring F to be monotonic, for the same reason we chose the weaker soundness condition (2.2): flow functions are generated by our framework, and proving that they are monotonic requires additional effort on the part of the analysis writer. By having termination and soundness be decoupled from the monotonicity of F, we allow analysis designers the option of not proving that F is monotonic. The drawback of not having F be monotonic is that the fixed point computed by $S_{\mathcal{A}}$ is not necessarily a least fixed point anymore. As a result, the solution returned by $S_{\mathcal{A}}$ is not guaranteed to be the most precise one.

2.4 Integrating Analysis and Transformation

Now that we have defined a single analysis followed by some transformations, we proceed to formalizing how our framework integrates an analysis with its transformations.

2.4.1 Definition

An *Integrated Analysis* is a tuple $\mathcal{IA} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, FR)$ where $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ is a complete lattice, $\alpha : D_c \to D$ is the abstraction function, and $FR : Node \times D^* \to D^* \cup Graph$ is a *flow-replacement function*. The flow-replacement function FR takes a node and a tuple of input abstract values, one per incoming edge to the node, and returns either a tuple of output abstract values, one per outgoing edge from the node, or a graph with which to replace the node.

An integrated analysis is an analysis which has been combined with its transformations. The flow replacement function can now return graph transformations that are taken into account during the fixed point computation, and used after the fixed point has been reached to make permanent transformations to the graph.

The meaning of an integrated analysis is defined in terms of an *associated AT-analysis*, for which the behavior has already been defined in section 2.3.1. Given an integrated analysis $\mathcal{IA} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, FR)$, we define the associated AT-analysis $\mathcal{AT}_{\mathcal{IA}}$ as $(\mathcal{A}, R)$, with $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$, where F and R are derived from FR as follows:

$$F(n, ds) = \begin{cases} FR(n, ds) & \text{if } FR(n, ds) \in D^* \\ SolveSubGraph_F(FR(n, ds), ds) & \text{otherwise} \end{cases}$$

$$SolveSubGraph_F(g, ds) = \overrightarrow{S_{\mathcal{A}}(g, ds)}(OutEdges_g)$$

$$R(n, ds) = \begin{cases} \epsilon & \text{if } FR(n, ds) \in D^* \\ SolveSubGraph_R(FR(n, ds), ds) & \text{otherwise} \end{cases}$$

$$SolveSubGraph_R(g, ds) = T(R, g, S_{\mathcal{A}}(g, ds))$$

The definition of F above shows how transformations are taken into account while the analysis is running. If FR returns a tuple of dataflow values, then that tuple is immediately returned. If, on the other hand, FR chooses to do a transformation, the replacement graph is recursively analyzed and the dataflow values computed for the output edges of the graph are returned. The next time the same node gets analyzed, FR can choose another graph transformation, or possibly no transformation at all. Transformations are only committed after the analysis has reached a final sound solution, as specified by the definition of R. If at the final dataflow solution, FR returns a tuple of dataflow values, then R returns ϵ, indicating that the analysis has chosen not to do a transformation. If, on the other hand, FR chooses a replacement graph, then R returns this replacement graph after transformations have been applied to it recursively. Although the definition of R above reanalyzes recursive graph replacements, an efficient implementation, such as the ones in Vortex and Whirlwind, can cache the solution of the last replacement graph computed by FR for each node, so that the transformation pass need not recompute them.

2.4.2 Soundness

An integrated analysis $\mathcal{IA}$ is sound if the associated AT-analysis $\mathcal{AT}_{\mathcal{IA}}$ is sound. We define here conditions that are sufficient to show that $\mathcal{AT}_{\mathcal{IA}}$ is sound, and therefore that $\mathcal{IA}$ is sound. Intuitively, we want the flow-replacement function FR to satisfy condition (2.2) when it returns a tuple of dataflow values, and condition (2.6) when it returns a replacement graph. Formally, this amounts to having $\mathcal{IA}$ be locally sound according to the following definition:

Definition 14 *We say that an integrated analysis* $\mathcal{IA} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, FR)$ *is locally sound iff it satisfies the following two local soundness properties:*

$$\begin{array}{l} \forall (n, cs, ds) \in Node \times D_c^* \times D^*. \\ \quad FR(n, ds) \in D^* \Rightarrow \\ \qquad [\overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq FR(n, ds)] \end{array} \tag{2.24}$$

$$\begin{array}{l} \forall (n, ds, g) \in Node \times D^* \times Graph. \\ \quad FR(n, ds) = g \Rightarrow \\ \qquad [\forall cs \in D_c^*. \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \\ \qquad\quad F_c(n, cs) \sqsubseteq \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)] \end{array} \tag{2.25}$$

Note that the first property is the same as (2.2) with F_a replaced by FR, except for the additional antecedent $FR(n, ds) \in D^*$, and the second property is the same as (2.6), with R replaced by FR.

Theorem 3 *If an integrated analysis* $\mathcal{IA}$ *is locally sound, then the associated AT-analysis* $\mathcal{AT}_{\mathcal{IA}}$ *is sound, and therefore* $\mathcal{IA}$ *is sound.*

Proving Theorem 3 involves showing that if FR satisfies properties (2.24) and (2.25), then F and R as defined in section 2.4.1 satisfy properties (2.2) and (2.6) respectively. Throughout this subsection, F and R will always refer to the definitions from section 2.4.1. The proof is by induction on the graph nesting structure. We give an intuitive description of what this means, and then we proceed with the proof.

Since we are not interested in proving soundness when the algorithm does not terminate, we assume there is no infinite nesting of graph replacements. This means that at some point in the recursion, there is a graph whose nodes do not request graph replacements. These nodes are the base case of the induction, and correspond in the definitions of F and R to the condition $FR(n, ds) \in D^*$. The inductive case is the one where a node does choose a graph transformation. The inductive hypothesis is that the property of interest holds for nodes in the replacement graph, and our goal is to prove that the property holds for the current node.

Proof that F satisfies property (2.2)

Base case. We are in the first case of the definition of F, the one where $FR(n, ds) \in D^*$. We therefore have $F(n, ds) = FR(n, ds)$, and then property (2.24) immediately implies (2.2).

Inductive case. We are in the second case of the definition of F, the one where $FR(n, ds) \in Graph$. The induction hypothesis is that property (2.2) holds for nodes in the recursive graph $g = FR(n, ds)$, and we want to prove that property (2.2) holds at the current node n. To do this, we assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, and show $\overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F(n, ds)$.

Since $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, we have from property (2.25):

$$F_c(n, cs) \sqsubseteq \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g) \tag{2.26}$$

By the induction hypothesis, F satisfies property (2.2) in g. Since in addition $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, we can then use Theorem 1 (with $\iota_c = cs$ and $\iota_a = ds$) to get:

$$\begin{array}{rc}
 & \widetilde{\alpha}(S_{\mathcal{C}}(g, cs)) \sqsubseteq S_{\mathcal{A}}(g, ds) \\
\Leftrightarrow & \overrightarrow{\alpha}(\overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)) \sqsubseteq \overrightarrow{S_{\mathcal{A}}(g, ds)}(OutEdges_g) \\
\Leftrightarrow & \overrightarrow{\alpha}(\overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)) \sqsubseteq F(n, ds) \ \ \text{(using def of } F\text{)} \\
\Leftrightarrow & \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F(n, ds) \quad \text{(using (2.26) and the monotonicity of } \alpha\text{)}
\end{array}$$

■

Proof that R satisfies property (2.6)

Base case. We are in the first case of the definition of R, the one where $FR(n, ds) \in D^*$. We therefore have $R(n, ds) = \epsilon$, and then property (2.6) holds trivially because the antecedent $R(n, ds) = g$ is false.

Inductive case. We are in the second case of the definition of R, the one where $FR(n, ds) \in Graph$. The induction hypothesis is that property (2.6) holds in the recursive graph $g = FR(n, ds)$, and we want to prove that property (2.6) holds at the current node n. To do this, we assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, we let $r = T(R, g, S_{\mathcal{A}}(g, ds)$, which implies that $r = R(n, ds)$, and we want to show that $F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r)$. Note that the g in (2.6) has been replaced here with r since it is r which equals $R(n, ds)$. g in our case stands for $FR(n, ds)$.

As in the previous proof, equation (2.26) holds because of property (2.25) combined with $\overrightarrow{\alpha}(cs) \sqsubseteq ds$.

By the inductive hypothesis, R satisfies property (2.6) for the nodes in g. In addition, we have already shown that F satisfies property (2.2), and we also know that $\overrightarrow{\alpha}(cs) \sqsubseteq ds$. We can therefore use Theorem 2 (with $\iota_c = cs$ and $\iota_a = ds$) to get:

$$\overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r) \sqsupseteq \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)$$

Combined with (2.26), this gives the required result:

$$F_c(n, cs) \sqsubseteq \overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r)$$

■

2.4.3 Termination

As in the case of an AT-analysis, the function $S_{\mathcal{A}}$ forces the solution to monotonically increase as iteration proceeds, even if the flow function F is not monotonic. If the designer of the analysis puts in the effort to prove that F is monotonic, then $S_{\mathcal{A}}$ computes the least fixed point. Otherwise, the result computed by $S_{\mathcal{A}}$ is not necessarily a least fixed point, but it is nevertheless sound as long as properties (2.24) and (2.25) hold.

However, having the solution monotonically increase is no longer sufficient to ensure termination: it is now possible for the flow functions to choose graph replacements that cause infinite recursion of nested graph analysis. For example, an inlining optimization could choose to inline a recursive function indefinitely. To ensure termination, we require that the user's graph replacements do not trigger such endless recursive transformations. Graph replacements either obviously simplify the program (such as deleting a node or replacing a complex node with several simpler ones), and thus cannot cause unbounded recursive graph replacements, or there are standard ways of avoiding endless recursive graph transformations (for instance, by marking selected nodes in the replacement graph as non-replaceable). Our framework does not enforce an arbitrary fixed bound on recursive graph replacements. Instead, we feel that individual dataflow analyses will have their own most appropriate solution, which can be explicitly implemented in the flow-replacement function. This non-termination issue with our framework is already present in any system that iteratively applies analyses and transformations. Such systems have either imposed some fixed bound on the number of iterations, or, as we do, require the analyses to avoid endless transformations.

2.4.4 Combining an AT-analysis

Our framework also allows a sound AT-analysis to be automatically converted into a sound integrated analysis. In particular, given an AT-analysis $\mathcal{AT} = (\mathcal{A}, R)$ where $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$, then the associated integrated analysis $\mathcal{IA}_{\mathcal{AT}}$ is defined as $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, FR)$, where FR is derived from F and R as follows:

$$FR(n, ds) = \begin{cases} F(n, ds) & \text{if } R(n, ds) = \epsilon \\ R(n, ds) & \text{if } R(n, ds) \neq \epsilon \end{cases}$$

Theorem 4 *If an AT-analysis $\mathcal{AT}$ is locally sound, then the associated integrated analysis $\mathcal{IA}_{\mathcal{AT}}$ is sound.*

Proof

Immediate from properties (2.2) and (2.6) ■

2.5 Combining Multiple Analyses

In this section, we define how our framework automatically combines several modular analyses, while still allowing mutually beneficial interactions.

2.5.1 Definition

The *Composition of k Integrated Analyses*, or a *Composed Analysis* for short, is a tuple $\mathcal{CA} = (\mathcal{IA}_1, \mathcal{IA}_2, \ldots, \mathcal{IA}_k)$, where each $\mathcal{IA}_i$ is an integrated analysis $(D_i, \sqcup_i, \sqcap_i, \sqsubseteq_i, \top_i, \bot_i, \alpha_i, FR_i)$.

Here again, we define the meaning of a composed analysis in terms of an *associated AT-analysis*. Given a composed analysis $\mathcal{CA} = (\mathcal{IA}_1, \mathcal{IA}_2, \ldots, \mathcal{IA}_k)$, we define the associated AT-analysis $\mathcal{AT}_{\mathcal{CA}}$ as $(\mathcal{A}, R)$, where $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$. We first define the lattice $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ of the composed analysis, then we define the composed abstraction function α, the composed flow function F and finally the composed replacement function R.

Composed lattice. The lattice $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$ of the composed analysis is the product of the individual lattices, namely:

- $D \triangleq D_1 \times D_2 \times \ldots \times D_k$

- $\sqcup$ is defined by $(a_1, \ldots, a_k) \sqcup (b_1, \ldots, b_k) \triangleq (a_1 \sqcup_1 b_1, \ldots, a_k \sqcup_k b_k)$
- $\sqcap$ is defined similarly to $\sqcup$
- $\sqsubseteq$ is defined by $(a_1, \ldots, a_k) \sqsubseteq (b_1, \ldots, b_k) \triangleq a_1 \sqsubseteq_1 b_1 \wedge \ldots \wedge a_k \sqsubseteq_k b_k$
- $\top \triangleq (\top_1, \top_2, \ldots, \top_k)$ and $\bot \triangleq (\bot_1, \bot_2, \ldots, \bot_k)$

Composed abstraction function. The abstraction function $\alpha : D_c \to D$ is defined by $\alpha(c) \triangleq (\alpha_1(c), \ldots, \alpha_k(c))$.

Composed flow function. Before defining F, we must first introduce two helper functions, $c2s$ and $s2c$. The first function, $c2s$ (which stands for "composed to single"), is used to extract the dataflow values of an individual analysis from the dataflow values of the composed analysis. Given an integer i, and an n-tuple of k-tuples, $c2s$ returns an n-tuple whose elements are the i^{th} entries of each k-tuple. Formally:

$$c2s(i, (x_1, \ldots, x_n)) \triangleq (x_1[i], \ldots, x_n[i])$$

For example, if $ds \in D^*$ is a tuple of input values to a node in the composed analysis, then $c2s(i, ds)$ is the tuple of input values to that node for the i^{th} component analysis.

The second function, $s2c$ (which stands for "single to composed") has the exact opposite role as $c2s$: it combines the dataflow values of individual analyses to form the dataflow values of the composed analysis. Formally, it is defined by

$$s2c(x_1, \ldots, x_k) \triangleq ((x_1[1], \ldots, x_k[1]), \ldots, (x_1[n], \ldots, x_k[n]))$$

where each x_i is an n-tuple. For example, if $x_1, \ldots, x_k$ are n-tuples, each one being the result of a single analysis for the n output edges of a given node, then $s2c(x_1, \ldots, x_k)$ is the output tuple of the composed analysis for that node. Also, note that $c2s(i, s2c(x_1, \ldots, x_k)) = x_i$.

We are now ready to give the definition of F:

$$F(n, ds) = s2c(res_1, \ldots, res_k)$$

where for each $i \in [1..k]$:

$$res_i = lres_i \sqcap_i \bigsqcap_{g \in gs_i}{}_i \; c2s(i, SolveSubGraph_F(g, ds))$$

$$lres_i = \begin{cases} fres_i & \text{if } fres_i \in D_i^* \\ c2s(i, SolveSubGraph_F(fres_i, ds)) & \text{otherwise} \end{cases}$$

$$fres_i = FR_i(n, c2s(i, ds))$$

$$gs_i = Graph \cap \bigcup_{j \in [1..k] \wedge j \neq i} \{fres_j\}$$

and

$$SolveSubGraph_F(g, ds) = \overrightarrow{S_{\mathcal{A}}(g, ds)}(OutEdges_g)$$

The above definition looks daunting but it is in fact quite simple. To compute the result res_i of the i^{th} analysis, the composed flow function first determines what the i^{th} analysis would do in isolation by evaluating

FR_i and storing the result in $fres_i$. The next step is to determine the dataflow value $lres_i$ that results from the selected action. $lres_i$ is either $fres_i$ if $fres_i$ is a tuple of dataflow values, or the result of a recursive analysis if $fres_i$ is a replacement graph. Finally, the expression for res_i takes into account not only the action of the i^{th} analysis, through the $lres_i$ term, but also the actions of *other* analyses, through the second term. This second term of res_i computes the result of the i^{th} analysis on all graph replacements (gs_i) selected by *other* analyses. Because graph replacements are required to be sound with respect to the concrete semantics, they are sound to apply for any analysis, not just the one that selected them. This means that the result produced by the i^{th} analysis on any graph replacement is sound given the current dataflow approximation. Doing a meet of these recursive results, each of which is sound, provides the most optimistic inference that can be soundly drawn.

This last term in the definition of res_i is important for two reasons. First, it allows analyses to communicate implicitly through graph replacements. If one analysis makes a transformation, then the chosen graph replacement will immediately be seen by other analyses. Second, it ensures a certain precision guarantee. Because all potential graph replacements are recursively analyzed, and the returned value is the most optimistic inference that can be drawn from these recursive results, we are guaranteed to get results which are at least as good as any inter-leaving of the individual analyses. Although analyzing all potential graph replacements is theoretically required to ensure this precision result, an implementation could choose to recursively analyze only a subset of the potential graph replacements. The Vortex and Whirlwind implementations in fact only analyze those graph replacements selected by the $PICK$ function defined below.

Composed replacement function. The definition of R relies on a cost function $PICK : 2^{Graph} \rightarrow Graph \cup \{\epsilon\}$ to select which graph replacement to apply if more than one analysis selects a transformation. Although the composed flow function recursively analyses all graph replacements, only one of these graphs can actually be applied once the analysis has reached fixed point. The $PICK$ function is used to make this decision: given a set of graphs, $PICK$ selects at most one of them to apply, which means that if $PICK(gs) = g$, then either $g = \epsilon$, or $g \in gs$. R can now be defined as follows:

$$R(n, ds) = \begin{cases} \epsilon & \text{if } PICK(gs) = \epsilon \\ SolveSubGraph_R(PICK(gs), ds) & \text{otherwise} \end{cases}$$

$$\text{where } gs = Graph \cap \bigcup_{j \in [1..k]} \{FR_j(n, c2s(j, ds))\}$$

$$SolveSubGraph_R(g, ds) = T(R, g, S_{\mathcal{A}}(g, ds))$$

This definition of R is very similar to the one for an integrated analysis from section 2.4.1, except that here the $PICK$ function selects which graph to apply from the set (gs) of all potential replacement graphs. If $PICK$ selects no transformation, then R does the same. If, however, $PICK$ chooses a replacement graph, then this replacement graph is returned after transformations have been applied to it recursively.

2.5.2 Soundness

A composed analysis $\mathcal{CA}$ is sound if the associated AT-analysis $\mathcal{AT}_{\mathcal{CA}}$ is sound. We say that a composed analysis $\mathcal{CA} = (\mathcal{IA}_1, \mathcal{IA}_2, \ldots, \mathcal{IA}_k)$ is *locally sound* if each integrated analysis $\mathcal{IA}_i$ is *locally sound* (according to definition 14).

Theorem 5 *If a composed analysis $\mathcal{CA}$ is locally sound, then the associated AT-analysis $\mathcal{AT}_{\mathcal{CA}}$ is sound, and therefore $\mathcal{CA}$ is sound.*

Theorem 5 says that if each integrated analysis has been shown to be sound (by showing that each one is locally sound), then the composed analysis is sound. Proving Theorem 5 involves showing that if each FR_i satisfies properties (2.24) and (2.25), then F and R as defined in section 2.5.1 satisfy properties (2.2) and (2.6) respectively. Throughout this subsection, F and R will always refer to the definitions from section 2.5.1.

Again, the proof is by induction on the graph nesting structure. The base case is the one where all analyses choose to propagate dataflow information, instead of doing a graph transformation. The inductive case is the one where at least one analysis selects a transformation. The inductive hypothesis is that the property of interest holds in all selected replacement graphs, and our goal is to show that the property holds at the current node.

Proof that F satisfies property (2.2)

Base case. We are in the case where no graph replacements are chosen, which means that $\forall i.fres_i \in D_i^*$ and $\forall i.gs_i = \emptyset$. As a result, $res_i = FR_i(n, c2s(i, ds))$, and then property (2.24) for each one of the FR_i together imply property (2.2).

Inductive case. This is the case where at least one analysis selects a transformation. The inductive hypothesis is that F satisfies property (2.2) for nodes in any replacement graph $g = FR_i(n, ds)$, and we need to show that F satisfies property (2.2) at the current node n. To do this, we assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, and show $\overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq F(n, ds)$, or $\forall i.\overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i res_i$. We also let gs be the set of all possible graph replacements, in other words $gs = Graph \cap \bigcup_{j\in[1..k]}\{FR_j(n, c2s(j, ds))\}$.

The proof proceeds with 3 claims that we prove individually, and which are then used together to arrive at the required result.

Claim:

$$\forall g \in gs.F_c(n, cs) \sqsubseteq \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g) \tag{2.27}$$

Proof: Let $g \in gs$ be the graph returned by an analysis, say the i^{th} analysis, so that $g = FR_i(n, c2s(i, ds))$.

Since $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, we have $\overrightarrow{\alpha_i}(cs) \sqsubseteq_i c2s(i, ds)$. Also, since the integrated analyses are all sound, FR_i satisfies property (2.25), which can then be used to get:

$$F_c(n, cs) \sqsubseteq \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)$$

□

Claim:

$$\begin{aligned}&\forall i.\forall g \in gs.\\ &\quad \overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i c2s(i, SolveSubGraph_F(g, ds))\end{aligned} \tag{2.28}$$

Proof: Let $g \in gs$. By the induction hypothesis, F satisfies property (2.2) in g. Since $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, we can then use Theorem 1 (with $\iota_c = cs$ and $\iota_a = ds$) to get:

$$\begin{array}{lc}
 & \widetilde{\alpha}(S_{\mathcal{C}}(g, cs)) \sqsubseteq S_{\mathcal{A}}(g, ds) \\
\Leftrightarrow & \overrightarrow{\alpha}(\overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)) \sqsubseteq \overrightarrow{S_{\mathcal{A}}(g, ds)}(OutEdges_g) \\
\Leftrightarrow & \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq \overrightarrow{S_{\mathcal{A}}(g, ds)}(OutEdges_g) \\
 & \text{(using (2.27) and the monotonicity of } \alpha\text{)} \\
\Leftrightarrow & \overrightarrow{\alpha}(F_c(n, cs)) \sqsubseteq SolveSubGraph_F(g, ds) \\
 & \text{(using def of } F\text{)} \\
\Leftrightarrow & \overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i c2s(i, SolveSubGraph_F(g, ds)) \\
 & \text{(projecting onto } i^{th} \text{ analysis)}
\end{array}$$

□

Claim:

$$\forall i. \overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i lres_i \tag{2.29}$$

Proof: There are two cases in the definition of $lres_i$, one where $fres_i \in D_i^*$, and one where $fres_i \in Graph$. If $fres_i \in D_i^*$, then $lres_i = FR_i(n, c2s(i, ds))$. Furthermore, we have $\overrightarrow{\alpha_i}(cs) \sqsubseteq_i c2s(i, ds)$ because of $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and then property (2.24) of FR_i implies $\overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i lres_i$. If $fres_i \in Graph$ then $lres_i = c2s(i, SolveSubGraph_f(fres_i, ds))$. Claim (2.28) then implies $\overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i lres_i$.

□

Now we are ready to prove $\forall i. \overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i res_i$. For the i^{th} analysis, from property (2.28) and the fact that $gs_i \subseteq gs$ we have, for any candidate replacement graph $g \in gs_i$:

$$\overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i c2s(i, SolveSubGraph_F(g, ds))$$

and from property (2.29), we have:

$$\overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i lres_i$$

Since $x \sqsubseteq a \wedge x \sqsubseteq b$ implies $x \sqsubseteq a \sqcap b$, we get:

$$\overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i lres_i \sqcap_i \bigsqcap_{g \in gs_i}{}_i c2s(i, SolveSubGraph_F(g, ds))$$

or $\overrightarrow{\alpha_i}(F_c(n, cs)) \sqsubseteq_i res_i$, which is what we wanted to show.

