The Name of the Title is Hope

JUNPENG ZHA, Department of Computer Science and Technology, Nanjing University

ACM Reference Format:
Junpeng Zha. 2018. The Name of the Title is Hope. *Proc. ACM Meas. Anal. Comput. Syst.* 37, 4, Article 111 (August 2018), 32 pages. https://doi.org/10.1145/1122445.1122456

In this document, we show how to define transformation rules to prove the correctness of the compiler optimizations.

1 LANGUAGES

The syntax of the language.

$$
\begin{array}{rcll}
\textit{(Addr)}\ \ x \ \in \ \ldots & \textit{(Reg)}\ \ r \ \in \ \ldots & \textit{(Lab)}\ \ l, \mathsf{f} \ \in \ \ldots & \\
\textit{(Expre)} & e & ::= & r \mid v \mid e+e \mid e-e \mid e*e \\
\textit{(Instr)} & i & ::= & \langle r := e \rangle_l \mid \langle r := [e] \rangle_l \mid \langle [e_1] := e_2 \rangle_l \mid \langle \mathtt{skip} \rangle_l \mid \langle \mathtt{print}\ r \rangle_l \\
 & & \mid & \langle \mathtt{jmp} \rangle_l \mid \langle \mathsf{be}\ e \rangle_{l_1, l_2} \\
\textit{(Code)} & C & ::= & \{ l \rightsquigarrow i \}^* \quad \text{where } \ l \in \textit{Lab} \\
\textit{(GE)} & ge & ::= & \{ x \rightsquigarrow v \}^* \\
\textit{(Init)} & \mathsf{Init} & \in & \textit{GE} \longrightarrow \textit{Mem} \times \textit{Reg}
\end{array}
$$

The program state.

$$
\begin{array}{rcll}
\textit{(Mem)} & M & ::= & \{ x \rightsquigarrow v \}^* \quad \text{where } \ x \in \textit{Addr} \\
\textit{(RegFile)} & R & ::= & \{ r \rightsquigarrow v \}^* \quad \text{where } \ r \in \textit{Reg} \\
\textit{(State)} & S & ::= & (M, R)
\end{array}
$$

The semantics of the language.

$$
\frac{C(\mathsf{pc}) = i \qquad i \vdash ((M, R), \mathsf{pc}) \longrightarrow ((M', R'), \mathsf{pc}')}{C \vdash ((M, R), \mathsf{pc}) \longrightarrow ((M', R'), \mathsf{pc}')}
$$

$$
\frac{[\![e]\!]_R = v}{\langle r := e \rangle_l \vdash ((M, R), \mathsf{pc}) \longrightarrow ((M, R\{ r \rightsquigarrow v \}), l)}
\qquad
\frac{[\![e]\!]_R = x \qquad M(x) = v}{\langle r := [e] \rangle_l \vdash ((M, R), \mathsf{pc}) \longrightarrow ((M, R\{ r \rightsquigarrow v \}), l)}
$$

$$
\frac{[\![e_1]\!]_R = x \qquad [\![e_2]\!]_R = v}{\langle [e_1] := e_2 \rangle_l \vdash ((M, R), \mathsf{pc}) \longrightarrow ((M\{ x \rightsquigarrow v \}, R), l)}
\qquad
\frac{}{\langle \mathtt{skip} \rangle_l \vdash ((M, R), \mathsf{pc}) \longrightarrow ((M, R), l)}
$$

$$
\frac{[\![e]\!]_R = v \qquad (l = l_1 \ \wedge \ v = 0) \vee (l = l_2 \ \wedge \ v = 1)}{\langle \mathsf{be}\ r \rangle_{l_1, l_2} \vdash ((M, R), \mathsf{pc}) \longrightarrow ((M, R), l)}
\qquad
\frac{R(r) = v}{\langle \mathtt{print}\ r \rangle_l \vdash ((M, R), \mathsf{pc}) \longrightarrow ((M, R), l)}
$$

Author's address: Junpeng Zha, Department of Computer Science and Technology, Nanjing University.

