
ROPAS

Research On Program Analysis System
National Creative Research Initiative Center 1998-2003
Programming Research Laboratory, School of Computer Science & Engineering
Seoul National University

ROPAS MEMO
2006-32
July 26, 2006

An Abstract Interpretation with the Interval Domain for C-like Programs

Jaeho Shin
netj@ropas.snu.ac.kr

School of Computer Science and Engineering
Seoul National University

Master's Thesis
Advisor: Kwangkeun Yi

Abstract

We defined an intermediate representation called G for representing programs written in imperative languages like C. We designed an abstract interpretation of G using the interval domain and proved its soundness. As a real world application, we implemented an abstract interpreter that analyzes array index ranges, and finds buffer overrun errors in real C programs. Implementation techniques used in our analyzer and its performance are presented.

1 Overview

1.1 Introduction

As the software industry matures, demands for static program analysis and verification tools are increasing. The industry understands software testing approach is neither efficient nor sufficient for quality assurance. Test runs can find some bugs at their best, but can never prove the software is free from bugs. Although ad-hoc static bug finding tools are becoming popular, they have the same problem as software testing, since they may still miss true errors. As software industry becomes more serious, people will eventually want to use conservative yet effective software verification tools.

In order to build good static analysis tools for C-like programs, we designed an abstract interpretation with the interval domain. To focus on the abstract interpretation, we designed a simple intermediate form called G. We believe programs written in languages similar to the C Programming Language[14] can be transformed into G straightforwardly. We defined the abstract interpretation of G, and proved its soundness on top of the Abstract Interpretation Framework[7]. It is a theoretical framework for static program analyzer we built, and will be a starting point for static program verifiers we plan to build.

This chapter gives a brief overview of our problem and approach, then discusses related work and directions. Chapter 2 describes the syntax and semantics of G form. Chapter 3 defines the abstract semantics of G using the integer interval domain[6], and Chapter 4 presents the proof of its soundness. Chapter 5 suggests some techniques of implementation that will improve the performance of the abstract interpreter. Chapter 6 shows a realistic application of our abstract interpretation on the buffer overrun problem.

1.2 Discussion

Related work Value Set Analysis[2] is an approach very similar to ours. The major difference is that they are targeted at analyzing executables to reject malicious plugins and

mobile codes or analyze worms and viruses, while we are analyzing source programs to help the programmers fix bugs in them. Analyzing executables clearly has the advantage that bugs introduced by compilers can also be detected. However, without source code information, it is very hard to provide useful debug information to program developers, and the analysis can only decide the correctness of the program.

Report of ASTRÉE[3, 4] mentions the idea of localizing operators in Section 5.3. No detail is provided but only a brief, incomplete explanation is there why localizing operators on their abstract environment is possible. We provide detailed arguments on what conditions make it possible to localize operators on abstract memory and how one can do it almost independently of other parts of the analyzer.

Huge body of work on buffer overrun errors are mentioned in Chapter 6.

Advantages of abstract interpretation We believe our strong point is the fact that abstract semantics resembles concrete semantics. Because of that, programmers can intuitively understand its results, and analysis designers can improve its accuracy in a natural way. Results of set-based analysis[12] or other constraint based analyses, especially for C-like programs, are hard to understand for program developers. Interpreting or summarizing those results for the user requires lots of effort. The abstraction is not explicitly handled so hard to control systematically. Moreover, there is no systematic guide for designing and establishing soundness of them as well as analyses based on type or effect systems. It is a huge burden for the designer that he must find the right way out of the full darkness all by himself. Various program analysis techniques based on data flow analyses framework are usually composed of several separate analyses, e.g. control flow, pointer and interval analyses. The separation of analyses makes improving the accuracy hard when the information analyzed at later steps are required by the early steps. Our approach does not suffer from such problem since it computes every information as a single fixpoint.

Future work The syntax of G form defined in this report is not expressive enough to represent all C programs. Neither explicit type castings nor implicit ones of C can be expressed in G. We will handle it appropriately by introducing memory and address operators with type size information into G.

The semantics of G needs to explicitly handle error conditions. Currently, we do not define any semantics for erroneous executions. Therefore, the abstract semantics after such points is $\bot$. However, since $\bot \sqsubseteq m$ for all m, it becomes ambiguous whether an abstract memory m also means erroneous state or not. Since we are interested in errors in programs, distinguishing these two are very important. A similar problem to this is in treatment of free variables. We define no semantics for use of free variables. Practically, there are always free variables in real programs, i.e. there are parts whose source code is not available. To make the analysis more robust, we must give explicit concrete semantics for free variables and corresponding sound abstract semantics.

Several improvements to the abstract semantics of G will significantly increase the accuracy. First, our abstract memory update must be improved to a sound, strong update. Although our analysis is flow sensitive, i.e. computes different abstract memory for each point, because of weak update, possible values of previous points get accumulated to later ones. We need strong update to fully benefit from our flow sensitivity. Second, we need to make it possible to distinguish values of array elements. Currently, we use a single abstract memory cell for array elements. This limits us from accurately analyzing programs with arrays having fixed contents. A systematic method for partitioning the indexes of elements must be carried out. Third, we need to isolate local variables of procedures. Since join or widening, narrowing on memory are unaware of reachability, such operation in one procedure will also be applied to cells of local variables of other procedures which are definitely unreachable from that point. Computing the reachability for every join, widening must be an expensive choice. Therefore, we can get some help from syntax, and simply isolate only the local variables to the owning procedure.

We are designing a systematic method for increasing the analysis accuracy to reduce the number of false alarms. Using the trace partitioning technique[17], we can control various

path and context sensitivities of the analysis. We will be adopting it into our analysis by extending the abstractions α_{ctrl}, α_{data} and domain of abstract addresses $A\hat{d}dr$.

To become a true verification tool, the analyzer must provide a way to eliminate false alarms. Although false alarms are inherently inevitable for static analysis, most of the verification process is a job too routined for humans to perform. Using the systematic way of increasing the analysis accuracy, we will build a persistent verification framework. We believe it will automate most of the analysis and verification process.

We are working on other applications, e.g. detection of memory leaks and uses of dangling pointers. They are also a major problem for C-like programs as well as buffer overruns. By slightly extending our instrumentation on the G form, we can check the result of the abstract interpretation to detect those errors.

2 G Programs

2.1 Representation

A G program P is a set of G procedures.

$$P \in Pgm = \wp(Proc)$$

A G procedure has its name, formal parameters[1], and a *CFG* which represents its body.

$$Proc = ProcId \times Var \times CFG$$

A control flow graph[1, §9.4] $\in CFG$

- is a directed graph whose nodes are basic blocks $\in Block$.
- has a unique $\texttt{ENTRY}_\mathsf{p}$ node where every flow begins.
- has a unique $\texttt{EXIT}_\mathsf{p}$ node where all flows began from $\texttt{ENTRY}_\mathsf{p}$ end.
- has two types of edges: flow edges and resume edges. Resume edges remember the node to resume after a procedure call from a call node.

$$\begin{aligned} CFG &= Flows \times Resumes \\ Flows &= Edges \\ Resumes &= Edges \\ Edges &= \wp(Block \times Block) \end{aligned}$$

There are three types of identifiers: local, global and field names. Name of procedure is also a global name.

local name	x	$\in$	Var
global name	X	$\in$	$GVar$
procedure name	p	$\in$	$ProcId \subseteq GVar$
field name	f	$\in$	$FieldId$
allocation site	t	$\in$	$AllocSite$

Basic block $b \in Block$ constituting a control flow graph is defined by the following syntax:

block	b	$\in$	$Block$
	b	::=	$\texttt{CALL}(e,e,e) \mid \texttt{ENTRY}_\mathsf{p} \mid \texttt{EXIT}_\mathsf{p} \mid c^*$
command	c	$\in$	Cmd
	c	::=	$\texttt{SET}(e,e) \mid \texttt{ESCAPE}(e) \mid \texttt{ASSUME}(r)$
		$\mid$	$\texttt{ALLOC}(e,s)_\mathsf{t} \mid \texttt{FREE}(e)$
expression	e	$\in$	$Expr$
	e	::=	$n \mid \mathsf{x} \mid \mathsf{X} \mid e \star e \mid e.\mathsf{f} \mid \$e$
	$\star$	::=	$+ \mid - \mid * \mid /$
relation	r	$\in$	Rel
	r	::=	$e \diamond e$
	$\diamond$	::=	$= \mid \texttt{!=} \mid < \mid \texttt{<=} \mid > \mid \texttt{>=}$
shape	s	$\in$	$Shape$
	s	::=	$[e] \mid \{\mathsf{f}^*\}$

[1] For simplicity, only one parameter is considered here.

For simplicity, let us assume every basic block $\in Block_P$ of a program P is always distinguishable. Under this assumption, keeping separate sets of edges for each procedure is not very useful. So from here on, let us only consider the two sets of edges of the program, i.e. $Flows_P$ and $Resumes_P$, and define procedures as $Proc = ProcId \times Var$. Nevertheless, let us use b^{p} to denote that b belongs to procedure p.

2.2 Semantics

2.2.1 Domain

Here are the domains used in our semantics definition:

$$\begin{array}{rcrcl}
\tau & \in & Trace & = & State^+ \\
\sigma & \in & State & = & Block \times Memory \times Dump \\
 & & & & \\
m & \in & Memory & = & Map \times Counter \times Alloc \\
M & \in & Map & = & Addr \xrightarrow{\text{fin}} Val \\
v & \in & Val & = & \mathbb{Z} + Addr \\
a & \in & Addr & = & GVar + Dump \times Var + \\
 & & & & Region \times Index + Region \times FieldId \\
r & \in & Region & = & Counter \times AllocSite \\
c & \in & Counter & = & \mathbb{N} \\
I & \in & Alloc & = & Region \xrightarrow{\text{fin}} (Size + FieldId^*) \\
 & & Size & = & \mathbb{N} \\
 & & Index & = & \mathbb{Z} \\
 & & & & \\
d & \in & Dump & = & (ProcId \times Block \times Addr)^*
\end{array}$$

2.2.2 Traces

Let us use $P \in Pgm$ to refer to the G program we are considering throughout this section. Its collecting semantics $[\![P]\!]\ \Sigma_0$ is the prefix closure of traces whose states are all connected by the transition $\longrightarrow$ and begins from an initial state in Σ_0.

$$[\![P]\!] \quad : \quad \wp(State) \to \wp(Trace)$$

$$[\![P]\!]\ \Sigma_0 = \{(\sigma_1, \cdots, \sigma_n) : Trace \mid \sigma_1 \in \Sigma_0, \sigma_i \longrightarrow \sigma_{i+1}\}$$

$[\![P]\!]$ can also be expressed as the least fixpoint of a semantic transfer function $\mathcal{F}$ greater than or equal to Σ_0.

$$[\![P]\!]\ \Sigma_0 = \mathrm{lfp}_{\Sigma_0}(\mathcal{F}\ P)$$

where

$$\mathcal{F} \quad : \quad Pgm \to \wp(Trace) \to \wp(Trace)$$

$$\mathcal{F}\ P\ T = T \cup \{(\sigma_1, \cdots, \sigma_n, \sigma_{n+1}) \mid (\sigma_1, \cdots, \sigma_n) \in T, \sigma_n \longrightarrow \sigma_{n+1}\}$$

Usually, if main is the entry procedure, and Map'_0 is the set of value mappings for free variables, then the set of initial states Σ_0 is determined as the following:

$$\begin{array}{rcl}
\Sigma_0 & = & \{(\mathtt{ENTRY}_{\mathsf{main}}, (M_0, 0, \emptyset), \epsilon) : State \mid \\
 & & M_0 = M\{\mathsf{p} \mapsto \mathsf{p} \mid (\mathsf{p}, \mathsf{x}) \in P\}, M \in Map'_0\}
\end{array}$$

2.2.3 Transition

The transition $\longrightarrow$ is a binary relation on $State$. $\mathcal{R}$ and $\mathcal{V}$, $\mathcal{A}$ defines the semantics of commands and expressions respectively.

$$(b, m, d) \quad \longrightarrow \quad (b', m', d')$$

if and only if

- when $b = \texttt{CALL}(e, e_0, e_1)$,

$$\begin{array}{rcllrcl} \mathsf{p}' & : & \mathit{ProcId} & & b' & = & \texttt{ENTRY}_{\mathsf{p}'} \\ \mathsf{p}' & = & \mathcal{A}\ d\ m\ e_0 & & d' & = & (\mathsf{p}', b_r, \mathcal{A}\ d\ m\ e) :: d \\ (\mathsf{p}', \mathsf{x}) & \in & P & & m' & = & (M\{(d', \mathsf{x}) \mapsto \mathcal{V}\ d\ m\ e_1\}, c, I) \\ (b, b_r) & \in & \mathit{Resumes}_P & & & & \text{where } (M, c, I) = m \end{array}$$

- when $b = \texttt{ENTRY}_{\mathsf{p}}$,

$$\begin{array}{rcl} (\texttt{ENTRY}_{\mathsf{p}}, b') & \in & \mathit{Flows}_P \\ (m', d') & = & (m, d) \end{array}$$

- when $b = \texttt{EXIT}_{\mathsf{p}'}$,

$$\begin{array}{rcl} (\mathsf{p}', b', \_) :: d' & = & d \\ m' & = & m \end{array}$$

- when $b = (c_1, \cdots, c_n)$,

$$\begin{array}{rcl} (b, b') & \in & \mathit{Flows}_P \\ m' & = & (\mathcal{R}\ d\ c_1; \cdots; \mathcal{R}\ d\ c_n)\ m \\ d' & = & d \end{array}$$

2.2.4 Reaction

Each command changes the memory part of the state. Some commands only change the value mapping, and some also modifies the allocation information.

$$\mathcal{R} \quad : \quad \mathit{Dump} \to \mathit{Cmd} \to \mathit{Memory} \to \mathit{Memory}$$

$$\begin{array}{rcl} \mathcal{R}\ d\ \texttt{SET}(e_a, e)\ (M, c, I) & = & (M\{\mathcal{A}\ d\ M\ e_a \mapsto \mathcal{V}\ d\ M\ e\}, c, I) \\ \mathcal{R}\ d\ \texttt{ESCAPE}(e)\ (M, c, I) & = & (M\{a \mapsto \mathcal{V}\ d\ M\ e\}, c, I) \\ & & \quad \text{where } (\_, \_, a) :: \_ = d \\ \mathcal{R}\ d\ \texttt{ASSUME}(r)\ (M, c, I) & = & (M, c, I) \text{ if } d, (M, c, I) \models r \\ \mathcal{R}\ d\ \texttt{ALLOC}(e_a, [e])_{\mathsf{t}}\ (M, c, I) & = & (M\{\mathcal{A}\ d\ M\ e_a \mapsto (r, 0)\}, \\ & & \quad c + 1, I\{r \mapsto \mathcal{V}\ d\ M\ e\}) \\ & & \quad \text{where } r = (c, \mathsf{t}) \\ \mathcal{R}\ d\ \texttt{ALLOC}(e_a, \{\vec{\mathsf{f}}\})_{\mathsf{t}}\ (M, c, I) & = & (M\{\mathcal{A}\ d\ M\ e_a \mapsto (r, \mathsf{f}_1)\}, \\ & & \quad c + 1, I\{r \mapsto \vec{\mathsf{f}}\}) \\ & & \quad \text{where } r = (c, \mathsf{t}) \\ \mathcal{R}\ d\ \texttt{FREE}(e)\ (M, c, I) & = & (M - \{a\}, c, I - \{a\}) \\ & & \quad \text{where } a = \mathcal{A}\ d\ M\ e \end{array}$$

2.2.5 Value

Expression defines a value *Val* under its context, dump *Dump* and value mapping *Map*.

$$\mathcal{V} \quad : \quad \mathit{Dump} \to \mathit{Map} \to \mathit{Expr} \to \mathit{Val}$$

$$\begin{array}{rcl} \mathcal{V}\ d\ M\ n & = & n \\ \mathcal{V}\ d\ M\ \mathsf{X} & = & \mathsf{X} \\ \mathcal{V}\ d\ M\ \mathsf{x} & = & (d, \mathsf{x}) \\ \mathcal{V}\ d\ M\ e_1 \star e_2 & = & \mathcal{V}\ d\ M\ e_1 \star \mathcal{V}\ d\ M\ e_2 \\ \mathcal{V}\ d\ M\ e.\mathsf{f} & = & (r, \mathsf{f}) \text{ where } (r, \_) = \mathcal{A}\ d\ M\ e \\ \mathcal{V}\ d\ M\ \$e & = & M(\mathcal{A}\ d\ M\ e) \end{array}$$

For convenience, let us overload $\mathcal{V}$ and also use it on *Memory*.

$$\mathcal{V} \quad : \quad \mathit{Dump} \to \mathit{Memory} \to \mathit{Expr} \to \mathit{Val}$$

$$\mathcal{V}\ d\ (M, \_, \_)\ e \quad = \quad \mathcal{V}\ d\ M\ e$$

2.2.6 Address

There are restrictions to some values so that they are not allowed to be used as addresses.

$$\begin{array}{rcl} \mathcal{A} & : & \mathit{Dump} \to \mathit{Map} \to \mathit{Expr} \to \mathit{Addr} \\ \mathcal{A}\, d\, M\, e & = & \mathcal{V}\, d\, M\, e \\ \mathcal{A} & : & \mathit{Dump} \to \mathit{Memory} \to \mathit{Expr} \to \mathit{Addr} \\ \mathcal{A}\, d\, m\, e & = & \mathcal{V}\, d\, m\, e \end{array}$$

2.2.7 Consistency

$d, m \models r$ reads "dump d and memory m are consistent with assertion r."

$$\frac{\mathcal{V}\, d\, m\, e_1 = \mathcal{V}\, d\, m\, e_2}{d, m \models e_1 = e_2} \qquad \frac{\mathcal{V}\, d\, m\, e_1 \neq \mathcal{V}\, d\, m\, e_2}{d, m \models e_1 \,!= e_2}$$

$$\frac{\mathcal{V}\, d\, m\, e_1 < \mathcal{V}\, d\, m\, e_2}{d, m \models e_1 < e_2} \qquad \frac{\mathcal{V}\, d\, m\, e_1 \geq \mathcal{V}\, d\, m\, e_2}{d, m \models e_1 >= e_2}$$

$$\frac{\mathcal{V}\, d\, m\, e_1 > \mathcal{V}\, d\, m\, e_2}{d, m \models e_1 > e_2} \qquad \frac{\mathcal{V}\, d\, m\, e_1 \leq \mathcal{V}\, d\, m\, e_2}{d, m \models e_1 <= e_2}$$

Order on $\mathbb{Z}$ are defined as usual, and order between two indexed addresses in $\mathit{Region} \times \mathit{Index}$ are defined as the order on Index if their Region coincides.