■

Proof that R satisfies property (2.6)

Base case. We are in the first case of the definition of R, the one where $PICK(gs) = \epsilon$. We therefore have $R(n, ds) = \epsilon$, and then property (2.6) holds trivially because the antecedent $R(n, ds) = g$ is false.

Inductive case. We are in the second case of the definition of R, the one where $PICK(gs) \in Graph$. The induction hypothesis is that property (2.6) holds for all graphs in gs, and in particular, that it holds for $g = PICK(gs)$. We now want to prove that property (2.6) holds at the current node n. To do this, we assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, we let $r = T(R, g, S_{\mathcal{A}}(g, ds)$, which implies that $r = R(n, ds)$, and we want to show that

$F_c(n, cs) = \overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r)$. Note that the g in (2.6) has been replaced here with r since it is r which equals $R(n, ds)$. g in our case stands for $PICK(gs)$.

From property (2.27), we have that:

$$F_c(n, cs) \sqsubseteq \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g) \tag{2.30}$$

By the inductive hypothesis, R satisfies property (2.6) for the nodes in g. In addition, we have already shown that F satisfies property (2.2), and we also know that $\overrightarrow{\alpha}(cs) \sqsubseteq ds$. We can therefore use Theorem 2 (with $\iota_c = cs$ and $\iota_a = ds$) to get:

$$\overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r) \sqsupseteq \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)$$

Combined with (2.30), this gives the required result:

$$F_c(n, cs) \sqsubseteq \overrightarrow{S_{\mathcal{C}}(r, cs)}(OutEdges_r)$$

■

2.5.3 Termination

Termination is handled in a similar way to the case of integrated analyses from section 2.4.3. The only difference is that the analysis designer must now show that the *composed analysis* does not cause endless recursive graph replacements. Even if each integrated analysis by itself does not cause infinite recursive analysis, the interaction between two analyses can. For example, two analyses can oscillate back and forth, the first one optimizing a statement that the second one reverts back to the original form. However, as long as the lattice has finite height, our framework does guarantee that non-termination will never be caused by infinite traversal of the lattice.

Chapter 3

Forward analyses and transformations

In this section, we assume that we are dealing with the following "forward" Rhodium program P:

```
define edge fact $EF_1(\ldots)$ with meaning $M_1$
...
define edge fact $EF_k(\ldots)$ with meaning $M_k$

$PR_1$
...
$PR_l$

$TR_1$
...
$TR_m$
```

TR_i above is a transformation rule. We assume that all node facts and all edge facts without meanings have been macro-expanded, so we are only left with edge facts that have meanings.

We also assume that all transformation and propagation rules are forward. In particular, each PR_i has the form `if` ψ `then` $EF_j(\ldots)$@`cfg_out`$[h]$, where ψ only refers to CFG input edges, and each TR_i has the form `if` ψ `then transform` s, where ψ only refers to CFG input edges.

3.1 Abstract semantics

We define the meaning of P using an AT-analysis $(\mathcal{A}_P, R_P)$. We first define the analysis $\mathcal{A}_P$, and then the replacement function R. In this section we only deal with intraprocedural analyses. Chapter 6 will show how Rhodium handles interprocedural analyses.

3.1.1 Analysis

We assume that the analysis is running on procedure p. The analysis for P is $\mathcal{A}_P = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, F, \alpha)$. The domain D of the analysis is:

$$D = 2^{Facts}$$

where

$$Facts = \{EF_i(t_1, \ldots, t_j) \mid i \in [1..k] \wedge t_1 \in Term \wedge \ldots \wedge t_j \in Term\}$$

The lattice of the analysis is:

$$(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot) = (D, \cap, \cup, \supseteq, \emptyset, Facts)$$

The flow function $F : Node \times D^* \to D^*$ gives the interpretation of nodes. Given a tuple of input dataflow values, one per incoming edge to the node, F produces a tuple of output dataflow values, one per outgoing edge from the node. It is defined as follows (where $stmtAt(n)$ denotes the statement at CFG node n):

$$\begin{aligned} F(n, ds)[h] = \{\theta(EF(t_1, \ldots, t_j)) \mid \exists i, \psi.\ \ & PR_i = (\texttt{if}\ \psi\ \texttt{then}\ EF(t_1, \ldots, t_j)@\texttt{cfg\_out}[h]) \wedge \\ & \theta[\![\psi]\!]^{ds}_{stmtAt(n)}\} \end{aligned} \tag{3.1}$$

The meaning of a formula ψ is given by a function $[\![\psi]\!] : Subst \times D^* \times Stmt \to bool$. We write $\theta[\![\psi]\!]^{ds}_s$ for $[\![\psi]\!](\theta, ds, s)$. $[\![\psi]\!]$ is defined as follows (we assume that case expressions have been expanded into foralls and conjunctions, and that each primitive p has a meaning $[\![p]\!]$):

$$\begin{array}{lcl}
\theta[\![\texttt{true}]\!]^{ds}_s & = & true \\
\theta[\![\texttt{false}]\!]^{ds}_s & = & false \\
\theta[\![!\ \psi]\!]^{ds}_s & = & \neg\theta[\![\psi]\!]^{ds}_s \\
\theta[\![\psi_1\ ||\ \psi_2]\!]^{ds}_s & = & \theta[\![\psi_1]\!]^{ds}_s \vee \theta[\![\psi_2]\!]^{ds}_s \\
\theta[\![\psi_1\ \&\&\ \psi_2]\!]^{ds}_s & = & \theta[\![\psi_1]\!]^{ds}_s \wedge \theta[\![\psi_2]\!]^{ds}_s \\
\theta[\![\psi_1 \Rightarrow \psi_2]\!]^{ds}_s & = & \theta[\![\psi_1]\!]^{ds}_s \Rightarrow \theta[\![\psi_2]\!]^{ds}_s \\
\theta[\![\texttt{forall}\ X : \tau.\ \psi]\!]^{ds}_s & = & \forall t : \tau.\ \ \theta[X \mapsto t][\![\psi]\!]^{ds}_s \\
\theta[\![\texttt{exists}\ X : \tau.\ \psi]\!]^{ds}_s & = & \exists t : \tau.\ \ \theta[X \mapsto t][\![\psi]\!]^{ds}_s \\
\theta[\![EF(t_1, \ldots, t_j)@\texttt{cfg\_in}[h]]\!]^{ds}_s & = & EF(\theta[\![t_1]\!]_s, \ldots, \theta[\![t_j]\!]_s) \in ds[h] \\
\theta[\![t_1 == t_2]\!]^{ds}_s & = & \theta[\![t_1]\!]_s = \theta[\![t_2]\!]_s \\
\theta[\![PrimPredSymbol(t_1, \ldots, t_j)]\!]^{ds}_s & = & [\![PrimPredSymbol]\!](\theta[\![t_1]\!]_s, \ldots, \theta[\![t_j]\!]_s)
\end{array}$$

where the semantics of a term t at a statement s under substitution θ is defined by:

$$\begin{array}{lcl}
\theta[\![\texttt{currStmt}]\!]_s & = & s \\
\theta[\![PrimFunctionSymbol(t_1, \ldots, t_j)]\!]_s & = & [\![PrimFunctionSymbol]\!](\theta[\![t_1]\!]_s, \ldots, \theta[\![t_j]\!]_s) \\
\theta[\![WilExpr]\!]_s & = & \theta(WilExpr)
\end{array}$$

The abstraction function α will be defined later.

3.1.2 Replacement function

The replacement function R is defined by:

$$R(n, ds) = \begin{cases} \mathit{singleNodeGraph}(n, \theta(s)) & \text{if } \exists i, \psi, \theta.\ \Big[TR_i = (\texttt{if } \psi \texttt{ then transform } s) \land \theta[\![\psi]\!]^{ds}_{\mathit{stmtAt}(n)}\Big] \\ \epsilon & \text{otherwise} \end{cases} \tag{3.2}$$

where $singleNodeGraph(n, s)$ creates a sub-graph with a single node n' that has as many input and output edges as n and that satisfies $\mathit{stmtAt}(n') = s$.

3.2 Abstraction

The meaning of an edge fact declaration:

$$\texttt{define edge fact } EF(X_1 : \tau_1, \ldots, X_n : \tau_n) \texttt{ with meaning } M$$

is given by $[\![EF]\!] : \tau_1 \times \ldots \times \tau_n \times \mathit{State} \to \mathit{bool}$ and is defined by:

$$[\![EF]\!](t_1, \ldots, t_n, \eta) = \theta(M)(\eta)$$

where $\theta = [X_1 \mapsto t_1, \ldots, X_n \mapsto t_n]$, $\theta(M)$ applies the substitution θ to M, and $P(\eta)$ evaluates a predicate P at a program state η.

The abstraction function $\alpha : D_c \to D$ is defined as:

$$\alpha(\eta s) = \{EF_i(t_1, \ldots, t_j) \mid 1 \le i \le k \land \forall \eta \in \eta s. [\![EF_i]\!](t_1, \ldots, t_j, \eta)\} \tag{3.3}$$

3.3 Conditions for soundness

These are local soundness conditions, which we ask the theorem prover to discharge. The conditions below work for arbitrary number of inputs and outputs to a node.

3.3.1 Propagation rule

Definition 15 *A forward propagation rule* `if` ψ `then` $EF(t_1, \ldots, t_n)$`cfg_out`$[h']$ *is said to be sound iff the following condition holds:*

$$\begin{array}{l} \forall (n, \eta, \eta', h, h', ds, \theta) \in \mathit{Node} \times \mathit{State}^2 \times \mathit{Natural}^2 \times D^* \times \mathit{Subst}. \\ \quad \left[\begin{array}{l} \theta[\![\psi]\!]^{ds}_{\mathit{stmtAt}(n)} \land \\ h, \eta \overset{n}{\hookrightarrow} h', \eta' \land \\ \mathit{allMeaningsHold}(ds[h], \eta) \end{array}\right] \quad \Rightarrow \quad [\![EF]\!](\theta(t_1), \ldots, \theta(t_n), \eta') \end{array} \tag{fwd-prop-sound}$$

where *allMeaningsHold* is defined as follows:

$$\begin{aligned} \mathit{allMeaningsHold}(d,\eta) \triangleq\ & \forall(EF,t_1,\ldots,t_j) \in \mathit{EdgeFact} \times \mathit{Term}^j . \\ & \qquad EF(t_1,\ldots,t_j) \in d \Rightarrow [\![EF]\!](t_1,\ldots,t_j,\eta) \end{aligned}$$

Intuitively, the above condition says the following. Suppose the propagate rule fires at a node n on some incoming facts ds. For every index h into ds, we assume that some state η on input edge h steps through n to η' on output edge h'. Further, we assume that the meanings of all the dataflow facts in $ds[h]$ hold of the state η. It then better be the case that the meaning of the propagated dataflow fact holds of η'.

The condition (fwd-prop-sound) can be restated as follows:

$$\begin{array}{l} \forall(n,\eta,\eta',h,h',\theta) \in \mathit{Node} \times \mathit{State}^2 \times \mathit{Natural}^2 \times \mathit{Subst}. \\ \quad \left[\begin{array}{l} \{\!\!\{\psi\}\!\!\}(\theta,n,h,\eta)\ \wedge \\ h,\eta \overset{n}{\hookrightarrow} h',\eta' \end{array}\right] \quad \Rightarrow \quad [\![EF]\!](\theta(t_1),\ldots,\theta(t_n),\eta') \end{array}$$

where $\{\!\!\{\psi\}\!\!\} : \mathit{Subst} \times \mathit{Node} \times \mathit{Natural} \times \mathit{State}$ is defined by:

$$\{\!\!\{\psi\}\!\!\}(\theta,n,h,\eta) \triangleq \exists\ ds \in D^*\ .\ (\theta[\![\psi]\!]^{ds}_{\mathit{stmtAt}(n)} \wedge \mathit{allMeaningsHold}(ds[h],\eta))$$

3.3.2 Transformation rule

Definition 16 *A forward transformation rule* `if` ψ `then transform` s *is said to be sound iff the following condition holds:*

$$\begin{array}{l} \forall(n,n',\eta,\eta',h,h',ds,\theta) \in \mathit{Node}^2 \times \mathit{State}^2 \times \mathit{Natural}^2 \times D^* \times \mathit{Subst}. \\ \quad \left[\begin{array}{l} \theta[\![\psi]\!]^{ds}_{\mathit{stmtAt}(n)}\ \wedge \\ h,\eta \overset{n}{\hookrightarrow} h',\eta'\ \wedge \\ \mathit{stmtAt}(n') = \theta(s)\ \wedge \\ \mathit{allMeaningsHold}(ds[h],\eta) \end{array}\right] \quad \Rightarrow \quad h,\eta \overset{n'}{\hookrightarrow} h',\eta' \end{array} \qquad \text{(fwd-trans-sound)}$$

Intuitively, the above condition says the following. Suppose the transformation rule fires at a node n on some incoming facts ds. For every index h into ds, we assume that some state η on input edge h steps through n to η' on output edge h'. Then η on edge h should also step to η' on edge h' through the transformed node n'.

3.4 Metatheory

3.4.1 Monotonicity

We have the following theorem about the monotonicity of F.

Theorem 6 (monotonicity of F) *If the syntactic form* ψ_1 `=>` ψ_2 *is disallowed, and the syntactic form* `!`ψ *is allowed only if* ψ *is an equality (i.e.* t_1 `==` t_2*) or a primitive (i.e.* $\mathit{PrimPredSymbol}(t_1,\ldots,t_j)$*) then* F *is monotonic.*

Theorem 6 can be used to implement a simple syntactic check that guarantees the monotonicity of F. For each rule, transform the antecedent as follows: convert ψ_1 `=>` ψ_2 to `!`ψ_1 `||` ψ_2, and then push all the negations to the inside in the standard way (through conjunctions, disjunctions, existentials, and universals). If after this conversion all the antecedents are in the form required by theorem 6, then F is guaranteed to be monotonic.

Proof of theorem 6

We assume the syntactic restrictions on the antecedents from theorem 6 and we need to show:

$$\forall\ n, ds_1, ds_2\ .\ ds_1 \sqsubseteq ds_2 \Rightarrow F(n, ds_1) \sqsubseteq F(n, ds_2)$$

Or, equivalently:

$$\forall\ n, ds_1, ds_2, h\ .\ ds_1 \sqsubseteq ds_2 \Rightarrow F(n, ds_1)[h] \sqsubseteq F(n, ds_2)[h]$$

Using the definition of $\sqsubseteq$, we need to show:

$$\forall\ n, ds_1, ds_2, h\ .\ ds_1 \sqsubseteq ds_2 \Rightarrow F(n, ds_1)[h] \supseteq F(n, ds_2)[h]$$

Using the definition of F, it is sufficient to show:

$$\forall\ \psi, ds_1, ds_2, \theta, s\ .\ (ds_1 \sqsubseteq ds_2 \wedge \theta[\![\psi]\!]_s^{ds_2}) \Rightarrow \theta[\![\psi]\!]_s^{ds_1}$$

Let $P(\psi) = \forall\ ds_1, ds_2, \theta, s\ .\ (ds_1 \sqsubseteq ds_2 \wedge \theta[\![\psi]\!]_s^{ds_2}) \Rightarrow \theta[\![\psi]\!]_s^{ds_1}$. We need to show $\forall \psi\ .\ P(\psi)$. We do this by induction on the syntactic form of ψ.

- Cases where ψ is `true`, `false`, t_1 `==` t_2, or $\mathit{PrimPredSymbol}(t_1, \ldots, t_j)$

  In all these cases $\theta[\![\psi]\!]_s^{ds}$ does not depend on ds.

  Therefore $\theta[\![\psi]\!]_s^{ds_1} = \theta[\![\psi]\!]_s^{ds_2}$, and so $P(\psi)$ holds.

- Case `!`ψ

  We need to show $P(!\psi)$.

  Because of the syntactic restrictions mentioned in theorem 6, this case only occurs with $\psi = (t_1{==}t_2)$ or $\psi = \mathit{PrimPredSymbol}(t_1, \ldots, t_j)$.

  In both of these cases, $\theta[\![\psi]\!]_s^{ds}$ does not depend on ds, so that $\theta[\![\psi]\!]_s^{ds_1} = \theta[\![\psi]\!]_s^{ds_2}$.

  This implies $\neg\theta[\![\psi]\!]_s^{ds_1} = \neg\theta[\![\psi]\!]_s^{ds_2}$.

  By the the definition of $\theta[\![\,\cdot\,]\!]_s^{ds}$ this implies $\theta[\![!\psi]\!]_s^{ds_1} = \theta[\![!\psi]\!]_s^{ds_2}$.

  And therefore $P(!\psi)$ holds.

- Case ψ_1 `||` ψ_2

  By the induction hypothesis we know $P(\psi_1)$ and $P(\psi_2)$, and we need to show $P(\psi_1$ `||` $\psi_2)$.

  To show $P(\psi_1$ `||` $\psi_2)$, we assume:

$$ds_1 \sqsubseteq ds_2 \wedge \theta[\![\psi_1 \ \texttt{||} \ \psi_2]\!]_s^{ds_2}$$

and show:

$$\theta[\![\psi_1 \ \texttt{||} \ \psi_2]\!]_s^{ds_1}$$

By the definition of $\theta[\![\cdot]\!]_s^{ds}$ We have:

$$\begin{array}{rcl}\theta[\![\psi_1 \ \texttt{||} \ \psi_2]\!]_s^{ds_2} & = & \theta[\![\psi_1]\!]_s^{ds_2} \vee \theta[\![\psi_2]\!]_s^{ds_2} \\ \theta[\![\psi_1 \ \texttt{||} \ \psi_2]\!]_s^{ds_1} & = & \theta[\![\psi_1]\!]_s^{ds_1} \vee \theta[\![\psi_2]\!]_s^{ds_1}\end{array}$$

Let $A = \theta[\![\psi_1]\!]_s^{ds_2}$, $B = \theta[\![\psi_2]\!]_s^{ds_2}$, $C = \theta[\![\psi_1]\!]_s^{ds_1}$, and $D = \theta[\![\psi_2]\!]_s^{ds_1}$.

Thus we are assuming $ds_1 \sqsubseteq ds_2$ and $A \vee B$ and trying to show $C \vee D$. We do case analysis on whether or not A holds.

- Case where A holds. Then by $ds_1 \sqsubseteq ds_2$ and $P(\psi_1)$, we get that C holds, and so $C \vee D$.
- Case where A does not hold. Then B must hold. Then by $ds_1 \sqsubseteq ds_2$ and $P(\psi_2)$, we get that D holds, and so $C \vee D$.

- Case ψ_1 `&&` ψ_2

  By the induction hypothesis we know $P(\psi_1)$ and $P(\psi_2)$, and we need to show $P(\psi_1 \ \texttt{\&\&} \ \psi_2)$.

  To show $P(\psi_1 \ \texttt{\&\&} \ \psi_2)$, we assume:

$$ds_1 \sqsubseteq ds_2 \wedge \theta[\![\psi_1 \ \texttt{\&\&} \ \psi_2]\!]_s^{ds_2}$$

and show:

$$\theta[\![\psi_1 \ \texttt{\&\&} \ \psi_2]\!]_s^{ds_1}$$

By the definition of $\theta[\![\cdot]\!]_s^{ds}$ We have:

$$\begin{array}{rcl}\theta[\![\psi_1 \ \texttt{\&\&} \ \psi_2]\!]_s^{ds_2} & = & \theta[\![\psi_1]\!]_s^{ds_2} \wedge \theta[\![\psi_2]\!]_s^{ds_2} \\ \theta[\![\psi_1 \ \texttt{\&\&} \ \psi_2]\!]_s^{ds_1} & = & \theta[\![\psi_1]\!]_s^{ds_1} \wedge \theta[\![\psi_2]\!]_s^{ds_1}\end{array}$$

Let $A = \theta[\![\psi_1]\!]_s^{ds_2}$, $B = \theta[\![\psi_2]\!]_s^{ds_2}$, $C = \theta[\![\psi_1]\!]_s^{ds_1}$, and $D = \theta[\![\psi_2]\!]_s^{ds_1}$.

Thus we are assuming $ds_1 \sqsubseteq ds_2$ and $A \wedge B$ and trying to show $C \wedge D$.

Because A holds, using $ds_1 \sqsubseteq ds_2$ and $P(\psi_1)$, we get that C holds.

Because B holds, using $ds_1 \sqsubseteq ds_2$ and $P(\psi_2)$, we get that D holds.

Thus $C \wedge D$ holds.

- Case $\psi_1 \Rightarrow \psi_2$

  Because of the syntactic restrictions mentioned in theorem 6, this case does not occur.

- Case $\texttt{forall}\ X:\tau\ .\ \psi$

  By the induction hypothesis, we know $P(\psi)$, and we need to show $P(\texttt{forall}\ X:\tau\ .\ \psi)$.

  To show $P(\texttt{forall}\ X:\tau\ .\ \psi)$, we assume:

  $$ds_1 \sqsubseteq ds_2 \wedge \theta[\![\texttt{forall}\ X:\tau\ .\ \psi]\!]_s^{ds_2}$$

  and show:

  $$\theta[\![\texttt{forall}\ X:\tau\ .\ \psi]\!]_s^{ds_1}$$

  By the definition of $\theta[\![\cdot]\!]_s^{ds}$ We have:

  $$\begin{array}{lcl}\theta[\![\texttt{forall}\ X:\tau\ .\ \psi]\!]_s^{ds_2} & = & \forall\ t:\tau\ .\ \theta[X\mapsto t][\![\psi]\!]_s^{ds_2}\\ \theta[\![\texttt{forall}\ X:\tau\ .\ \psi]\!]_s^{ds_1} & = & \forall\ t:\tau\ .\ \theta[X\mapsto t][\![\psi]\!]_s^{ds_1}\end{array}$$

  Let $A = \forall\ t:\tau\ .\ \theta[X\mapsto t][\![\psi]\!]_s^{ds_2}$ and $B = \forall\ t:\tau\ .\ \theta[X\mapsto t][\![\psi]\!]_s^{ds_1}$.

  Thus we are assuming $ds_1 \sqsubseteq ds_2$ and A and trying to show B.

  To show B, pick a $t:\tau$, and show $\theta[X\mapsto t][\![\psi]\!]_s^{ds_1}$.

  Instantiating A with t, we get $\theta[X\mapsto t][\![\psi]\!]_s^{ds_2}$.

  Using $ds_1 \sqsubseteq ds_2$ and $P(\psi)$, we get $\theta[X\mapsto t][\![\psi]\!]_s^{ds_1}$ (Note that the θ in the $P(\psi)$ quantifier gets instantiated with $\theta[X\mapsto t]$).

- Case $\texttt{exists}\ X:\tau\ .\ \psi$

  By the induction hypothesis, we know $P(\psi)$, and we need to show $P(\texttt{exists}\ X:\tau\ .\ \psi)$.

  To show $P(\texttt{exists}\ X:\tau\ .\ \psi)$, we assume:

  $$ds_1 \sqsubseteq ds_2 \wedge \theta[\![\texttt{exists}\ X:\tau\ .\ \psi]\!]_s^{ds_2}$$

  and show:

  $$\theta[\![\texttt{exists}\ X:\tau\ .\ \psi]\!]_s^{ds_1}$$

  By the definition of $\theta[\![\cdot]\!]_s^{ds}$ We have:

  $$\begin{array}{lcl}\theta[\![\texttt{exists}\ X:\tau\ .\ \psi]\!]_s^{ds_2} & = & \exists\ t:\tau\ .\ \theta[X\mapsto t][\![\psi]\!]_s^{ds_2}\\ \theta[\![\texttt{exists}\ X:\tau\ .\ \psi]\!]_s^{ds_1} & = & \exists\ t:\tau\ .\ \theta[X\mapsto t][\![\psi]\!]_s^{ds_1}\end{array}$$

  Let $A = \exists\ t:\tau\ .\ \theta[X\mapsto t][\![\psi]\!]_s^{ds_2}$ and $B = \exists\ t:\tau\ .\ \theta[X\mapsto t][\![\psi]\!]_s^{ds_1}$.