2 FINAL THEOREM

We show the final theorem in this section. Our goal is to prove the correctness of the compiler optimizations. A compiler optimization includes an analyzer and a translater.

$$\begin{array}{rlcl} \textit{(Analyzer)} & \textsf{Analyzer} & \in & \textit{Code} \times \textit{GE} \times \textit{Lab} \rightharpoonup \textit{AnalyRes} \\ \textit{(Translater)} & \textsf{Translater} & \in & \textit{Code} \times \textit{AnalyRes} \rightharpoonup \textit{Code} \end{array}$$

The result of the analyses *AnalyRes* will be defined in the following.

A compiler optimization is correct, if the target code generated refines the source code. We define the refinement relation below.

Definition 2.1 (Refinement).

$$\begin{array}{rcl} C' \subseteq_{ge,\mathsf{f}} C & ::= & \forall M, R.\ (\mathsf{Init}(ge) = (M, R) \wedge \mathsf{Safe}(C, (M, R), \mathsf{f})) \\ & & \qquad \Longrightarrow\ (C', (M, R), \mathsf{f}) \subseteq (C, (M, R), \mathsf{f}) \end{array}$$

$$\text{where}\quad \mathsf{Safe}(C, (M, R), \mathsf{f})\ ::=\ \neg(C \vdash (M, R, \mathsf{f}) \longrightarrow^* \textbf{abort}).$$

We give the final theorem below.

Theorem 2.2 (Final Theorem). For any C, C', f, ge and A, if $\mathsf{Analyzer}(C, ge, \mathsf{f}) = A$ and $\mathsf{Translater}(C, A) = C'$, then $C' \subseteq_{ge,\mathsf{f}} C$.

The correctness proof of the Theorem. 2.2 can be divided into the proof of the two lemmas shown below.

Lemma 2.3 (Optimization Correctness). For any C, C', f, ge and A, if $\mathsf{Analyzer}(C, ge, \mathsf{f}) = A$ and $\mathsf{Translater}(C, A) = C'$, then there exists I such that $I \vdash_{ge,\mathsf{f}} (C, A) \sim C'$.

Lemma 2.4 (Soundness of Transformation rule).

$$I \vdash_{ge,\mathsf{f}} (C, A) \sim C' \implies C' \subseteq_{ge,\mathsf{f}} C$$

In Lemma. 2.3, we prove the correctness of the optimization by proving whether the source code, the target code and the analyses result can pass the checking of the transformation rules.

The transformation rule in the form of "$I \vdash_{ge,\mathsf{f}} (C, A) \sim C'$" will be defined in Sec. 5.

The Lemma. 2.4 shows that the soundness of the transformation rule can ensure that the target code refines the source code.

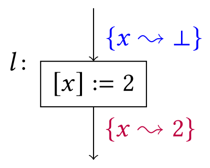
3 THE DEFINITION OF THE ANALYSIS RESULT

We define the result of the analysis *(AnalyRes)* mentioned previously in this section. In this work, we focus on the dataflow analysis mainly.

$$\begin{array}{rlcl} \textit{(AI)} & L & ::= & \ldots \\ \textit{(AIM)} & \mathbb{L} & ::= & \{l_1 \rightsquigarrow L_1, \ldots, l_2 \rightsquigarrow L_n\} \\ \textit{(AnalyRes)} & A & ::= & (\mathbb{L}_i, \mathbb{L}_o) \end{array}$$

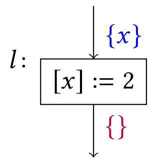
The result of the dataflow analysis gives each node an abstract interpretation. We show the abstract interpretation in the value analysis, the liveness analysis and the common subexpression elimination below.

Value Analysis :



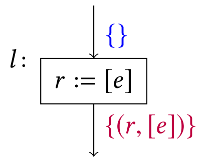
$$\begin{aligned} (AVal) \quad \hat{v} \;&\in\; Val \cup \{\bot, \top\} \\ a_v \;&\in\; \mathcal{P}((Addr \rightharpoonup AVal) \cup (Reg \rightharpoonup AVal)) \end{aligned}$$

Liveness Analysis :



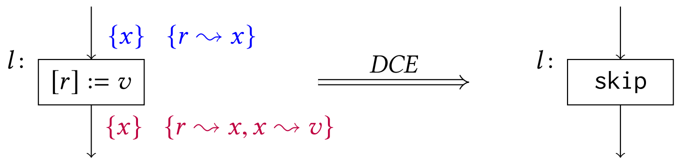
$$a_l \;\in\; \mathcal{P}(Addr \cup Reg)$$