2.2.8 Operations

Let us consider four binary operations, addition(+), subtraction(-), multiplication(*), and division(/) denoted as $\star$ in the previous definitions.

Operations on $\mathbb{Z}$ are defined as usual.

$$\begin{array}{rcl} +, -, *, / & : & \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \\ n + n' & = & n + n' \\ n - n' & = & n - n' \\ n * n' & = & n \times n' \\ n / n' & = & n \div n' \end{array}$$

Addition can shift indexed addresses,

$$\begin{array}{rcl} + & : & (\mathit{Region} \times \mathit{Index}) \times \mathbb{Z} \to (\mathit{Region} \times \mathit{Index}) \\ (r, i) + n & = & (r, i + n) \\ + & : & \mathbb{Z} \times (\mathit{Region} \times \mathit{Index}) \to (\mathit{Region} \times \mathit{Index}) \\ n + (r, i) & = & (r, n + i) \end{array}$$

and subtraction can compute the distance between two.

$$\begin{array}{rcl} - & : & (\mathit{Region} \times \mathit{Index}) \times (\mathit{Region} \times \mathit{Index}) \to \mathbb{Z} \\ (r, i) - (r, j) & = & i - j \end{array}$$

3 G Program Analysis

3.1 Abstraction

3.1.1 Abstract domain

$$
\begin{array}{lcll}
\hat{S} & \in & \hat{Summary} & = \hat{Graph} \times \hat{Table} \times \hat{Dump} \\
\hat{G} & \in & \hat{Graph} & = Edges \\
\hat{T} & \in & \hat{Table} & = Block \xrightarrow{\text{fin}} \hat{Memory} \\
\hat{m} & \in & \hat{Memory} & = \hat{Map} \times \hat{Alloc} \\
\hat{M} & \in & \hat{Map} & = \hat{Addr} \xrightarrow{\text{fin}} \hat{Value} \\
\hat{V} & \in & \hat{Value} & = \wp(\hat{Val}) \\
\hat{A} & \in & \hat{Address} & = \wp(\hat{Addr}) \\
\hat{v} & \in & \hat{Val} & = \hat{\mathbb{Z}} + \hat{Addr} \\
\hat{a} & \in & \hat{Addr} & = GVar + ProcId \times Var + \\
 & & & \quad \hat{Region} \times \hat{Index} + \hat{Region} \times FieldId \\
\hat{I} & \in & \hat{Alloc} & = \hat{Region} \xrightarrow{\text{fin}} (\hat{Size} \times \wp(FieldId^*)) \\
\hat{r} & \in & \hat{Region} & = AllocSite \\
 & & \hat{Size} & = \hat{\mathbb{Z}} \\
\hat{i} & \in & \hat{Index} & = \hat{\mathbb{Z}} \\
\hat{D} & \in & \hat{Dump} & = ProcId \xrightarrow{\text{fin}} \hat{Address}
\end{array}
$$

Order defined on each abstract domain respects the subset order of the corresponding concrete domain. Order on $\hat{Alloc}$ are pointwisely defined. Intervals in $\hat{\mathbb{Z}}$ have exactly the same order to the subset order they mean. $\hat{Addr}$ and $\hat{Val}$ have order naturally derived from $\hat{\mathbb{Z}}$, and set of them in $\hat{Address}$ and $\hat{Value}$ are ordered if and only if all elements they contain are ordered.

3.1.2 Galois connections

We define the concretization part of each Galois connection. The abstractions are uniquely defined as following thanks to the property of Galois connection[8, theorem 5.3.0.5].

$$
\begin{array}{rcl}
 & & \wp(X) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{X} \\
\alpha\, X' & = & \bigsqcap \{\hat{x} \in \hat{X} \mid X' \subseteq \gamma\, \hat{x}\}
\end{array}
$$

3.1.3 Trace abstraction

Set of traces a program P generates is summarized by three abstract components: $\hat{Graph}$, $\hat{Table}$, and $\hat{Dump}$.

$$
\begin{array}{rcl}
\wp(Trace) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{Graph} \times \hat{Table} \times \hat{Dump} = \hat{Summary} & & \\
\gamma\, (\hat{G}, \hat{T}, \hat{D}) & = & \gamma_{ctrl}\, \hat{G} \cap \gamma_{data}\, \hat{T} \cap \gamma'_{Dump}\, \hat{D}
\end{array}
$$

Control flow abstraction

$$
\begin{array}{rcl}
 & & \wp(Trace) \underset{\alpha_{ctrl}}{\overset{\gamma_{ctrl}}{\leftrightarrows}} \hat{Graph} \\
\gamma_{ctrl}\, \hat{G} & = & \{((b_1, \_, \_), \cdots, (b_n, \_, \_)) \in Trace \mid \\
 & & \quad \forall i \in \{1, \cdots, n-1\} : (b_i, b_{i+1}) \in \hat{G}\}
\end{array}
$$

Data state abstraction

$$
\begin{array}{rcl}
 & & \wp(Trace) \underset{\alpha_{data}}{\overset{\gamma_{data}}{\leftrightarrows}} \hat{Table} \\
\gamma_{data}\, \hat{T} & = & \gamma_{State}\, \{(b, m, \_) \in State \mid m \in \gamma_{Memory}\, (\hat{T}(b))\}
\end{array}
$$

Dump abstraction

$$\wp(Trace) \underset{\alpha'_{Dump}}{\overset{\gamma'_{Dump}}{\leftrightarrows}} \hat{Dump}$$

$$\gamma'_{Dump}\ \hat{D} \quad = \quad \gamma_{State}\ \{(\_, \_, d) \in State \mid d \in \gamma_{Dump}\ \hat{D}\}$$

$$\wp(Dump) \underset{\alpha_{Dump}}{\overset{\gamma_{Dump}}{\leftrightarrows}} \hat{Dump}$$

$$\begin{array}{rcl} \gamma_{Dump}\ \hat{D} & = & \{d \in Dump \mid d = (\mathsf{p}, \_, a) :: \_\}, \\ & & \mathsf{p} = \eta_{Dump}\ d, a \in \gamma_{Addr}\ \hat{D}(\mathsf{p}) \end{array}$$

$$\eta_{Dump} \quad : \quad Dump \to ProcId$$

$$\eta_{Dump}\ ((\mathsf{p}, \_, \_) :: \_) \quad = \quad \mathsf{p}$$

States of traces without order

$$\wp(Trace) \underset{\alpha_{State}}{\overset{\gamma_{State}}{\leftrightarrows}} \wp(State)$$

$$\gamma_{State}\ \Sigma \quad = \quad \{(\sigma_1, \cdots, \sigma_n) \mid \forall i \in \{1, \cdots, n\} : \sigma_i \in \Sigma\}$$

3.1.4 Memory abstraction

Memories

$$\wp(Memory) \underset{\alpha_{Memory}}{\overset{\gamma_{Memory}}{\leftrightarrows}} \hat{Memory}$$

$$\begin{array}{rcl} \gamma_{Memory}\ (\hat{M}, \hat{I}) & = & \{(M, \_, I) \in Memory \mid \\ & & M \in \gamma_{Map}\ \hat{M}, I \in \gamma_{Alloc}\ \hat{I}\} \end{array}$$

Value mappings

$$\wp(Map) \underset{\alpha_{Map}}{\overset{\gamma_{Map}}{\leftrightarrows}} \hat{Map}$$

$$\begin{array}{rcl} \gamma_{Map}\ \hat{M} & = & \{M \in Map \mid \forall a \mapsto v \in M, \hat{a} \mapsto \hat{V} \in \hat{M} : \\ & & \eta_{Addr}\ a \sqsubseteq \hat{a} \Longrightarrow v \in \gamma_{Val}\ \hat{V}, \\ & & \exists \hat{a}' \in \mathrm{dom}(\hat{M}) : \eta_{Addr}\ a \sqsubseteq \hat{a}'\} \end{array}$$

$$\eta_{Addr} \quad : \quad Addr \to \hat{Addr}$$

$$\begin{array}{rcl} \eta_{Addr}\ \mathsf{X} & = & \mathsf{X} \\ \eta_{Addr}\ (d, \mathsf{x}) & = & (\eta_{Dump}\ d, \mathsf{x}) \\ \eta_{Addr}\ (r, n) & = & (\eta_{Region}\ r, [n, n]) \\ \eta_{Addr}\ (r, \mathsf{f}) & = & (\eta_{Region}\ r, \mathsf{f}) \end{array}$$

Allocation information

$$\wp(Alloc) \underset{\alpha_{Alloc}}{\overset{\gamma_{Alloc}}{\leftrightarrows}} \hat{Alloc}$$

$$\begin{array}{rcl} \gamma_{Alloc}\ \hat{I} & = & \{I \in Alloc \mid \forall r \mapsto i \in I : \\ & & i \in \gamma_{\mathbb{Z}}\ \hat{n} \cup F, (\hat{n}, F) = \hat{I}(\eta_{Region}\ r)\} \end{array}$$

$$\eta_{Region} \quad : \quad Region \to \hat{Region}$$

$$\eta_{Region}\ (\_, \mathsf{t}) \quad = \quad \mathsf{t}$$

3.1.5 Value abstraction

Values

$$\wp(Val) \underset{\alpha_{Val}}{\overset{\gamma_{Val}}{\leftrightarrows}} \wp(\hat{Val}) = \hat{Value}$$

$$\gamma_{Val}\ \hat{V} \quad = \quad \bigcup\{\gamma_{\mathbb{Z}}\ \hat{n} \mid \hat{n} \in \hat{V} \cap \hat{\mathbb{Z}}\} \cup \gamma_{Addr}\ (\hat{V} - \hat{\mathbb{Z}})$$

Addresses

$$\wp(Addr) \underset{\alpha_{Addr}}{\overset{\gamma_{Addr}}{\leftrightarrows}} \wp(\hat{Addr}) = \hat{Address}$$
$$\gamma_{Addr}\ \hat{A} \quad = \quad \{a \mid \eta_{Addr}\ a \sqsubseteq \hat{a}, \hat{a} \in \hat{A}\}$$

Numerals

$$\wp(\mathbb{Z}) \underset{\alpha_{\mathbb{Z}}}{\overset{\gamma_{\mathbb{Z}}}{\leftrightarrows}} \hat{\mathbb{Z}}$$
$$\begin{array}{lcl}
\gamma_{\mathbb{Z}}\ \hat{\bot}_{\hat{\mathbb{Z}}} & = & \emptyset \\
\gamma_{\mathbb{Z}}\ [n,m] & = & \{z \in \mathbb{Z} \mid n \le z \le m\} \\
\gamma_{\mathbb{Z}}\ [n,+\infty] & = & \{z \in \mathbb{Z} \mid n \le z\} \\
\gamma_{\mathbb{Z}}\ [-\infty,m] & = & \{z \in \mathbb{Z} \mid z \le m\} \\
\gamma_{\mathbb{Z}}\ [-\infty,+\infty] & = & \mathbb{Z}
\end{array}$$

3.2 Abstract semantics

Let us define the abstract semantics $[\![\hat{P}]\!]$ of a G program P from some $\hat{S}_0 \in \hat{Summary}$ that covers all initial states, as the least fixed point of the abstract semantic transfer function $\hat{\mathcal{F}}\ P$ greater than or equal to $\hat{S}_0$.

$$[\![\hat{P}]\!] \quad : \quad \hat{Summary} \to \hat{Summary}$$
$$[\![\hat{P}]\!]\ \hat{S}_0 \quad = \quad \mathrm{lfp}_{\hat{S}_0}(\hat{\mathcal{F}}\ P) \quad \text{where}$$

$$\hat{\mathcal{F}} \quad : \quad Pgm \to \hat{Summary} \to \hat{Summary}$$
$$\hat{\mathcal{F}}\ P\ (\hat{G},\hat{T},\hat{D}) \quad = \quad (\hat{G}',\hat{T}',\hat{D}') \quad \text{where}$$

$$\begin{array}{lll}
\hat{G}' = \hat{G} & \cup & Flows_P \\
& \cup & \bigcup\{\ \{(b^{\mathsf{p}}, \mathtt{ENTRY}_{\mathsf{p}'}), (\mathtt{EXIT}_{\mathsf{p}'}, b')\} \mid b^{\mathsf{p}} = \mathtt{CALL}(_, e_0, _), \\
& & \qquad (_, b^{\mathsf{p}}) \in \hat{G},\ (b^{\mathsf{p}}, b') \in Resumes_P, \\
& & \qquad b^{\mathsf{p}} \mapsto \hat{m} \in \hat{T},\ \mathsf{p}' \in \hat{\mathcal{A}}\ \mathsf{p}\ \hat{m}\ e_0,\ (\mathsf{p}', _) \in P \quad \}
\end{array}$$

$$\begin{array}{lll}
\hat{T}' = \hat{T} & \sqcup & \{\ \mathtt{ENTRY}_{\mathsf{p}'} \mapsto \hat{m}' \mid b^{\mathsf{p}} = \mathtt{CALL}(_, e_0, e_1), \\
& & \qquad (_, b^{\mathsf{p}}) \in \hat{G}, \\
& & \qquad b^{\mathsf{p}} \mapsto \hat{m} \in \hat{T},\ \mathsf{p}' \in \hat{\mathcal{A}}\ \mathsf{p}\ \hat{m}\ e_0,\ (\mathsf{p}', \mathsf{x}) \in P, \\
& & \qquad (\hat{M}, \hat{I}) = \hat{m},\ \hat{m}' = (\hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}'\ \hat{m}\ \mathsf{x} \hat{\mapsto} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e_1\}, \hat{I}) \quad \} \\
& \sqcup & \{\ b' \mapsto \hat{m}' \mid b \in \{\mathtt{ENTRY}_{\mathsf{p}}, \mathtt{EXIT}_{\mathsf{p}}\}, \\
& & \qquad (b, b') \in \hat{G}, \\
& & \qquad b \mapsto \hat{m}' \in \hat{T} \quad \} \\
& \sqcup & \{\ b' \mapsto \hat{m}' \mid b^{\mathsf{p}} = (c_1, \cdots, c_n), \\
& & \qquad (b^{\mathsf{p}}, b') \in \hat{G}, \\
& & \qquad b^{\mathsf{p}} \mapsto \hat{m} \in \hat{T},\ \hat{m}' = (\hat{\mathcal{R}}\ \mathsf{p}\ c_1; \cdots; \hat{\mathcal{R}}\ \mathsf{p}\ c_n)\ \hat{m} \quad \}
\end{array}$$

$$\begin{array}{lll}
\hat{D}' = \hat{D} & \sqcup & \{\ \mathsf{p}' \mapsto \hat{\mathcal{A}}\ \mathsf{p}\ \hat{m}\ e \mid b^{\mathsf{p}} = \mathtt{CALL}(e, e_0, _), \\
& & \qquad (_, b^{\mathsf{p}}) \in \hat{G}, \\
& & \qquad b^{\mathsf{p}} \mapsto \hat{m} \in \hat{T},\ \mathsf{p}' \in \hat{\mathcal{A}}\ \mathsf{p}\ \hat{m}\ e_0,\ (\mathsf{p}', _) \in P \quad \}
\end{array}$$

where $\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ c\ \hat{m}$, $\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e$ and $\hat{\mathcal{A}}\ \mathsf{p}\ \hat{m}\ e$ respectively defines the abstract semantics of command c, expression e under $\mathsf{p} \in ProcId$, $\hat{D} \in \hat{Dump}$, and $\hat{m} \in \hat{Memory}$.