  Thus we are assuming $ds_1 \sqsubseteq ds_2$ and A and trying to show B.

  From A we know there exists t such that $\theta[X\mapsto t][\![\psi]\!]_s^{ds_2}$.

  Using $ds_1 \sqsubseteq ds_2$ and $P(\psi)$, we get $\theta[X\mapsto t][\![\psi]\!]_s^{ds_1}$ (Note that the θ in the $P(\psi)$ quantifier gets instantiated with $\theta[X\mapsto t]$).

  Thus B holds, with t being the witness to the existential quantifier in B.

- Case $EF(t_1, \ldots, t_j)@\texttt{cfg\_in}[h]$

  We need to show $P(EF(t_1, \ldots, t_j)@\texttt{cfg\_in}[h])$.

  To show this, assume:

$$ds_1 \sqsubseteq ds_2 \wedge \theta[\![EF(t_1, \ldots, t_j)@\texttt{cfg\_in}[h]]\!]_s^{ds_2}$$

  and show:

$$\theta[\![EF(t_1, \ldots, t_j)@\texttt{cfg\_in}[h]]\!]_s^{ds_1}$$

  By the definition of $\theta[\![\cdot]\!]_s^{ds}$ We have:

$$\begin{array}{rcl}\theta[\![EF(t_1, \ldots, t_j)@\texttt{cfg\_in}[h]]\!]_s^{ds_2} & = & EF(\theta[\![t_1]\!]_s, \ldots, \theta[\![t_j]\!]_s) \in ds_2[h] \\ \theta[\![EF(t_1, \ldots, t_j)@\texttt{cfg\_in}[h]]\!]_s^{ds_1} & = & EF(\theta[\![t_1]\!]_s, \ldots, \theta[\![t_j]\!]_s) \in ds_1[h]\end{array}$$

  Let $A = EF(\theta[\![t_1]\!]_s, \ldots, \theta[\![t_j]\!]_s) \in ds_2[h]$ and $B = EF(\theta[\![t_1]\!]_s, \ldots, \theta[\![t_j]\!]_s) \in ds_1[h]$.

  Thus we are assuming $ds_1 \sqsubseteq ds_2$ and A and trying to show B.

  Since $ds_1 \sqsubseteq ds_2$, we have $ds_1[h] \sqsubseteq ds_2[h]$.

  By the definition of $\sqsubseteq$, we get $ds_1[h] \supseteq ds_2[h]$.

  $ds_1[h] \supseteq ds_2[h]$ combined with A then gives B.

■

3.4.2 Soundness

We need to show that if conditions (fwd-prop-sound) and (fwd-trans-sound) hold for all propagation rules and transformation rules, and we perform all the transformations, then the concrete fixed point of the original procedure and the concrete fixed point of the transformed procedure are the same. This condition is stated in the following theorem:

Theorem 7 (main forward soundness) *If all propagation rules and transformation rules in a forward Rhodium program P are sound, then the induced AT-analysis $(\mathcal{A}_P, R_P)$ is sound.*

Theorem 7 follows from the following two lemmas and theorem 2:

Lemma 3 *If all propagation rules in a forward Rhodium program P are sound then F as defined in* (3.1) *is sound.*

Lemma 4 *If all transformation rules in a forward Rhodium program P are sound, then R as defined in* (3.2) *is sound.*

All we need to do now is prove lemmas 3 and 4. Before doing this, we establish the following helper lemma:

Lemma 5

$$\forall(\eta, cs, ds, h) \in \mathit{State} \times D_c^* \times D \times \mathit{Natural}\ .\ (\overrightarrow{\alpha}(cs) \sqsubseteq ds \wedge \eta \in cs[h]) \Rightarrow \mathit{allMeaningsHold}(ds[h], \eta)$$

Proof of lemma 5

We assume ($\overrightarrow{\alpha}(cs) \sqsubseteq ds \wedge \eta \in cs[h]$), and we need to show:

$$\begin{array}{l} \forall (EF, t_1, \ldots, t_j) \in \mathit{EdgeFact} \times \mathit{Term}^j . \\ \qquad \mathit{EF}(t_1, \ldots, t_j) \in ds[h] \Rightarrow [\![\mathit{EF}]\!](t_1, \ldots, t_j, \eta) \end{array}$$

To do this, pick $(EF, t_1, \ldots, t_j) \in \mathit{EdgeFact} \times \mathit{Term}^j$, assume $EF(t_1, \ldots, t_j) \in ds[h]$ and show:

$$[\![EF]\!](t_1, \ldots, t_j, \eta) \tag{3.4}$$

From the assumptions we know that $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, which means $\forall i \,.\, \alpha(cs[i]) \sqsubseteq ds[i]$, and using the definition of $\sqsubseteq$, $\forall i \,.\, \alpha(cs[i]) \supseteq ds[i]$. Thus, from $EF(t_1, \ldots, t_j) \in ds[h]$, we get:

$$EF(t_1, \ldots, t_j) \in \alpha(cs[h])$$

Using the definition of α, this means that:

$$\forall \eta \in cs[h] \,.\, [\![EF]\!](t_1, \ldots, t_j, \eta)$$

From the assumptions we know that $\eta \in cs[h]$, and thus we get $[\![EF]\!](t_1, \ldots, t_j, \eta)$, which is what we had to show in (3.4).

■

Proof of lemma 3

We need to show:

$$\begin{array}{l} \forall (n, cs, ds) \in \mathit{Node} \times D_c^* \times D^* \\ \qquad \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \overrightarrow{\alpha}(F_c(n, ds)) \sqsubseteq F(n, ds) \end{array}$$

Pick $(n, cs, ds) \in \mathit{Node} \times D_c^* \times D^*$, assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$, and show $\overrightarrow{\alpha}(F_c(n, ds)) \sqsubseteq F(n, ds)$.

To show $\overrightarrow{\alpha}(F_c(n, ds)) \sqsubseteq F(n, ds)$, we need to show that $\forall\ h' \,.\, \alpha(F_c(n, ds)[h']) \sqsubseteq F(n, ds)[h']$.

So pick h', and show $\alpha(F_c(n, ds)[h']) \sqsubseteq F(n, ds)[h']$, which, using the definition of $\sqsubseteq$ is $\alpha(F_c(n, ds)[h']) \supseteq F(n, ds)[h']$. To show this, pick $x \in F(n, ds)[h']$, and show that:

$$x \in \alpha(F_c(n, ds)[h']) \tag{3.5}$$

Using the definition of F from (3.1), $x \in F(n, ds)[h']$ implies that there is a θ, ψ, i such that:

$$x = \theta(EF(t_1, \ldots, t_j)) \tag{3.6}$$

$$PR_i = (\texttt{if}\ \psi\ \texttt{then}\ EF(t_1, \ldots, t_j) @ \texttt{cfg\_out}[h]) \tag{3.7}$$

$$\theta [\![\psi]\!]^{ds}_{\mathit{stmtAt}(n)} \tag{3.8}$$

Using the definition of α from (3.3), we get:

$$\alpha(F_c(n, ds)[h']) = \{EF_i(t_1, \ldots, t_j) \mid 1 \leq i \leq k \wedge \forall \eta' \in F_c(n, ds)[h'] \ . \ [\![EF_i]\!](t_1, \ldots, t_j, \eta')\} \tag{3.9}$$

To show (3.5), because of (3.6), we must show that $\theta(EF(t_1, \ldots, t_j)) \in \alpha(F_c(n, ds)[h'])$. Using (3.9), this amounts to showing:

$$\forall \eta' \in F_c(n, ds)[h'] \ . \ [\![EF]\!](\theta(t_1), \ldots, \theta(t_j), \eta')$$

Pick $\eta' \in F_c(n, ds)[h']$, and show:

$$[\![EF]\!](\theta(t_1), \ldots, \theta(t_j), \eta') \tag{3.10}$$

From the definition of F_c in (2.1), we know that there exists η, h such that:

$$\eta \in cs[h] \tag{3.11}$$

$$h, \eta \overset{n}{\hookrightarrow} h', \eta' \tag{3.12}$$

Because all propagation rules in the Rhodium program P are sound, we have that PR_i satisfies (fwd-prop-sound).

We instantiate (fwd-prop-sound): the first two conditions of the antecedent are met from (3.8) and (3.12), and the third condition of the antecedent follows from (3.11), $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and lemma 5. By instantiating (fwd-prop-sound), we get (3.10), which is what we had to show.

■

Proof of lemma 4

We need to show:

$$\begin{aligned} &\forall (n, ds, g) \in Node \times D^* \times Graph. \\ &\quad R(n, ds) = g \Rightarrow \\ &\quad\quad [\forall cs \in D_c^*. \overrightarrow{\alpha}(cs) \sqsubseteq ds \Rightarrow \\ &\quad\quad\quad F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)] \end{aligned} \tag{3.13}$$

Pick $(n, ds, g) \in Node \times D^* \times Graph$, assume $R(n, ds) = g$, then pick $cs \in D_c^*$, assume $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and show:

$$F_c(n, cs) \sqsubseteq_c \overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g)$$

By the definition of R from equation (3.2), and from $R(n, ds) = g$, we know that there exists i, ψ and θ such that:

$$TR_i = (\texttt{if } \psi \texttt{ then transform } s)$$

$$g = singleNodeGraph(n, \theta(s))$$

$$\theta[\![\psi]\!]^{ds}_{stmtAt(n)} \tag{3.14}$$

$$stmtAt(n') = \theta(s) \text{ where } n' \text{ is the node from the single-node graph } g \tag{3.15}$$

Because g is a single node CFG, we have:

$$\overrightarrow{S_{\mathcal{C}}(g, cs)}(OutEdges_g) = F_c(n', ds)$$

Thus, we need to show:

$$F_c(n, cs) \sqsubseteq_c F_c(n', cs)$$

Or:

$$\forall\ k\ .F_c(n, cs)[k] \sqsubseteq_c F_c(n', cs)[k]$$

Pick a k, and show:

$$F_c(n, cs)[k] \sqsubseteq_c F_c(n', cs)[k]$$

By the definition of $\sqsubseteq$, this is:

$$F_c(n, cs)[k] \subseteq F_c(n', cs)[k]$$

To show this, pick $\eta_{out} \in F_c(n, cs)[k]$, and show:

$$\eta_{out} \in F_c(n', cs)[k] \tag{3.16}$$

Since $\eta_{out} \in F_c(n, cs)[k]$, by the definition of F_c from equation (2.1), we know that there exists $\eta_{in} \in State$ and $i \in Natural$ such that:

$$\eta_{in} \in cs[i] \tag{3.17}$$

$$i, \eta_{in} \overset{n}{\hookrightarrow} k, \eta_{out} \tag{3.18}$$

Because all transformation rules in the Rhodium program P are sound, we know that PR_i satisfies (fwd-trans-sound).

We instantiate (fwd-trans-sound): the first three conditions of the antecedent are met from (3.14), (3.15) and (3.18), and the fourth condition of the antecedent follows from (3.17), $\overrightarrow{\alpha}(cs) \sqsubseteq ds$ and lemma 5. By instantiating (fwd-trans-sound), we get get $i, \eta_{in} \overset{n'}{\hookrightarrow} k, \eta_{out}$.

Since $i, \eta_{in} \overset{n'}{\hookrightarrow} k, \eta_{out}$, and since $\eta_{in} \in cs[i]$ from (3.17), by the definition of F_c from equation (2.1), we get that $\eta_{out} \in F_c(n', cs)$, which is what we had to show in (3.16).

■

Chapter 4

Backward analyses and transformations

In this section, we assume that we are dealing with the following "backward" Rhodium program P:

```
define edge fact $EF_1(\ldots)$ with meaning $M_1$
...
define edge fact $EF_k(\ldots)$ with meaning $M_k$

$PR_1$
...
$PR_l$

$TR_1$
...
$TR_m$
```

Like in the forward case, each TR_i above is a transformation rule, and we assume that all node facts and all edge facts without meanings have been macro-expanded.

Finally, we assume that all transformation and propagation rules are backward. In particular, each PR_i has the form `if` ψ `then` $EF_j(\ldots)$@`cfg_in`$[h]$, where ψ only refers to cfg output edges, and each TR_i has the form `if` ψ `then transform` s, where ψ only refers to cfg output edges.

4.1 Abstract semantics

Again, we define the meaning of P using an AT-analysis $(\mathcal{A}_P, R_P)$. The definitions are identical to the forward case, except that `cfg_out` becomes `cfg_in` and vice-versa.

In particular, everything is the same, except for the following changes:

$$F(n, ds)[h] = \{\theta(EF(t_1, \ldots, t_j)) \mid \exists i, \psi. \quad PR_i = (\texttt{if } \psi \texttt{ then } EF(t_1, \ldots, t_j)@\texttt{cfg\_in}[h]) \wedge \theta[\![\psi]\!]^{ds}_{stmtAt(n)}\} \tag{4.1}$$

$$\theta[\![EF(t_1, \ldots, t_j)@\texttt{cfg\_out}[h]]\!]^{ds}_{s} \quad = \quad EF(\theta[\![t_1]\!]_s, \ldots, \theta[\![t_j]\!]_s) \in ds[h]$$

4.2 Concrete semantics

We use a weakest precondition concrete semantics, as described in section 3.2 of [2]. In particular, the backward concrete semantics is given in terms of an arbitrary condition C. At each program point, we want to compute the weakest condition P such that if execution is started at that program point with condition P, execution ends with condition C.

There are two notions of weakest precondition: strict weakest precondition (*wp*) and liberal weakest precondition (*wlp*). The difference between the two is in how they handle errors and termination. The strict weakest precondition of P through S is the weakest condition Q such that S terminates when started in a state satisfying Q **and** the state after executing S satisfies P. The liberal weakest precondition of P through S is the weakest condition Q such that **if** S terminates without errors when started in a state satisfying Q, **then** the state after executing S satisfies P.

The two are related in the following way:

$$wp(S, P) = wp(S, true) \wedge wlp(S, P)$$

In our formalism, we use the strict weakest precondition. The liberal version is not well suited for our purposes because it would allow the transformed program to run forever even if the original program did not. Suppose we set the end condition C to be $x = 3$. Then the programs $x := 3$ and *loop_forever* have the same liberal weakest precondition semantics.

The concrete backward analysis is defined as:

$$\mathcal{C} = (D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c, F_c, id)$$

where the lattice is:

$$(D_c, \sqcup_c, \sqcap_c, \sqsubseteq_c, \top_c, \bot_c) = (Pred, \wedge, \vee, \Leftarrow, false, true)$$

The concrete flow function F_c is defined as:

$$F_c(n, Ps)[h] = \bigvee_{i \in [1..Len(out(n))]} wp_p(Ps[i], out(n)[i], in(n)[h])$$

where $wp : Procedure \times Pred \times Edge \times Edge \rightarrow Pred$ is the strict weakest precondition function: $wp_p(P, e_{dst}, e_{src})$ is the weakest condition Q such that if the procedure p is stepped once starting at program point (edge) e_{src} in a state satisfying Q, execution will end up at program point e_{dst} in a state satisfying P.

The strict weakest precondition solution is the greatest fixed point of the precondition equations, not the least fixed point (or alternatively, it's the least fixed point in the inverted lattice).[1]

We have two options for the notion of semantics preservation based on whether or not we want to preserve non-termination. If we want to preserve non-termination, then we must use exact equality of the WP semantics to compare the original and the transformed program: the condition computed at each program point in the original program must be equal to the condition computed at the same program point in the transformed program. If we want to relax the requirement of preserving non-termination (in other words, allow programs that run forever or hit errors to be transformed in any way), then we want implication of the WP semantics: the condition in the original program must imply the condition in the transformed program. For now, we will not require non-termination preservation (meaning that if the original program does not terminate, the transformed program can do anything).

4.3 Abstraction

The meaning of an edge fact declaration:

$$\texttt{define edge fact } EF(X_1 : \tau_1, \ldots, X_n : \tau_n) \texttt{ with meaning } R$$

is given by $[\![EF]\!] : \tau_1 \times \ldots \times \tau_n \times \mathit{Pred} \to \mathit{bool}$ and is defined by:

$$[\![EF]\!](t_1, \ldots, t_n, P) = \forall(\eta_1, \eta_2) \in \mathit{State}^2 \ . \ \theta(R)(\eta_1, \eta_2) \Rightarrow P(\eta_1) = P(\eta_2)$$

where $\theta = [X_1 \mapsto t_1, \ldots, X_n \mapsto t_n]$, $\theta(R)$ applies the substitution θ to R, and $P(\eta)$ evaluates a predicate P at a program state η.

We only accept R's that match program points, meaning that $R(\eta_1, \eta_2) \Rightarrow \mathit{edge}(\eta_1) = \mathit{edge}(\eta_2)$. We achieve this in the implementation by having the user define a different relation, namely R', which is not allowed to mention the program point part of the state. We then define $R(\eta_1, \eta_2) \triangleq R'(\eta_1, \eta_2) \wedge \mathit{edge}(\eta_1) = \mathit{edge}(\eta_2)$.

The abstraction function $\alpha : D_c \to D$ is defined as:

$$\alpha(P) = \{EF_i(t_1, \ldots, t_n) \mid 1 \leq i \leq k \wedge [\![EF_i]\!](t_1, \ldots, t_n, P)\}$$

4.4 Conditions for soundness

4.4.1 Propagation rule

Definition 17 *A backward propagation rule* `if` ψ `then` $EF(t_1, \ldots, t_n)$`cfg_in`$[h']$ *is said to be sound iff the following condition holds:*

[1]To use liberal weakest preconditions, we would have to change wp_p to the liberal version wlp_p and then use the least fixed point instead of the greatest fixed point

$$\forall (P, P', ds, n, \theta, h, p) \in \mathit{Pred} \times \mathit{Pred} \times D^* \times \mathit{Node} \times \mathit{Subst} \times \mathit{Natural} \times \mathit{Proc}.$$
$$\left[\begin{array}{l} \theta[\![\psi]\!]^{ds}_{\mathit{stmtAt}(n)} \wedge \\ P' = wp_p(P, \mathit{out}(n)[h], \mathit{in}(n)[h']) \wedge \\ \forall (EF', t'_1, \ldots, t'_j) \in \mathit{EdgeFact} \times \mathit{Term}^j. \\ \qquad EF'(t'_1, \ldots, t'_j) \in ds[h] \Rightarrow [\![EF']\!](t'_1, \ldots, t'_j, P) \end{array}\right] \Rightarrow [\![EF]\!](\theta(t_1), \ldots, \theta(t_n), P') \tag{bwd-prop-sound}$$

4.4.2 Transformation rule

Definition 18 *A backward transformation rule* `if` ψ `then transform` s *is said to be sound iff the following condition holds:*

$$\forall (P, P', P'', ds, n, \theta, h, p, p') \in \mathit{Pred}^3 \times D^* \times \mathit{Node} \times \mathit{Subst} \times \mathit{Natural} \times \mathit{Proc}^2.$$
$$\left[\begin{array}{l} \theta[\![\psi]\!]^{ds}_{\mathit{stmtAt}(n)} \wedge \\ P' = wp_p(P, \mathit{out}(n)[h], \mathit{in}(n)[h'])) \wedge \\ P'' = wp_{p'}(P, \mathit{out}(n')[h], \mathit{in}(n')[h'])) \wedge \\ \mathit{stmtAt}(n') = \theta(s) \wedge \\ \forall (EF', t'_1, \ldots, t'_j) \in \mathit{EdgeFact} \times \mathit{Term}^j. \\ \qquad EF'(t'_1, \ldots, t'_j) \in ds[h] \Rightarrow [\![EF']\!](t'_1, \ldots, t'_j, P) \end{array}\right] \Rightarrow (P' \Rightarrow P'') \tag{bwd-trans-sound}$$

4.5 Metatheory

The main soundness theorem for the backward case is similar to the forward case:

Theorem 8 (main backward soundness) *If all propagation rules and transformation rules in a backward Rhodium program P are sound, then the induced AT-analysis $(\mathcal{A}_P, R_P)$ is sound.*

Theorem 8 follows from the following two lemmas and theorem 2:

Lemma 6 *If all propagation rules in a backward Rhodium program P are sound then F as defined in* (4.1) *is sound.*

Lemma 7 *If all transformation rules in a backward Rhodium program P are sound, then R as defined in* (3.2) *is sound.*

We do not yet have the proofs of lemmas 6 and 7, but we foresee no problems in making these proofs go through. Furthermore, we need to update the composing framework from section 2.3 in order to handle our weakest precondition semantics. This will be not be hard, because the proofs in section 2.3 for the most part do not refer to the actual semantics of the concrete domain. We are currently working on these proofs.

Chapter 5

Framework for flow-insensitive analyses

In order to define flow-insensitive analyses, we adapt the composing dataflow analyses and transformations framework so that flow functions take a map from edges to dataflow information, instead of just a tuple of the input dataflow information. We call such flow functions map flow functions.

The concrete map flow function $F_{mc} : \mathit{Node} \times (\mathit{Edge} \to D_c) \to D_c^*$ is defined as:

$$F_{mc}(n, m_c) = F_c(n, \overrightarrow{m_c}(in(n)))$$

The condition for soundness of a flow function is then:

Definition 19 *A map flow function F_a is said to be sound iff the following condition holds:*

$$\begin{gathered}\forall (n, m_c, m_a) \in \mathit{Node} \times (\mathit{Edge} \to D_c) \times (\mathit{Edge} \to D_a). \\ \alpha(m_c) \sqsubseteq m_a \Rightarrow \alpha(F_{mc}(n, m_c)) \sqsubseteq F_a(n, m_a)\end{gathered}$$

Lemma 8 (condition for $(\mathcal{A}, R)$ to be sound with map flow functions) *Given an AT-analysis $(\mathcal{A}, R)$, if the map flow function F_a of $\mathcal{A}$ is sound and R is sound, then $(\mathcal{A}, R)$ is sound.*

The proof of lemma 8 would be almost identical to the proof of lemma 2. We are currently working on adapting the proof of lemma 2 to lemma 8.

Given a flow sensitive analysis $\mathcal{A} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F)$ with flow-function $F : \mathit{Node} \times D^* \to D^*$, we define the flow-insensitive version of this analysis $\mathcal{A}_{fi} = (D, \sqcup, \sqcap, \sqsubseteq, \top, \bot, \alpha, F_{fi})$, where the flow-insensitive flow function is a map flow function $F_{fi} : \mathit{Node} \times (\mathit{Edge} \to D) \to D^*$ defined as:

$$\begin{aligned}&F_{fi}(n, m)[k] = \bigsqcup S \\ &\quad \text{where } S = \{ds[j] \mid ds = F(n, m(in(n))) \wedge n \in \mathit{Node} \wedge 0 \le k \le len(ds)\}\end{aligned} \tag{5.1}$$

F_{fi} computes the join of all the results returned by all nodes in the CFG. The join is returned for all nodes on all outgoing edges. As a result, all edges in the CFG will have the same dataflow value throughout the analysis. The execution engine can therefore keep only one copy of this value. Also, the execution engine can perform the regular chaotic iteration optimization of executing only one node at a time (and keeping a worklist), instead of executing all nodes on each iteration.