Available Expression Analysis :



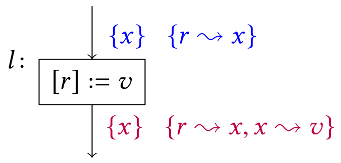
$$\begin{aligned} (VExp) \quad ve \;&::=\; (r, e) \mid (r, [e]) \\ a_e \;&\in\; \mathcal{P}(VExp) \end{aligned}$$

An analyzer in a compiler optimization may do more than one analysis. For example, the *dead code elimination* optimization in CompCert does both the value analysis and the liveness analysis as shown below.


l:
$\{x\}$ $\{r \rightsquigarrow x\}$
$[r] := v$
$\{x\}$ $\{r \rightsquigarrow x, x \rightsquigarrow v\}$
DCE
l:
skip

The instruction "$[r] := v$" is a dead code, since "$\{r \rightsquigarrow x\}$" says that the register r points to the memory location x, and "$\{x : \circ\}$" says that the memory location x will not be read before the next write.

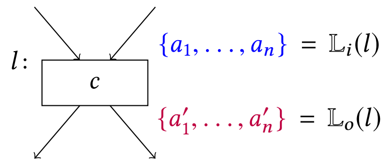
Thus, we define the set of abstract interpretation L *(AISet)*, as the collection of abstract interpretations of each node. We give an example of the specific L below.


l:
$\{x\}$ $\{r \rightsquigarrow x\}$
$[r] := v$
$\{x\}$ $\{r \rightsquigarrow x, x \rightsquigarrow v\}$

$$\begin{aligned} L \quad &::= \quad \{\{x\}, \{r \rightsquigarrow x\}\} \\ L' \quad &::= \quad \{\{x\}, \{r \rightsquigarrow x, x \rightsquigarrow v\}\} \end{aligned}$$

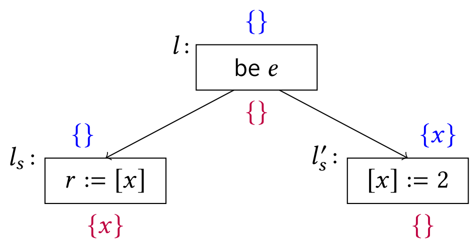
Then, we define the mapping of the set of abstract interpretations $\mathbb{L}$ *(AISetM)*, which is a partial mapping from the label to the set of the abstract interpretations.

Finally, we define the result of the analysis below, which is a pair of *AISetM*. Here, the $\mathbb{L}_i$ maps the label of each node to the set of abstract interpretations, which describe the program state before the execution of the instruction of the node, and the $\mathbb{L}_o$ maps the label of each node to the set of abstract interpretations, which describe the program state after the execution of the instruction of the node. We show the meaning of $\mathbb{L}_i$ and $\mathbb{L}_o$ below.



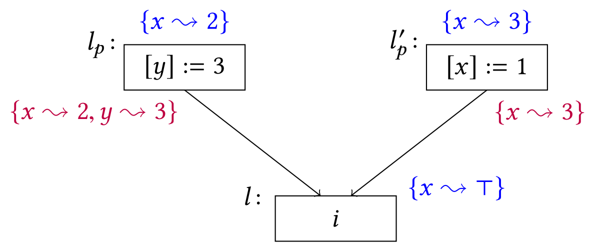
$l:$
c
$\{a_1, \ldots, a_n\} = \mathbb{L}_i(l)$
$\{a'_1, \ldots, a'_n\} = \mathbb{L}_o(l)$

The abstract interpretation, which describes the program state after the execution of the node labelled l, may be different from the abstract interpretation, which describes the program state before the execution of the node l's successor node l_s, as the example shown below.



$$f(\{x\}, r := [x]) = \{\}$$
$$f(\{\}, [x] := 2) = \{x\}$$
$$\{\} \cap \{x\} = \{\}$$

Similarly, The abstract interpretation, which describes the program state before the execution of the node labelled l, may be different from the abstract interpretation, which describes the program state after the execution of the node l's predecessor node l_p, as the example shown below.