3.2.1 Reaction

$\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ c\ \hat{m}$ defines how a single command c under some context $\hat{D}$ and p changes the abstract memory $\hat{m}$.

$$\hat{\mathcal{R}} \quad : \quad \hat{Dump} \to ProcId \to Cmd \to \hat{Memory} \to \hat{Memory}$$

$$\begin{array}{lcl}
\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{SET}(e_a, e)\ (\hat{M}, \hat{I}) & = & (\hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a \hat{\mapsto} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e\}, \hat{I}) \\
\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{ESCAPE}(e)\ (\hat{M}, \hat{I}) & = & (\hat{M}\{\hat{D}(\mathsf{p}) \hat{\mapsto} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e\}, \hat{I}) \\
\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{ASSUME}(r)\ (\hat{M}, \hat{I}) & = & \hat{\mathcal{P}}\ \mathsf{p}\ r\ (\hat{M}, \hat{I}) \\
\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{ALLOC}(e_a, [e])_{\mathsf{t}}\ (\hat{M}, \hat{I}) & = & (\hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a \hat{\mapsto} \{(\hat{r}, [0,0])\}\}, \\
 & & \hat{I}\{\hat{r} \mapsto \hat{I}(\hat{r}) \sqcup (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e, \emptyset)\}) \\
 & & \quad \text{where } \hat{r} = \mathsf{t} \\
\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{ALLOC}(e_a, \{\vec{\mathsf{f}}\})_{\mathsf{t}}\ (\hat{M}, \hat{I}) & = & (\hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a \hat{\mapsto} \{(\hat{r}, \mathsf{f}_1)\}\}, \\
 & & \hat{I}\{\hat{r} \mapsto \hat{I}(\hat{r}) \sqcup (\hat{\perp}_{\hat{\mathbb{Z}}}, \{\vec{\mathsf{f}}\})\}) \\
 & & \quad \text{where } \hat{r} = \mathsf{t} \\
\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{FREE}(e)\ (\hat{M}, \hat{I}) & = & (\hat{M}, \hat{I})
\end{array}$$

3.2.2 Lookup and update

Abstract value mapping $\hat{M} \in \hat{Map}$ is always used with set of abstract addresses $\hat{A} \in \hat{Address}$ in these semantics definitions. Hence, let us define two abbreviations here that extend lookup and update on $\hat{Map} = \hat{Addr} \xrightarrow{\text{fin}} \hat{Value}$ to $\hat{Address} \to \hat{Value}$.

First, let us define set of overlapping addresses as following.

$$overlap(\hat{a}, \hat{A}) = \{\hat{a}' \in \hat{A} \mid \hat{a} \sqcap \hat{a}' \neq \hat{\perp}_{\hat{Addr}}\}$$

Lookup with $\hat{A}$ is defined as following.

$$\hat{M}\,?\,\hat{A} \quad = \quad \bigsqcup(\{\hat{M}(\hat{a}') \mid \hat{a}' \in overlap(\hat{a}, \mathrm{dom}(\hat{M})), \hat{a} \in \hat{A}\})$$

Update with $\hat{A}$ is defined as following.

$$\begin{array}{lcl}
\hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\} & = & \hat{M} \\
 & \sqcup & \{\hat{a}' \mapsto \hat{V} \mid \hat{a}' \in overlap(\hat{a}, \mathrm{dom}(\hat{M})), \hat{a} \in \hat{A}\} \\
 & \sqcup & \{\hat{a} \mapsto \hat{V} \mid \hat{a} \in \hat{A} - \hat{Region} \times \hat{Index}\} \\
 & \sqcup & \{(\hat{r}, [-\infty, +\infty]) \mapsto \hat{V} \mid (\hat{r}, _) \in \hat{A}\}
\end{array}$$

3.2.3 Value

$\hat{\mathcal{V}}$ defines abstract value for each expression of a procedure p, which is an abstraction of possible values such expression can compute under some value mapping $\hat{M}$.

$$\hat{\mathcal{V}} \quad : \quad ProcId \to \hat{Map} \to Expr \to \hat{Value}$$

$$\begin{array}{lcl}
\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ n & = & \{[n, n]\} \\
\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ \mathsf{X} & = & \{\mathsf{X}\} \\
\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ \mathsf{x} & = & \{(\mathsf{p}, \mathsf{x})\} \\
\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e_1 \star e_2 & = & \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e_1 \hat{\star} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e_2 \\
\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e.\mathsf{f} & = & \{(\hat{r}, \mathsf{f}) \mid (\hat{r}, _) \in \hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e\} \\
\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ \$e & = & \hat{M}\,?\,\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e
\end{array}$$

For convenience, let us overload $\hat{\mathcal{V}}$ and also use it on $\hat{Memory}$.

$$\hat{\mathcal{V}} \quad : \quad ProcId \to \hat{Memory} \to Expr \to \hat{Value}$$

$$\hat{\mathcal{V}}\ \mathsf{p}\ (\hat{M}, \_)\ e \quad = \quad \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e$$

3.2.4 Address

$\hat{\mathcal{A}}$ defines abstract address for each expression.

$$\begin{array}{rcl} \hat{\mathcal{A}} & : & \mathit{ProcId} \to \hat{\mathit{Map}} \to \mathit{Expr} \to \hat{\mathit{Address}} \\ \hat{\mathcal{A}}\,\mathsf{p}\,\hat{M}\,e & = & \hat{\mathcal{V}}\,\mathsf{p}\,\hat{M}\,e \end{array}$$

$$\begin{array}{rcl} \hat{\mathcal{A}} & : & \mathit{ProcId} \to \hat{\mathit{Memory}} \to \mathit{Expr} \to \hat{\mathit{Address}} \\ \hat{\mathcal{A}}\,\mathsf{p}\,\hat{m}\,e & = & \hat{\mathcal{V}}\,\mathsf{p}\,\hat{m}\,e \end{array}$$

3.2.5 Pruning

Pruning $\hat{\mathcal{P}}$ defines a refined abstract memory according to the relation being assumed.

$$\begin{array}{rcl} \hat{\mathcal{P}} & : & \mathit{ProcId} \to \mathit{Rel} \to \hat{\mathit{Memory}} \to \hat{\mathit{Memory}} \\ \hat{\mathcal{P}}\,\mathsf{p}\,r\,(\hat{M},\hat{I}) & = & (\bigsqcap(\{\hat{M}\{\hat{A} \hookmapsto \hat{V}\} \mid \hat{A} = \hat{\mathcal{A}}\,\mathsf{p}\,(\hat{M},\hat{I})\,e, \\ & & \quad e \in \mathcal{L}\,r, \$e \diamond e' \in \mathcal{N}\,e\,r \cup \mathcal{N}\,e\,r^t, \\ & & \quad \hat{V} = \hat{M}\,?\,\hat{A} \sqcap \mathcal{S}_\diamond\,(\hat{\mathcal{V}}\,\mathsf{p}\,(\hat{M},\hat{I})\,e'), \\ & & \quad \hat{V} \neq \hat{\bot}_{\hat{\mathit{Value}}}\} \cup \{\hat{M} \mid e \diamond e' = r, \\ & & \quad \hat{\mathcal{V}}\,\mathsf{p}\,(\hat{M},\hat{I})\,e \sqcap \mathcal{S}_\diamond\,(\hat{\mathcal{V}}\,\mathsf{p}\,(\hat{M},\hat{I})\,e') \neq \hat{\bot}_{\hat{\mathit{Value}}}\}, \hat{I}) \end{array}$$

$\mathcal{S}_\diamond$ defines an abstract value which is an abstraction of possible concrete values that can satisfy the relation $\diamond$ with a value among the meaning of the given abstract value.

$$\begin{array}{rcl} \mathcal{S}_\diamond & : & \hat{\mathit{Value}} \to \hat{\mathit{Value}} \\ \mathcal{S}_{=}\,\hat{V} & = & \hat{V} \\ \mathcal{S}_{!=}\,\hat{V} & = & \hat{\top}_{\hat{\mathit{Value}}} \\ \mathcal{S}_{<}\,\hat{V} & = & \{[-\infty, u-1] \mid [l,u] \in \hat{V}\} \\ & \cup & \{(\hat{r}, [-\infty, u-1]) \mid (\hat{r}, [l,u]) \in \hat{V}\} \\ \mathcal{S}_{>=}\,\hat{V} & = & \{[l, +\infty] \mid [l,u] \in \hat{V}\} \\ & \cup & \{(\hat{r}, [l, +\infty]) \mid (\hat{r}, [l,u]) \in \hat{V}\} \\ \mathcal{S}_{>}\,\hat{V} & = & \{[l+1, +\infty] \mid [l,u] \in \hat{V}\} \\ & \cup & \{(\hat{r}, [l+1, +\infty]) \mid (\hat{r}, [l,u]) \in \hat{V}\} \\ \mathcal{S}_{<=}\,\hat{V} & = & \{[-\infty, u] \mid [l,u] \in \hat{V}\} \\ & \cup & \{(\hat{r}, [-\infty, u]) \mid (\hat{r}, [l,u]) \in \hat{V}\} \end{array}$$

$\mathcal{L}$ gives a set of expressions that computes an address whose value mapping in the memory can be modified.

$$\begin{array}{rcl} \mathcal{L} & : & \mathit{Expr} \to \wp(\mathit{Expr}) \\ \mathcal{L}\,\$e & = & \{e\} \\ \mathcal{L}\,e \star e' & = & \mathcal{L}\,e \cup \mathcal{L}\,e' \\ \mathcal{L}\,\_ & = & \emptyset \end{array}$$

$$\begin{array}{rcl} \mathcal{L} & : & \mathit{Rel} \to \wp(\mathit{Expr}) \\ \mathcal{L}\,e \diamond e' & = & \mathcal{L}\,e \cup \mathcal{L}\,e' \end{array}$$

$\mathcal{N}\,e$ defines all equivalent relations with the given one whose left hand side is $\$e$.

$$\begin{array}{rcl} \mathcal{N} & : & \mathit{Expr} \to \mathit{Rel} \to \wp(\mathit{Rel}) \\ \mathcal{N}\,e\,\$e \diamond e' & = & \{\$e \diamond e'\} \\ \mathcal{N}\,e\,(e_1 + e_2) \diamond e_3 & = & \mathcal{N}\,e\,e_1 \diamond (e_3 - e_2) \cup \mathcal{N}\,e\,e_2 \diamond (e_3 - e_1) \\ \mathcal{N}\,e\,(e_1 - e_2) \diamond e_3 & = & \mathcal{N}\,e\,e_1 \diamond (e_3 + e_2) \cup \mathcal{N}\,e\,e_2 \diamond^t (e_1 - e_3) \\ \mathcal{N}\,e\,(n * e_1) \diamond e_2 & = & \mathcal{N}\,e\,(e_1 * n) \diamond e_2 \\ \mathcal{N}\,e\,(e_1 * n) \diamond e_2 & = & \mathcal{N}\,e\,e_1 \diamond (e_2 / n) \text{ if } n > 0 \\ \mathcal{N}\,e\,(e_1 / n) \diamond e_2 & = & \mathcal{N}\,e\,e_1 \diamond (e_2 * n) \text{ if } n > 0 \\ \mathcal{N}\,e\,(e_1 * n) \diamond e_2 & = & \mathcal{N}\,e\,e_1 \diamond^t (e_2 / n) \text{ if } n < 0 \\ \mathcal{N}\,e\,(e_1 / n) \diamond e_2 & = & \mathcal{N}\,e\,e_1 \diamond^t (e_2 * n) \text{ if } n < 0 \\ \mathcal{N}\,e\,\_ & = & \emptyset \end{array}$$

r^t is the relation that has the same meaning as r but its left and right hand sides swapped.

$$\begin{array}{rcl} \cdot^t & : & Rel \to Rel \\ (e \diamond e')^t & = & e' \diamond^t e \text{ where} \\ & & \begin{array}{cccccc} =^t & = & =, & !=^t & = & !=, \\ <^t & = & >, & >=^t & = & <=, \\ >^t & = & <, & <=^t & = & >= \end{array} \end{array}$$

3.2.6 Operations

$$\hat{+}, \hat{-}, \hat{*}, \hat{/} \quad : \quad \hat{Value} \times \hat{Value} \to \hat{Value}$$

$$\begin{array}{rcl} \hat{V} \hat{+} \hat{V}' & = & \{\hat{n} \hat{+} \hat{n}' \mid \hat{n} \in \hat{V}, \hat{n}' \in \hat{V}'\} \\ & \cup & \{(\hat{r}, \hat{n} \hat{+} \hat{n}') \mid (\hat{r}, \hat{n}) \in \hat{V}, \hat{n}' \in \hat{V}'\} \\ & \cup & \{(\hat{r}, \hat{n} \hat{+} \hat{n}') \mid \hat{n} \in \hat{V}, (\hat{r}, \hat{n}') \in \hat{V}'\} \\ \hat{V} \hat{-} \hat{V}' & = & \{\hat{n} \hat{-} \hat{n}' \mid \hat{n} \in \hat{V}, \hat{n}' \in \hat{V}'\} \\ & \cup & \{\hat{n} \hat{-} \hat{n}' \mid (\hat{r}, \hat{n}) \in \hat{V}, (\hat{r}, \hat{n}') \in \hat{V}'\} \\ \hat{V} \hat{*} \hat{V}' & = & \{\hat{n} \hat{*} \hat{n}' \mid \hat{n} \in \hat{V}, \hat{n}' \in \hat{V}'\} \\ \hat{V} \hat{/} \hat{V}' & = & \{\hat{n} \hat{/} \hat{n}' \mid \hat{n} \in \hat{V}, \hat{n}' \in \hat{V}'\} \end{array}$$

$$\hat{+}, \hat{-}, \hat{*}, \hat{/} \quad : \quad \hat{\mathbb{Z}} \times \hat{\mathbb{Z}} \to \hat{\mathbb{Z}}$$

$$\begin{array}{rcl} [a,b] \hat{+} [c,d] & = & [a+c, b+d] \text{ where } \forall z \in \mathbb{Z} : \\ & & \quad -\infty + z = z + -\infty = -\infty + -\infty = -\infty, \\ & & \quad +\infty + z = z + +\infty = +\infty + +\infty = +\infty \\ [a,b] \hat{-} [c,d] & = & [a-d, b-c] \text{ where } \forall z \in \mathbb{Z} : \\ & & \quad -\infty - z = z - +\infty = -\infty - +\infty = -\infty, \\ & & \quad +\infty - z = z - -\infty = +\infty - -\infty = +\infty \\ [a,b] \hat{*} [c,d] & = & [\min X, \max X] \text{ where} \\ & & \quad X = \{a \times c, a \times d, b \times c, b \times d\} \\ [a,b] \hat{/} [c,d] & = & [-\infty, +\infty] \text{ if } c \le 0 \le d \\ & = & [\min X, \max X] \text{ otherwise where} \\ & & \quad X = \{a \div c, a \div d, b \div c, b \div d\} \end{array}$$

4 Soundness

4.1 Semantics

Theorem 1 *Abstract semantics is sound, i.e.*

$$\begin{array}{ll} & \Sigma \subseteq \gamma\ \hat{S} \\ \Longrightarrow & \forall P \in Pgm : \mathrm{lfp}_{\Sigma}(\mathcal{F}\ P) \subseteq \gamma\ (\mathrm{lfp}_{\hat{S}}(\hat{\mathcal{F}}\ P)) \end{array}$$

Proof Since $\hat{\mathcal{F}}\ P$ is monotone, by *fixpoint transfer theorem*[8, theorem 7.1.0.4] with Lemma 1, we have the soundness of abstract semantics. □

Lemma 1 *Transition $\hat{\mathcal{F}}$ is sound, i.e.*

$$\begin{array}{ll} & T \subseteq \gamma\ \hat{S} \\ \Longrightarrow & \forall P \in Pgm : \mathcal{F}\ P\ T \subseteq \gamma\ (\hat{\mathcal{F}}\ P\ \hat{S}) \end{array}$$

Proof Since $\mathcal{F}\ P$ and $\hat{\mathcal{F}}\ P$ are both extensive, we only need to check all new trace $\tau' \in \mathcal{F}\ P\ T$ such that $\tau' \notin T$ is also in $\gamma\ (\hat{\mathcal{F}}\ P\ \hat{S})$. If $\tau = (\sigma_1, \cdots, \sigma) \in T$, then by definition of $\mathcal{F}$, new traces are

$$\tau' = (\sigma_1, \cdots, \sigma, \sigma') \in \mathcal{F}\ P\ T.$$

such that $\sigma \longrightarrow \sigma'$. So, let us consider each case of transition,

$$(b, m, d) \longrightarrow (b', m', d')$$

where $\sigma = (b, m, d)$ and $\sigma' = (b', m', d')$. Let us use $(\hat{G}, \hat{T}, \hat{D})$ to refer $\hat{S}$ and $(\hat{G}', \hat{T}', \hat{D}') = \hat{S}'$ to refer $\hat{\mathcal{F}}\ P\ \hat{S}$. We are going to check whether $\tau' \in \gamma\ (\hat{G}', \hat{T}', \hat{D}')$ holds when $\tau \in \gamma\ (\hat{G}, \hat{T}, \hat{D})$. $m \in \gamma_{Memory}\ \hat{T}(b)$ and $d \in \gamma_{Dump}\ \hat{D}$.