Theorem 9 (main flow-insensitive soundness) *If F is sound then F_{fi} is sound.*

Proof of theorem 9

We need to show

$$\forall (n, m_c, m) \in \mathit{Node} \times (\mathit{Edge} \rightarrow D_c) \times (\mathit{Edge} \rightarrow D). \\ \alpha(m_c) \sqsubseteq m \Rightarrow \alpha(F_{mc}(n, m_c)) \sqsubseteq F_{fi}(n, m)$$

Pick $(n, m_c, m) \in \mathit{Node} \times (\mathit{Edge} \rightarrow D_c) \times (\mathit{Edge} \rightarrow D)$, assume $\alpha(m_c) \sqsubseteq m$, and show $\alpha(F_{mc}(n, m_c)) \sqsubseteq F_{fi}(n, m)$.

To show $\alpha(F_{mc}(n, m_c)) \sqsubseteq F(n, m)$, we need to show $F_{mc}(n, m_c)[k] \sqsubseteq F_{fi}(n, m)[k]$.

We have:

$$\begin{array}{lll} & \alpha(m_c) \sqsubseteq m & \\ \Rightarrow & \alpha(m_c(in(n))) \sqsubseteq m(in(n)) & \\ \Rightarrow & \alpha(F_c(n, m_c(in(n)))) \sqsubseteq F(n, m(in(n))) & \text{(soundness of } F\text{)} \\ \Rightarrow & \alpha(F_{mc}(n, m_c)) \sqsubseteq F(n, m(in(n))) & \text{(definition of } F_{mc}\text{)} \\ \Rightarrow & \alpha(F_{mc}(n, m_c))[k] \sqsubseteq F(n, m(in(n)))[k] & \end{array}$$

Since $F(n, m(in(n)))[k] \in S$ in (5.1), we have $F(n, m(in(n)))[k] \sqsubseteq \bigsqcup S$.

Therefore $F(n, m(in(n)))[k] \sqsubseteq F_{fi}(n, m)[k]$.

And by transitivity we get $\alpha(F_{mc}(n, m_c))[k] \sqsubseteq F_{fi}(n, m)[k]$.

■

Chapter 6

Framework for interprocedural analyses

6.1 Concrete semantics

For the concrete semantics, we use a trace semantics. A trace t is a sequence of machine configurations: $t = [\delta_1, \ldots, \delta_n]$. We denote by *Trace* the set of all traces.

Given a trace $t = [\delta_1, \ldots, \delta_n]$, where $\delta_n = (e, \eta)$, we use *lastMachineConfig*(e) to denote δ_n and *lastState* to denote η.

Definition 20 *The trace transition function* $\rightarrow \;\subseteq \mathit{Trace} \times \mathit{Trace}$ *is defined as:*

$$[\delta_1, \ldots, \delta_n] \rightarrow [\delta_1, \ldots, \delta_n, \delta_{n+1}] \text{ where } \delta_n \rightarrow \delta_{n+1}$$

So the concrete interprocedural domain is

$$(D_{ic}, \sqcup_{ic}, \sqcap_{ic}, \sqsubseteq_{ic}, \top_{ic}, \bot_{ic}) = (2^{\mathit{Trace}}, \cup, \cap, \subseteq, \mathit{Trace}, \emptyset)$$

The concrete flow function is:

$$F_{ic} : \mathit{Node} \times (\mathit{Edge} \rightarrow D_{ic}) \rightarrow D_{ic}^{*}$$

Given a node n and a map m providing the information at each edge $F_{ic}(n, m)$ returns a tuple of outgoing concrete facts, one for each outgoing edge from the node.

At a call site, $F_{ic}(n, m)$ returns a pair, with $F_{ic}(n, m)[0]$ being the information to propagate to the return site and $F_{ic}(n, m)[1]$ being the information to propagate to the callee's entry edge.

The definition of F_{ic} is split based on whether or not we are at a call site.

If $\mathit{stmtAt}(n) \neq$ [`call`...] then:

$$F_{ic}(n,m)[k] = \{t \mid \begin{array}{l} edge(lastMachineConfig(t)) = out(n)[k] \wedge \\ \exists\, t' \in Trace, i \in Natural\ .\ [t' \in m(in(n)[i]) \wedge t' \to t]\} \end{array}$$

If $stmtAt(n) = [$`call` $fn]$ then:

$$\begin{array}{lcl} F_{ic}(n,m)[0] & = & \{step(t) \mid t \in m(outEdge(fn)) \wedge callSite(t) = n\} \\ F_{ic}(n,m)[1] & = & \{step(t) \mid t \in m(in(n)[0])\} \end{array}$$

where:

- $step(t) = t'$ iff $t \to t'$. The *step* function is partial, but in the uses above, it is well defined because program execution cannot fail at a call site or at a return site.
- $callSite(t)$ is the call site of the latest stack frame. In particular, $callSite(t) = node(currentCall(t))$, where $currentCall(t)$ returns the machine configuration in t that produced the call to the currently executing function. Alternatively, $callSite(t) = n$ iff $pop(stack(lastState(t))) = ((n, \_, \_), \_)$.
- $outEdge(fn)$ returns fn's CFG edge right before the return statement.

6.2 Abstract semantics

6.2.1 Domain

We assume a lattice of contour keys $(CK, \sqcup_{ck}, \sqcap_{ck}, \sqsubseteq_{ck}, \top_{ck}, \bot_{ck})$.

Recall that the abstract domain is $(D, \sqcup, \sqcap, \sqsubseteq, \top, \bot)$.

We define the domain of the interprocedural analysis as $(D_i, \sqcup_i, \sqcap_i, \sqsubseteq_i, \top_i, \bot_i)$ where:

$$\begin{array}{lcl} D_i & \triangleq & CK \to D \\ f_1 \sqcup_i f_2 & \triangleq & \lambda\, ck\ .\ f_1(ck) \sqcup f_2(ck) \\ f_1 \sqcap_i f_2 & \triangleq & \lambda\, ck\ .\ f_1(ck) \sqcap f_2(ck) \\ f_1 \sqsubseteq_i f_2 & \triangleq & \forall\, ck\ .\ f_1(ck) \sqsubseteq f_2(ck) \\ \top_i & \triangleq & \lambda\, ck\ .\ \top \\ \bot_i & \triangleq & \lambda\, ck\ .\ \bot \end{array}$$

The domain D_i is a domain of total maps from CK to D. The domain of partial maps described in the POPL paper [4] can be seen as an implementation strategy for the domain of total maps. Partial maps can be used to represent total maps if the missing entries have implicit mappings (for example all missing entries can map to $\bot$).

6.2.2 Translating information from callers to callees and vice-versa

The function $callerToCallee : Node \times D \to D$ translates information from the caller to the callee at call site: given a call site n, and some information d coming into the call site, $callerToCallee(n, d)$ returns the information at the entry to the caller. This function translates facts from the caller to the callee.

Similarly the function *calleeToCaller* : $\mathit{Node} \times D \rightarrow D$ translates facts from the callee's return instruction to the calling site: given a call site n, and some information d coming into the return statement of the function which is called at n, $\mathit{calleeToCaller}(n, d)$ returns the information at the return site of n. This function translates facts from the callee to the caller.

Definition 21 *The translation function callerToCallee is said to be sound iff it satisfies the following condition:*

$$\forall (n, \eta, \eta', h, h', d) \in \mathit{Node} \times \mathit{State}^2 \times \mathit{Natural}^2 \times D \; . \\ \left[\begin{array}{l} \mathit{stmtAt}(n) = (x := p(y)) \wedge \\ h, \eta \xrightarrow{n} h', \eta' \wedge \\ \mathit{allMeaningsHold}(d, \eta) \end{array} \right] \Rightarrow \quad \mathit{allMeaningsHold}(\mathit{callerToCallee}(n, d), \eta') \tag{6.1}$$

Definition 22 *The translation function calleeToCaller is said to be sound iff it satisfies the following condition:*

$$\forall (n, n_{call}, \eta, \eta', h, h', d) \in \mathit{Node}^2 \times \mathit{State}^2 \times \mathit{Natural}^2 \times D \; . \\ \left[\begin{array}{l} \mathit{stmtAt}(n) = (\texttt{return } x) \wedge \\ h, \eta \xrightarrow{n} h', \eta' \wedge \\ \mathit{pop}(\mathit{stack}(\eta)) = ((n_{call}, \_, \_), \_) \wedge \\ \mathit{allMeaningsHold}(d, \eta) \end{array} \right] \Rightarrow \quad \mathit{allMeaningsHold}(\mathit{calleeToCaller}(n_{call}, d), \eta') \tag{6.2}$$

The following two lemmas follow from the above definitions of soundness of *callerToCallee* and *calleeToCaller*.

Lemma 9 *If callerToCallee is sound then:*

$$\forall (n, d, X) \in \mathit{Node} \times D \times 2^{\mathit{Trace}} \\ [\forall t \in X \; . \; \mathit{stmtAt}(\mathit{node}(\mathit{lastMachineConfig}(t))) = (x := p(y))] \Rightarrow \\ [\alpha(\{\mathit{lastState}(t) \mid t \in X\}) \sqsubseteq d \Rightarrow \alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid t \in X\}) \sqsubseteq \mathit{callerToCallee}(n, d)]$$

Lemma 10 *If calleeToCaller is sound then:*

$$\forall (n, d, X) \in \mathit{Node} \times D \times 2^{\mathit{Trace}} \\ [\forall t \in X \; . \; \mathit{callSite}(t) = n \wedge \mathit{stmtAt}(\mathit{node}(\mathit{lastMachineConfig}(t))) = (\texttt{return } x)] \Rightarrow \\ [\alpha(\{\mathit{lastState}(t) \mid t \in X\}) \sqsubseteq d \Rightarrow \alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid t \in X\}) \sqsubseteq \mathit{calleeToCaller}(n, d)]$$

We use $t[x \mapsto y]$ to denote the term t with x replaced by y, and we use $EF(t_1, \ldots, t_j)[x \mapsto y]$ to denote $EF(t_1[x \mapsto y], \ldots, t_j[x \mapsto y])$.

We now define *value facts*. These facts have the property that they are preserved through assignments by just replacing the assigned variable in the fact.

Definition 23 *The set valueFacts of edge facts is defined as: $EF \in$ valueFacts iff the following condition holds:*

$$\forall (n, \eta, \eta', x, y, t_1, \ldots, t_j) \in \mathit{Node} \times \mathit{State}^2 \in \mathit{Var}^2 \times \mathit{Term}^j \; . \\ \left[\begin{array}{l} [\![EF]\!](t_1, \ldots, t_j, \eta) \wedge \\ \mathit{stmtAt}(n) = (x := y) \wedge \\ 0, \eta \xrightarrow{n} 0, \eta' \end{array} \right] \Rightarrow [\![EF]\!](t_1[y \mapsto x], \ldots, t_j[y \mapsto x], \eta')$$

We now define sound translation functions *callerToCallee* and *calleeToCaller*:

Definition 24 *If $stmtAt(n) = (x := p(b))$ then:*

$$callerToCallee(n, d) = \{EF(t_1, \ldots, t_j)[b \mapsto formal_p] \mid \begin{array}{l} EF(t_1, \ldots, t_j) \in d \wedge EF \in valueFacts \wedge \\ \forall\ 1 \leq i \leq j\ .\ (t_i = b \vee t_i \in Const)\} \end{array}$$

Definition 25 *If $stmtAt(n) = (x := p(b))$ and the return statement of p is* `return` *r then:*

$$calleeToCaller(n, d) = \{EF(t_1, \ldots, t_j)[r \mapsto x] \mid \begin{array}{l} EF(t_1, \ldots, t_j) \in d \wedge EF \in valueFacts \wedge \\ \forall\ 1 \leq i \leq j\ .\ (t_i = r \vee t_i \in Const)\} \end{array}$$

These translation functions are very conservative: only value facts are are propagated and the translation is performed only for formals and return values. As future work, we plan to investigate more precise translation functions.

6.2.3 Abstract flow function

The interprocedural flow function is:

$$F_i : Node \times (Edge \rightarrow D_i) \rightarrow D_i^*$$

Given a node n and a map m providing the information at each edge $F_i(n, m)$ returns a tuple of outgoing dataflow facts, one for each outgoing edge from the node.

At a call site, $F_i(n, m)$ returns a pair, with $F_i(n, m)[0]$ being the information to propagate to the return site and $F_i(n, m)[1]$ being the information to propagate to the callee's entry edge.

The definition of F_i is split based on whether or not we are at a call site.

If $stmtAt(n) \neq [\texttt{call} \ldots]$ then:

$$F_i(n, m)[k] = \lambda\, ck\ .\ F(n, [d_1, \ldots, d_j])[k] \text{ where } d_i = m(in(n)[i])(ck) \text{ for } i \in [1..j]$$

If $stmtAt(n) = [\texttt{call}\ fn]$ then:

$$F_i(n, m)[0] \quad = \quad \lambda\, ck\ .\ calleeToCaller(n, m(outEdge(fn))(BC(ck, m(in(n)[0])(ck), n)))$$

For the transfer function strategy:

$$F_i(n, m)[1] \quad = \quad \lambda\, ck\ .\ callerToCallee(n, ck)$$

For all other strategies:

$$F_i(n, m)[1] \quad = \quad \lambda\, ck\ .\ \bigsqcup\{callerToCallee(n, m(in(n)[0])(ck')) \mid ck = BC(ck', m(in(n)[0])(ck'), n)\}$$

The function $BC : (CK \times D \times Node) \rightarrow CK$ determines the contour key to be used for analyzing the callee of a particular call site. If a call site n to function fn is analyzed in contour key ck and dataflow fact d, then $BC(ck, d, n)$ returns the contour key to use in analyzing fn for call site n.

$F_i(n,m)[1]$ is defined more conservatively for the transfer function strategy in order to make the proofs go through. We are currently looking at how to unify the definition of $F_i(n,m)[1]$ for the transfer function strategy with the general definition of $F_i(n,m)[1]$.

The abstraction function $\alpha_i : D_{ic} \rightarrow D_i$ is defined as follows:

$$\alpha_i(ts) = \lambda\, ck\ .\ \alpha(\{lastState(t) \mid t \in ts \wedge matchesContour(t, ck)\}$$

where:

- α is the abstraction function of the intraprocedural (in our case α is defined as in sections 3.2 and 4.3, but the following formalization does not in any way depend on the details of those definitions)
- $matchesContour(t, ck)$ says whether or not a trace t matches contour ck:

$$matchesContour(t, ck) = \begin{cases} true & \text{if } t = [\texttt{call main}] \wedge ck = \top_{ck} \\ false & \text{if } t = [\texttt{call main}] \wedge ck \neq \top_{ck} \\ \begin{array}{ll} \exists\, ck' \in CK, d \in D\ . & \alpha(\{currentCall(t)\}) \sqsubseteq d \wedge \\ & ck = BC(ck', d, node(currentCall(t))) \wedge \\ & matchesContour(prefix(t, currentCall(t)), ck') \end{array} & \text{otherwise} \end{cases}$$

- [`call main`] is the trace that is about to execute a call to main. The program is started in the trace [`call main`; S_1], where S_1 is the first statement in `main`. The trace [`call main`; S_1] is about to execute S_1. As a result, the call to `main` is never executed in the concrete semantics – it is there so that $currentCall(t)$ is always well defined (see below).
- $currentCall(t)$ returns the machine configuration in t that produced the call to the currently executing function. The call to main is at the beginning of all traces so that $currentCall(t)$ is always defined, even when t is about to execute a statement in `main`.
- $prefix(t, \delta)$ returns the prefix of t up to and including machine configuration δ. It is not defined if δ does not occur in t.

We have the following lemma about $matchesContour$:

Lemma 11

$$\begin{array}{l} \forall\ (t_1, t_2) \in Trace^2\ . \\ \quad t_1 \neq [\texttt{call main}] \wedge t_2 \neq [\texttt{call main}] \wedge currentCall(t_1) = currentCall(t_2) \Rightarrow \\ \qquad \forall\ ck \in CK\ .\ matchesContour(t_1, ck) = matchesContour(t_2, ck) \end{array}$$

Proof of lemma 11

The proof is immediate from the fact that if $t_1 \neq$ [`call main`] $\wedge\ t_2 \neq$ [`call main`], then t_1 appears in $matchesContour(t_1, ck)$ only as $currentCall(t_1)$ and t_2 appears in $matchesContour(t_2, ck)$ only as $currentCall(t_2)$.

■

6.3 Call strings strategy

If we want to implement the k-length call-string strategy, then $\mathit{CK} = \mathit{list}[\mathit{string}]$ and $\mathit{BC}(\mathit{ck}, d, n) = \mathit{concat}(\mathit{ck}, [\mathit{fn}(n)]).\mathit{last}(k)$ where $\mathit{fn}(n)$ is the name of the function called at node n, concat is the list concatenation function ($\mathit{concat} : \mathit{list}[\mathit{string}] \times \mathit{list}[\mathit{string}] \rightarrow \mathit{list}[\mathit{string}]$), and $l.\mathit{last\_k}(k)$ returns the a list containing the last k elements of l ($\mathit{last\_k} : \mathit{list}[\mathit{string}] \times \mathit{nat} \rightarrow \mathit{list}[\mathit{string}]$). Note that we use $a.f(b)$ as sugar for $f(a, b)$.

The BC function in this case does not depend on d and so we also define the simpler version $\mathit{BC}_{[str]}(\mathit{ck}, n)$ that does not take the d parameter. The simpler version is defined as $\mathit{BC}_{[str]}(\mathit{ck}, n) = \mathit{concat}(\mathit{ck}, [\mathit{fn}(n)]).\mathit{lastk}(k)$.

We call $\mathit{matchesContour}_{[str]}$ the resulting $\mathit{matchesContour}$ function.

We define the function $\mathit{last\_k\_calls} : \mathit{Trace} \times \mathit{Natural} \rightarrow \mathit{list}[\mathit{string}]$ so that $t.\mathit{last\_k\_calls}(k)$ returns a list with the last k function names on the call stack. We then have the following theorem:

Lemma 12 $\mathit{matchesContour}_{[str]}(t, \mathit{ck}) = (t.\mathit{last\_k\_calls}(k) = \mathit{ck})$

Proof of lemma 12

By induction on the length of the trace t.

■

Note that for convenience, we have defined BC in such a way so that we include the current function being analyzed in the call string. Thus, when we analyze function f when it is called from g, the call string is $[g, f]$, not just $[g]$. This means that our k is one bigger than the k from k-CFA. For example, for 1-CFA, we must actually set $k = 2$.

Lemma 13 *If a trace t is about to execute a call at node n and $\mathit{matchesContour}_{[str]}(\mathit{step}(t), \mathit{ck})$ and $\mathit{matchesContour}_{[str]}(t, \mathit{ck}')$ then $\mathit{ck} = \mathit{BC}_{[str]}(\mathit{ck}', n)$.*

6.4 Transfer function strategy

If we want to implement the calling-context strategy, then $\mathit{CK} = D$, and $\mathit{BC}(\mathit{ck}, d, n) = d$. In this case, we call $\mathit{matchesContour}_{[trans]}$ the resulting $\mathit{matchesContour}$ function.

Lemma 14

$$\mathit{matchesContour}_{[trans]}(t, \mathit{ck}) = \begin{cases} \mathit{true} & \mathit{if}\ t = [\texttt{call main}] \wedge \mathit{ck} = \top_{ck} \\ \mathit{false} & \mathit{if}\ t = [\texttt{call main}] \wedge \mathit{ck} \neq \top_{ck} \\ \alpha(\{\mathit{currentCall}(t)\}) \sqsubseteq \mathit{ck}\ \wedge & \\ \exists\, \mathit{ck}' \in \mathit{CK}\ . & \mathit{otherwise} \\ \quad \mathit{matchesContour}_{[trans]}(\mathit{prefix}(t, \mathit{currentCall}(t)), \mathit{ck}') & \end{cases}$$

Proof of lemma 14

Only the third case needs to be shown equivalent. We take the third case from the definition of *matchesContour*, and expand *BC*:

$$\begin{array}{lll} & \exists\, ck' \in CK, d \in D\ . & \alpha(\{currentCall(t)\}) \sqsubseteq d\ \wedge \\ & & ck = d\ \wedge \\ & & matchesContour_{[trans]}(prefix(t, currentCall(t)), ck') \\[1em] & \exists\, ck' \in CK, d \in D\ . & \alpha(\{currentCall(t)\}) \sqsubseteq ck\ \wedge \\ \Longleftrightarrow & & ck = d\ \wedge \\ & & matchesContour_{[trans]}(prefix(t, currentCall(t)), ck') \end{array}$$

$$\begin{array}{ll} & \alpha(\{currentCall(t)\}) \sqsubseteq ck\ \wedge \\ \Longleftrightarrow & \quad \exists\, ck' \in CK, d \in D\ .\ \ ck = d\ \wedge \\ & \qquad\qquad\qquad\qquad matchesContour_{[trans]}(prefix(t, currentCall(t)), ck') \\[1em] \Longleftrightarrow & \alpha(\{currentCall(t)\}) \sqsubseteq ck\ \wedge \\ & \exists\, ck' \in CK\ .\ matchesContour_{[trans]}(prefix(t, currentCall(t)), ck') \end{array}$$

■

6.5 Soundness

We want to show that F_i is sound if F is sound. The definition of soundness of F_i is the same as that for F, except that it is adapted for the fact that we have a global map as input to F_i (instead of just a tuple of dataflow facts for the input edges of the node).

Definition 26 *An interprocedural flow function F_i is said to be sound iff it satisfies the following condition:*

$$\begin{array}{l} \forall (n, m_{ic}, m_i) \in Node \times (Edge \to D_{ic}) \times (Edge \to D_i)\ . \\ \qquad \alpha_i(m_{ic}) \sqsubseteq m_i \Rightarrow \alpha_i(F_{ic}(n, m_{ic})) \sqsubseteq_i F_i(n, m_i) \end{array}$$

Theorem 10 (main interprocedural soundness) *If F, callerToCallee and calleeToCaller are all sound then F_i is sound.*

Proof of theorem 10

We need to show:

$$\begin{array}{l} \forall (n, m_{ic}, m_i) \in Node \times (Edge \to D_{ic}) \times (Edge \to D_i)\ . \\ \qquad \alpha_i(m_{ic}) \sqsubseteq m_i \Rightarrow \alpha_i(F_{ic}(n, m_{ic})) \sqsubseteq_i F_i(n, m_i) \end{array}$$

Pick $n \in Node$, $m_{ic} \in (Edge \to D_{ic})$ and $m_i \in (Edge \to D_i)$ and assume $\alpha_i(m_{ic}) \sqsubseteq_i m_i$. We now want to show $\alpha_i(F_{ic}(n, m_{ic})) \sqsubseteq_i F_i(n, m_i)$.

- **Case where** $stmtAt(n) = [Call\ fn]$

  There are two output edges to a call node, so we must show $\alpha_i(F_{ic}(n, m_{ic})[0]) \sqsubseteq_i F_i(n, m_i)[0]$ and $\alpha_i(F_{ic}(n, m_{ic})[1]) \sqsubseteq_i F_i(n, m_i)[1]$.

– **Proof of** $\alpha_i(F_{ic}(n, m_{ic})[0]) \sqsubseteq_i F_i(n, m_i)[0]$

Let $f_1 = \alpha_i(F_{ic}(n, m_{ic})[0])$ and $f_2 = F_i(n, m_i)[0]$. We must show $\forall\, ck \in CK \;.\; f_1(ck) \sqsubseteq f_2(ck)$. Pick $ck \in CK$, and show $f_1(ck) \sqsubseteq f_2(ck)$.