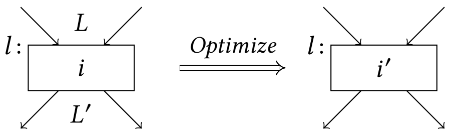
$\{x \rightsquigarrow 2\}$
$l_p:$
$[y] := 3$
$\{x \rightsquigarrow 2, y \rightsquigarrow 3\}$
$\{x \rightsquigarrow 3\}$
$l'_p:$
$[x] := 1$
$\{x \rightsquigarrow 3\}$
$l:$
i
$\{x \rightsquigarrow \top\}$

$$f(\{x \rightsquigarrow 2\}, [y] := 3) = \{x \rightsquigarrow 2, y \rightsquigarrow 3\}$$
$$f(\{x \rightsquigarrow 3\}, [x] := 1) = \{x \rightsquigarrow 3\}$$
$$\{x \rightsquigarrow 2, y \rightsquigarrow 3\} \cup \{x \rightsquigarrow 3\} = \{x \rightsquigarrow \top\}$$

4 THE ASSERTION LANUGAGE

As we have introduced, we want to define a set of transformation rules to check the correctness of the compiler optimization. The transformation rule checks whether the transformation is legel according to the result of the analysis. Thus, the rules that checks the correctness of transforming an individual instruction from the source code to the corresponding instruction in the target code may be in the following form.

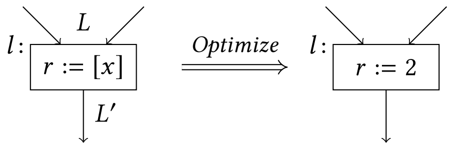
$$\frac{\textit{some side conditions}\ldots}{\vdash \{L\}\, i\, \{L'\} \sim i'}$$



$$A = (\mathbb{L}_i, \mathbb{L}_o)$$
$$\mathbb{L}_i(l) = L \qquad \mathbb{L}_o(l) = L'$$

We use a code transformation in constant propagation as an example.

$$\frac{L = \{\{x \rightsquigarrow 2\}\}}{\vdash \{L\}\, \langle r := [x]\rangle_{l_1}\, \{L'\} \sim \langle r := 2\rangle_{l_1}}$$

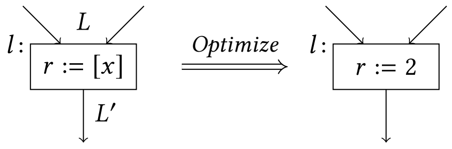


$$A = (\mathbb{L}_i, \mathbb{L}_o)$$
$$\mathbb{L}_i(l) = L = \{\{x \rightsquigarrow 2\}\} \qquad \mathbb{L}_o(l) = L = \{\{x \rightsquigarrow 2, r \rightsquigarrow 2\}\}$$

However, the transformation rule in such form is not general, since different analyses will produce different forms of analysis results.

So, we define a general assertion language to replace the specific analysis result to describe the program state. The basic idea is show below. I is a converter that converts a specific form of the abstract interpretation to the general assertion.

$$\frac{I(L) \Rrightarrow (x \mapsto 2)}{\vdash \{I(L)\}\, \langle r := [x]\rangle_{l_1}\, \{I(L')\} \sim \langle r := 2\rangle_{l_1}}$$



$$A = (\mathbb{L}_i, \mathbb{L}_o)$$
$$\mathbb{L}_i(l) = L = \{\{x \rightsquigarrow 2\}\} \qquad \mathbb{L}_o(l) = L = \{\{x \rightsquigarrow 2, r \rightsquigarrow 2\}\}$$
$$I(L) = (x \mapsto 2) \qquad I(L') = (x \mapsto 2) \wedge (r = 2)$$

We define the syntax of the assertion language in Fig. 1.

We define the semantics of the assertion language in Fig. 2. Here, the *is* is the set of identifiers, which include registers and memory locations.

$$\textit{(IdSet)} \quad \textit{is} \quad \in \quad \mathcal{P}(\textit{Addr} \cup \textit{Reg})$$

$$\begin{array}{llll} \textit{(Asrt)} & p & ::= & e \mapsto e' \mid e = e' \mid \mathsf{liveE}(e) \mid \mathsf{liveX}(e) \mid p[e/r] \\ & & \mid & p_1 * p_2 \mid p_1 \land p_2 \mid p_1 \lor p_2 \mid p_1 \mathrel{-\!\!*} p_2 \mid \exists v.p \mid \forall v.p \end{array}$$