When $b = \texttt{CALL}(e, e_0, e_1)$, by definition of $\longrightarrow$,

$$\mathsf{p}' = \mathcal{A}\ d\ m\ e_0 \quad \text{and} \quad b' = \texttt{ENTRY}_{\mathsf{p}'}$$

By Corollary 1,

$$\mathsf{p}' = \mathcal{A}\ d\ m\ e_0 \in \gamma_{Addr}\ (\hat{\mathcal{A}}\ \mathsf{p}\ \hat{m}\ e_0).$$

By definition of γ_{Addr},

$$\mathsf{p}' \in \gamma_{Addr}\ \hat{A} \iff \mathsf{p}' \in \hat{A}.$$

Hence, by definition of $\hat{\mathcal{F}}$, we have

$$(b, \texttt{ENTRY}_{\mathsf{p}'}) = (b, b') \in \hat{G}'.$$

Now, by definition of $\longrightarrow$,

$$d' = (\mathsf{p}', b_r, \mathcal{A}\ d\ m\ e) :: d.$$

By definition, $\eta_{Dump}\ d' = \mathsf{p}'$. Because of the following,

$$\begin{array}{lcl} \mathcal{A}\ d\ m\ e & \in & \gamma_{Addr}\ (\hat{\mathcal{A}}\ \mathsf{p}\ \hat{m}\ e) \\ & & \quad \text{(Corollary 1)} \\ & \subseteq & \gamma_{Addr}\ (\hat{D}'(\mathsf{p}')) \\ & & \quad \text{(definition of } \hat{\mathcal{F}}\text{, } \gamma_{Addr} \text{ is monotone)} \end{array}$$

definition of γ_{Dump} gives,

$$d' \in \gamma_{Dump}\ \hat{D}'.$$

By definition of $\longrightarrow$,

$$m' = (M\{(d', \mathsf{x}) \mapsto \mathcal{V}\ d\ m\ e_1\}, c, I)$$

where $(\mathsf{p}', \mathsf{x}) \in P$. By definition of $\mathcal{A}$ and γ_{Addr},

$$\begin{array}{lcl} (d', \mathsf{x}) & = & \mathcal{A}\ d'\ m\ \mathsf{x} \\ & & \quad \text{(definition of } \mathcal{A}\text{)} \\ & \in & \gamma_{Addr}\ (\hat{\mathcal{A}}\ \mathsf{p}'\ \hat{m}\ \mathsf{x}) \\ & & \quad \text{(Corollary 1)} \end{array}$$

By Lemma 6,

$$\mathcal{V}\ d\ m\ e_1 \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e_1).$$

By Lemma 5,

$$M\{(d', \mathsf{x}) \mapsto \mathcal{V}\ d\ m\ e_1\} \in \gamma_{Map}\ \hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}'\ \hat{m}\ \mathsf{x} \hat{\longmapsto} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e_1\},$$

and therefore,

$$m' \in \gamma_{Memory}\ \hat{m}' = \gamma_{Memory}\ (\hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}'\ \hat{m}\ \mathsf{x} \hat{\longmapsto} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e_1\}, \hat{I}).$$

Since γ_{Memory} is monotone and by definition of $\hat{\mathcal{F}}$,

$$m' \in \gamma_{Memory}\ \hat{m}' \subseteq \gamma_{Memory}\ \hat{T}'(\texttt{ENTRY}_{\mathsf{p}'}) = \gamma_{Memory}\ \hat{T}'(b').$$

Hence, we have $\tau' \in \gamma\ (\hat{G}', \hat{T}', \hat{D}')$.

When $b = \texttt{ENTRY}_{\mathsf{p}}$, by definition of $\longrightarrow$, $(\texttt{ENTRY}_{\mathsf{p}}, b') \in \mathit{Flows}_P$, $m' = m$ and $d' = d$. Since $\mathit{Flows}_P \subseteq \hat{G}'$, $(\texttt{ENTRY}_{\mathsf{p}}, b') \in \hat{G}'$. Since γ_{Memory} is monotone,

$$m' = m \in \gamma_{\mathit{Memory}}\ \hat{T}(b') \subseteq \gamma_{\mathit{Memory}}\ \hat{T}'(b').$$

Since γ_{Dump} is monotone,

$$d' = d \in \gamma_{\mathit{Dump}}\ \hat{D} \subseteq \gamma_{\mathit{Dump}}\ \hat{D}'.$$

Therefore, $\tau' \in \gamma\ (\hat{G}', \hat{T}', \hat{D}')$.

When $b = \texttt{EXIT}_{\mathsf{p}'}$, by definition of $\longrightarrow$,

$$(\mathsf{p}', b', \_) :: d' = d \quad \text{and} \quad m' = m$$

The only case where $(\mathsf{p}', b', \_)$ is pushed onto d' is $b'' = \texttt{CALL}(e, e_0, e_1)$. Therefore, $(b'', b') \in \mathit{Resumes}_P$ and by definition of $\hat{\mathcal{F}}$, $(\texttt{ENTRY}_{\mathsf{p}'}, b') \in \hat{G}$. Since d' already appeared in the τ with b'', by monotony of γ_{Dump}, we have

$$d' \in \gamma_{\mathit{Dump}}\ \hat{D} \subseteq \gamma_{\mathit{Dump}}\ \hat{D}'.$$

By monotony of γ_{Memory}, we have

$$m' = m \in \gamma_{\mathit{Memory}}\ \hat{T}(b') \subseteq \gamma_{\mathit{Memory}}\ \hat{T}'(b').$$

Therefore, $\tau' \in \gamma\ (\hat{G}', \hat{T}', \hat{D}')$.

When $b = (c_1, \cdots, c_n)$, let us first define each m_i and $\hat{m}_i$ as following.

$$\begin{array}{rclcrcl} m_1 & = & m & & \hat{m}_1 & = & \hat{m} \\ m_{i+1} & = & \mathcal{R}\ d\ c_i\ m_i & & \hat{m}_{i+1} & = & \hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ c_i\ \hat{m}_i \\ m' & = & m_{n+1} & & \hat{m}' & = & \hat{m}_{n+1} \end{array}$$

where $\hat{m} = \hat{T}(b)$. We are going to show $m \in \gamma_{\mathit{Memory}}\ \hat{m}'$ by induction on the length of commands.
Since $\tau \in \gamma\ \hat{S}$ and $\hat{m} = \hat{T}(b)$, by definition of γ, $m \in \gamma_{\mathit{Memory}}\ \hat{m}$, we have the base case,

$$m_1 \in \gamma_{\mathit{Memory}}\ \hat{m}_1.$$

By Lemma 2, we have the inductive case,

$$\begin{array}{rl} & m_i \in \gamma_{\mathit{Memory}}\ \hat{m}_i \\ \Longrightarrow & \mathcal{R}\ d\ c_i\ m_i \in \gamma_{\mathit{Memory}}\ (\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ c_i\ \hat{m}_i) \\ \Longleftrightarrow & m_{i+1} \in \gamma_{\mathit{Memory}}\ \hat{m}_{i+1} \end{array}$$

Hence, by induction, we have $m' \in \gamma_{\mathit{Memory}}\ \hat{m}'$.
Definition of $\longrightarrow$ gives $d' = d$ and $(b, b') \in \mathit{Flows}_P$. Since $\mathit{Flows}_P \subseteq \hat{G}'$, $(b, b') \in \hat{G}'$. Since γ_{Dump} is monotone, $d' = d \in \gamma_{\mathit{Dump}}\ \hat{D} \subseteq \gamma_{\mathit{Dump}}\ \hat{D}'$.

$$\begin{array}{rcl} m' & \in & \gamma_{\mathit{Memory}}\ \hat{m}' \\ & \subseteq & \gamma_{\mathit{Memory}}\ (\hat{T}(b') \sqcup \hat{m}') \\ & \subseteq & \gamma_{\mathit{Memory}}\ \hat{T}'(b') \end{array}$$

Therefore $\tau' \in \gamma\ (\hat{G}', \hat{T}', \hat{D}')$.

From the above analysis on every τ' that can be generated from possible transitions, we have

$$\tau' \in \gamma\ (\hat{G}', \hat{T}', \hat{D}').$$

Thus,

$$\mathcal{F}\ P\ T \subseteq \gamma\ (\hat{\mathcal{F}}\ P\ \hat{S}).$$

□

4.2 Reaction

Lemma 2 *Reaction $\hat{\mathcal{R}}$ is sound, i.e.*

$$\begin{array}{ll} & d \in \gamma_{Dump}\ \hat{D}, \\ & \eta_{Dump}\ d = \mathsf{p}, \\ & m \in \gamma_{Memory}\ \hat{m} \\ \Longrightarrow & \forall c \in Cmd : \mathcal{R}\ d\ c\ m \in \gamma_{Memory}\ (\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ c\ \hat{m}) \end{array}$$

Proof Let us consider each case of c. Let $(M, c, I) = m$ and $(\hat{M}, \hat{I}) = \hat{m}$ where $M \in \gamma_{Map}\ \hat{M}$ and $I \in \gamma_{Alloc}\ \hat{I}$.

When $c = \texttt{SET}(e_a, e)$, by definition of $\mathcal{R}$,

$$\mathcal{R}\ d\ \texttt{SET}(e_a, e)\ (M, c, I) = (M\{\mathcal{A}\ d\ M\ e_a \mapsto \mathcal{V}\ d\ M\ e\}, c, I).$$

By Corollary 1,

$$\mathcal{A}\ d\ M\ e_a \in \gamma_{Addr}\ (\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a)$$

and by Lemma 6,

$$\mathcal{V}\ d\ M\ e \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e).$$

By Lemma 5,

$$M\{\mathcal{A}\ d\ M\ e_a \mapsto \mathcal{V}\ d\ M\ e\} \in \gamma_{Map}\ \hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a \hat{\longmapsto} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e\}$$

and hence,

$$(M\{\mathcal{A}\ d\ M\ e_a \mapsto \mathcal{V}\ d\ M\ e\}, c, I) \in \gamma_{Memory}\ (\hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a \hat{\longmapsto} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e\}, \hat{I}).$$

Therefore, by definition of $\hat{\mathcal{R}}$,

$$\mathcal{R}\ d\ \texttt{SET}(e_a, e)\ (M, c, I) \in \gamma_{Memory}\ (\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{SET}(e_a, e)\ (\hat{M}, \hat{I})).$$

When $c = \texttt{ESCAPE}(e)$, by definition of $\mathcal{R}$,

$$\mathcal{R}\ d\ \texttt{ESCAPE}(e)\ (M, c, I) = (M\{a \mapsto \mathcal{V}\ d\ M\ e\}, c, I)$$

where $(\_, \_, a) :: \_ = d$. By Lemma 6,

$$\mathcal{V}\ d\ M\ e \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e).$$

Since $d \in \gamma_{Dump}\ \hat{D}$ and $\eta_{Dump}\ d = \mathsf{p}$, by definition of γ_{Dump},

$$a \in \gamma_{Addr}\ \hat{D}(\mathsf{p}).$$

By Lemma 5,

$$M\{a \mapsto \mathcal{V}\ d\ M\ e\} \in \gamma_{Map}\ \hat{M}\{\hat{D}(\mathsf{p}) \hat{\longmapsto} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e\}$$

and hence,

$$(M\{a \mapsto \mathcal{V}\ d\ M\ e\}, c, I) \in \gamma_{Memory}\ (\hat{M}\{\hat{D}(\mathsf{p}) \hat{\longmapsto} \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e\}, \hat{I}).$$

Therefore, by definition of $\hat{\mathcal{R}}$,

$$\mathcal{R}\ d\ \texttt{ESCAPE}(e)\ (M, c, I) \in \gamma_{Memory}\ (\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{ESCAPE}(e)\ (\hat{M}, \hat{I})).$$

When $c = \texttt{ASSUME}(r)$,

$$\mathcal{R}\ d\ \texttt{ASSUME}(r)\ (M, c, I) = (M, c, I)$$

where $M, I \models r$. By Lemma 3,

$$(M, c, I) \in \gamma_{Memory}\ (\hat{\mathcal{P}}\ \mathsf{p}\ r\ (\hat{M}, \hat{I})).$$

Therefore, by definition of $\hat{\mathcal{R}}$,

$$\mathcal{R}\ d\ \texttt{ASSUME(}r\texttt{)}\ (M, c, I) \in \gamma_{Memory}\ (\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{ASSUME(}r\texttt{)}\ (\hat{M}, \hat{I})).$$

When $c = \texttt{ALLOC(}e_a\texttt{, [}e\texttt{])}_{\mathsf{t}}$, by definition of $\mathcal{R}$,

$$\begin{array}{l}\mathcal{R}\ d\ \texttt{ALLOC(}e_a\texttt{, [}e\texttt{])}_{\mathsf{t}}\ (M, c, I)\\ = (M\{\mathcal{A}\ d\ M\ e_a \mapsto (r, 0)\}, c + 1, I\{r \mapsto \mathcal{V}\ d\ M\ e\})\end{array}$$

where $r = (c, \mathsf{t})$. By definition of η_{Region},

$$\eta_{Region}\ r = \mathsf{t}.$$

By Corollary 1,

$$\mathcal{A}\ d\ M\ e_a \in \gamma_{Addr}\ (\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a)$$

and by definition of γ_{Val},

$$(r, 0) \in \gamma_{Val}\ \{(\hat{r}, [0, 0])\}.$$

By Lemma 5,

$$M\{\mathcal{A}\ d\ M\ e_a \mapsto (r, 0)\} \in \gamma_{Map}\ \hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a \hat{\longmapsto} \{(\hat{r}, [0, 0])\}\}.$$

By Lemma 6,

$$\mathcal{V}\ d\ M\ e \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e)$$

and hence,

$$I\{r \mapsto \mathcal{V}\ d\ M\ e\} \in \gamma_{Alloc}\ \hat{I}\{\hat{r} \mapsto \hat{I}(\hat{r}) \sqcup (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e, \emptyset)\}.$$

Therefore, we have

$$\begin{array}{l}(M\{\mathcal{A}\ d\ M\ e_a \mapsto (r, 0)\}, c + 1, I\{r \mapsto \mathcal{V}\ d\ M\ e\}) \in\\ \gamma_{Memory}\ (\hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a \hat{\longmapsto} \{(\hat{r}, [0, 0])\}\}, \hat{I}\{\hat{r} \mapsto \hat{I}(\hat{r}) \sqcup (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e, \emptyset)\}).\end{array}$$

By definition of $\hat{\mathcal{R}}$,

$$\mathcal{R}\ d\ \texttt{ALLOC(}e_a\texttt{, [}e\texttt{])}_{\mathsf{t}}\ (M, c, I) \in \gamma_{Memory}\ (\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{ALLOC(}e_a\texttt{, [}e\texttt{])}_{\mathsf{t}}\ (\hat{M}, \hat{I})).$$

When $c = \texttt{ALLOC(}e_a\texttt{, \{}\vec{\mathsf{f}}\texttt{\})}_{\mathsf{t}}$, by definition of $\mathcal{R}$,

$$\begin{array}{l}\mathcal{R}\ d\ \texttt{ALLOC(}e_a\texttt{, \{}\vec{\mathsf{f}}\texttt{\})}_{\mathsf{t}}\ (M, c, I)\\ = (M\{\mathcal{A}\ d\ M\ e_a \mapsto (r, \mathsf{f}_1)\}, c + 1, I\{r \mapsto \vec{\mathsf{f}}\})\end{array}$$

where $r = (c, \mathsf{t})$. By definition of η_{Region},

$$\eta_{Region}\ r = \mathsf{t}.$$

By Corollary 1,

$$\mathcal{A}\ d\ M\ e_a \in \gamma_{Addr}\ (\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a)$$

and by definition of γ_{Val},

$$(r, \mathsf{f}_1) \in \gamma_{Val}\ \{(\hat{r}, \mathsf{f}_1)\}.$$

By Lemma 5,

$$M\{\mathcal{A}\ d\ M\ e_a \mapsto (r, \mathsf{f}_1)\} \in \gamma_{Map}\ \hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a \hat{\longmapsto} \{(\hat{r}, \mathsf{f}_1)\}\}.$$

By definition of γ_{Alloc},

$$I\{r \mapsto \vec{\mathsf{f}}\} \in \gamma_{Alloc}\ \hat{I}\{\hat{r} \mapsto \hat{I}(\hat{r}) \sqcup (\hat{\perp}_{\hat{\mathbb{Z}}}, \{\vec{\mathsf{f}}\})\}.$$

Therefore, we have

$$\begin{array}{l}(M\{\mathcal{A}\ d\ M\ e_a \mapsto (r, \mathsf{f}_1)\}, c + 1, I\{r \mapsto \vec{\mathsf{f}}\}) \in\\ \gamma_{Memory}\ (\hat{M}\{\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a \hat{\longmapsto} \{(\hat{r}, \mathsf{f}_1)\}\}, \hat{I}\{\hat{r} \mapsto \hat{I}(\hat{r}) \sqcup (\hat{\perp}_{\hat{\mathbb{Z}}}, \{\vec{\mathsf{f}}\})\}).\end{array}$$