Using the definition of α_i and of F_{ic}, we get:

$$\begin{array}{ll} f_1(ck) = \alpha(T) & \\ \text{where } T = \{lastState(step(t)) \mid & t \in m_{ic}(outEdge(fn)) \wedge \\ & callSite(t) = n \wedge matchesContour(step(t), ck)\}) \end{array}$$

Using the definition of F_i, we get:

$$\begin{array}{l} f_2(ck) = calleeToCaller(n, m_i(outEdge(fn))(x)) \\ \text{where } x = BC(ck, m_i(in(n)[0])(ck), n) \end{array}$$

Since $\alpha_i(m_{ic}) \sqsubseteq_i m_i$, we get:

$$\begin{array}{ll} & \alpha_i(m_{ic}(outEdge(fn))) \sqsubseteq_i m_i(outEdge(fn)) \\ \Rightarrow & \forall\, ck \;.\; \alpha_i(m_{ic}(outEdge(fn)))(ck) \sqsubseteq m_i(outEdge(fn))(ck) \\ \Rightarrow & \alpha_i(m_{ic}(outEdge(fn)))(x) \sqsubseteq m_i(outEdge(fn))(x) \\ \Rightarrow & \alpha(\{lastState(t) \mid t \in m_{ic}(outEdge(fn)) \wedge matchesContour(t, x)\} \sqsubseteq m_i(outEdge(fn))(x) \end{array}$$

Using lemma 10 with $X = \{t \mid t \in m_{ic}(outEdge(fn)) \wedge matchesContour(t, x)\}$ we get:

$$\begin{array}{l} \alpha(\{lastState(step(t)) \mid t \in m_{ic}(outEdge(fn)) \wedge matchesContour(t, x)\} \sqsubseteq \\ \qquad\qquad\qquad\qquad callerToCallee(n, m_i(outEdge(fn))(x)) \end{array}$$

Or:

$$\alpha(\{lastState(step(t)) \mid t \in m_{ic}(outEdge(fn)) \wedge matchesContour(t, x)\} \sqsubseteq f_2(ck)$$

Let $S = \{lastState(step(t)) \mid t \in m_{ic}(outEdge(fn)) \wedge matchesContour(t, x)\}$ so that:

$$\alpha(S) \sqsubseteq f_2(ck)$$

All we need to show now is that for $t \in m_{ic}(outEdge(fn))$, $[callSite(t) = n \wedge matchesContour(step(t), ck)] \Rightarrow matchesContour(t, x)$. For if this is the case, then $T \subseteq S$ and then by the monotonicity of α we get $\alpha(T) \sqsubseteq \alpha(S)$, or $f_1(ck) \sqsubseteq \alpha(S)$, which by transitivity of $\sqsubseteq$ gives us $f_1(ck) \sqsubseteq f_2(ck)$.

So we pick $t \in m_{ic}(outEdge(fn))$, and assume $callSite(t) = n \wedge matchesContour(step(t), ck)$, and we need to show $matchesContour(t, x)$.

Because $t \in m_{ic}(outEdge(fn))$, we know that $t \neq$ [`call main`], and so we are the third case of the definition of *matchesContour*. Thus, we need to show:

$$\begin{array}{ll} \exists\, ck' \in CK, d \in D \;. & \alpha(\{currentCall(t)\}) \sqsubseteq d \wedge \\ & x = BC(ck', d, node(currentCall(t))) \wedge \\ & matchesContour(prefix(t, currentCall(t)), ck') \end{array}$$

The claim is that setting $ck' = ck$ and $d = m_i(in(n)[0])(ck)$ makes the existential valid.

To show this, we need to show:

$$\begin{array}{l} \alpha(\{\mathit{currentCall}(t)\}) \sqsubseteq m_i(\mathit{in}(n)[0])(\mathit{ck}) \wedge \\ x = \mathit{BC}(\mathit{ck}, m_i(\mathit{in}(n)[0])(\mathit{ck}), \mathit{node}(\mathit{currentCall}(t))) \wedge \\ \mathit{matchesContour}(\mathit{prefix}(t, \mathit{currentCall}(t)), \mathit{ck}) \end{array}$$

We do these in reverse order:

* **Proof of** $\mathit{matchesContour}(\mathit{prefix}(t, \mathit{currentCall}(t)), \mathit{ck})$
  Since $t \in m_{ic}(\mathit{outEdge}(fn))$, t is about to step out of a call. As a result, $\mathit{step}(t)$ is at the return site of the call node n.
  Let $t' = \mathit{prefix}(t, \mathit{currentCall}(t))$. t' is the prefix of the trace t right about to execute the call node n. Thus, t' is the trace right before the call node n is executed, and $\mathit{step}(t)$ is the trace once execution has returned from the call.
  Thus, t' and $\mathit{step}(t)$ are at the same calling level, meaning that $\mathit{currentCall}(t') = \mathit{currentCall}(\mathit{step}(t))$.
  We also know that n cannot be a call to `main`, since the original call to `main` is not executed or analyzed. Thus, $t' \neq$ [`call main`]. Also, $\mathit{step}(t)$ cannot be [`call main`] since [`call main`] cannot result from stepping.
  Using lemma 11 with $t_1 = t'$ and $t_2 = \mathit{step}(t)$, we get $\mathit{matchesContour}(t') = \mathit{currentCall}(\mathit{step}(t))$.
  Since we know $\mathit{matchesContour}(\mathit{step}(t), \mathit{ck})$, then $\mathit{matchesContour}(t', \mathit{ck})$, or equivalently $\mathit{matchesContour}(\mathit{prefix}(t, \mathit{currentCall}(t)), \mathit{ck})$.
* **Proof of** $x = \mathit{BC}(\mathit{ck}, m_i(\mathit{in}(n)[0])(\mathit{ck}), \mathit{node}(\mathit{currentCall}(t)))$
  We already know that $x = \mathit{BC}(\mathit{ck}, m_i(\mathit{in}(n)[0](\mathit{ck})), n)$. We also assumed that $n = \mathit{callSite}(t)$. Expanding the definition of $\mathit{callSite}$, we get $n = \mathit{node}(\mathit{currentCall}(t))$. And so we therefore get $x = \mathit{BC}(\mathit{ck}, m_i(\mathit{in}(n)[0])(\mathit{ck}), \mathit{node}(\mathit{currentCall}(t)))$.
* **Proof of** $\alpha(\{\mathit{currentCall}(t)\}) \sqsubseteq m_i(\mathit{in}(n)[0])(\mathit{ck})$
  Let $t' = \mathit{prefix}(t, \mathit{currentCall}(t))$. By the definition of $prefix$, we get that $\mathit{currentCall}(t) = \mathit{lastState}(t')$.
  So we need to show $\alpha(\{\mathit{lastState}(t')\}) \sqsubseteq m_i(\mathit{in}(n)[0])(\mathit{ck})$.
  Since $\alpha_i(m_{ic}) \sqsubseteq_i m_i$, we get:

$$\begin{array}{ll} & \alpha_i(m_{ic}(\mathit{in}(n)[0])) \sqsubseteq_i m_i(\mathit{in}(n)[0]) \\ \Rightarrow & \forall\, \mathit{ck}\ .\ \alpha_i(m_{ic}(\mathit{in}(n)[0]))(\mathit{ck}) \sqsubseteq m_i(\mathit{in}(n)[0])(\mathit{ck}) \\ \Rightarrow & \alpha_i(m_{ic}(\mathit{in}(n)[0]))(\mathit{ck}) \sqsubseteq m_i(\mathit{in}(n)[0])(\mathit{ck}) \\ \Rightarrow & \alpha(\{\mathit{lastState}(t) \mid t \in m_{ic}(\mathit{in}(n)[0]) \wedge \mathit{matchesContour}(t, \mathit{ck})\} \sqsubseteq m_i(\mathit{in}(n)[0])(\mathit{ck}) \end{array}$$

  We have already shown that $\mathit{matchesContour}(t', \mathit{ck})$.
  Furthermore, because $t \in m_{ic}(\mathit{outEdge}(fn))$ and t' is a prefix of t, by the definition of the concrete semantics we must have $t' \in m_{ic}(\mathit{in}(n)[0]$.
  Thus, we get that:

$$\begin{array}{lll} & \{\mathit{lastState}(t')\} \subseteq \{\mathit{lastState}(t) \mid t \in m_{ic}(\mathit{in}(n)[0]) \wedge \mathit{matchesContour}(t, \mathit{ck})\} & \\ \Rightarrow & \alpha(\{\mathit{lastState}(t')\}) \sqsubseteq \alpha(\{\mathit{lastState}(t) \mid t \in m_{ic}(\mathit{in}(n)[0]) \wedge \mathit{matchesContour}(t, \mathit{ck})\}) & \text{monotonic } \alpha \\ \Rightarrow & \alpha(\{\mathit{lastState}(t')\}) \sqsubseteq m_i(\mathit{in}(n)[0])(\mathit{ck}) & \text{transitivity of } \sqsubseteq \end{array}$$

– **Proof of** $\alpha_i(F_{ic}(n, m_{ic})[1]) \sqsubseteq_i F_i(n, m_i)[1]$

* **Transfer function strategy**
  Let $f_1 = \alpha_i(F_{ic}(n, m_{ic})[1])$ and $f_2 = F_i(n, m_i)[1]$. We must show $\forall\, ck \in CK \,.\, f_1(ck) \sqsubseteq f_2(ck)$.
  Pick $ck \in CK$, and show $f_1(ck) \sqsubseteq f_2(ck)$.
  Using the definition of α_i and of F_{ic}, we get:

$$f_1(ck) = \alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid t \in m_{ic}(\mathit{in}(n)[0]) \wedge \mathit{matchesContour}_{[trans]}(\mathit{step}(t), ck)\}) \tag{6.3}$$

  We know that $\mathit{step}(t)$ cannot be equal to [`call main`], since $\mathit{step}(t)$ has already taken at least one step. Thus, we can expand the definition of $\mathit{matchesContour}_{[trans]}(\mathit{step}(t), ck)$ in (6.3) into the third case:

$$\begin{aligned} f_1(ck) = \alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid\ & t \in m_{ic}(\mathit{in}(n)[0]) \wedge \\ & \alpha(\{\mathit{currentCall}(\mathit{step}(t))\}) \sqsubseteq ck \ \wedge \\ & \exists\, ck' \in CK \ . \\ & \quad \mathit{matchesContour}_{[trans]}(\mathit{prefix}(t, \mathit{currentCall}(\mathit{step}(t))), ck')\}) \end{aligned}$$

  Since $\mathit{currentCall}(\mathit{step}(t)) = \mathit{lastState}(t)$, we get:

$$\begin{aligned} f_1(ck) = \alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid\ & t \in m_{ic}(\mathit{in}(n)[0]) \wedge \\ & \alpha(\{\mathit{lastState}(t)\}) \sqsubseteq ck \ \wedge \\ & \exists\, ck' \in CK \ . \\ & \quad \mathit{matchesContour}_{[trans]}(\mathit{prefix}(t, \mathit{lastState}(t)), ck')\}) \end{aligned}$$

  Let T be the set that is a parameter to α above so that $f_1(ck) = \alpha(T)$.
  Let $f_1'(ck)$ be defined as:

$$\begin{aligned} f_1'(ck) = \alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid\ & t \in m_{ic}(\mathit{in}(n)[0]) \wedge \\ & \alpha(\{\mathit{lastState}(\mathit{step}(t))\}) \sqsubseteq \mathit{callerToCallee}(n, ck) \ \wedge \\ & \exists\, ck' \in CK \ . \\ & \quad \mathit{matchesContour}_{[trans]}(\mathit{prefix}(t, \mathit{lastState}(t)), ck')\}) \end{aligned} \tag{6.4}$$

  Let S be the set that is the parameter to α above, so that $f_1'(ck) = \alpha(S)$.
  From lemma 9, we know that:

$$[\alpha(\{\mathit{currentCall}(\mathit{step}(t))\}) \sqsubseteq ck] \Rightarrow [\alpha(\{\mathit{lastState}(\mathit{step}(t))\}) \sqsubseteq \mathit{callerToCallee}(n, ck)]$$

  Thus $T \subseteq S$ and by the monotonicity of α, $\alpha(T) \sqsubseteq \alpha(S)$, or $f_1(ck) \sqsubseteq f_1'(ck)$.
  We wanted to show $f_1(ck) \sqsubseteq f_2(ck)$. Because of $f_1(ck) \sqsubseteq f_1'(ck)$ and the transitivity of $\sqsubseteq$, it suffices to show $f_1'(ck) \sqsubseteq f_2(ck)$
  Because of the condition $\alpha(\{\mathit{lastState}(\mathit{step}(t))\}) \sqsubseteq \mathit{callerToCallee}(n, ck)$ in (6.4), we have $\forall\, \eta \in S \,.\, \alpha(\{\eta\}) \sqsubseteq \mathit{callerToCallee}(n, ck)$.
  From the continuity of α, we know that for any set X of elements from 2^{State} we have $\alpha(\bigsqcup X) \sqsubseteq \bigsqcup\{\alpha(x) \mid x \in X\}$. Since $\sqcup$ is $\cup$ we get:

$$\alpha(\bigcup X) \sqsubseteq \bigsqcup\{\alpha(x) \mid x \in X\} \tag{6.5}$$

  If we let $X = \{\{\eta\} \mid \eta \in S\}$ then from (6.5) we get:

$$\begin{aligned}
& \alpha(\bigcup X) \sqsubseteq \bigsqcup\{\alpha(x) \mid x \in X\} \\
\Rightarrow\quad & \alpha(S) \sqsubseteq \bigsqcup\{\alpha(\{\eta\}) \mid \eta \in S\} \\
\Rightarrow\quad & f_1'(ck) \sqsubseteq \bigsqcup\{\alpha(\{\eta\}) \mid \eta \in S\}
\end{aligned}$$

By the monotonicity of $\sqcup$, which says $(a \sqsubseteq b) \wedge (c \sqsubseteq d) \Rightarrow (a \sqcup c) \sqsubseteq (b \sqcup d)$, and from $\forall\, \eta \in S\ .\ \alpha(\{\eta\}) \sqsubseteq \mathit{callerToCallee}(n, ck)$, we get:

$$\bigsqcup\{\alpha(\{\eta\}) \mid \eta \in S\} \sqsubseteq \mathit{callerToCallee}(n, ck)$$

By transitivity of $\sqsubseteq$ we get:

$$f_1'(ck) \sqsubseteq \mathit{callerToCallee}(n, ck)$$

Using the definition of F_i, we get:

$$f_2(ck) = \mathit{callerToCallee}(n, ck)$$

And therefore:

$$f_1'(ck) \sqsubseteq f_2(ck)$$

$*$ **Call-strings strategy**
Let $f_1 = \alpha_i(F_{ic}(n, m_{ic})[1])$ and $f_2 = F_i(n, m_i)[1]$. We must show $\forall\, ck \in CK\ .\ f_1(ck) \sqsubseteq f_2(ck)$.
Pick $ck \in CK$, and show $f_1(ck) \sqsubseteq f_2(ck)$.
Using the definition of α_i and of F_{ic}, we get:

$$f_1(ck) = \alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid t \in m_{ic}(\mathit{in}(n)[0]) \wedge \mathit{matchesContour}(\mathit{step}(t), ck)\})$$

Using the definition of F_i, we get:

$$f_2(ck) = \bigsqcup\{\mathit{callerToCallee}(n, m_i(\mathit{in}(n)[0])(ck')) \mid ck = BC(ck', m_i(\mathit{in}(n)[0])(ck'), n)\}$$

Or equivalently:

$$\begin{gathered}
f_2(ck) = \textstyle\bigsqcup_{ck' \in X}\{\mathit{callerToCallee}(n, m_i(\mathit{in}(n)[0])(ck'))\} \\
\text{where } X = \{ck' \mid ck = BC(ck', m_i(\mathit{in}(n)[0])(ck'), n)\}
\end{gathered} \tag{6.6}$$

Since $\alpha_i(m_{ic}) \sqsubseteq_i m_i$, we get:

$$\begin{aligned}
& \alpha_i(m_{ic}(\mathit{in}(n)[0])) \sqsubseteq_i m_i(\mathit{in}(n)[0]) \\
\Rightarrow\quad & \forall\, ck\ .\ \alpha_i(m_{ic}(\mathit{in}(n)[0]))(ck) \sqsubseteq m_i(\mathit{in}(n)[0])(ck) \\
\Rightarrow\quad & \forall\, ck\ .\ \alpha(\{\mathit{lastState}(t) \mid t \in m_{ic}(\mathit{in}(n)[0]) \wedge \mathit{matchesContour}(t, ck)\}) \sqsubseteq m_i(\mathit{in}(n)[0])(ck)
\end{aligned}$$

Since $t \in m_{ic}(\mathit{in}(n)[0])$, t is about to execute a call. Using lemma 9 with $X = \{t \mid t \in m_{ic}(\mathit{in}(n)[0]) \wedge \mathit{matchesContour}(t, ck)\}$, we get:

$$\begin{aligned}
\forall\, ck\ .\ \alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid t \in m_{ic}(\mathit{in}(n)[0]) \wedge \mathit{matchesContour}(t, ck)\}) \sqsubseteq \\
\mathit{callerToCallee}(n, m_i(\mathit{in}(n)[0])(ck))
\end{aligned} \tag{6.7}$$

Since we know $\forall\ t \in Traces\ .\ \exists\ ck \in CK\ .\ matchesContour(t, ck)$, we can re-express the definition of f_1 as:

$$f_1(ck) = \alpha(\textstyle\bigcup_{ck' \in CK}\{lastState(step(t)) \mid \begin{array}{l} t \in m_{ic}(in(n)[0]) \wedge \\ matchesContour(step(t), ck) \wedge \\ matchesContour(t, ck')\}) \end{array} \tag{6.8}$$

All we did here was re-express the argument to α as a union over all the ck', and for each ck' we only look at those traces that satisfy $matchesContour(t, ck')$. This definition is equivalent to the original one because we know that each trace t will be covered by some ck'.
Let $P(t, ck') \triangleq t \in m_{ic}(in(n)[0]) \wedge matchesContour(step(t), ck) \wedge matchesContour(t, ck')$ so that (6.8) becomes:

$$f_1(ck) = \alpha(\bigcup_{ck' \in CK}\{lastState(step(t)) \mid P(t, ck')\}) \tag{6.9}$$

Using (6.5) and (6.9) we get:

$$f_1(ck) \sqsubseteq \bigsqcup_{ck' \in CK}\{\alpha(\{lastState(step(t)) \mid P(t, ck')\})\} \tag{6.10}$$

Since we are using call-strings, from lemma 13 we get:

$$P(t, ck') \Rightarrow ck = BC(ck', m_i(in(n)[0])(ck'), n)$$

Then (6.10) becomes:

$$f_1(ck) \sqsubseteq \bigsqcup_{ck' \in CK}\{\alpha(\{lastState(step(t)) \mid P(t, ck') \wedge ck = BC(ck', m_i(in(n)[0])(ck'), n)\})\} \tag{6.11}$$

which becomes:

$$\begin{array}{l} f_1(ck) \sqsubseteq \bigsqcup_{ck' \in X}\{\alpha(\{lastState(step(t)) \mid P(t, ck')\})\} \\ \text{where } X = \{ck' \mid ck = BC(ck', m_i(in(n)[0])(ck'), n)\} \end{array} \tag{6.12}$$

Equation (6.12) is equivalent to (6.11) because for whatever ck' that is not in X, $ck = BC(ck', m_i(in(n)[0])(ck'), n)$ will be false in (6.11), making the set that is a parameter to α the empty set. Since $\alpha(\emptyset) = \bot$, this ck' does not affect the join in (6.11), and thus can be omitted.
Now, notice the parallel between equations (6.6) and (6.12).
By the monotonicity of $\sqcup$, which says $(a \sqsubseteq b) \wedge (c \sqsubseteq d) \Rightarrow (a \sqcup c) \sqsubseteq (b \sqcup d)$, all we need to show now is that for any $ck' \in X$, we have:

$$\alpha(\{lastState(step(t)) \mid P(t, ck')\}) \sqsubseteq callerToCallee(n, m_i(in(n)[0])(ck')) \tag{6.13}$$

We therefore pick a $ck' \in X$ and prove (6.13). We instantiate (6.7) with ck' to get:

$$\begin{array}{l} \alpha(\{lastState(step(t)) \mid t \in m_{ic}(in(n)[0]) \wedge matchesContour(t, ck')\}) \sqsubseteq \\ \qquad\qquad callerToCallee(n, m_i(in(n)[0])(ck')) \end{array} \tag{6.14}$$

Since $P(t, ck') \Rightarrow [t \in m_{ic}(in(n)[0]) \wedge \mathit{matchesContour}(t, ck')\}]$, we get:

$$\{\mathit{lastState}(\mathit{step}(t)) \mid P(t, ck')\} \subseteq \{\mathit{lastState}(\mathit{step}(t)) \mid t \in m_{ic}(in(n)[0]) \wedge \mathit{matchesContour}(t, ck')\}$$

which by the monotonicity of α gives:

$$\alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid P(t, ck')\}) \sqsubseteq \alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid t \in m_{ic}(in(n)[0]) \wedge \mathit{matchesContour}(t, ck')\})$$

which combined with (6.14) and transitivity of $\sqsubseteq$ gives:

$$\alpha(\{\mathit{lastState}(\mathit{step}(t)) \mid P(t, ck')\}) \sqsubseteq m_i(in(n)[0])(ck')$$

This completes the proof of (6.13), which is what we needed to prove.

- **Case where** $\mathit{stmtAt}(n) \neq [\mathit{Call} \ldots]$

We want to show $\alpha_i(F_{ic}(n, m_{ic})[k]) \sqsubseteq_i F_i(n, m_i)[k]$ for $k \in [1..l]$, where l is the length of $out(n)$.

Let $f_1 = \alpha_i(F_{ic}(n, m_{ic})[k])$ and $f_2 = F_i(n, m_i)[k]$. We must show $\forall\ ck \in CK\ .\ f_1(ck) \sqsubseteq f_2(ck)$. Pick $ck \in CK$, and show $f_1(ck) \sqsubseteq f_2(ck)$.