Fig. 1. The syntax of the assertion language

$$\begin{array}{lll}
(M, R, is) \models e \mapsto e' & ::= & \exists x, v.\, [\![e]\!]_R = x \land [\![e']\!]_R = v \land M(x) = v \\
(M, R, is) \models e = e' & ::= & [\![e]\!]_R = [\![e']\!]_R \\
(M, R, is) \models \mathsf{liveE}(e) & ::= & \mathsf{fv}(e) \cap is = \emptyset \\
(M, R, is) \models \mathsf{liveX}(e) & ::= & \exists x.\, [\![e]\!]_R = x \land x \notin is \\
(M, R, is) \models p[e/r] & ::= & \exists v.\, [\![e]\!]_R = v \land (M, R\{r \rightsquigarrow v\}, is) \models p \\
(M, R, is) \models p_1 * p_2 & ::= & \exists M_1, M_2.\, \mathrm{dom}(M_1) \cap \mathrm{dom}(M_2) = \emptyset \land M = M_1 \cap M_2 \\
 & & \qquad \land (M_1, R, is) \models p_1 \land (M_2, R, is) \models p_2 \\
(M, R, is) \models p_1 \land p_2 & ::= & (M, R, is) \models p_1 \land (M, R, is) \models p_2 \\
(M, R, is) \models p_1 \lor p_2 & ::= & (M, R, is) \models p_1 \lor (M, R, is) \models p_2 \\
(M, R, is) \models p_1 \mathrel{-\!\!*} p_2 & ::= & \forall M'.\, ((M', R, is) \models p_1 \land \mathrm{dom}(M') \cap \mathrm{dom}(M) = \emptyset) \\
 & & \qquad \Longrightarrow (M \cup M', R, is) \models p_2 \\
(M, R, is) \models \exists v.p & ::= & \exists v.\, (M, R, is) \models p \\
(M, R, is) \models \forall v.p & ::= & \forall v.\, (M, R, is) \models p
\end{array}$$

Fig. 2. The semantics of the assertion language

We use the "fv(e)" to collect the registers used in the expression e.

$$\mathsf{fv}(e) \quad ::= \quad \begin{cases} \{r\} & \textit{if } e = r \\ \emptyset & \textit{if } e = n \\ \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) & \textit{if } e = e_1 + e_2 \\ \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) & \textit{if } e = e_1 - e_2 \end{cases}$$

We define an invariant I to describe the relation between the specific analysis and the assertion.

$$(AInv) \quad I \quad \in \quad AI \longrightarrow Asrt$$

The invariant I *(AInv)* will be instantiated when meeting different analysis results. Below, we give some examples.

Auxiliary definition for value analysis:

$$\begin{aligned} I'_v(a_v) \quad ::= \quad & (\forall r \in \mathrm{dom}(a_v).\, (\exists v.\, r = v \wedge a_v(r) = v) \vee (r = - \wedge a_v(r) \in \{\bot, \top\})) \\ & \wedge(\forall x \in \mathrm{dom}(a_v).\, (\exists v.\, x \mapsto v \wedge a_v(x) = v) \vee (x \mapsto - \wedge a_v(x) \in \{\bot, \top\})) \end{aligned}$$

Constant propagation (Value analysis a_v) :

$$I_c(a_v) \quad ::= \quad I'_v(a_v) \wedge (\forall r.\, \mathrm{liveE}(r)) \wedge (\forall x.\, \mathrm{liveX}(x))$$

Dead code elimination (Value analysis a_v + Liveness analysis a_l) :

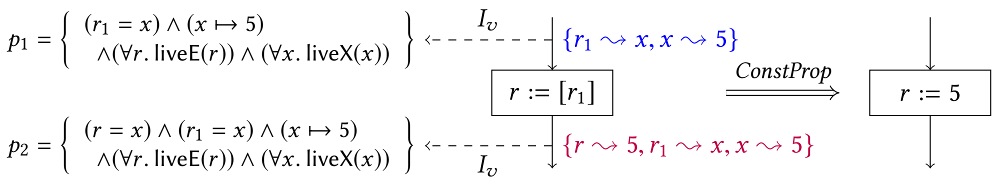
$$\begin{aligned} I_d(\{a_v, a_l\}) \quad ::= \quad & I'_v(\{a_v\}) \\ & \wedge (\forall r \notin a_l.\, \mathrm{liveE}(r)) \wedge (\forall r \in a_l.\, \neg\mathrm{liveE}(r))) \\ & \wedge (\forall x \notin a_l.\, \mathrm{liveX}(x) \wedge (\forall x \in a_l.\, \neg\mathrm{liveX}(x)) \end{aligned}$$