By definition of $\hat{\mathcal{R}}$,

$$\mathcal{R}\ d\ \texttt{ALLOC}(e_a, \{\vec{\texttt{f}}\})_\texttt{t}\ (M, c, I) \in \gamma_{Memory}\ (\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{ALLOC}(e_a, \{\vec{\texttt{f}}\})_\texttt{t}\ (\hat{M}, \hat{I})).$$

When $c = \texttt{FREE}(e)$, by definition of $\mathcal{R}$,

$$\mathcal{R}\ d\ \texttt{FREE}(e)\ (M, c, I) = (M - \{a\}, c, I - \{a\})$$

where $a = \mathcal{A}\ d\ M\ e$. By definition of γ_{Map} and $M \in \gamma_{Map}\ \hat{M}$,

$$I - \{a\} \in \gamma_{Alloc}\ \hat{I}.$$

Similarly by definition of γ_{Alloc} and $I \in \gamma_{Alloc}\ \hat{I}$,

$$I - \{a\} \in \gamma_{Alloc}\ \hat{I}.$$

Therefore, we have

$$(M - \{a\}, c, I - \{a\}) \in \gamma_{Memory}\ (\hat{M}, \hat{I}).$$

By definition of $\hat{\mathcal{R}}$,

$$\mathcal{R}\ d\ \texttt{FREE}(e)\ (M, c, I) \in \gamma_{Memory}\ (\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ \texttt{FREE}(e)\ (\hat{M}, \hat{I})).$$

Thus, for any case of $c \in Cmd$, we have

$$\mathcal{R}\ d\ c\ (M, c, I) \in \gamma_{Memory}\ (\hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ c\ (\hat{M}, \hat{I})).$$

□

Lemma 3 *Pruning $\hat{\mathcal{P}}$ is sound, i.e.*

$$\begin{array}{ll} & d, m \models r, \\ & \mathsf{p} = \eta_{Dump}\ d, \\ & m \in \gamma_{Memory}\ \hat{m} \\ \Longrightarrow & m \in \gamma_{Memory}\ (\hat{\mathcal{P}}\ \mathsf{p}\ r\ \hat{m}) \end{array}$$

Proof First, we show a proposition of $\mathcal{S}_\diamond$.

$$\begin{array}{ll} & d, m \models e \diamond e' \\ \Longrightarrow & \hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e \sqcap \mathcal{S}_\diamond\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e') \neq \hat{\perp}_{Value} \end{array}$$

By definition of $\models$, the premise becomes

$$\mathcal{V}\ d\ m\ e \diamond \mathcal{V}\ d\ m\ e'.$$

By Lemma 6,

$$\mathcal{V}\ d\ m\ e \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e),$$
$$\mathcal{V}\ d\ m\ e' \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e').$$

Since γ_{Val} is multiplicative, we may show the following instead.

$$\gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e) \cap \gamma_{Val}\ (\mathcal{S}_\diamond\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e')) \neq \emptyset.$$

So, we will actually show the following for each case of $\diamond$.

$$\mathcal{V}\ d\ m\ e \in \gamma_{Val}\ (\mathcal{S}_\diamond\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e')).$$

When $\diamond = =$,

$$\begin{array}{rcl} \mathcal{V}\ d\ m\ e & = & \mathcal{V}\ d\ m\ e' \\ & & \text{(definition of } \models) \\ & \in & \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e') \\ & & \text{(assumption)} \\ & = & \gamma_{Val}\ (\mathcal{S}_{=}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e')) \\ & & \text{(definition of } \mathcal{S}) \end{array}$$

When $\diamond = !=$,

$$\begin{array}{rcl} \mathcal{V}\ d\ m\ e & \in & \gamma_{Val}\ \hat{\top}_{\hat{Value}} \\ & & \text{(assumption)} \\ & = & \gamma_{Val}\ (\mathcal{S}_{!=}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e')) \\ & & \text{(definition of } \mathcal{S}) \end{array}$$

When $\diamond = <$, by definition of $\models$,

$$\mathcal{V}\ d\ m\ e < \mathcal{V}\ d\ m\ e'.$$

If $n = \mathcal{V}\ d\ m\ e'$, $l \leq n \leq u$, then

$$\mathcal{V}\ d\ m\ e < n \leq u,$$

$$\mathcal{V}\ d\ m\ e \leq u - 1,$$

and by definition of $\mathcal{S}$,

$$\mathcal{V}\ d\ m\ e \in \gamma_{Val}\ (\mathcal{S}_{<}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e')).$$

Same argument applies when $(r, n) = \mathcal{V}\ d\ m\ e'$.
Other cases for $\diamond \in \{<=, >, >=\}$ are also justified by very similar arguments.

Next, we check relations defined by $\mathcal{N}$, and transpositioned relation are all equivalent with the original one.

$$d, m \models r^t \iff d, m \models r \iff \forall r' \in \mathcal{N}\ e\ r : d, m \models r'$$

The transposition r^t is obviously the equivalent relation to r, since it swaps the position of the two expressions, and changes the $\diamond$ to the corresponding inverted one.

$$\begin{array}{rcl} (e \diamond e')^t & = & e' \diamond^t e \text{ where} \\ & & \begin{array}{llllll} =^t & = & =, & !=^t & = & !=, \\ <^t & = & >, & >=^t & = & <=, \\ >^t & = & <, & <=^t & = & >= \end{array} \end{array}$$

Equivalence of relations defined by $\mathcal{N}\ e$ is shown by structural induction. The first base case is obvious, since it is the identical relation.

$$\mathcal{N}\ e\ \$e \diamond e' = \{\$e \diamond e'\}$$

The second base case holds vacuously.

$$\mathcal{N}\ e\ \_ = \emptyset$$

All other inductive cases hold because of the commutativity of arithmetic operators and equivalence of transposition of terms.

Now, we show the conclusion on $\hat{\mathcal{P}}$ using previous arguments.
When there is no $e \in \mathcal{L}\ r$ such that $\hat{V} \neq \hat{\bot}_{\hat{Value}}$, every consistent memory m with relation r is in $\hat{\mathcal{P}}\ \mathsf{p}\ r\ \hat{m}$ by our first argument on $\mathcal{S}$.
Otherwise, let us first recall the definition of $\bigsqcap$ that $\bigsqcap$ is the greatest one among the lower bounds.

$$\forall a \in A : x \sqsubseteq a \Longrightarrow x \sqsubseteq \bigsqcap A.$$

By arguments on $\mathcal{N}$ and $\mathcal{S}$, and Lemma 5, greatestness of $\bigsqcap$, we have the same conclusion. □

4.3 Memory manipulation

Lemma 4 *Lookup is sound, i.e.*

$$\begin{array}{ll} & a \in \gamma_{Addr}\ \hat{A}, \\ & M \in \gamma_{Map}\ \hat{M} \\ \Longrightarrow & M(a) \in \gamma_{Val}\ (\hat{M}\,?\,\hat{A}) \end{array}$$

Proof First, suppose $\gamma_{Addr}\ \hat{A} \not\subseteq \gamma_{Addr}\ \mathrm{dom}(\hat{M})$. There exists $a \in \gamma_{Addr}\ \hat{A}$ but $a \notin \gamma_{Addr}\ \mathrm{dom}(\hat{M})$. However, from $M \in \gamma_{Map}\ \hat{M}$ and definition of γ_{Map}, we have $\mathrm{dom}(M) \subseteq \gamma_{Addr}\ \mathrm{dom}(\hat{M})$. This means $a \notin \gamma_{Addr}\ \mathrm{dom}(\hat{M})$ cannot have mapping in M. Hence $M(a)$ is not defined, and the conclusion holds vacuously. So, we only need to consider the other possibility, $a \in \gamma_{Addr}\ \mathrm{dom}(\hat{M})$.

From here on, without loss of generality, let us assume $\gamma_{Addr}\ \hat{A} \subseteq \gamma_{Addr}\ \mathrm{dom}(\hat{M})$. By definition of γ_{Addr},

$$\forall a \in \gamma_{Addr}\ \hat{A} : \eta_{Addr}\ a \sqsubseteq \hat{a} \in \hat{A}, \eta_{Addr}\ a \sqsubseteq \hat{a}' \in \mathrm{dom}(\hat{M}).$$

Since $\eta_{Addr}\ a \neq \hat{\bot}_{A\hat{d}dr}$, and by greatestness of $\sqcap$, i.e. $a \sqsubseteq b, a \sqsubseteq c \Longrightarrow a \sqsubseteq b \sqcap c$, here we have,

$$\hat{\bot}_{A\hat{d}dr} \sqsubset \eta_{Addr}\ a \sqsubseteq \hat{a} \sqcap \hat{a}'.$$

Hence, $\hat{a} \sqcap \hat{a}' \neq \hat{\bot}_{A\hat{d}dr}$.

$$\begin{array}{rcl} M(a) & \in & \{v \mid a \mapsto v \in M, a \in \gamma_{Addr}\ \hat{A}\} \\ & & \quad (a \in \gamma_{Addr}\ \hat{A}) \\ & = & \{v \mid a \mapsto v \in M, \eta_{Addr}\ a \sqsubseteq \hat{a}, \hat{a} \in \hat{A}\} \\ & & \quad (\text{definition of } \gamma_{Addr}) \\ & \subseteq & \{v \mid a \mapsto v \in M, \eta_{Addr}\ a \sqsubseteq \hat{a}, \hat{a} \in \hat{A}, \\ & & \qquad \eta_{Addr}\ a \sqsubseteq \hat{a}', \hat{a}' \mapsto \hat{V}' \in \hat{M}, v \in \gamma_{Val}\ \hat{V}'\} \\ & & \quad (M \in \gamma_{Map}\ \hat{M}, \gamma_{Addr}\ \hat{A} \subseteq \gamma_{Addr}\ \mathrm{dom}(\hat{M})) \\ & \subseteq & \gamma_{Val}\ \bigsqcup\{\hat{V}' \mid a \mapsto v \in M, \eta_{Addr}\ a \sqsubseteq \hat{a}, \hat{a} \in \hat{A}, \\ & & \qquad \eta_{Addr}\ a \sqsubseteq \hat{a}', \hat{a}' \mapsto \hat{V}' \in \hat{M}, v \in \gamma_{Val}\ \hat{V}'\} \\ & & \quad (\gamma_{Val} \text{ is monotone}) \\ & \subseteq & \gamma_{Val}\ \bigsqcup\{\hat{V}' \mid \hat{a}' \mapsto \hat{V}' \in \hat{M}, \hat{a} \in \hat{A}, \hat{a} \sqcap \hat{a}' \neq \hat{\bot}_{A\hat{d}dr}\} \\ & & \quad (\text{previous argument}) \\ & = & \gamma_{Val}\ \bigsqcup\{\hat{M}(\hat{a}') \mid \hat{a}' \in overlap(\hat{a}, \mathrm{dom}(\hat{M})), \hat{a} \in \hat{A}\} \\ & & \quad (\text{definition of } overlap(\hat{a}, \mathrm{dom}(\hat{M}))) \\ & \subseteq & \gamma_{Val}\ (\hat{M}\,?\,\hat{A}) \\ & & \quad (\text{definition of lookup, } \gamma_{Val} \text{ is monotone}) \end{array}$$

Thus, lookup with set of addresses on abstract value mapping is sound. □

Lemma 5 *Update is sound, i.e.*

$$\begin{array}{ll} & a \in \gamma_{Addr}\ \hat{A}, \\ & v \in \gamma_{Val}\ \hat{V}, \\ & M \in \gamma_{Map}\ \hat{M} \\ \Longrightarrow & M\{a \mapsto v\} \in \gamma_{Map}\ \hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\} \end{array}$$

Proof By definition of update,

$$\begin{array}{rcl} \hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\} & = & \hat{M} \\ & \sqcup & \{\hat{a}' \mapsto \hat{V} \mid \hat{a}' \in overlap(\hat{a}, \mathrm{dom}(\hat{M})), \hat{a} \in \hat{A}\} \\ & \sqcup & \{\hat{a} \mapsto \hat{V} \mid \hat{a} \in \hat{A} - \hat{Region} \times \hat{Index}\} \\ & \sqcup & \{(\hat{r}, [-\infty, +\infty]) \mapsto \hat{V} \mid (\hat{r}, \_) \in \hat{A}\} \end{array}$$

In order to show that $M\{a \mapsto v\}$ is contained in $\gamma_{Map}\ \hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\}$, we check the following.

$$\begin{array}{l}\forall a' \mapsto v' \in M\{a \mapsto v\}, \hat{a}' \mapsto \hat{V}' \in \hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\} : \\ \quad \eta_{Addr}\ a' \sqsubseteq \hat{a}' \Longrightarrow v' \in \gamma_{Val}\ \hat{V}', \\ \quad \exists \hat{a}'' \in \mathrm{dom}(\hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\}) : \eta_{Addr}\ a' \sqsubseteq \hat{a}''\end{array}$$

Let $a' \mapsto v' \in M\{a \mapsto v\}$ and $\hat{a}' \mapsto \hat{V}' \in \hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\}$.

We will first check whether $\eta_{Addr}\ a' \sqsubseteq \hat{a}'$ implies $v' \in \gamma_{Val}\ \hat{V}'$. So, let us assume $\eta_{Addr}\ a' \sqsubseteq \hat{a}'$.

- If $a' \neq a$, then $a' \mapsto v' \in M$. Because $M \in \gamma_{Map}\ \hat{M} \subseteq \gamma_{Map}\ \hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\}$, we have $v' \in \gamma_{Val}\ \hat{V}'$.
- Otherwise, $a' = a$ and $v' = v$.
  - Suppose $\hat{a}' \in \hat{A}$.
    - When $a \in Region \times Index$, $\hat{a}' = (\hat{r}, \_)$ for some $\hat{r} \in \hat{Region}$. By $(\hat{r}, [-\infty, +\infty]) \mapsto \hat{V}$, we have $v' = v \in \gamma_{Val}\ \hat{V} \subseteq \gamma_{Val}\ \hat{V}'$.
    - Otherwise, by $\hat{a}' \mapsto \hat{V}$, $\hat{V} \sqsubseteq \hat{V}'$. Since γ_{Val} is monotone, $v' = v \in \gamma_{Val}\ \hat{V} \subseteq \gamma_{Val}\ \hat{V}'$.
  - Suppose $\hat{a}' \notin \hat{A}$. Since $a \in \gamma_{Addr}\ \hat{A}$, for some $\hat{a} \in \hat{A}$, $\eta_{Addr}\ a \sqsubseteq \hat{a}$. From $\eta_{Addr}\ a \sqsubseteq \hat{a}'$ and $\eta_{Addr}\ a \neq \hat{\bot}_{A\hat{d}dr}$, we have $\hat{a} \sqcap \hat{a}' \neq \hat{\bot}_{A\hat{d}dr}$. Therefore, $\hat{a}' \in overlap(\hat{a}, \mathrm{dom}(\hat{M}))$. By $\hat{a}' \mapsto \hat{V}$, $\hat{V} \sqsubseteq \hat{V}'$, Since γ_{Val} is monotone, we have $v' = v \in \gamma_{Val}\ \hat{V} \subseteq \gamma_{Val}\ \hat{V}'$.

Now, let us check whether there exists $\hat{a}' \in \mathrm{dom}(\hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\})$ such that $\eta_{Addr}\ a' \sqsubseteq \hat{a}'$.

- If $a' \neq a$, then $a' \mapsto v' \in M$. Because $M \in \gamma_{Map}\ \hat{M} \subseteq \gamma_{Map}\ \hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\}$, there exists $\hat{a}' \in \mathrm{dom}(\hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\})$ such that $\eta_{Addr}\ a' \sqsubseteq \hat{a}'$.
- Otherwise, $a' = a$. Since $a \in \gamma_{Addr}\ \hat{A}$, there exists $\hat{a} \in \hat{A}$ such that $\eta_{Addr}\ a' \sqsubseteq \hat{a}$.
  - When $a \in Region \times Index$, $\hat{a} = (\hat{r}, \_) \in \hat{A}$ for some $\hat{r} \in \hat{Region}$. Hence, there exists $(\hat{r}, [-\infty, +\infty]) \in \mathrm{dom}(\hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\})$ such that $\eta_{Addr}\ a' \sqsubseteq (\hat{r}, [-\infty, +\infty])$.
  - Otherwise, there exists $\hat{a} \in \mathrm{dom}(\hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\})$ such that $\eta_{Addr}\ a \sqsubseteq \hat{a}$.

Thus, we have the conclusion,

$$M\{a \mapsto v\} \in \gamma_{Map}\ \hat{M}\{\hat{A} \hat{\mapsto} \hat{V}\}.$$

□

4.4 Evaluation

Lemma 6 *Value evaluation $\hat{\mathcal{V}}$ is sound, i.e.*

$$\begin{array}{ll} & \eta_{Dump}\ d = \mathsf{p}, \\ & m \in \gamma_{Memory}\ \hat{m} \\ \Longrightarrow & \forall e \in Expr : \mathcal{V}\ d\ m\ e \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{m}\ e)\end{array}$$

Proof $\mathcal{V}\ d\ (M, \_, \_) = \mathcal{V}\ d\ M$ and $\hat{\mathcal{V}}\ \mathsf{p}\ (\hat{M}, \_) = \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}$. Therefore, let us show $\mathcal{V}\ d\ M\ e \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e)$ when $M \in \gamma_{Map}\ \hat{M}$ instead. This is proved by structural induction on e.