Using the definition of α_i and F_{ic}, we get:

$$\begin{aligned} f_1(ck) = \alpha(\{\mathit{lastState}(t) \mid \quad & \mathit{matchesContour}(t, ck) \wedge \mathit{edge}(\mathit{lastMachineConfig}(t)) = out(n)[k] \wedge \\ & \exists\, t' \in \mathit{Trace}, i \in \mathit{Natural}\ .\ [t' \in m_{ic}(in(n)[i]) \wedge t' \to t]\}) \end{aligned} \tag{6.15}$$

Using the definition of F_i, we get:

$$f_2(ck) = F(n, [d_1, \ldots, d_j])[k] \text{ where } d_i = m_i(in(n)[i])(ck) \text{ for } i \in [1..j] \tag{6.16}$$

From the soundness of F, we know:

$$\begin{aligned} & \forall (n, cs, ds) \in \mathit{Node} \times D_c^* \times D^*\ . \\ & \quad \alpha(cs) \sqsubseteq ds \Rightarrow \alpha(F_c(n, cs)) \sqsubseteq F(n, ds) \end{aligned} \tag{6.17}$$

Let $cs \triangleq [c_1, \ldots, c_j]$ where $c_i = \{\mathit{lastState}(t) \mid t \in m_{ic}(in(n)[i]) \wedge \mathit{matchesContour}(t, ck)\}$ for $i \in [1..j]$

Let $ds \triangleq [d_1, \ldots, d_j]$ where $d_i = m_i(in(n)[i](ck)$ for $i \in [1..j]$

We have:

$$\begin{array}{lll} & \alpha_i(m_{ic}) \sqsubseteq_i m_i & \\ \Rightarrow & \forall i \in [1..j]\ .\ \alpha_i(m_{ic}(in(n)[i])) \sqsubseteq_i m_i(in(n)[i]) & \\ \Rightarrow & \forall i \in [1..j]\ .\ \alpha_i(m_{ic}(in(n)[i]))(ck) \sqsubseteq_i m_i(in(n)[i])(ck) & \\ \Rightarrow & \forall i \in [1..j]\ .\ \alpha_i(m_{ic}(in(n)[i]))(ck) \sqsubseteq_i d_i & (\text{defn of } d_i) \\ \Rightarrow & \forall i \in [1..j]\ .\ \alpha(\{\mathit{lastState}(t) \mid t \in m_{ic}(in(n)[i]) \wedge \mathit{matchesContour}(t, ck)\}) \sqsubseteq d_i & (\text{defn of } \alpha_i) \\ \Rightarrow & \forall i \in [1..j]\ .\ \alpha(c_i) \sqsubseteq d_i & (\text{defn of } c_i) \\ \Rightarrow & \alpha(cs) \sqsubseteq ds & (\text{defn of } cs \text{ and } ds) \\ \Rightarrow & \alpha(F_c(n, cs)) \sqsubseteq F(n, ds) & (\text{using (6.17)}) \\ \Rightarrow & \alpha(F_c(n, cs)[k]) \sqsubseteq F(n, ds)[k] & \\ \Rightarrow & \alpha(F_c(n, cs)[k]) \sqsubseteq f_2(ck) & (\text{using (6.16)}) \end{array}$$

Because of transitivity of $\sqsubseteq$, to show $f_1(ck) \sqsubseteq f_2(ck)$, we now only need to show:

$$f_1(ck) \sqsubseteq \alpha(F_c(n, cs)[k]) \tag{6.18}$$

From equation (6.15), we have:

$$\begin{aligned}
&f_1(ck) = \alpha(X) \\
&\quad \text{where } X = \{\mathit{lastState}(t) \mid \quad \mathit{matchesContour}(t, ck) \wedge \mathit{edge}(\mathit{lastMachineConfig}(t)) = \mathit{out}(n)[k] \wedge \\
&\qquad\qquad\qquad\qquad\qquad\quad \exists\, t' \in \mathit{Trace}, i \in \mathit{Natural}\ .\ [t' \in m_{ic}(\mathit{in}(n)[i]) \wedge t' \to t]\})
\end{aligned}$$

Using the definition of F_c from equation (2.1), we get:

$$\begin{aligned}
&\alpha(F_c(n, cs)[k]) = \alpha(Y) \\
&\quad \text{where } Y = \{\eta \mid \exists\, \eta' \in \mathit{State}, i \in \mathit{Natural}\ .\ [\eta' \in cs[i] \wedge i, \eta' \overset{n}{\hookrightarrow} k, \eta]\})
\end{aligned}$$

Thus (6.18) reduces to $\alpha(X) \sqsubseteq \alpha(Y)$, which by the monotonicity of α, reduces to showing that $X \sqsubseteq_c Y$, or $X \subseteq Y$.

So we assume $\eta \in X$, and show $\eta \in Y$.

Because $\eta \in X$, we know that there exists $t \in \mathit{Trace}$, $t' \in \mathit{Trace}$ and $i \in Natural$ such that:

$$\eta = \mathit{lastState}(t) \tag{6.19}$$
$$\mathit{matchesContour}(t, ck)\ \wedge \tag{6.20}$$
$$\mathit{edge}(\mathit{lastMachineConfig}(t)) = \mathit{out}(n)[k]\ \wedge \tag{6.21}$$
$$t' \in m_{ic}(\mathit{in}(n)[i])\ \wedge \tag{6.22}$$
$$t' \to t \tag{6.23}$$

To show that $\eta \in Y$, we need to show:

$$\exists\, \eta' \in \mathit{State}, i' \in \mathit{Natural}\ .\ [\eta' \in cs[i'] \wedge i', \eta' \overset{n}{\hookrightarrow} k, \eta] \tag{6.24}$$

The claim is that $\eta' = \mathit{lastState}(t')$ and $i' = i$ make the existential (6.24) valid. To show this we need to show:

$$\mathit{lastState}(t') \in cs[i]\ \wedge \tag{6.25}$$
$$i, \mathit{lastState}(t') \overset{n}{\hookrightarrow} k, \eta \tag{6.26}$$

Because of (6.19), (6.26) is equivalent to $i, \mathit{lastState}(t') \overset{n}{\hookrightarrow} k, \mathit{lastState}(t)$.

Because $\mathit{stmtAt}(n)$ is not a call, this is equivalent to $i, \mathit{lastState}(t') \overset{n}{\to} k, \mathit{lastState}(t)$, which follows from (6.21), (6.22) and (6.23).

To show (6.25), we recall that from the definition of cs, we have:

$$cs[i] = \{\mathit{lastState}(t) \mid t \in m_{ic}(\mathit{in}(n)[i]) \wedge \mathit{matchesContour}(t, \mathit{ck})\}$$

Thus to show (6.25), we just need to show:

$$t' \in m_{ic}(\mathit{in}(n)[i]) \ \wedge \tag{6.27}$$
$$\mathit{matchesContour}(t', \mathit{ck}) \tag{6.28}$$

We know (6.27) from (6.22).

When t' steps to t, it cannot be executing a call because $\mathit{stmtAt}(n) \neq [\mathit{Call} \ldots]$. Therefore $\mathit{currentCall}(t') = \mathit{currentCall}(t)$. Using lemma 11 with $t_1 = t$ and $t_2 = t'$, we get $\mathit{matchesContour}(t, \mathit{ck}) = \mathit{matchesContour}(t', \mathit{ck})$. Since we know $\mathit{matchesContour}(t, \mathit{ck})$ from (6.20), we get $\mathit{matchesContour}(t', \mathit{ck})$, which shows (6.28).

■

Bibliography

[1] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles CA, January 1977.

[2] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio TX, January 1979.

[3] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland OR, January 2002.

[4] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automatically proving the correctness of program analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Long Beach OR, January 2005.