Common subexpression elimination (Value analysis a_v + Avaliable expression analysis a_e) :

$$\begin{aligned} I_e(\{a_v, a_e\}) \quad ::= \quad & I'_v(\{a_v\}) \\ & \wedge (\forall (r, [e]) \in a_e.\, e \mapsto r) \wedge (\forall (r, e) \in a_e.\, e = r) \\ & \wedge (\forall r.\, \mathrm{liveE}(r)) \wedge (\forall x.\, \mathrm{liveX}(x)) \end{aligned}$$

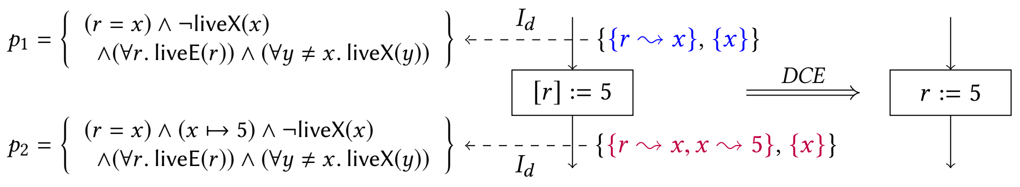
We use some digrams to illustrate the use of the invariant I *(AInv)* and the assertion p *(Asrt)* in defining the transformation rules.

Constant propagation.


$p_1 = \left\{ \begin{array}{l} (r_1 = x) \wedge (x \mapsto 5) \\ \wedge(\forall r.\, \mathrm{liveE}(r)) \wedge (\forall x.\, \mathrm{liveX}(x)) \end{array} \right\}$
I_v
$\{r_1 \rightsquigarrow x, x \rightsquigarrow 5\}$
$r := [r_1]$
ConstProp
$r := 5$
$p_2 = \left\{ \begin{array}{l} (r = x) \wedge (r_1 = x) \wedge (x \mapsto 5) \\ \wedge(\forall r.\, \mathrm{liveE}(r)) \wedge (\forall x.\, \mathrm{liveX}(x)) \end{array} \right\}$
I_v
$\{r \rightsquigarrow 5, r_1 \rightsquigarrow x, x \rightsquigarrow 5\}$

$$\frac{p_1 \Rightarrow (r_1 \mapsto 5)}{\vdash \{p_1\}\, \langle r := [r_1] \rangle_l\, \{p_2\} \sim \langle r := 5 \rangle_l}$$

Dead code elimination.

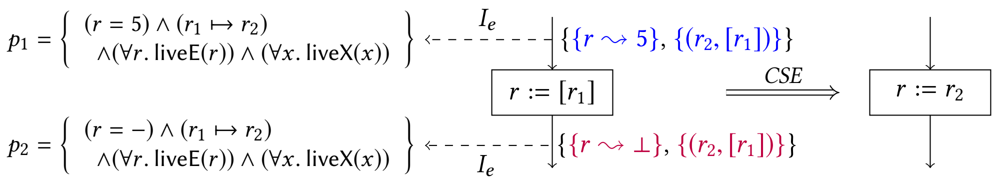

$p_1 = \left\{ \begin{array}{l} (r = x) \wedge \neg\mathrm{liveX}(x) \\ \wedge(\forall r.\, \mathrm{liveE}(r)) \wedge (\forall y \neq x.\, \mathrm{liveX}(y)) \end{array} \right\}$
I_d
$\{\{r \rightsquigarrow x\}, \{x\}\}$
$[r] := 5$
DCE
$r := 5$
$p_2 = \left\{ \begin{array}{l} (r = x) \wedge (x \mapsto 5) \wedge \neg\mathrm{liveX}(x) \\ \wedge(\forall r.\, \mathrm{liveE}(r)) \wedge (\forall y \neq x.\, \mathrm{liveX}(y)) \end{array} \right\}$
I_d
$\{\{r \rightsquigarrow x, x \rightsquigarrow 5\}, \{x\}\}$