When $e = n$,

$$\mathcal{V}\ d\ M\ n = n \in \gamma_{Val}\ \{[n, n]\} = \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ n).$$

When $e = \mathsf{X}$,

$$\mathcal{V}\ d\ M\ \mathsf{X} = \mathsf{X} \in \gamma_{Val}\ \{\mathsf{X}\} = \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ \mathsf{X}).$$

When $e = \mathsf{x}$, since $\eta_{Dump}\ d = \mathsf{p}$,

$$\mathcal{V}\ d\ M\ \mathsf{x} = (d, \mathsf{x}) \in \gamma_{Val}\ \{(\mathsf{p}, \mathsf{x})\} = \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ \mathsf{x}).$$

When $e = e_1 \star e_2$, assuming $\mathcal{V}\ d\ M\ e_i \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e_i)$ for $i \in \{1, 2\}$,

$$\begin{array}{rcl}
\mathcal{V}\ d\ M\ e_1 \star e_2 & = & \mathcal{V}\ d\ M\ e_1 \star \mathcal{V}\ d\ M\ e_2 \\
& & \quad \text{(definition of } \mathcal{V}\text{)} \\
& \in & \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e_1\ \hat{\star}\ \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e_2) \\
& & \quad \text{(induction hypothesis and Lemma 7)} \\
& = & \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e_1 \star e_2) \\
& & \quad \text{(definition of } \hat{\mathcal{V}}\text{)}.
\end{array}$$

When $e = e'.\mathsf{f}$, assuming $\mathcal{V}\ d\ M\ e' \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e')$,

$$\begin{array}{rcl}
\mathcal{V}\ d\ M\ e'.\mathsf{f} & = & (r, \mathsf{f}) \text{ where } (r, _) = \mathcal{A}\ d\ M\ e' \\
& & \quad \text{(definition of } \mathcal{V}\text{)} \\
& = & (r, \mathsf{f}) \text{ where } (r, _) = \mathcal{V}\ d\ M\ e' \\
& & \quad \text{(definition of } \mathcal{A}\text{)} \\
& \in & \{(r, \mathsf{f}) \mid (r, _) \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e')\} \\
& & \quad \text{(induction hypothesis)} \\
& = & \gamma_{Val}\ \{(\hat{r}, \mathsf{f}) \mid (\hat{r}, _) \in \hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e'\} \\
& & \quad \text{(definition of } \gamma_{Val}, \gamma_{Addr}\text{)} \\
& = & \gamma_{Val}\ \{(\hat{r}, \mathsf{f}) \mid (\hat{r}, _) \in \hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e'\} \\
& & \quad \text{(definition of } \hat{\mathcal{A}}\text{)} \\
& = & \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e'.\mathsf{f}) \\
& & \quad \text{(definition of } \hat{\mathcal{V}}\text{)}
\end{array}$$

When $e = \$e'$, assuming $\mathcal{V}\ d\ M\ e' \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e')$,

$$\begin{array}{rcl}
\mathcal{V}\ d\ M\ \$e' & = & M(\mathcal{A}\ d\ M\ e') \\
& & \quad \text{(definition of } \mathcal{V}\text{)} \\
& = & M(\mathcal{V}\ d\ M\ e') \\
& & \quad \text{(definition of } \mathcal{A}\text{)} \\
& \in & \{M(a) \mid a \in \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e')\} \\
& & \quad \text{(induction hypothesis)} \\
& = & \{M(a) \mid a \in \gamma_{Addr}\ (\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e')\} \\
& & \quad (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ e' = \hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e') \\
& \subseteq & \gamma_{Val}\ (\hat{M}\,?\,\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e') \\
& & \quad \text{(Lemma 4)} \\
& = & \gamma_{Val}\ (\hat{\mathcal{V}}\ \mathsf{p}\ \hat{M}\ \$e') \\
& & \quad \text{(definition of } \hat{\mathcal{V}}\text{)}
\end{array}$$

Thus, by structural induction, we have shown $\hat{\mathcal{V}}$ is sound. □

Corollary 1 *Address evaluation $\hat{\mathcal{A}}$ is sound, i.e.*

$$\begin{array}{ll}
& \eta_{Dump}\ d = \mathsf{p}, \\
& m \in \gamma_{Memory}\ \hat{m} \\
\Longrightarrow & \forall e \in Expr : \mathcal{A}\ d\ m\ e \in \gamma_{Addr}\ (\hat{\mathcal{A}}\ \mathsf{p}\ \hat{m}\ e)
\end{array}$$

Proof Since definition of both $\mathcal{A}$, $\hat{\mathcal{A}}$ are restrictions of $\mathcal{V}$, $\hat{\mathcal{V}}$ respectively, soundness of $\hat{\mathcal{A}}$ follows from Lemma 6. □

Lemma 7 *Binary operations +, -, *, / are sound, i.e.*

$$\begin{array}{ll}
& \star \in \{+, -, *, /\} \\
& v \in \gamma_{Val}\ \hat{V}, v' \in \gamma_{Val}\ \hat{V}' \\
\Longrightarrow & v \star v' \in \gamma_{Val}\ (\hat{V}\ \hat{\star}\ \hat{V}')
\end{array}$$

Proof Let us consider each binary operator separately.

Addition $\hat{+}$ on intervals is sound since

$$\begin{array}{rcl}\{x+y \mid x \in \gamma_{\mathbb{Z}}\,[a,b], y \in \gamma_{\mathbb{Z}}\,[c,d]\} & = & \{x+y \mid a \le x \le b, c \le y \le d\} \\ & = & \{x+y \mid a+c \le x+y \le b+d\} \\ & = & \{z \mid a+c \le z \le b+d\} \\ & = & \gamma_{\mathbb{Z}}\,[a+c, b+d] \\ & = & \gamma_{\mathbb{Z}}\,([a,b] \hat{+} [c,d]).\end{array}$$

$\hat{+}$ as a shift operator on indexed address is also sound since

$$\begin{array}{ll} & (r,i) \in \gamma_{Val}\,\hat{V}, \quad n \in \gamma_{Val}\,\hat{V}' \\ \Longrightarrow & (\hat{r},\hat{i}) \in \hat{V} \text{ where } \eta_{Region}\, r = \hat{r}, i \in \gamma_{\mathbb{Z}}\,\hat{i}, \\ & \hat{n} \in \hat{V}' \text{ where } n \in \gamma_{\mathbb{Z}}\,\hat{n} \\ \Longrightarrow & i+n \in \gamma_{\mathbb{Z}}\,(\hat{i} \hat{+} \hat{n}) \\ \Longrightarrow & (r,i)+n = (r, i+n) \in \gamma_{Addr}\,(\hat{r}, \hat{i} \hat{+} \hat{n}) \\ \Longrightarrow & (r,i)+n \in \gamma_{Val}\,(\hat{V} \hat{+} \hat{V}') \text{ since } (\hat{r}, \hat{i} \hat{+} \hat{n}) \in \hat{V} \hat{+} \hat{V}' \text{ by definition.}\end{array}$$

Subtraction $\hat{-}$ on intervals is sound since

$$\begin{array}{rcl}\{x-y \mid x \in \gamma_{\mathbb{Z}}\,[a,b], y \in \gamma_{\mathbb{Z}}\,[c,d]\} & = & \{x-y \mid a \le x \le b, c \le y \le d\} \\ & = & \{x-y \mid a-d \le x-y \le b-c\} \\ & = & \{z \mid a-d \le z \le b-c\} \\ & = & \gamma_{\mathbb{Z}}\,[a-d, b-c] \\ & = & \gamma_{\mathbb{Z}}\,([a,b] \hat{-} [c,d]).\end{array}$$

$\hat{-}$ as a distance operator on indexed addresses is also sound since

$$\begin{array}{ll} & (r,i) \in \gamma_{Val}\,\hat{V}, \quad (r,i') \in \gamma_{Val}\,\hat{V}' \\ \Longrightarrow & (\hat{r},\hat{i}) \in \hat{V} \text{ where } \eta_{Region}\, r = \hat{r}, i \in \gamma_{\mathbb{Z}}\,\hat{i}, \\ & (\hat{r},\hat{i}') \in \hat{V}' \text{ where } i' \in \gamma_{\mathbb{Z}}\,\hat{i}' \\ \Longrightarrow & (r,i)-(r,i') = i-i' \in \gamma_{\mathbb{Z}}\,(\hat{i} \hat{-} \hat{i}') \\ \Longrightarrow & (r,i)-(r,i') \in \gamma_{Val}\,(\hat{V} \hat{-} \hat{V}') \text{ since } \hat{i} \hat{-} \hat{i}' \in \hat{V} \hat{-} \hat{V}' \text{ by definition.}\end{array}$$

Multiplication $\hat{*}$ on intervals is sound since

$$\begin{array}{rcl}\{x*y \mid x \in \gamma_{\mathbb{Z}}\,[a,b], y \in \gamma_{\mathbb{Z}}\,[c,d]\} & = & \{x \times y \mid a \le x \le b, c \le y \le d\} \\ & = & \{x \times y \mid \min X \le x \times y \le \max X\} \\ & = & \{z \mid \min X \le z \le \max X\} \\ & = & \gamma_{\mathbb{Z}}\,[\min X, \max X] \\ & = & \gamma_{\mathbb{Z}}\,([a,b] \hat{*} [c,d])\end{array}$$

where $X = \{a \times c, a \times d, b \times c, b \times d\}$.

Division $\hat{/}$ on intervals is sound because of the following two arguments. When $c \le 0 \le d$,

$$\begin{array}{rcl}\{x/y \mid x \in \gamma_{\mathbb{Z}}\,[a,b], y \in \gamma_{\mathbb{Z}}\,[c,d]\} & = & \{x \div y \mid a \le x \le b, c \le y \le d\} \\ & \subseteq & \mathbb{Z} \\ & = & \gamma_{\mathbb{Z}}\,[-\infty, +\infty] \\ & = & \gamma_{\mathbb{Z}}\,([a,b] \hat{/} [c,d])\end{array}$$

Otherwise when $c \le d < 0$ or $0 < c \le d$,

$$\begin{array}{rcl}\{x/y \mid x \in \gamma_{\mathbb{Z}}\,[a,b], y \in \gamma_{\mathbb{Z}}\,[c,d]\} & = & \{x \div y \mid a \le x \le b, c \le y \le d\} \\ & = & \{x \div y \mid \min X \le x \div y \le \max X\} \\ & = & \{z \mid \min X \le z \le \max X\} \\ & = & \gamma_{\mathbb{Z}}\,[\min X, \max X] \\ & = & \gamma_{\mathbb{Z}}\,([a,b] \hat{/} [c,d])\end{array}$$

where $X = \{a \div c, a \div d, b \div c, b \div d\}$. □

$$
\begin{array}{rcl}
W & \in & \mathit{Worklist} \\
b, b' & \in & \mathit{Block}_P \\
\mathit{old}, \mathit{new} & \in & \hat{\mathit{Memory}} \\
\hat{G} & \in & \hat{\mathit{Graph}} \\
\hat{T}_{in}, \hat{T}_{out} & \in & \hat{\mathit{Table}} \\
\hat{D} & \in & \hat{\mathit{Dump}} \\
\hat{\mathcal{F}}_b & : & \hat{\mathit{Graph}} \times \hat{\mathit{Memory}} \times \hat{\mathit{Dump}} \to \hat{\mathit{Graph}} \times \hat{\mathit{Memory}} \times \hat{\mathit{Dump}}
\end{array}
$$

input$(W, (\hat{G}, \hat{T}_{in}, \hat{D}))$
$\hat{T}_{out} := \hat{\bot}_{\hat{\mathit{Table}}}$
until $(\mathit{isempty}(W))$:
  $(b, W) := \mathit{choose}(W)$; $\mathit{old} := \hat{T}_{out}(b)$
  $(\hat{G}, \mathit{new}, \hat{D}) := \hat{\mathcal{F}}_b\ (\hat{G}, \hat{T}_{in}(b), \hat{D})$
  if $(\mathit{isloophead}(\hat{G}, b))$ $\mathit{new} := \mathit{old} \triangledown \mathit{new}$
  $\hat{T}_{out}(b) := \mathit{new}$
  foreach $(b, b') \in \hat{G}$:
    if $(\mathit{new} \not\sqsubseteq \hat{T}_{in}(b'))$ $\hat{T}_{in}(b') := \mathit{new}$; $W := \mathit{add}(W, b')$
output$(\hat{G}, \hat{T}_{in}, \hat{D})$

Figure 1: Worklist algorithm

5 Implementation

5.1 Worklist algorithm

We use the worklist algorithm shown in Figure 1 to reduce the amount of computations for finding an approximation of the least fixpoint. This algorithm can be understood as a block-scopic, incremental version of $\hat{\mathcal{F}}$ in the definition of the abstract semantics. This algorithm keeps track of state of each block and only computes the one that changes. $\hat{\mathcal{F}}_b$ is the semantic transfer function that computes the effect of a single block b. $\hat{T}_{in}$ and $\hat{T}_{out}$ records the input and output state of each block respectively. $\hat{T}_{in}$ is the $\hat{T}$ we have seen in the previous definitions. After the widening iterations, narrowing iterations are done by replacing $\not\sqsupseteq$ for $\not\sqsubseteq$ and $\triangle$ for $\triangledown$.

5.1.1 Wait at join

A worklist managing strategy called *wait at join* can be used to reduce the amount of computation in many cases. When one manages the worklist in a LIFO(last in first out) manner, the computation is done depth first order. Computations for blocks following a join point are repeated as many as the number of branches in worst cases. The heuristic of this strategy is to wait at such join points rather than continuing computation. The computation suspended on join points and their descendants are resumed after no other place needs to be computed.

To implement this, two sub-worklists are used: *active* A and *waiting* W. Both are managed in a LIFO manner.

$$(A, W) \quad \in \quad \mathit{Worklist} \quad = \quad \wp(\mathit{Block}_P) \times \wp(\mathit{Block}_P)$$

When choosing blocks, active has priority over waiting.

$$\mathit{choose} \quad : \quad \mathit{Worklist} \to \mathit{Block}_P \times \mathit{Worklist}$$

$$
\begin{array}{rcll}
\mathit{choose}\ (A, W) & = & (b, (A - \{b\}, W)) & \text{if } b \in A \\
\mathit{choose}\ (\emptyset, W) & = & (b, (\emptyset, W - \{b\})) & \text{if } b \in W
\end{array}
$$

Join blocks which have more than two predecessors are always added to waiting, and others are added to active.

$$\begin{array}{rcll} add & : & Worklist \times Block_P \to Worklist & \\ add\ ((A,W),b) & = & (A, W \cup \{b\}) & \text{if } (b',b),(b'',b) \in \hat{G}, b' \neq b'' \\ & = & (A \cup \{b\}, W) & \text{otherwise} \end{array}$$

5.2 Widening/Narrowing

Widening is used for finishing the fixpoint computation in finite steps, and narrowing for recovering some accuracy lost by widening. For intervals, we use the same widening and narrowing defined in [9]. The widening extrapolates unstable bounds to zero or infinity, and the narrowing improves those bounds.

$$\begin{array}{rcll} & & \nabla & : \quad \hat{\mathbb{Z}} \times \hat{\mathbb{Z}} \to \hat{\mathbb{Z}} \\ \bot & \nabla & \hat{n} & = \quad \hat{n} \\ \hat{n} & \nabla & \bot & = \quad \hat{n} \\ [n,m] & \nabla & [n',m'] & = \quad [0 \leq n' < n\ ?\ 0 : n' < n\ ?\ -\infty : n, \\ & & & \quad\ m < m' \leq 0\ ?\ 0 : m < m'\ ?\ +\infty : m] \end{array}$$

$$\begin{array}{rcll} & & \triangle & : \quad \hat{\mathbb{Z}} \times \hat{\mathbb{Z}} \to \hat{\mathbb{Z}} \\ \bot & \triangle & \hat{n} & = \quad \bot \\ \hat{n} & \triangle & \bot & = \quad \bot \\ [n,m] & \triangle & [n',m'] & = \quad [n \leq 0 \leq n'\ ?\ n' : n = -\infty\ ?\ n' : n, \\ & & & \quad\ m' \leq 0 \leq m\ ?\ m' : m = +\infty\ ?\ m' : m] \end{array}$$

Widening and narrowing on other domains are natural extensions of those on intervals. Widenings on finite maps are defined pointwisely. Widening between two indexed addresses $\hat{Region} \times \hat{Index}$ of same region applies interval widening to the index part. Widening on set of abstract values are defined as applying widening only between interval elements and between indexed addresses. Other values, e.g. variables, field addresses are simply joined since only finite number of them are generated for a single program.

5.3 Localizing operators

5.3.1 Motivation

Every abstract memory of each block in our abstract interpretation can contain entries of local variables of other procedures. This not only affects the accuracy of join and widening operators but also increases the cost of those operators. If we assume the number of variables is proportional to the size of the program, the operators on abstract memories have time complexity quadratic to the program size.