Appendix A

Rhodium code

We present here all of our Rhodium code. This code is copy-and-pasted from our current Rhodium file. The following code has been checked for soundness automatically using our automated soundness checking strategy.

```
-- Some global declarations using simple naming conventions
-- (X,Y,Z,I,J,A,B are vars, C's are contants, etc.), so that we don't
-- have to declare variables on each rule.
decl X:Var, Y:Var, C:Const, C1:Const, C2:Const, C3:Const, L1:Label, L2:Label,
     A:Var, B:Var, P:Fun, Z:Var, E:NonCallExpr, E1:NonCallExpr,
     E2:NonCallExpr, E3:NonCallExpr, S:Stmt, Ed:Edge,
     OP:BinaryOp, I:Var, J:Var, L:Loc, HS:HeapSummary, HS1:HeapSummary,
     HS2:HeapSummary, N:CFGNode, BE:BaseExpr, BE1:BaseExpr, BE2:BaseExpr in

-------------------------------------------------------------------------
--
-- stmt fact
--
-------------------------------------------------------------------------

define node fact stmt(S:Stmt) = eq(S, currStmt)

-------------------------------------------------------------------------
--
-- syntacticDef
--
-------------------------------------------------------------------------

define node fact syntacticDef(Z:Var) =
      case currStmt of
         [skip]                  => false
         [decl X]                => eq(Z,X)
         [decl X[I]]             => eq(Z,X)
         [if X goto L1 else L2]  => false
         [if C goto L1 else L2]  => false
```

```
        [X := C]                 => eq(Z,X)
        [X := Y]                 => eq(Z,X)
        [*X := Y]                => false
        [X := *Y]                => eq(Z,X)
        [X := &Y]                => eq(Z,X)
        [X := new]               => eq(Z,X)
        [X := A ‘OP B]           => eq(Z,X)
        [X := A ‘OP C]           => eq(Z,X)
        [X := C ‘OP B]           => eq(Z,X)
        [X := C1 ‘OP C2]         => eq(Z,X)
        [X := P(Y)]              => eq(Z,X)
        [X := new [I]]          => eq(Z,X)
        [X := A[I]]              => eq(Z,X)
        [X := *(A[I])]           => eq(Z,X)
        [*(A[I]) := X]           => false
        [*(A[I]) := C]           => false
        [X := (&A)[I]]           => eq(Z,X)
        [X := *((&A)[I])]        => eq(Z,X)
        [*((&A)[I]) := X]        => false
        [*((&A)[I]) := C]        => false
        [return X]               => false
     endcase

------------------------------------------------------------------------
--
-- syntacticDefConserv
--   This defines a version of syntacticDef that is conservative for
--   assignment of an array element’s value to a variable.  This
--   conservative definition is used only in one rule for
--   equalsTimes. That rule terminates with the conservative version
--   but not with the less conservative version.
--
------------------------------------------------------------------------

define node fact syntacticDefConserv(Z:Var) =
     case currStmt of
        [skip]                   => false
        [decl X]                 => eq(Z,X)
        [decl X[I]]              => eq(Z,X)
        [if X goto L1 else L2]   => false
        [if C goto L1 else L2]   => false
        [X := C]                 => eq(Z,X)
        [X := Y]                 => eq(Z,X)
        [*X := Y]                => false
        [X := *Y]                => eq(Z,X)
        [X := &Y]                => eq(Z,X)
        [X := new]               => eq(Z,X)
        [X := A ‘OP B]           => eq(Z,X)
        [X := A ‘OP C]           => eq(Z,X)
        [X := C ‘OP B]           => eq(Z,X)
        [X := C1 ‘OP C2]         => eq(Z,X)
        [X := P(Y)]              => eq(Z,X)
```

```
         [X := new [I]]           => eq(Z,X)
         [X := A[I]]              => eq(Z,X)
         [X := *(A[I])]           => true
         [*(A[I]) := X]           => false
         [*(A[I]) := C]           => false
         [X := (&A)[I]]           => eq(Z,X)
         [X := *((&A)[I])]        => eq(Z,X)
         [*((&A)[I]) := X]        => false
         [*((&A)[I]) := C]        => false
         [return X]               => false
      endcase

-------------------------------------------------------------------------
--
-- syntacticUse
--
-------------------------------------------------------------------------

define node fact syntacticUse(Z:Var) =
      case currStmt of
         [skip]                 => false
         [decl X]               => false
         [decl X[I]]            => false
         [if X goto L1 else L2] => eq(Z,X)
         [if C goto L1 else L2] => false
         [X := C]               => false
         [X := Y]               => eq(Z,Y)
         [*X := Y]              => eq(Z,Y) || eq(Z,X)
         [X := *Y]              => eq(Z,Y)
         [X := &Y]              => eq(Z,Y)
         [X := new]             => false
         [X := A ‘OP B]         => eq(Z,A) || eq(Z,B)
         [X := A ‘OP C]         => eq(Z,A)
         [X := C ‘OP B]         => eq(Z,B)
         [X := C1 ‘OP C2]       => false
         [X := P(Y)]            => eq(Z,Y)
         [X := new [I]]         => eq(Z,I)
         [X := A[I]]            => eq(Z,I) || eq(Z,A)
         [X := *(A[I])]         => eq(Z,I) || eq(Z,A)
         [*(A[I]) := X]         => eq(Z,A) || eq(Z,I) || eq(Z,X)
         [*(A[I]) := C]         => eq(Z,A) || eq(Z,I)
         [X := (&A)[I]]         => eq(Z,I) || eq(Z,A)
         [X := *((&A)[I])]      => eq(Z,I) || eq(Z,A)
         [*((&A)[I]) := X]      => eq(Z,A) || eq(Z,I) || eq(Z,X)
         [*((&A)[I]) := C]      => eq(Z,A) || eq(Z,I)
         [return X]             => eq(Z,X)
     endcase

-------------------------------------------------------------------------
--
-- syntacticUnchanged
--
```

```
------------------------------------------------------------------------

define node fact syntacticUnchanged(E:NonCallExpr) =
      case E of
         [Z]           => !syntacticDef(Z)@currNode
         [&Z]          => !syntacticDef(Z)@currNode
         [C]           => true
         [C1 ‘OP C2]  => true
         [X ‘OP Y]    => !syntacticDef(X)@currNode && !syntacticDef(Y)@currNode
         [X ‘OP C]    => !syntacticDef(X)@currNode
         [C ‘OP X]    => !syntacticDef(X)@currNode
         [*Z]         => !syntacticDef(Z)@currNode
         [*(A[I])]    => !syntacticDef(A)@currNode && !syntacticDef(I)@currNode
         [A[I]]       => !syntacticDef(A)@currNode && !syntacticDef(I)@currNode
         [*((&A)[I])] => !syntacticDef(A)@currNode && !syntacticDef(I)@currNode
         [(&A)[I]]    => !syntacticDef(A)@currNode && !syntacticDef(I)@currNode
      endcase


------------------------------------------------------------------------
--
-- mayDef
--
------------------------------------------------------------------------

define node fact mayDef(Z:Var) =
      case currStmt of
         [skip]                 => syntacticDef(Z)@currNode
         [decl X]               => syntacticDef(Z)@currNode
         [decl X[I]]            => syntacticDef(Z)@currNode
         [if X goto L1 else L2] => syntacticDef(Z)@currNode
         [if C goto L1 else L2] => syntacticDef(Z)@currNode
         [X := C]               => syntacticDef(Z)@currNode
         [X := Y]               => syntacticDef(Z)@currNode
         [*X := Y]              => syntacticDef(Z)@currNode ||
                                   mayPointTo(X,Z)@cfg_in
         [X := *Y]              => syntacticDef(Z)@currNode
         [X := &Y]              => syntacticDef(Z)@currNode
         [X := new]             => syntacticDef(Z)@currNode
         [X := A ‘OP B]         => syntacticDef(Z)@currNode
         [X := A ‘OP C]         => syntacticDef(Z)@currNode
         [X := C ‘OP B]         => syntacticDef(Z)@currNode
         [X := C1 ‘OP C2]       => syntacticDef(Z)@currNode
         [X := P(Y)]            => true --syntacticDef(Z)@currNode ||
                                   --!isNotTainted(Z)@cfg_in
         [X := new [I]]         => syntacticDef(Z)@currNode
         [X := A[I]]            => syntacticDef(Z)@currNode
         [X := *(A[I])]         => syntacticDef(Z)@currNode
         [*(A[I]) := X]         => syntacticDef(Z)@currNode
         [*(A[I]) := C]         => syntacticDef(Z)@currNode
         [X := (&A)[I]]         => syntacticDef(Z)@currNode
         [X := *((&A)[I])]      => syntacticDef(Z)@currNode
```

```
        [*((&A)[I]) := X]      => syntacticDef(Z)@currNode
        [*((&A)[I]) := C]      => syntacticDef(Z)@currNode
        [return X]             => syntacticDef(Z)@currNode
    endcase


-----------------------------------------------------------------------
--
-- mayDefConserv
--   This defines a version of mayDef that is conservative
--   because it uses the conservative definition of syntacticDef.
--
-----------------------------------------------------------------------

define node fact mayDefConserv(Z:Var) =
      case currStmt of
        [skip]                 => syntacticDefConserv(Z)@currNode
        [decl X]               => syntacticDefConserv(Z)@currNode
        [decl X[I]]            => syntacticDefConserv(Z)@currNode
        [if X goto L1 else L2] => syntacticDefConserv(Z)@currNode
        [if C goto L1 else L2] => syntacticDefConserv(Z)@currNode
        [X := C]               => syntacticDefConserv(Z)@currNode
        [X := Y]               => syntacticDefConserv(Z)@currNode
        [*X := Y]              => syntacticDefConserv(Z)@currNode ||
                                  mayPointTo(X,Z)@cfg_in
        [X := *Y]              => syntacticDefConserv(Z)@currNode
        [X := &Y]              => syntacticDefConserv(Z)@currNode
        [X := new]             => syntacticDefConserv(Z)@currNode
        [X := A ‘OP B]         => syntacticDefConserv(Z)@currNode
        [X := A ‘OP C]         => syntacticDefConserv(Z)@currNode
        [X := C ‘OP B]         => syntacticDefConserv(Z)@currNode
        [X := C1 ‘OP C2]       => syntacticDefConserv(Z)@currNode
        [X := P(Y)]            => true --syntacticDef(Z)@currNode ||
                                  --!isNotTainted(Z)@cfg_in
        [X := new [I]]         => syntacticDefConserv(Z)@currNode
        [X := A[I]]            => syntacticDefConserv(Z)@currNode
        [X := *(A[I])]         => syntacticDefConserv(Z)@currNode
        [*(A[I]) := X]         => syntacticDefConserv(Z)@currNode
        [*(A[I]) := C]         => syntacticDefConserv(Z)@currNode
        [X := (&A)[I]]         => syntacticDefConserv(Z)@currNode
        [X := *((&A)[I])]      => syntacticDefConserv(Z)@currNode
        [*((&A)[I]) := X]      => syntacticDefConserv(Z)@currNode
        [*((&A)[I]) := C]      => syntacticDefConserv(Z)@currNode
        [return X]             => syntacticDefConserv(Z)@currNode
    endcase

-----------------------------------------------------------------------
--
-- mayUse
--
-----------------------------------------------------------------------
```

```
define node fact mayUse(Z:Var) =
      case currStmt of
         [skip]                 => syntacticUse(Z)@currNode
         [decl X]               => syntacticUse(Z)@currNode
         [decl X[I]]            => syntacticUse(Z)@currNode
         [if X goto L1 else L2] => syntacticUse(Z)@currNode
         [if C goto L1 else L2] => syntacticUse(Z)@currNode
         [X := C]               => syntacticUse(Z)@currNode
         [X := Y]               => syntacticUse(Z)@currNode
         [*X := Y]              => syntacticUse(Z)@currNode
         [X := *Y]              => syntacticUse(Z)@currNode
                                   || mayPointTo(Y,Z)@cfg_in
         [X := &Y]              => syntacticUse(Z)@currNode
         [X := new]             => syntacticUse(Z)@currNode
         [X := A ‘OP B]         => syntacticUse(Z)@currNode
         [X := A ‘OP C]         => syntacticUse(Z)@currNode
         [X := C ‘OP B]         => syntacticUse(Z)@currNode
         [X := C1 ‘OP C2]       => syntacticUse(Z)@currNode
         [X := P(Y)]            => true
         [X := new [I]]         => true
         [X := A[I]]            => true
         [X := *(A[I])]         => true
         [*(A[I]) := X]         => true
         [*(A[I]) := C]         => true
         [X := (&A)[I]]         => true
         [X := *((&A)[I])]      => true
         [*((&A)[I]) := X]      => true
         [*((&A)[I]) := C]      => true
         [return X]             => syntacticUse(Z)@currNode
      endcase

------------------------------------------------------------------------
--
-- unchanged
--
------------------------------------------------------------------------

define node fact unchanged(E:NonCallExpr) =
      case E of
         [Z]           => !mayDef(Z)@currNode
         [&Z]          => !mayDef(Z)@currNode
         [C]           => true
         [C1 ‘OP C2]   => true
         [X ‘OP Y]     => !mayDef(X)@currNode && !mayDef(Y)@currNode
         [X ‘OP C]     => !mayDef(X)@currNode
         [C ‘OP X]     => !mayDef(X)@currNode
         [*Z]          =>
            case currStmt of
               [skip]                 => !mayDef(Z)@currNode
               [decl X]               => !mayDef(Z)@currNode
               [decl X[I]]            => !mayDef(Z)@currNode
               [if X goto L1 else L2] => !mayDef(Z)@currNode
               [if C goto L1 else L2] => !mayDef(Z)@currNode
               [X := C]               => !mayDef(Z)@currNode &&
                                         dnpHeapSummary(Z,X)@cfg_in
               [X := Y]               => !mayDef(Z)@currNode &&
                                         dnpHeapSummary(Z,X)@cfg_in
               [*X := Y]              => false
               [X := *Y]              => !mayDef(Z)@currNode &&
                                         dnpHeapSummary(Z,X)@cfg_in
               [X := &Y]              => !mayDef(Z)@currNode &&
                                         dnpHeapSummary(Z,X)@cfg_in
               [X := new]             => !mayDef(Z)@currNode &&
                                         dnpHeapSummary(Z,X)@cfg_in
               [X := A ‘OP B]         => !mayDef(Z)@currNode &&
                                         dnpHeapSummary(Z,X)@cfg_in
               [X := A ‘OP C]         => !mayDef(Z)@currNode &&
                                         dnpHeapSummary(Z,X)@cfg_in
               [X := C ‘OP B]         => !mayDef(Z)@currNode &&
                                         dnpHeapSummary(Z,X)@cfg_in
               [X := C1 ‘OP C2]       => !mayDef(Z)@currNode &&
                                         dnpHeapSummary(Z,X)@cfg_in
               [X := P(Y)]            => false
               [X := new [I]]         => false
               [X := A[I]]            => false
               [X := *(A[I])]         => false
               [*(A[I]) := X]         => false
               [*(A[I]) := C]         => false
               [X := (&A)[I]]         => false
               [X := *((&A)[I])]      => false
               [*((&A)[I]) := X]      => false
               [*((&A)[I]) := C]      => false
               [return X]             => !mayDef(Z)@currNode
            endcase
         [A[I]]        => false
         [*(A[I])]     => false
         [(&A)[I]]     => false
         [*((&A)[I])]  => false

      endcase

------------------------------------------------------------------------
--
-- unchangedConserv
--   This defines a version of unchanged that is conservative
--   because is uses the conservative definition of mayDef.
--
------------------------------------------------------------------------

define node fact unchangedConserv(E:NonCallExpr) =
      case E of
         [Z]           => !mayDefConserv(Z)@currNode
         [&Z]          => !mayDefConserv(Z)@currNode
         [C]           => true
         [C1 ‘OP C2]   => true
         [X ‘OP Y]     => !mayDefConserv(X)@currNode &&
                          !mayDefConserv(Y)@currNode
         [X ‘OP C]     => !mayDefConserv(X)@currNode
         [C ‘OP X]     => !mayDefConserv(X)@currNode
         [*Z]          =>
            case currStmt of
               [skip]                  => !mayDefConserv(Z)@currNode
               [decl X]                => !mayDefConserv(Z)@currNode
               [decl X[I]]             => !mayDefConserv(Z)@currNode
               [if X goto L1 else L2]  => !mayDefConserv(Z)@currNode
               [if C goto L1 else L2]  => !mayDefConserv(Z)@currNode
               [X := C]                => !mayDefConserv(Z)@currNode &&
                                          doesNotPointTo(Z,X)@cfg_in
               [X := Y]                => !mayDefConserv(Z)@currNode &&
                                          doesNotPointTo(Z,X)@cfg_in
               [*X := Y]               => false
               [X := *Y]               => !mayDefConserv(Z)@currNode &&
                                          doesNotPointTo(Z,X)@cfg_in
               [X := &Y]               => !mayDefConserv(Z)@currNode &&
                                          doesNotPointTo(Z,X)@cfg_in
               [X := new]              => !mayDefConserv(Z)@currNode &&
                                          doesNotPointTo(Z,X)@cfg_in
               [X := A ‘OP B]          => !mayDefConserv(Z)@currNode &&
                                          doesNotPointTo(Z,X)@cfg_in
               [X := A ‘OP C]          => !mayDefConserv(Z)@currNode &&
                                          doesNotPointTo(Z,X)@cfg_in
               [X := C ‘OP B]          => !mayDefConserv(Z)@currNode &&
                                          doesNotPointTo(Z,X)@cfg_in
               [X := C1 ‘OP C2]        => !mayDefConserv(Z)@currNode &&
                                          doesNotPointTo(Z,X)@cfg_in
               [X := P(Y)]             => false
               [X := new [I]]          => false
               [X := A[I]]             => false
               [X := *(A[I])]          => false
               [*(A[I]) := X]          => false
               [*(A[I]) := C]          => false
               [X := (&A)[I]]          => false
               [X := *((&A)[I])]       => false
               [*((&A)[I]) := X]       => false
               [*((&A)[I]) := C]       => false
               [return X]              => !mayDefConserv(Z)@currNode
            endcase
         [A[I]]        => false
         [*(A[I])]     => false
         [(&A)[I]]     => false
         [*((&A)[I])]  => false

      endcase

-------------------------------------------------------------------------
--
```

```
-- addrTaken
--
------------------------------------------------------------------------

define node fact addrTaken(Z:Var) =
      case currStmt of
         [skip]                 => false
         [decl X]               => false
         [decl X[I]]            => false
         [if X goto L1 else L2] => false
         [if C goto L1 else L2] => false
         [X := C]               => false
         [X := Y]               => false
         [*X := Y]              => false
         [X := *Y]              => false
         [X := &Y]              => eq(Z,Y)
         [X := new]             => false
         [X := A ‘OP B]         => false
         [X := A ‘OP C]         => false
         [X := C ‘OP B]         => false
         [X := C1 ‘OP C2]       => false
         [X := P(Y)]            => false
         [X := new [I]]         => false
         -- This may be able to be made less conservative
         [X := A[I]]            => true
         [X := *(A[I])]         => false
         [*(A[I]) := X]         => false
         [*(A[I]) := C]         => false
         -- This may be able to be made less conservative
         [X := (&A)[I]]         => true
         [X := *((&A)[I])]      => false
         [*((&A)[I]) := X]      => false
         [*((&A)[I]) := C]      => false
         [return X]             => false
      endcase

------------------------------------------------------------------------
--
-- isNotTainted analysis
--
------------------------------------------------------------------------

-- We stopped using this analysis
-- define edge fact isNotTainted(X:Var)
--   -- The meaning says that for any location L, the result of
--   -- evaluating *L (using evalLocDeref, which is a primitive, much
--   -- like evalExpr) is not equal to &X. In other words, no store
--   -- location points to X.
--   with meaning  forall L:Loc . isLoc(L) => neq(evalLocDeref(eta, L),
--                                                evalExpr(eta, [&X]))
--
--   if stmt([decl X])@currNode
--   then isNotTainted(X)@cfg_out
--
--   if isNotTainted(X)@cfg_in && !addrTaken(X)@currNode
--   then isNotTainted(X)@cfg_out

-----------------------------------------------------------------------
--
-- Andersen pointer analysis
--
-----------------------------------------------------------------------

define edge fact doesNotPointTo(X:Var, Y:Var)
   with meaning neq(X, [&Y])

   if stmt([*A := B])@currNode &&
      doesNotPointTo(B, Y)@cfg_in &&
      doesNotPointTo(X, Y)@cfg_in
   then doesNotPointTo(X, Y)@cfg_out

   if doesNotPointTo(X, Y)@cfg_in && !mayDef(X)@currNode
   then doesNotPointTo(X, Y)@cfg_out

   if stmt([X := &A])@currNode && hasBeenDeclared(Y)@cfg_in && neq(A, Y)
   then doesNotPointTo(X, Y)@cfg_out

   if stmt([X := BE1 ‘OP BE2])@currNode && hasBeenDeclared(Y)@cfg_in
   then doesNotPointTo(X, Y)@cfg_out

   if stmt([X := A])@currNode && doesNotPointTo(A, Y)@cfg_in
   then doesNotPointTo(X, Y)@cfg_out

   if stmt([*A := B])@currNode &&
      mustPointTo(A, X)@cfg_in &&
      doesNotPointTo(B, Y)@cfg_in
   then doesNotPointTo(X, Y)@cfg_out

   if stmt([X := *A])@currNode &&
      doesNotPointToHeap(A)@cfg_in &&
      forall B:Var . !doesNotPointTo(A,B)@cfg_in =>
                     doesNotPointTo(B,Y)@cfg_in
   then doesNotPointTo(X,Y)@cfg_out


define virtual edge fact mayPointTo(X:Var, Y:Var)@Ed = !dnpHeapSummary(X,Y)@Ed

define edge fact mustPointTo(X:Var, Y:Var)
   with meaning eq(X, [&Y])

   if stmt([X := &Y])@currNode
   then mustPointTo(X, Y)@cfg_out

   if mustPointTo(X, Y)@cfg_in && !mayDef(X)@currNode
   then mustPointTo(X, Y)@cfg_out

   if stmt([X := A])@currNode && mustPointTo(A,Y)@cfg_in
   then mustPointTo(X, Y)@cfg_out

define edge fact hasBeenDeclared(X:Var)
   -- This witness says that X is not stuck in the machine state eta,
   -- which basically means that X is in the domain of the environment
   -- component of eta. The predicate symbol isExprNotStuck is a primitive.
   with meaning isExprNotStuck(eta, X)

   if stmt([decl X])@currNode
   then hasBeenDeclared(X)@cfg_out

   if hasBeenDeclared(X)@cfg_in
   then hasBeenDeclared(X)@cfg_out

define edge fact doesNotPointToHeap(X:Var)
   with meaning exists Y:Var. eq(X, [&Y])

   if stmt([X := &Y])@currNode
   then doesNotPointToHeap(X)@cfg_out

   if doesNotPointToHeap(X)@cfg_in && !mayDef(X)@currNode
   then doesNotPointToHeap(X)@cfg_out

-------------------------------------------------------------------------
--
-- Self assignment removal
--
-------------------------------------------------------------------------

if stmt([X := X])@currNode
then transform [skip]

-------------------------------------------------------------------------
--
-- Branch folding
--
-------------------------------------------------------------------------

if stmt([if true goto L1 else L2])@currNode
then transform [if true goto L1 else L1]

if stmt([if false goto L1 else L2])@currNode
then transform [if true goto L2 else L2]

-------------------------------------------------------------------------
--
-- Constant propagation
--
-------------------------------------------------------------------------
```

```
define node fact baseExprHasConstValue(BE:BaseExpr, C:Const) =
        case BE of
            [X]     => hasConstValue(X, C)@cfg_in
            [C1]    => eq(C, C1)
        endcase

define edge fact hasConstValue(X:Var,C:Const)
   with meaning eq(X,C)

   if stmt([X := BE])@currNode && baseExprHasConstValue(BE, C)@currNode
   then hasConstValue(X,C)@cfg_out

   if stmt([X := BE1 ‘OP BE2])@currNode &&
      baseExprHasConstValue(BE1,C1)@currNode &&
      baseExprHasConstValue(BE2,C2)@currNode &&
      eq(C, newConst(applyBinaryOp(OP, getConst(C1), getConst(C2))))
   then hasConstValue(X, C)@cfg_out

   if hasConstValue(X,C)@cfg_in && !mayDef(X)@currNode
   then hasConstValue(X,C)@cfg_out

   if stmt([X := Y])@currNode && hasConstValue(Y,C)@cfg_in
   then transform [X := C]

   if stmt([*(A[I]) := Y])@currNode && hasConstValue(Y,C)@cfg_in
   then transform [*(A[I]) := C]

   if stmt([X := BE1 ‘OP BE2])@currNode &&
      baseExprHasConstValue(BE1, C1)@currNode &&
      baseExprHasConstValue(BE2, C2)@currNode &&
      eq(C, newConst(applyBinaryOp(OP, getConst(C1), getConst(C2))))
   then transform [X := C]

define edge fact hasConstValueArray(A:Var, I:Var, C:Const)
    with meaning eq([*(A[I])], C)

    if stmt([*(A[I]) := BE])@currNode &&
       baseExprHasConstValue(BE,C)@currNode
    then hasConstValueArray(A,I,C)@cfg_out

    if hasConstValueArray(A,I,C)@cfg_in &&
        !mayDefArray(A)@currNode &&
        !mayDef(I)@currNode &&
        !mayDefArrayElem(A,I)@currNode
    then hasConstValueArray(A,I,C)@cfg_out

    if stmt([X := *(A[I])])@currNode && hasConstValueArray(A,I,C)@cfg_in
    then transform [X := C]

-------------------------------------------------------------------------
--
-- CSE
```

```
--
------------------------------------------------------------------------

define edge fact exprAvail(X:Var, E:NonCallExpr)
   with meaning eq(X, E) &&
             (isConst(evalExpr(eta,X)) || (isLoc(evalExpr(eta,X))))

   if stmt([X := E])@currNode && syntacticUnchanged(E)@currNode
   then exprAvail(X, E)@cfg_out

   if exprAvail(X, E)@cfg_in && !mayDef(X)@currNode && unchanged(E)@currNode
   then exprAvail(X, E)@cfg_out

   if stmt([X := E])@currNode && exprAvail(Z, E)@cfg_in
   then transform [X := Z]

   -- Code sinking
   if stmt([skip])@currNode && exprAvail(X, E)@cfg_in
   then transform [X := E]

------------------------------------------------------------------------
--
-- Copy prop
--
------------------------------------------------------------------------

define edge fact varEqual(X:Var, Y:Var)
   with meaning eq(X, Y)

   if stmt([X := Y])@currNode
   then varEqual(X, Y)@cfg_out

   if stmt([X := Y])@currNode
   then varEqual(Y, X)@cfg_out

   if varEqual(X, Y)@cfg_in && !mayDef(X)@currNode && !mayDef(Y)@currNode
   then varEqual(X, Y)@cfg_out

   if stmt([X := Y])@currNode && varEqual(Y, Z)@cfg_in
   then transform [X := Z]

------------------------------------------------------------------------
--
-- Load removal
--
------------------------------------------------------------------------

if stmt([X := *Y])@currNode && mustPointTo(Y, Z)@cfg_in
then transform [X := Z]

------------------------------------------------------------------------
--
```

```
-- equalsTimes
--
-----------------------------------------------------------------------

define edge fact equalsTimes(X:NonCallExpr, Y:NonCallExpr, Z:NonCallExpr)
   with meaning eq(evalExpr(eta, X),
                   applyBinaryOp(*, evalExpr(eta, Y), evalExpr(eta, Z)))


   if stmt([X := I * C])@currNode && neq(X, I)
   then equalsTimes(X, I, C)@cfg_out

   -- This rule uses the conservative version of unchanged because Simplify
   -- does not terminate when the less conservative definition is used.
   if equalsTimes(E1, E2, C)@cfg_in &&
      unchangedConserv(E1)@currNode &&
      unchangedConserv(E2)@currNode
   then equalsTimes(E1, E2, C)@cfg_out

   if stmt([I := I + C1])@currNode && neq(X, I) &&
      equalsTimes(X, I, C2)@cfg_in
   then equalsTimes(X, [I-C1], C2)@cfg_out

   if stmt([X := X + C1])@currNode &&
      neq(X, I) && equalsTimes(X, [I-C2], C3)@cfg_in &&
      eq(C1, newConst(applyBinaryOp(*, getConst(C2), getConst(C3))))
   then equalsTimes(X, I, C3)@cfg_out

-- Simpler version of the above rule, when C2 == 1
--    if stmt([X := X + C])@currNode &&
--       neq(X, I) && equalsTimes(X, [I-1], C)@cfg_in
--    then equalsTimes(X, I, C)@cfg_out

   if stmt([X := X + C1])@currNode &&
      neq(X, I) && equalsTimes(X, I, C2)@cfg_in
   then equalsTimes([X-C1], I, C2)@cfg_out

   if stmt([I := I + C1])@currNode &&
      neq(X, I) && equalsTimes([X-C2], I, C3)@cfg_in &&
      eq(C2, newConst(applyBinaryOp(*, getConst(C1), getConst(C3))))
   then equalsTimes(X, I, C3)@cfg_out

-- Simpler version of the above rule, when C1 == 1
--    if stmt([I := I + 1])@currNode &&
--       neq(X, I) && equalsTimes([X-C], I, C)@cfg_in
--    then equalsTimes(X, I, C)@cfg_out

   if stmt([Y := I * C])@currNode &&
      equalsTimes(X, I, C)@cfg_in
   then transform [Y := X]

-----------------------------------------------------------------------
```

```
--
-- mayDefAllocSiteSummary(N): says if the current statement may define
-- some location in summary N.
--
-----------------------------------------------------------------------

define node fact mayDefAllocSiteSummary(N:CFGNode) =
      case currStmt of
         [skip]                  => false
         [decl X]                => false
         [decl X[I]]             => false
         [if X goto L1 else L2]  => false
         [if C goto L1 else L2]  => false
         [X := C]                => !varNotInAllocSiteSummary(X, N)@cfg_in
         [X := Y]                => !varNotInAllocSiteSummary(X, N)@cfg_in
         [*X := Y]               => !dnpHeapSummary(X, N)@cfg_in
         [X := *Y]               => !varNotInAllocSiteSummary(X, N)@cfg_in
         [X := &Y]               => !varNotInAllocSiteSummary(X, N)@cfg_in
         [X := new]              => !varNotInAllocSiteSummary(X, N)@cfg_in
         [X := A `OP B]          => !varNotInAllocSiteSummary(X, N)@cfg_in
         [X := A `OP C]          => !varNotInAllocSiteSummary(X, N)@cfg_in
         [X := C `OP B]          => !varNotInAllocSiteSummary(X, N)@cfg_in
         [X := C1 `OP C2]        => !varNotInAllocSiteSummary(X, N)@cfg_in
         [X := P(Y)]             => true
         [X := new [I]]          => true
         [X := A[I]]             => true
         [X := *(A[I])]          => true
         [*(A[I]) := X]          => true
         [*(A[I]) := C]          => true
         [X := (&A)[I]]          => true
         [X := *((&A)[I])]       => true
         [*((&A)[I]) := X]       => true
         [*((&A)[I]) := C]       => true
         [return X]              => false
      endcase

-----------------------------------------------------------------------
--
-- The parser for Rhodium does not currently support state extensions
-- or meanings that use state extensions. As a result, state
-- extensions and meanings that use state extensions must be written
-- in the Simplify input language using "prim" blocks. See the
-- following example. It is straightforward to extend the Rhodium
-- parser to support these contructs directly, and generate the
-- Simplify code automatically. We are currently working on this.
--
-----------------------------------------------------------------------

-----------------------------------------------------------------------
-- State extension for heap summaries:
--      partitions: location ==> Node
-----------------------------------------------------------------------
```

```
prim {

   (BG_PUSH (FORALL (P Prg Lhs Rhs)
      (IMPLIES (EQ (stmtAt P Prg) (Assgn Lhs Rhs))
         (FORALL (Env S Mem Stack StateExt)
            (IMPLIES (isMachineState Prg P Env S Mem Stack)
               (EQ (partitions (stepStateExt Prg P Env S Mem Stack StateExt))
                   (partitions StateExt)))))))

   (BG_PUSH (FORALL (P Prg X)
      (IMPLIES (EQ (stmtAt P Prg) (AssgnNew (Var X)))
         (FORALL (Env S Mem Stack StateExt)