Reducing the cost of operations on abstract memory is the key to improving the analysis speed. Each command and block of G form can only change a few number of entries in the memory, so one can expect the operators on abstract memory to have time complexity linear to such small number. Since operations on abstract memory are used multiple times in every step of fixpoint iteration of the worklist algorithm, reducing their cost will speed up each step and make the entire analysis faster.

We want the operators to perform the actual computation only for the locally changed entries. We will call this modification *localizing* the operators. Let us consider a typical case of join after branches that looks like Figure 2. At the join point, $\hat{M}\Delta_1 \sqcup \hat{M}\Delta_2$ must be computed. Instead of scanning all the entries of $\hat{M}\Delta_1$ and $\hat{M}\Delta_2$, we want the operators to compute the join only for Δ_1 and Δ_2, and skip the common parts of $\hat{M}$. A similar idea to this approach was briefly mentioned in the report of ASTRÉE[3, §6.2].

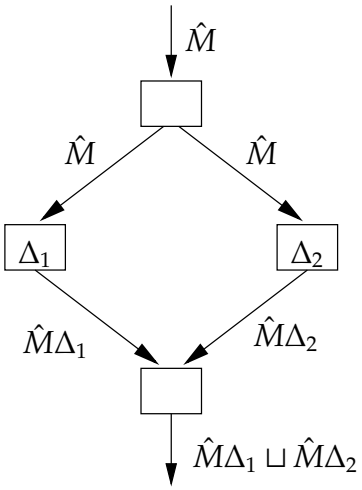

Figure 2: Typical case of join after branches.

5.3.2 Observation

Localizing the operators are possible, when one uses idempotent join, widening, narrowing operators on values and pointwise extensions for maps. Our operators $\star \in \{\sqcup, \triangledown, \triangle\}$ used in the abstract interpretation suffices such conditions.

$$\begin{array}{rcl} x \star x & = & x \\ (\hat{M} \star \hat{M}')(x) & = & \hat{M}(x) \star \hat{M}'(x) \\ (\hat{I} \star \hat{I}')(x) & = & \hat{I}(x) \star \hat{I}'(x) \end{array}$$

An operator $\star : Y \times Y \to Y$ is *idempotent* if and only if $\forall y \in Y : y \star y = y$. Operator $\star : (X \xrightarrow{\text{fin}} Y) \times (X \xrightarrow{\text{fin}} Y) \to (X \xrightarrow{\text{fin}} Y)$ is *pointwisely defined* if and only if

$$(A \star B)(x) = A(x) \star B(x)$$

where $A, B : X \xrightarrow{\text{fin}} Y$ and $\star$ is also defined on Y.

Theorem 2 *Idempotent operators extended to finite maps are localizable if defined pointwisely.*

$\star : Y \times Y \to Y$ *is idempotent*,
$\star : (X \xrightarrow{fin} Y) \times (X \xrightarrow{fin} Y) \to (X \xrightarrow{fin} Y)$ *is pointwisely defined*,
$A, B, C : X \xrightarrow{fin} Y$,
$\text{dom}(A) \cap \text{dom}(B) = \emptyset, \quad \text{dom}(A) \cap \text{dom}(C) = \emptyset$
$\Longrightarrow \quad (AB) \star (AC) = A(B \star C)$

Proof For all $x \in X$, if $x \in \text{dom}(A)$, then

$$\begin{array}{rcll} ((AB) \star (AC))(x) & = & (AB)(x) \star (AC)(x) & \text{(pointwise definition of } \star) \\ & = & A(x) \star A(x) & \text{(domain are disjoint)} \\ & = & A(x) & \text{(idempotence of } \star). \end{array}$$

Otherwise,

$$\begin{array}{rcll} ((AB) \star (AC))(x) & = & (AB)(x) \star (AC)(x) & \text{(pointwise definition of } \star) \\ & = & B(x) \star C(x) & \text{(domain are disjoint)} \\ & = & (B \star C)(x) & \text{(pointwise definition of } \star). \end{array}$$

Hence, we have

$$\begin{array}{rcll} ((AB) \star (AC))(x) & = & A(x) & \text{if } x \in \text{dom}(A) \\ & = & (B \star C)(x) & \text{otherwise} \end{array}$$

whose right hand side can be rewritten as

$$((B \star C)A)(x).$$

Since domains are disjoint, it is equal to

$$(A(B \star C))(x).$$

Thus, we have shown

$$(AB) \star (AC) = A(B \star C)$$

□

5.3.3 Journaling memory

In order to keep track of changes, we attach a journal to the abstract memory. Let us call this *journaling memory*. A journaling memory $\hat{J} \in \mathit{JMem}$ is either 1) $\Diamond\ \hat{M}$ with the empty journal $\Diamond$, or 2) $\langle \hat{J} \rangle_\delta\ \hat{M}$ with a journal $\langle \hat{J} \rangle_\delta$ that means $\hat{M}$ was derived by modifying the entries of δ from $\hat{J}$.

$$\begin{array}{lcllcl} \hat{J} & \in & \mathit{JMem} & j & \in & \mathit{Jnl} \\ \hat{J} & ::= & j\ \hat{M} & j & ::= & \Diamond \mid \langle \hat{J} \rangle_\delta \end{array}$$

where $\hat{M} \in \hat{\mathit{Map}}$ and $\delta \subseteq \hat{\mathit{Addr}}$.

Proxy operators The lookup and update operations on *JMem* are exactly the same ones on the outermost $\hat{\mathit{Map}}$ part. The only difference is that update operator adds the address to the outermost journal if it exists. Since these operators are the only ones used in the semantics definition of c and e, and they remain exactly the same as before, none of the semantics part needs to be modified.

$$\begin{array}{rcl} \cdot(\cdot) & : & \mathit{JMem} \to \hat{\mathit{Addr}} \to \hat{\mathit{Val}} \\ (j\ \hat{M})(a) & = & \hat{M}(a) \end{array}$$

$$\begin{array}{rcl} \cdot\{\cdot \mapsto \cdot\} & : & \mathit{JMem} \to \hat{\mathit{Addr}} \to \hat{\mathit{Val}} \to \mathit{JMem} \\ (\Diamond\ \hat{M})\{a \mapsto v\} & = & \Diamond\ \hat{M}\{a \mapsto v\} \\ (\langle \hat{J} \rangle_\delta\ \hat{M})\{a \mapsto v\} & = & \langle \hat{J} \rangle_{\delta \cup \{a\}}\ \hat{M}\{a \mapsto v\} \end{array}$$

Controlling depth of journal Operators *wrap* and *peel* controls the level of journals. *wrap* records the current journaling memory by wrapping it with a new journal,

$$\begin{array}{rcl} \mathit{wrap} & : & \mathit{JMem} \to \mathit{JMem} \\ \mathit{wrap}(\hat{J}) & = & \langle \hat{J} \rangle_\emptyset\ \hat{M} \text{ where } j\ \hat{M} = \hat{J} \end{array}$$

and *peel* peels off the outermost journal and merges the changes into the journal inside it.

$$\begin{array}{rcl} \mathit{peel} & : & \mathit{JMem} \to \mathit{JMem} \\ \mathit{peel}(\Diamond\ \hat{M}) & = & \Diamond\ \hat{M} \\ \mathit{peel}(\langle \Diamond\ \hat{M}' \rangle_\delta\ \hat{M}) & = & \Diamond\ \hat{M} \\ \mathit{peel}(\langle \langle \hat{J} \rangle_{\delta'}\ \hat{M}' \rangle_\delta\ \hat{M}) & = & \langle \hat{J} \rangle_{\delta' \cup \delta}\ \hat{M} \end{array}$$

Both do not touch any entries in the outermost $\hat{\mathit{Map}}$ part.

Depth of journals The number of nested journals n of $\hat{J} \in \mathit{JMem}$ are denoted $\hat{J}^n$. It is defined inductively as following.

$$\frac{}{(\Diamond\ \hat{M})^0} \qquad \frac{\hat{J}^n}{(\langle \hat{J} \rangle_\delta\ \hat{M})^{n+1}}$$

Difference and common journal Difference and common part between two *JMem* needs to be known to localize the operators. $\ominus$ gathers the addresses of modified entries in two maps.

$$\ominus \quad : \quad JMem \to JMem \to \wp(\hat{Addr})$$

$$\begin{array}{rcll}
(\Diamond\ \hat{M}_1) & \ominus & (\Diamond\ \hat{M}_2) & = \mathrm{dom}(\hat{M}_1) \cup \mathrm{dom}(\hat{M}_2) \\
(\langle \hat{J}_1 \rangle_{\delta_1}\ \hat{M}_1) & \ominus & (\Diamond\ \hat{M}_2) & = \mathrm{dom}(\hat{M}_1) \cup \mathrm{dom}(\hat{M}_2) \\
(\Diamond\ \hat{M}_1) & \ominus & (\langle \hat{J}_2 \rangle_{\delta_2}\ \hat{M}_2) & = \mathrm{dom}(\hat{M}_1) \cup \mathrm{dom}(\hat{M}_2)
\end{array}$$

$$\begin{array}{rcll}
(\langle \hat{J} \rangle_{\delta_1}\ \hat{M}_1) & \ominus & (\langle \hat{J} \rangle_{\delta_2}\ \hat{M}_2) & = \delta_1 \cup \delta_2 \\
(\langle \hat{J} \rangle_{\delta_1}\ \hat{M}_1) & \ominus & (\hat{J}) & = \delta_1 \\
(\hat{J}) & \ominus & (\langle \hat{J} \rangle_{\delta_2}\ \hat{M}_2) & = \delta_2
\end{array}$$

$$(\hat{J}_1^{n_1}) \ \ominus \ (\hat{J}_2^{n_2}) \quad = \quad \begin{cases} (peel(\hat{J}_1)) \ominus (\hat{J}_2) & \text{if } n_1 > n_2 \\ (\hat{J}_1) \ominus (peel(\hat{J}_2)) & \text{otherwise} \end{cases}$$

$\odot$ computes the common journal that can be used after merging two maps.

$$\odot \quad : \quad JMem \to JMem \to Jnl$$

$$\begin{array}{rcll}
(\Diamond\ \hat{M}_1) & \odot & (\Diamond\ \hat{M}_2) & = \Diamond \\
(\langle \hat{J}_1 \rangle_{\delta_1}\ \hat{M}_1) & \odot & (\Diamond\ \hat{M}_2) & = \Diamond \\
(\Diamond\ \hat{M}_1) & \odot & (\langle \hat{J}_2 \rangle_{\delta_2}\ \hat{M}_2) & = \Diamond
\end{array}$$

$$\begin{array}{rcll}
(\langle \hat{J} \rangle_{\delta_1}\ \hat{M}_1) & \odot & (\langle \hat{J} \rangle_{\delta_2}\ \hat{M}_2) & = \langle \hat{J} \rangle_{\emptyset} \\
(\langle \hat{J} \rangle_{\delta_1}\ \hat{M}_1) & \odot & (\hat{J}) & = \langle \hat{J} \rangle_{\emptyset} \\
(\hat{J}) & \odot & (\langle \hat{J} \rangle_{\delta_2}\ \hat{M}_2) & = \langle \hat{J} \rangle_{\emptyset}
\end{array}$$

$$(\hat{J}_1^{n_1}) \ \odot \ (\hat{J}_2^{n_2}) \quad = \quad \begin{cases} (peel(\hat{J}_1)) \odot (\hat{J}_2) & \text{if } n_1 > n_2 \\ (\hat{J}_1) \odot (peel(\hat{J}_2)) & \text{otherwise} \end{cases}$$

Selective operators Finally, let us define the selective operators $\star \in \{\sqcup, \triangledown, \triangle\}$ as following.

$$\star \quad : \quad JMem \to JMem \to JMem$$

$$(j_1\ \hat{M}_1) \star (j_2\ \hat{M}_2) \quad = \quad j\ (\hat{M}_1 \star_\delta \hat{M}_2)$$

$$\begin{array}{rcl}
\text{where} \quad \delta & = & (j_1\ \hat{M}_1) \ominus (j_2\ \hat{M}_2) \\
\text{and} \quad j & = & \begin{cases} \langle \hat{J} \rangle_{\delta' \cup \delta} & \text{if } \langle \hat{J} \rangle_{\delta'} = (j_1\ \hat{M}_1) \odot (j_2\ \hat{M}_2) \\ \Diamond & \text{otherwise} \end{cases}
\end{array}$$

$\star_\delta$ is the corresponding operator on $\hat{Map}$ that does the actual computation only for the entries of δ.

5.3.4 Localizing operators

Journaling rules If we apply *wrap* appropriately, the selective operators can operate only on local changes. The heuristic is to apply *wrap* on maps that might join later with slight differences, so the unmodified common part can be skipped later. Two following *wrap*ing rules handle this.

1. *wrap* output of nodes that have more than two successors.
2. *wrap* output of loop heads.

The first rule is because the map propagated to branches will probably join later. The second is because output of the loop head will eventually join at itself. These *wrap*ing can be done at the worklist algorithm after computing the effect of a single block with $\hat{\mathcal{F}}_b$. This is the only modification needs to be made for the analyzer.

Examples Let us look at some examples. Figure 3 shows the branch example with localized operators. Comparing it to the case with normal operators in Figure 2, one can see the cost of join has been reduced to the number of local changes in Δ_1 or Δ_2. Figure 4 shows a loop without localized operations. Figure 5 illustrates each join cost after an iteration of the loop body is reduced to size of local changes. Although it is not shown in the figure, the cost of widening operation also applied on output of each loop head is also reduced. Section 6.3.2 shows the experiment result measuring the speed up of this technique in the array index range analyzer, Airac5.

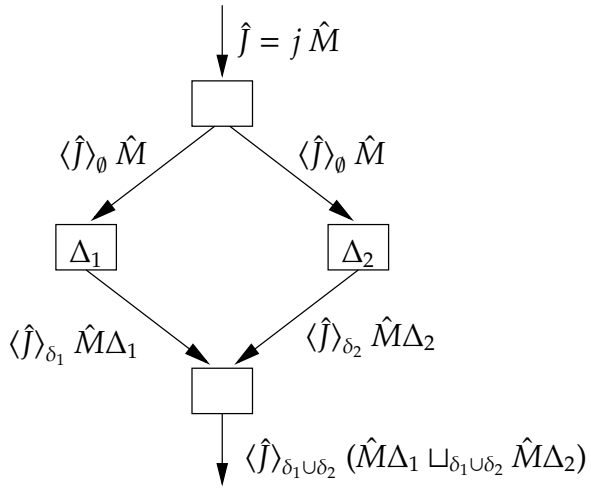

Figure 3: The branch with localized join

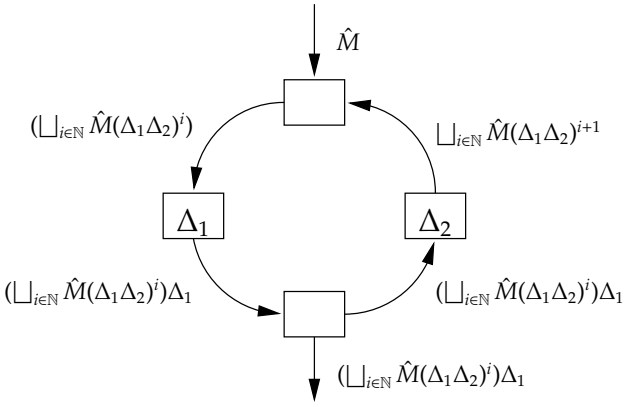

Figure 4: A loop without localized join

Another view One may achieve the same goal by collecting changes made in the path from the immediate dominator[1, §10.4] to the join point. However, traditional methods for computing dominators is not trivial to extend to work efficiently for interprocedural graphs which can change during the analysis. Moreover, we believe mentioning the graph structure in the definition of operators of maps increases the complexity to understand for readers. Our aim was to localize the operators independently of other parts of the analyzer. Our technique of localizing operators are done by keeping track of changes in them and few modifications were made to other parts of the analyzer.

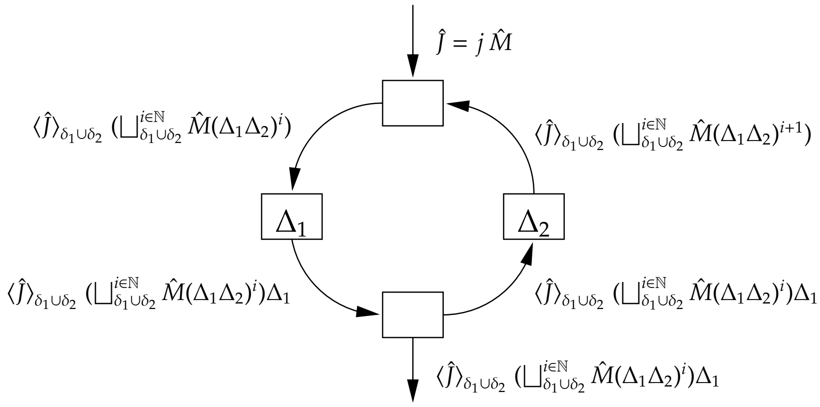

Figure 5: A Loop with localized join

6 Application: Array Index Range Analysis

6.1 Buffer overrun error

Buffer overrun is a root cause of major number of security breaches and malfunctions of software. Intruder manipulates the stack of privileged processes using buffer overrun errors and executes programs they want with the same privileges. Accessing area out of allocated boundaries may cause the program to stop, or even continue with totally unknown value which leads to undefined and mysterious behaviors. Some programming environments use runtime checks around every memory access to prevent this error. It can prevent the program from having undefined behavior, but does not fix the program from stopping. Besides, they have great impact on the performance of the program.