            (IMPLIES (isMachineState Prg P Env S Mem Stack)
               (EQ (partitions (stepStateExt Prg P Env S Mem Stack StateExt))
                   (store (partitions StateExt) (alloc Mem) P)))))))

   (BG_PUSH (FORALL (P Prg Arr Len)
      (IMPLIES (EQ (stmtAt P Prg) (AssgnNewArray (Var Arr) (Var Len)))
         (FORALL (Env S Mem Stack StateExt Z)
            (IMPLIES (isMachineState Prg P Env S Mem Stack)
               (AND
                  (IMPLIES (EQ Z (alloc Mem))
                           (EQ (select (partitions
                                          (stepStateExt Prg P Env S
                                                        Mem Stack StateExt))
                                       Z)
                               P))

                  (FORALL (I)
                     (IMPLIES (AND (>= I 0)
                                   (< I (evalExprNS Env S (Var Len)))
                                   (EQ Z (select (allocN (rest Mem)
                                                         (evalExprNS
                                                            Env
                                                            S
                                                            (Var Len)))
                                                 I)))
                              (EQ (select (partitions (stepStateExt Prg P
                                                         Env S Mem Stack
                                                         StateExt))
                                          Z)
                                  P)))
                  (IMPLIES (AND (NEQ Z (alloc Mem))
                                (FORALL (I)
                                   (NOT
                                      (AND (>= I 0)
                                           (< I
                                              (evalExprNS Env S (Var Len)))
                                           (EQ Z (select (allocN
                                                            (rest Mem)
                                                            (evalExprNS
                                                                 Env
                                                                 S
                                                            (Var Len)))
                                                         I))))))

                           (EQ (select (partitions (stepStateExt Prg P Env S
                                                      Mem Stack StateExt))
                                       Z)
                               (select (partitions StateExt) Z)))))))))


   (BG_PUSH (FORALL (P Prg X)
      (IMPLIES (EQ (stmtAt P Prg) (VarDecl (Var X)))
         (FORALL (Env S Mem Stack StateExt)
            (IMPLIES (isMachineState Prg P Env S Mem Stack)
               (EQ (partitions (stepStateExt Prg P Env S Mem Stack StateExt))
                   (store (partitions StateExt) (alloc Mem) P)))))))

   (BG_PUSH (FORALL (P Prg Arr Len)
      (IMPLIES (EQ (stmtAt P Prg) (ArrayDecl (Var Arr) (Var Len)))
         (FORALL (Env S Mem Stack StateExt Z)
            (IMPLIES (isMachineState Prg P Env S Mem Stack)
               (AND
                  (IMPLIES (EQ Z (alloc Mem))
                           (EQ (select (partitions
                                          (stepStateExt Prg P Env S
                                                        Mem Stack StateExt))
                                       Z)
                               P))

                  (FORALL (I)
                     (IMPLIES (AND (>= I 0)
                                   (< I (evalExprNS Env S (Var Len)))
                                   (EQ Z (select (allocN (rest Mem)
                                                         (evalExprNS
                                                            Env
                                                            S
                                                            (Var Len)))
                                                 I)))
                              (EQ (select (partitions (stepStateExt Prg P
                                                        Env S Mem Stack
                                                        StateExt))
                                          Z)
                                  P)))

                  (IMPLIES (AND (NEQ Z (alloc Mem))
                                (FORALL (I)
                                   (NOT
                                      (AND (>= I 0)
                                           (< I
                                              (evalExprNS Env S (Var Len)))
                                           (EQ Z (select (allocN
                                                    (rest Mem)
                                                    (evalExprNS
                                                         Env
                                                         S
                                                    (Var Len)))
                                                 I))))))
                             (EQ (select (partitions (stepStateExt Prg P
                                                        Env S Mem Stack
                                                        StateExt))
                                         Z)
                                 (select (partitions StateExt) Z)))))))))


    (BG_PUSH (FORALL (P Prg X FnName Arg)
       (IMPLIES (EQ (stmtAt P Prg) (Call (Var X) (Fun FnName) (Var Arg)))
          (FORALL (Env S Mem Stack StateExt)
             (IMPLIES (isMachineState Prg P Env S Mem Stack)
                (EQ (partitions (stepStateExt Prg P Env S Mem Stack StateExt))
                    (partitions StateExt)))))))

    (BG_PUSH (FORALL (P Prg)
       (IMPLIES (EQ (stmtAt P Prg) Skip)
          (FORALL (Env S Mem Stack StateExt)
             (IMPLIES (isMachineState Prg P Env S Mem Stack)
                (EQ (partitions (stepStateExt Prg P Env S Mem Stack StateExt))
                    (partitions StateExt)))))))

    (BG_PUSH (FORALL (P Prg E P1 P2)
       (IMPLIES (EQ (stmtAt P Prg) (Branch E P1 P2))
          (FORALL (Env S Mem Stack StateExt)
             (IMPLIES (isMachineState Prg P Env S Mem Stack)
                (EQ (partitions (stepStateExt Prg P Env S Mem Stack StateExt))
                    (partitions StateExt)))))))


    (BG_PUSH (FORALL (P Prg X)
       (IMPLIES (EQ (stmtAt P Prg) (Return (Var X)))
          (FORALL (Env S Mem Stack StateExt)
             (IMPLIES (isMachineState Prg P Env S Mem Stack)
                (EQ (partitions (stepStateExt Prg P Env S Mem Stack StateExt))
                    (partitions StateExt)))))))

    (BG_PUSH (FORALL (Loc)
       (IMPLIES (isLoc Loc)
          (isCFGNode (select (partitions StateExtIn) Loc)))))

 }

------------------------------------------------------------------------
--
-- varNotInAllocSiteSummary(X,N): says that variable X is distinct from
-- all locations in summary N.
```

```
--
------------------------------------------------------------------------

define edge fact varNotInAllocSiteSummary(X:Var, N:CFGNode)
   with meaning varNotInAllocSiteSummary_meaning_prim(eta, etaExtension, X, N)

   -- The meaning is defined using Simplify code.
   prim signature varNotInAllocSiteSummary_meaning_prim(State,
                                                        StateExtension,
                                                        Var,
                                                        CFGNode):Bool {
      (DEFPRED (varNotInAllocSiteSummary_meaning_prim
                         Env Store StateExtension X N))

      (BG_PUSH (FORALL (X N)
         (IMPLIES (AND (isVarName X) (isCFGNode N))
            (FORALL (Env Store StateExtension)
               (IFF (varNotInAllocSiteSummary_meaning_prim
                         Env Store StateExtension (Var X) N)
                    (AND (isExprNotStuck Env Store (Ref (Var X)))
                         (NEQ (select (partitions StateExtension)
                                      (evalExpr Env Store (Ref (Var X))))
                           N)))))))
   }

   if stmt([decl X])@currNode && neq(currNode, N)
   then varNotInAllocSiteSummary(X, N)@cfg_out

   if varNotInAllocSiteSummary(X, N)@cfg_in
   then varNotInAllocSiteSummary(X, N)@cfg_out


------------------------------------------------------------------------
--
-- Pointer analysis with summaries
--
------------------------------------------------------------------------

define edge fact dnpHeapSummary(HS1:HeapSummary, HS2:HeapSummary)
   with meaning dnpHeapSummary_meaning_prim(eta, etaExtension, HS1, HS2)

   -- The meaning is defined using Simplify code.
   prim signature dnpHeapSummary_meaning_prim(State,
                                              StateExtension,
                                              HeapSummary,
                                              HeapSummary):Bool {

      (DEFPRED (dnpHeapSummary_meaning_prim Env Store StateExtension HS1 HS2))

      (BG_PUSH (FORALL (HS1 HS2)
         (IMPLIES (AND (isVarExpr HS1) (isVarExpr HS2))
            (FORALL (Env Store StateExtension)
               (IFF (dnpHeapSummary_meaning_prim Env Store StateExtension
                                                 HS1 HS2)
                    (AND (isExprNotStuck Env Store HS1)
                         (isExprNotStuck Env Store (Ref HS2))
                         (NEQ (evalExpr Env Store HS1)
                              (evalExpr Env Store (Ref HS2))))))))

      (BG_PUSH (FORALL (HS1 HS2)
         (IMPLIES (AND (isCFGNode HS1) (isVarExpr HS2))
            (FORALL (Env Store StateExtension)
               (IFF (dnpHeapSummary_meaning_prim Env Store StateExtension
                                                 HS1 HS2)
                    (AND (isExprNotStuck Env Store (Ref HS2))
                         (FORALL (Loc)
                            (IMPLIES
                               (AND (isLoc Loc)
                                    (inStoreDomain Store Loc)
                                    (EQ (select (partitions StateExtension)
                                                Loc)
                                        HS1))
                               (NEQ (select Store Loc)
                                    (evalExpr Env Store
                                              (Ref HS2))))))))))

      (BG_PUSH (FORALL (HS1 HS2)
         (IMPLIES (AND (isVarExpr HS1) (isCFGNode HS2))
            (FORALL (Env Store StateExtension)
               (IFF (dnpHeapSummary_meaning_prim Env Store StateExtension
                                                 HS1 HS2)
                    (AND (isExprNotStuck Env Store HS1)
                         (IMPLIES (isLoc (evalExpr Env Store HS1))
                                  (NEQ (select (partitions StateExtension)
                                               (evalExpr Env Store HS1))
                                       HS2))))))))

      (BG_PUSH (FORALL (HS1 HS2)
         (IMPLIES (AND (isCFGNode HS1) (isCFGNode HS2))
            (FORALL (Env Store StateExtension)
               (IFF (dnpHeapSummary_meaning_prim Env Store StateExtension
                                                 HS1 HS2)
                    (FORALL (Loc)
                       (IMPLIES
                          (AND (isLoc Loc)
                               (inStoreDomain Store Loc)
                               (EQ (select (partitions StateExtension)
                                           Loc)
                                   HS1)
                               (isLoc (select Store Loc)))
                          (NEQ (select (partitions StateExtension)
                                       (select Store Loc))
                               HS2))))))))
         }

         -- Introduction rules that don't depend on any dnpHeapSummary
         -- incoming facts
         if stmt([X := &A])@currNode && hasBeenDeclared(Y)@cfg_in && neq(A, Y)
         then dnpHeapSummary(X, Y)@cfg_out

         if stmt([X := &A])@currNode && varNotInAllocSiteSummary(A, N)@cfg_in
         then dnpHeapSummary(X, N)@cfg_out

         if stmt([X := BE1 ‘OP BE2])@currNode && hasBeenDeclared(Y)@cfg_in
         then dnpHeapSummary(X, Y)@cfg_out

         if stmt([X := BE1 ‘OP BE2])@currNode
         then dnpHeapSummary(X, N)@cfg_out

         { norun }
         if stmt([X := new])@currNode && neq(currNode, N)
         then dnpHeapSummary(X, N)@cfg_out

         if stmt([X := new])@currNode && hasBeenDeclared(Y)@cfg_in
         then dnpHeapSummary(X, Y)@cfg_out

         -- Preservation rules
         if dnpHeapSummary(X, HS)@cfg_in && !mayDef(X)@currNode
         then dnpHeapSummary(X, HS)@cfg_out

         if dnpHeapSummary(N, HS)@cfg_in && !mayDefAllocSiteSummary(N)@currNode
         then dnpHeapSummary(N, HS)@cfg_out

         -- Update left-hand side info based on right-hand-side info
         if stmt([X := Y])@currNode && dnpHeapSummary(Y, HS)@cfg_in
         then dnpHeapSummary(X, HS)@cfg_out

         if stmt([*X := Z])@currNode && mustPointTo(X, Y)@cfg_in &&
            dnpHeapSummary(Z, HS)@cfg_in
         then dnpHeapSummary(Y, HS)@cfg_out

         if stmt([X := *Y])@currNode &&
            forall HS1:HeapSummary. !dnpHeapSummary(Y, HS1)@cfg_in =>
                                    dnpHeapSummary(HS1, HS2)@cfg_in
         then dnpHeapSummary(X, HS2)@cfg_out


      ------------------------------------------------------------------------
      --
      -- Array functionality tests
      --
      ------------------------------------------------------------------------

      --define edge fact arrayAccess(A:Var, I:Var, X:Var)
      --    with meaning eq([*(A[I])],X)
```

```
--
--    -- This shows that after A[I] is assigned to X, the two are equal
--    if stmt([X := *(A[I])])@currNode && neq(X,A) && neq(X,I)
--    then arrayAccess(A,I,X)@cfg_out
--
--
--define edge fact arrayUpdate(A:Var, I:Var, X:Var)
--    with meaning eq([*(A[I])],X)
--
--    -- This shows that after X is assigned to A[I], the two are equal
--    if stmt([*(A[I]) := X])@currNode
--    then arrayUpdate(A,I,X)@cfg_out
--
--define edge fact accessEqual(A:Var, I:Var, X:Var)
--    with meaning eq([*(A[I])],[*(A[X])])
--
--    -- This shows that if A[I] has been assigned a value and X is equal to I,
--    --   then A[I] = A[X]
--    if stmt([*(A[I]) := Y])@currNode && varEqual(I,X)@cfg_in
--    then accessEqual(A,I,X)@cfg_out



-------------------------------------------------------------------------
--
-- arraysUnaliased - gaurantees that two arrays are not aliased
--
-------------------------------------------------------------------------
define edge fact arraysUnaliased(A:Var, B:Var)
   with meaning neq(A,B)

   -- The first case
   if stmt([A := new [I]])@currNode && hasBeenDeclared(B)@cfg_in && neq(A,B)
   then arraysUnaliased(A,B)@cfg_out

   -- The other first case
   if stmt([A := new [I]])@currNode && hasBeenDeclared(B)@cfg_in && neq(A,B)
   then arraysUnaliased(B,A)@cfg_out

   -- Propagating the fact
   if arraysUnaliased(A,B)@cfg_in && !mayDef(A)@currNode &&
       !mayDef(B)@currNode && neq(A,B)
   then arraysUnaliased(A,B)@cfg_out


-------------------------------------------------------------------------
--
-- mayDefArray - present when a statement may change the contents of
-- array Z
--
-------------------------------------------------------------------------
```

```
define node fact mayDefArray(Z:Var) =
      case currStmt of
         [skip]                  => mayDef(Z)@currNode
         [decl X]                => mayDef(Z)@currNode
         [decl X[I]]             => mayDef(Z)@currNode
         [if X goto L1 else L2]  => mayDef(Z)@currNode
         [if C1 goto L1 else L2] => mayDef(Z)@currNode
         [X := C1]               => mayDef(Z)@currNode
         [X := Y]                => mayDef(Z)@currNode
         [*X := Y]               => mayDef(Z)@currNode
                                    || !arraysUnaliased(Z,X)@cfg_in
         [X := *Y]               => mayDef(Z)@currNode
                                    || !arraysUnaliased(Z,Y)@cfg_in
         [X := &Y]               => mayDef(Z)@currNode
         [X := new]              => mayDef(Z)@currNode
         [X := A ‘OP B]          => mayDef(Z)@currNode
         [X := A ‘OP C1]         => mayDef(Z)@currNode
         [X := C1 ‘OP B]         => mayDef(Z)@currNode
         [X := C1 ‘OP C2]        => mayDef(Z)@currNode
         [X := P(Y)]             => mayDef(Z)@currNode
                                    || mayUse(Z)@currNode
         [X := new [I]]          => mayDef(Z)@currNode
         [X := A[I]]             => mayDef(Z)@currNode
         [X := *(A[I])]          => mayDef(Z)@currNode
         [*(A[I]) := X]          => !arraysUnaliased(A,Z)@cfg_in
         [*(A[I]) := C1]         => !arraysUnaliased(A,Z)@cfg_in
         [X := (&A)[I]]          => mayDef(Z)@currNode
         [X := *((&A)[I])]       => mayDef(Z)@currNode
         [*((&A)[I]) := X]       => false
         [*((&A)[I]) := C1]      => false
         [return X]              => mayDef(Z)@currNode
     endcase


------------------------------------------------------------------------
--
-- mayDefArrayElem
--
------------------------------------------------------------------------

define node fact mayDefArrayElem(Z:Var, J:Var) =
      case currStmt of
         [skip]                  => false
         [decl X]                => false
         [decl X[I]]             => false
         [if X goto L1 else L2]  => false
         [if C1 goto L1 else L2] => false
         [X := C1]               => false
         [X := Y]                => false
         [*X := Y]               => true
         [X := *Y]               => false
         [X := &Y]               => false
```

```
        [X := new]              => false
        [X := A ‘OP B]          => false
        [X := A ‘OP C1]         => false
        [X := C1 ‘OP B]         => false
        [X := C1 ‘OP C2]        => false
        [X := P(Y)]             => true
        [X := new [I]]          => false
        [X := A[I]]             => false
        [X := *(A[I])]          => false
        [*(A[I]) := X]          => true
        [*(A[I]) := C1]         => true
        [X := (&A)[I]]          => false
        [X := *((&A)[I])]       => false
        [*((&A)[I]) := X]       => true
        [*((&A)[I]) := C1]      => true
        [return X]              => true
    endcase


------------------------------------------------------------------------
--
-- equalsPlus - shows that X is equal to Y + Z
--
------------------------------------------------------------------------

define edge fact equalsPlus(X:NonCallExpr, Y:NonCallExpr, Z:NonCallExpr)
    with meaning eq(evalExpr(eta, X),
                    applyBinaryOp(+, evalExpr(eta, Y), evalExpr(eta, Z)))

   if stmt([X := I + C])@currNode && neq(X, I)
   then equalsPlus(X, I, C)@cfg_out

   if equalsPlus(E1, E2, C)@cfg_in &&
      unchanged(E1)@currNode &&
      unchanged(E2)@currNode
   then equalsPlus(E1, E2, C)@cfg_out


------------------------------------------------------------------------
--
-- varEqualArray - shows that X equals A[I]
--
------------------------------------------------------------------------

define edge fact varEqualArray(X:Var, A:Var, I:Var)
   with meaning eq(X, [*(A[I])])

   -- The initial creation of the fact
   if stmt([X := *(A[I])])@currNode && neq(X,A) && neq(X,I)
   then varEqualArray(X,A,I)@cfg_out

   -- The other initial creation of the fact
   if stmt([*(A[I]) := X])@currNode && neq(X,A) && neq(X,I)
   then varEqualArray(X,A,I)@cfg_out

   -- Propagating the fact
   if varEqualArray(X,A,I)@cfg_in &&
      !mayDefArray(A)@currNode &&
      !mayDef(I)@currNode &&
      !mayDef(X)@currNode && !mayDefArrayElem(A,I)@currNode
   then varEqualArray(X,A,I)@cfg_out


   -- The vital case for our transformation
   if stmt([Y := I + BE])@currNode && varEqualArray(X,A,J)@cfg_in
      && equalsPlus(J,I,BE)@cfg_in && !mayDef(X)@currNode
      && !mayDefArray(A)@currNode && unchanged(BE)@currNode
   then varEqualArray(X,A,Y)@cfg_out

-- This case is now covered by the previous case
--   if stmt([I := I + C])@currNode && varEqualArray(X,A,J)@cfg_in &&
--      equalsPlus(J,I,C)@cfg_in && neq(X,I)
--   then varEqualArray(X,A,I)@cfg_out

   -- This is the important transformation
   if stmt([X := *(A[I])])@currNode && varEqualArray(Y,A,J)@cfg_in &&
      (varEqual(I,J)@cfg_in || eq(I,J))
   then transform [X := Y]

   if stmt([X := *(A[I])])@currNode && varEqualArray(X,A,J)@cfg_in &&
      (varEqual(I,J)@cfg_in || eq(I,J))
   then transform [skip]


------------------------------------------------------------------------
--
-- isInt
--
------------------------------------------------------------------------
define edge fact isInt(X:Var)
    with meaning exists C:Const . eq(evalExpr(eta,X),evalExpr(eta,C))

   if stmt([X := BE1 ‘OP BE2])@currNode
      && baseExprHasConstValue(BE1, C)@currNode
      && baseExprHasConstValue(BE2, C)@currNode
   then isInt(X)@cfg_out

   if isInt(X)@cfg_in && !mayDef(X)@currNode
   then isInt(X)@cfg_out


define node fact baseExprIsInt(BE:BaseExpr) =
        case BE of
            [X] => isInt(X)@cfg_in
            [C] => true
        endcase


------------------------------------------------------------------------
--
-- isTrue
--
------------------------------------------------------------------------

define edge fact isTrue(B:Var)
   with meaning eq(evalExpr(eta,[B]), evalExpr(eta, [true]))

   if stmt([B := X <= BE])@currNode && neq(X,B) && unchanged(BE)@currNode
      && leq(X,BE)@cfg_in
   then isTrue(B)@cfg_out

   if stmt([B := X >= BE])@currNode && neq(X,B) && unchanged(BE)@currNode
      && geq(X,BE)@cfg_in
   then isTrue(B)@cfg_out

    if stmt([B := X < BE])@currNode && neq(X,B) && unchanged(BE)@currNode
      && lt(X,BE)@cfg_in
   then isTrue(B)@cfg_out

   if stmt([B := X > BE])@currNode && neq(X,B) && unchanged(BE)@currNode
      && gt(X,BE)@cfg_in
   then isTrue(B)@cfg_out

   if isTrue(B)@cfg_in && !mayDef(B)@currNode
   then isTrue(B)@cfg_out


------------------------------------------------------------------------
--
-- isFalse
--
------------------------------------------------------------------------

define edge fact isFalse(B:Var)
   with meaning eq(evalExpr(eta,[B]), evalExpr(eta, [false]))

   if stmt([B := X < BE])@currNode && neq(X,B) && unchanged(BE)@currNode
      && geq(X,BE)@cfg_in
   then isFalse(B)@cfg_out

   if stmt([B := X > BE])@currNode && neq(X,B) && unchanged(BE)@currNode
      && leq(X,BE)@cfg_in
   then isFalse(B)@cfg_out

   if stmt([B := X <= BE])@currNode && neq(X,B) && unchanged(BE)@currNode
      && gt(X,BE)@cfg_in
   then isFalse(B)@cfg_out

   if stmt([B := X >= BE])@currNode && neq(X,B) && unchanged(BE)@currNode
      && lt(X,BE)@cfg_in
   then isFalse(B)@cfg_out

   if isFalse(B)@cfg_in && !mayDef(B)@currNode
   then isFalse(B)@cfg_out


------------------------------------------------------------------------
--
-- lt & gt
--
------------------------------------------------------------------------

define edge fact lt(E1:NonCallExpr, E2:NonCallExpr)
   with meaning eq([E1 < E2], [true])

define edge fact gt(E1:NonCallExpr, E2:NonCallExpr)
   with meaning eq([E1 > E2], [true])

   if hasConstValue(X,C1)@cfg_in && hasConstValue(Y,C2)@cfg_in
      && !mayDef(X)@currNode && !mayDef(Y)@currNode
      && eq(applyBinaryOp(<, getConst(C1), getConst(C2)),
            getConst([true]))
   then lt(X,C2)@cfg_out && gt(Y, C1)@cfg_out &&
        lt(X,Y)@cfg_out && gt(Y,X)@cfg_out &&
        lt(C1,Y)@cfg_out && gt(C2,X)@cfg_out

   if stmt([X := Y])@currNode && neq(X,Y) && lt(Y,BE)@cfg_in
      && unchanged(BE)@currNode
   then lt(X,BE)@cfg_out && gt(BE,X)@cfg_out

   if lt(X,BE)@cfg_in && !mayDef(X)@currNode && unchanged(BE)@currNode
   then lt(X,BE)@cfg_out && gt(BE,X)@cfg_out

   if stmt([X := Y])@currNode && neq(X,Y) && gt(Y,BE)@cfg_in
      && unchanged(BE)@currNode
   then gt(X,BE)@cfg_out && lt(BE,X)@cfg_out

   if gt(X,BE)@cfg_in && !mayDef(X)@currNode && unchanged(BE)@currNode
   then gt(X,BE)@cfg_out && lt(BE,X)@cfg_out

------------------------------------------------------------------------
--
-- Currently, the checker can only handle single input single output
-- nodes. For multiple inputs or multiples outputs, as a temporary
-- solution, we expand the cases by hand. See the leq and geq facts
-- below for some examples. Here is the general strategy for
-- generating the cases by hand.
--
```

```
-- There are only two nodes that have multiple input or output edges:
-- merge nodes and if nodes.
--
-- * Merge nodes:
--
--     To check:
--
--        if stmt([merge])@currNode && F(...)@cfg_in[0] && G(...)@cfg_in[1]
--        then H(...)@cfg_out
--
--     We by hand write:
--
--        if stmt([skip])@currNode && F(...)@cfg_in
--        then H(...)@cfg_out
--
--        if stmt([skip])@currNode && H(...)@cfg_in
--        then G(...)@cfg_out
--
-- * If nodes:
--
--     To check:
--
--        if stmt([if B goto L1 else L2])@currNode && psi
--        then F(...)@cfg_out[0]
--
--     We by hand write:
--
--        if stmt([if B goto L1 else L2])@currNode && psi && isFalse(B)
--        then F(...)@cfg_out
--
--     To check:
--
--        if stmt([if B goto L1 else L2])@currNode && psi
--        then F(...)@cfg_out[1]
--
--     We by hand write:
--
--        if stmt([if B goto L1 else L2])@currNode && psi && isTrue(B)
--        then F(...)@cfg_out
--
-------------------------------------------------------------------------

-------------------------------------------------------------------------
--
-- leq & geq. These two facts implement the inRange analysis. We split
-- the inRange fact into leq & geq for more flexibility. The following
-- analysis also handles symbolic ranges.
--
-------------------------------------------------------------------------

decl lo:Const, hi:Const, C1:Const, C2:Const, C3:Const, C4:Const, C:Const in
```

```
define edge fact leq(E1:NonCallExpr, E2:NonCallExpr)
   with meaning eq([E1 <= E2], [true])

define edge fact geq(E1:NonCallExpr, E2:NonCallExpr)
   with meaning eq([E1 >= E2], [true])

   if stmt([X := BE])@currNode
      && baseExprIsInt(BE)@currNode
   then leq(X, BE)@cfg_out && geq(X, BE)@cfg_out

   if leq(X,BE)@cfg_in && !mayDef(X)@currNode && unchanged(BE)@currNode
   then leq(X,BE)@cfg_out

   if geq(X,BE)@cfg_in && !mayDef(X)@currNode && unchanged(BE)@currNode
   then geq(X,BE)@cfg_out

   -- The following two cases correspong to the merge:
   if stmt([skip])@currNode
      && geq(X,C1)@cfg_in && leq(X,C2)@cfg_in
   then geq(X, newConst(min(getConst(C1), getConst(C3))))@cfg_out &&
        leq(X, newConst(max(getConst(C2), getConst(C4))))@cfg_out

   if stmt([skip])@currNode
      && geq(X,C3)@cfg_in && leq(X,C4)@cfg_in
   then geq(X, newConst(min(getConst(C1), getConst(C3))))@cfg_out &&
        leq(X, newConst(max(getConst(C2), getConst(C4))))@cfg_out

   if stmt([X := BE1 + BE2])@currNode
      && geq(BE1,C1)@cfg_in && leq(BE1,C2)@cfg_in
      && geq(BE2,C3)@cfg_in && leq(BE2,C4)@cfg_in
   then geq(X, newConst(applyBinaryOp(+, getConst(C1), getConst(C3))))@cfg_out
     && leq(X, newConst(applyBinaryOp(+, getConst(C2), getConst(C4))))@cfg_out


   -- The following cases are used to extract information from branches.
   if stmt([if B goto L1 else L2])@currNode && equalsLess(B,X,C)@cfg_in
      && isTrue(B)@cfg_in
      && leq(X,C2)@cfg_in
   then leq(X, newConst(min(applyBinaryOp(-,getConst(C),getConst([1])),
                            getConst(C2))))@cfg_out

   if stmt([if B goto L1 else L2])@currNode && equalsLess(B,X,C)@cfg_in
      && isFalse(B)@cfg_in
      && geq(X,C1)@cfg_in
   then geq(X, newConst(max(getConst(C), getConst(C1))))@cfg_out

   if stmt([if B goto L1 else L2])@currNode && equalsLessEq(B,X,C)@cfg_in
      && isTrue(B)@cfg_in
      && leq(X,C2)@cfg_in
   then leq(X, newConst(min(getConst(C), getConst(C2))))@cfg_out

   if stmt([if B goto L1 else L2])@currNode && equalsLessEq(B,X,C)@cfg_in
```

```
      && isFalse(B)@cfg_in
      && geq(X,C1)@cfg_in
   then geq(X, newConst(max(applyBinaryOp(+,getConst(C),getConst([1])),
                            getConst(C1))))@cfg_out

   if stmt([if B goto L1 else L2])@currNode && equalsGreater(B,X,C)@cfg_in
      && isTrue(B)@cfg_in
      && geq(X,C1)@cfg_in
   then geq(X, newConst(max(applyBinaryOp(+,getConst(C),getConst([1])),
                            getConst(C1))))@cfg_out

   if stmt([if B goto L1 else L2])@currNode && equalsGreater(B,X,C)@cfg_in
      && isFalse(B)@cfg_in
      && leq(X,C2)@cfg_in
   then leq(X, newConst(min(getConst(C), getConst(C2))))@cfg_out

   if stmt([if B goto L1 else L2])@currNode && equalsGreaterEq(B,X,C)@cfg_in
      && isTrue(B)@cfg_in
      && geq(X,C1)@cfg_in
   then geq(X, newConst(max(getConst(C), getConst(C1))))@cfg_out

   if stmt([if B goto L1 else L2])@currNode && equalsGreaterEq(B,X,C)@cfg_in
      && isFalse(B)@cfg_in
      && leq(X,C2)@cfg_in
   then leq(X, newConst(min(applyBinaryOp(-,getConst(C),getConst([1])),
                            getConst(C2))))@cfg_out

end

-------------------------------------------------------------------------
--
-- equalsLess analysis
--
-------------------------------------------------------------------------

define edge fact equalsLess(B:Var, E1:NonCallExpr, E2:NonCallExpr)
    with meaning eq(B, [E1 < E2])

   if stmt([B := BE1 < BE2])@currNode && unchanged(BE1)@currNode &&
      unchanged(BE2)@currNode
   then equalsLess(B,BE1,BE2)@cfg_out

   if equalsLess(B,BE1,BE2)@cfg_in
      && !mayDef(B)@currNode && unchanged(BE1)@currNode
      && unchanged(BE2)@currNode
   then equalsLess(B,BE1,BE2)@cfg_out


-------------------------------------------------------------------------
--
-- equalsLessEq analysis
--
```

```
-------------------------------------------------------------------------

define edge fact equalsLessEq(B:Var, E1:NonCallExpr, E2:NonCallExpr)
    with meaning eq(B, [E1 <= E2])

   if stmt([B := BE1 <= BE2])@currNode && unchanged(BE1)@currNode &&
      unchanged(BE2)@currNode
   then equalsLessEq(B,BE1,BE2)@cfg_out

   if equalsLessEq(B,BE1,BE2)@cfg_in
      && !mayDef(B)@currNode && unchanged(BE1)@currNode
      && unchanged(BE2)@currNode
   then equalsLessEq(B,BE1,BE2)@cfg_out


-------------------------------------------------------------------------
--
-- equalsGreater analysis
--
-------------------------------------------------------------------------

define edge fact equalsGreater(B:Var, E1:NonCallExpr, E2:NonCallExpr)
    with meaning eq(B, [E1 > E2])

   if stmt([B := BE1 > BE2])@currNode && unchanged(BE1)@currNode &&
      unchanged(BE2)@currNode
   then equalsGreater(B,BE1,BE2)@cfg_out

   if equalsGreater(B,BE1,BE2)@cfg_in
      && !mayDef(B)@currNode && unchanged(BE1)@currNode
      && unchanged(BE2)@currNode
   then equalsGreater(B,BE1,BE2)@cfg_out


-------------------------------------------------------------------------
--
-- equalsGreaterEq analysis
--
-------------------------------------------------------------------------

define edge fact equalsGreaterEq(B:Var, E1:NonCallExpr, E2:NonCallExpr)
    with meaning eq(B, [E1 >= E2])

   if stmt([B := BE1 >= BE2])@currNode && unchanged(BE1)@currNode &&
      unchanged(BE2)@currNode
   then equalsGreaterEq(B,BE1,BE2)@cfg_out

   if equalsGreaterEq(B,BE1,BE2)@cfg_in
      && !mayDef(B)@currNode && unchanged(BE1)@currNode
      && unchanged(BE2)@currNode
   then equalsGreaterEq(B,BE1,BE2)@cfg_out
```

```
end
```