We present static analysis of array index ranges using our abstract interpretation here. By conservatively analyzing array index ranges and bounds, all buffer overrun errors can be detected. The implementation called Airac5[18] is able to analyze real C programs. It found number of real bugs in the Linux Kernel and some GNU software with reasonable cost. The analysis results of some free and open source software are shown in Section 6.3.1. Although our goal is to guide the programmer to fix the error and make programs more robust, the analysis results can also be used for removing runtime checks in the target program to improve its running performance.

Huge amount of work is done on detecting buffer overrun or overflow errors. There is an extensive survey[16] on causes of buffer overrun error and its solutions. Some[13, 10] suggest methods inserting run-time bound checks to detect buffer overruns, while some others[5, 20] try to eliminate those inserted checks that are safe to remove, to reduce useless computation. Others[15, 21, 19, 11] present static analyses that detect buffer overrun errors at compile-time, so that programmers can fix them.

6.2 Array index range analysis

For a G program P, buffer overrun errors can be found from an approximation $(\hat{G}, \hat{T}, \hat{D}) \in \widehat{Summary}$ of the least fixpoint of $\hat{\mathcal{F}}\ P$. Bad addresses can be found by checking all the indexes of addresses used for memory access against the size of the region. We can find program codes that are using such addresses by simply running the abstract interpretation once more with the computed fixpoint, and gather used addresses with allocation information for inspection.

Let us formalize this inspection process.

We define the set of buffer overrun errors found from a fixpoint $(\_, \hat{T}, \hat{D}) \in \widehat{Summary}$ as

$$\{(\hat{A}, \hat{I}) \in \mathcal{I}_b\ b\ \hat{D}\ \hat{m} \mid b \mapsto \hat{m} \in \hat{T}, Overrun(\hat{A}, \hat{I})\}.$$

$\mathit{Overrun}((\hat{r},\hat{i}),\hat{I})$ denotes the fact that an address $(\hat{r},\hat{i}) \in \hat{\mathit{Region}} \times \hat{\mathit{Index}}$ used under some allocation context $\hat{I}$ can cause overrun because

$$\{\hat{i}\} \sqcap \mathcal{S}_{>=}\,\{s\} \neq \hat{\bot}_{\hat{\mathit{Value}}} \quad \text{or} \quad \{\hat{i}\} \sqcap \mathcal{S}_{<}\,\{[0,0]\} \neq \hat{\bot}_{\hat{\mathit{Value}}}$$

where $\hat{r} \mapsto (s,\_) \in \hat{I}$.

$\mathcal{I}_b, \mathcal{I}_c, \mathcal{I}_e$ gathers the set in $\wp(\mathit{AddrUsed})$ containing all address $\hat{A}$ used for updating or looking up the abstract memory during the interpretation of block, command, and expression respectively. This is exactly the same set containing all the address $\hat{A}$ appeared as $\hat{M}\{\hat{A} \hat{\longmapsto} \_\}$ or $\hat{M}\,?\,\hat{A}$ in the definition of $\hat{\mathcal{F}}$ or $\hat{\mathcal{R}}, \hat{\mathcal{V}}$. *AddrUsed* has address with the information when it was used.

$$\mathit{AddrUsed} = \hat{\mathit{Addr}} \times \hat{\mathit{Alloc}}$$

$$\mathcal{I}_b \quad : \quad \mathit{Block}_P \to \hat{\mathit{Dump}} \to \hat{\mathit{Memory}} \to \wp(\mathit{AddrUsed})$$

$$\begin{aligned}
\mathcal{I}_b\ b^{\mathsf{p}}\ \hat{D}\ (\hat{M},\hat{I}) \quad = \bigcup(\quad & \{\mathcal{I}_b\ (e,\mathsf{p})\ (\hat{M},\hat{I}) \cup \mathcal{I}_b\ (e_0,\mathsf{p})\ (\hat{M},\hat{I}) \cup \\
& \quad \mathcal{I}_b\ (e_1,\mathsf{p})\ (\hat{M},\hat{I}) \cup \{(b^{\mathsf{p}}, \hat{\mathcal{A}}\ \mathsf{p}'\ \hat{M}\ \mathsf{x}, \hat{I})\} \mid \\
& \quad b^{\mathsf{p}} = \mathtt{CALL}(e,e_0,e_1), \\
& \quad \mathsf{p}' \in \hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_0,\ (\mathsf{p}',\mathsf{x}) \in P\ \} \cup \\
& \{\mathcal{I}_b\ (c_i,\mathsf{p})\ \hat{D}\ \hat{m}_i \mid b^{\mathsf{p}} = (c_1,\cdots,c_n), \\
& \quad \hat{m}_1 = (\hat{M},\hat{I}),\ \hat{m}_{i+1} = \hat{\mathcal{R}}\ \hat{D}\ \mathsf{p}\ c_i\ \hat{m}_i\ \} \quad)
\end{aligned}$$

$$\mathcal{I}_c \quad : \quad \mathit{Cmd} \times \mathit{ProcId} \to \hat{\mathit{Dump}} \to \hat{\mathit{Memory}} \to \wp(\mathit{AddrUsed})$$

$$\begin{aligned}
\mathcal{I}_c\ (\mathtt{SET}(e_a,e),\mathsf{p})\ \hat{D}\ (\hat{M},\hat{I}) &= \mathcal{I}_c\ (e_a,\mathsf{p})\ (\hat{M},\hat{I}) \cup \mathcal{I}_c\ (e,\mathsf{p})\ (\hat{M},\hat{I}) \\
&\cup \{(\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e_a,\hat{I})\} \\
\mathcal{I}_c\ (\mathtt{ESCAPE}(e),\mathsf{p})\ \hat{D}\ (\hat{M},\hat{I}) &= \mathcal{I}_c\ (e,\mathsf{p})\ (\hat{M},\hat{I}) \cup \{(\hat{D}\ \mathsf{p},\hat{I})\} \\
\mathcal{I}_c\ (\mathtt{ASSUME}(e \diamond e'),\mathsf{p})\ \hat{D}\ (\hat{M},\hat{I}) &= \mathcal{I}_c\ (e,\mathsf{p})\ (\hat{M},\hat{I}) \cup \mathcal{I}_c\ (e',\mathsf{p})\ (\hat{M},\hat{I}) \\
\mathcal{I}_c\ (\mathtt{ALLOC}(e_a,\{\vec{\mathsf{f}}\}),\mathsf{p})\ \hat{D}\ (\hat{M},\hat{I}) &= \mathcal{I}_c\ (e_a,\mathsf{p})\ (\hat{M},\hat{I}) \\
\mathcal{I}_c\ (\mathtt{ALLOC}(e_a,[e]),\mathsf{p})\ \hat{D}\ (\hat{M},\hat{I}) &= \mathcal{I}_c\ (e_a,\mathsf{p})\ (\hat{M},\hat{I}) \cup \mathcal{I}_c\ (e,\mathsf{p})\ (\hat{M},\hat{I}) \\
\mathcal{I}_c\ (\mathtt{FREE}(e),\mathsf{p})\ \hat{D}\ (\hat{M},\hat{I}) &= \mathcal{I}_c\ (e,\mathsf{p})\ (\hat{M},\hat{I})
\end{aligned}$$

$$\mathcal{I}_e \quad : \quad \mathit{Expr} \times \mathit{ProcId} \to \hat{\mathit{Memory}} \to \wp(\mathit{AddrUsed})$$

$$\begin{aligned}
\mathcal{I}_e\ (e \star e',\mathsf{p})\ (\hat{M},\hat{I}) &= \mathcal{I}_e\ (e,\mathsf{p})\ (\hat{M},\hat{I}) \cup \mathcal{I}_e\ (e',\mathsf{p})\ (\hat{M},\hat{I}) \\
\mathcal{I}_e\ (e.\mathsf{f},\mathsf{p})\ (\hat{M},\hat{I}) &= \mathcal{I}_e\ (e,\mathsf{p})\ (\hat{M},\hat{I}) \\
\mathcal{I}_e\ (\$e,\mathsf{p})\ (\hat{M},\hat{I}) &= \mathcal{I}_e\ (e,\mathsf{p})\ (\hat{M},\hat{I}) \cup \{(\hat{\mathcal{A}}\ \mathsf{p}\ \hat{M}\ e,\hat{I})\} \\
\mathcal{I}_e\ (_,\mathsf{p})\ (\hat{M},\hat{I}) &= \emptyset
\end{aligned}$$

6.3 Implementation: Airac5

We have implemented an abstract interpreter that can analyze real C programs. It is called Airac5[18]. First, it transforms programs written in ANSI C with some GNU extensions into G programs. Then it computes an approximation of the least fixpoint which is then inspected to find buffer overrun alarms. Airac5 uses all the implementation techniques described in Chapter 5.

6.3.1 Performance

We have analyzed some small device drivers and modules of Linux Kernel 2.6.4, as well as large programs of GNU. Table 1 and 2 shows analysis speed and accuracy data of Airac5. All analysis was done in a Linux 2.6 system running on a Pentium4 3.2GHz box with 4GB of main memory. We inlined every procedure one depth from the root of the call tree. Procedures in each file were analyzed separately ignoring the side effects of procedures in other files. **LOC** are the number of line of the C source codes, before preprocessing them. **Time** are the CPU times spent during analysis. **#Alarms** are the number of alarms Airac5 raised, and **#Bugs** are the number of real bugs we have confirmed in that program.

In order to effectively analyze real world C programs, we used some assumptions that may affect the soundness of our analysis. We assumed free variables can have any value,

Table 1: Airac5's performance on Linux Kernel 2.6.4

Filename	LOC	Time(sec)	#Alarms	#Bugs
vmax302.c	246	161.38	1	1
cdc-acm.c	849	199.15	2	2
atkbd.c	944	526.75	2	2
eata_pio.c	984	2319.11	1	1
ip6_output.c	1110	7857.39	0	0
xfrm_user.c	1201	5974.44	9	1
keyboard.c	1256	1089.25	18	1
af_inet.c	1273	9825.47	25	2
usb-midi.c	2206	2784.22	4	2
aty128fb.c	2466	972.09	2	1
mptbase.c	6158	14928.75	7	1

Table 2: Airac5's performance on free or open source software

Program	LOC	Time(sec)	#Alarms	#Bugs
tar-1.13	20258	24783.40	424	1
bison-1.875	25907	20340.19	323	
gzip-1.2.4a	7327	18401.30	374	
sed-4.0.8	6053	51516.45	463	
grep-2.5.1	9297	33325.10	275	
Doom		471.09	413	1

and procedures without source codes have no side effect. For large programs, we analyzed procedures in each file separately, not considering the side effects of procedures in other files. We do not raise alarms for memory access expressions when the size of the target region is not known since they are usually noises due to our approximation. Although there is no semantics defined after erroneous commands, we continued the analysis beyond those points in order to gather all possible errors. We ignore type castings and assume every memory regions are used as the same type they were allocated. Very limited number of library functions that have side effects on memory were considered during the analysis.

6.3.2 Speed up of localizing operators

We implemented the technique for localizing operators suggested in Section 5.3 and measured its speed up. Table 3 shows the speed up of some programs in Linux Kernel 2.6.4. All the experiments were performed also on a Pentium4 3.2GHz, 4GB main memory system. Analyses using localized operators were almost always faster, and sometimes it was nearly twice as faster than using normal operators.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proc. Compiler Construction (LNCS 2985)*, pages 5–23. Springer Verlag, 2004.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis,*

Table 3: Speed up of localizing operators

Name	Without (s)	With (s)	Improvement
usb-serial.i	548.79	283.85	48.27%
usb-midi.i	3353.51	1841.59	45.08%
wavelan.i	10165.58	5741.34	43.52%
skge.i	49749.93	34551.73	30.54%
isdn_tty.i	19962.56	15711.49	21.29%
cdc-acm.i	172.66	144.22	16.47%
mptbase.i	17149.42	15013.55	12.45%
de4x5.i	14555.33	13301.8	8.61%
af_inet6.i	5336.33	5154.56	3.40%
i2o_proc.i	33556.94	35543.75	- 5.92%
		Average	15.62%
		Standard Deviation	16.01%

Transformation. Essays Dedicated to Neil D. Jones, LNCS 2566, pages 85–108. Springer-Verlag, October 2002.

[4] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 196–207, New York, NY, USA, 2003. ACM Press.

[5] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 321–333, New York, NY, USA, 2000. ACM Press.

[6] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[7] 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 ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[8] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[9] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP '92: Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, London, UK, 1992. Springer-Verlag.

[10] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[11] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, New York, NY, USA, 2003. ACM Press.

[12] Nevin Heintze. Set-based analysis of ml programs. In *Proceedings of the SIGPLAN Conference on Lisp and Functional Programming*, 1994.

[13] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[14] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[15] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 2001 USENIX Security Symposium*, pages 177–190, 2001.

[16] Kyung-Suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exper.*, 33(5):423–460, 2003.

[17] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, Lecture Notes in Computer Science, pages 5–20. Springer-Verlag, 2005.

[18] Jaeho Shin, Jaehwang Kim, Hakjoo Oh, Yikwon Hwang, and Kwangkeun Yi. Airac5: Array index range analyzer for c. `http://ropas.snu.ac.kr/2005/airac5/`.

[19] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, 2000.

[20] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, New York, NY, USA, 1998. ACM Press.

[21] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336, New York, NY, USA, 2003. ACM Press.

A Notation

A.1 Functions, Fixpoints

A.1.1 Composition

$$\begin{array}{rcl} \circ & : & (A \to B) \to (C \to A) \to (C \to B) \\ (f \circ g)\, x & = & f\,(g\,x) \end{array}$$

$$\begin{array}{rcl} ; & : & (A \to B) \to (B \to C) \to (A \to C) \\ (f ; g)\, x & = & g\,(f\,x) \end{array}$$

A.1.2 Fixpoint

For a monotone function $f : A \to A$ on a complete partially ordered set A, let us use $\mathrm{lfp}_x f$ to denote its least fixpoint greater than or equal to $x \in A$.

$$\mathrm{lfp}_x f = f\,(\mathrm{lfp}_x f) \qquad x \sqsubseteq \mathrm{lfp}_x f$$

$$\forall a \in A : a = f\,a, x \sqsubseteq a \Longrightarrow \mathrm{lfp}_x f \sqsubseteq a$$

A.2 Product and Sum

A.2.1 Product

$A \times B$ is a Cartesian product of sets A and B. $(a, b) \in A \times B$ if and only if $a \in A$ and $b \in B$. One may naturally extend this binary product to product of finite number of sets. Below, $(a_1, \cdots, a_n) \in A_1 \times \cdots \times A_n$.

Projection

$$(a_1, \cdots, a_n)_i \quad = \quad a_i$$

Insertion

$$\begin{array}{lcl} a :: \epsilon & = & (a) \\ a :: (a_1, \cdots, a_n) & = & (a, a_1, \cdots, a_n) \end{array}$$

A.2.2 Sum

$A + B$ is a disjunctive sum of sets A and B. $c_A \in A + B$ if and only if $c \in A$ and $c_B \in A + B$ if and only if $c \in B$. We omit the subscript $_A$ or $_B$ when it is clear where the element belongs to.

A.3 Finite maps

Let us write $A \xrightarrow{\text{fin}} B$ to denote set of finite number of mappings from A to B. A mapping from A to B is a pair of an element in A and one in B, written $a \mapsto b$ where $a \in A, b \in B$, and a finite map is a set of such mappings. There are several operators defined over finite maps. Let M and N be of type $A \xrightarrow{\text{fin}} B$.

$$\begin{array}{lcl} \text{dom}(\cdot) & : & (A \xrightarrow{\text{fin}} B) \to \wp(A) \\ \text{dom}(M) & = & \{a \in A \mid a \mapsto b \in M\} \end{array}$$

$$\begin{array}{lcl} \cdot - \cdot & : & (A \xrightarrow{\text{fin}} B) \times \wp(A) \to (A \xrightarrow{\text{fin}} B) \\ M - A' & : & \{a \mapsto b \in M \mid a \notin A'\} \end{array}$$

$$\begin{array}{lcl} \cdot\cdot & : & (A \xrightarrow{\text{fin}} B) \times (A \xrightarrow{\text{fin}} B) \to (A \xrightarrow{\text{fin}} B) \\ MN & = & (M - \text{dom}(N)) \cup N \end{array}$$

$$\begin{array}{lcl} \cdot(\cdot) & : & (A \xrightarrow{\text{fin}} B) \times A \to B \\ M(a) & = & b \text{ where } a \mapsto b \in M \end{array}$$