$$\frac{}{\vdash \{p_1 \wedge r = x\}\, \langle [r] := 5 \rangle_l\, \{p_2 \wedge \neg\mathrm{liveX}(x)\} \sim \langle \mathtt{skip} \rangle_l}$$

$$
\begin{array}{rcl}
\mathsf{succ}(c) & ::= & \begin{cases} \{l\} & \textit{if } c \in \{\langle r := e\rangle_l, \langle r := [e]\rangle_l, \langle [e_1] := e_2\rangle_l, \\ & \qquad \langle \mathtt{jmp}\rangle_l, \langle \mathtt{skip}\rangle_l\} \\ \{l_1, l_2\} & \textit{if } c = \langle \mathsf{be}\ e\rangle_{l_1,l_2} \end{cases} \\
\mathsf{pred}(C, l) & ::= & \{l_p \mid l \in \mathsf{succ}(C(l_p))\} \\
\mathsf{liveE}(e_1, \ldots, e_n) & ::= & \mathsf{liveE}(e_1) \land \ldots \land \mathsf{liveE}(e_n) \\
p \Rightarrow p' & ::= & \forall M, R, is.\ (M, R, is) \models p \implies (M, R, is) \models p' \\
p \Rrightarrow p' & ::= & \forall M, R, is.\ (M, R, is) \models p \implies \exists is'.\ is \subseteq is' \land (M, R, is') \models p' \\
p_1 \oplus p_2 & ::= & (p_1 \land p_2) \lor (p_1 \mathrel{-\!\!*} p_2) \\
\mathsf{ALStaE}(p, e) & ::= & \forall M, R, is.\ (M, R, is) \models p \implies (M, R, is \backslash \mathsf{fv}(e)) \models p \\
\mathsf{ALStaX}(p, e) & ::= & \forall M, R, is, x.\ ((M, R, is) \models p \land [\![e]\!]_R = x) \implies (M, R, is \backslash x) \models p
\end{array}
$$

Fig. 3. Some auxiliary definitions

Common subexpression elimination.


$p_1 = \left\{ \begin{array}{l} (r = 5) \land (r_1 \mapsto r_2) \\ \land(\forall r.\ \mathsf{liveE}(r)) \land (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$
I_e
$\{\{r \rightsquigarrow 5\}, \{(r_2, [r_1])\}\}$
$r := [r_1]$
CSE
$r := r_2$
$p_2 = \left\{ \begin{array}{l} (r = -) \land (r_1 \mapsto r_2) \\ \land(\forall r.\ \mathsf{liveE}(r)) \land (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$
I_e
$\{\{r \rightsquigarrow \bot\}, \{(r_2, [r_1])\}\}$

$$\frac{p_1 \Rightarrow (r_1 \mapsto r_2 \land \mathsf{liveE}(r_2))}{\vdash \{p_1\}\ \langle r := [r_1]\rangle_l\ \{p_2\} \sim \langle r := r_2\rangle}$$

5 TRANSFORMATION RULES

We define the transformation rules in Fig. 4. Some auxiliary definitions are defined in Fig. 3. We introduce each rule in details below.

Well-formed Optimization. The Wf-Opt rule is the top rule. It divides the correctness proof of the optimization into two parts: the *Well-formed Analysis*, which checks the correctness of the results of the analysis, and the *Well-formed Translater*, which checks the correctness of the code transformation if the relied analysis is correct.

Well-formed Analysis. The wf-Analysis is the top rule of *Well-formed Analysis* check. The result of the analysis A includes two parts as we have introduced: $\mathbb{L}_i$ records the set of abstract interpretations describing the states before the execution of each node, and $\mathbb{L}_o$ records the set of abstract interpretations describing the state after the execution of each node. *First*, the abstract interpretations for describing the initial states should be correct ("$\exists is.\ (\mathsf{Init}(ge), is) \models I(\mathbb{L}_i(\mathsf{f}))$"); *Second*, we check that the abstract interpretations of each node is correct. We use

Well-formed Optimization:

$$\frac{I, C \vdash_{ge,\mathsf{f}} A \qquad I, A \vdash C \sim C'}{I \vdash_{ge,\mathsf{f}} (C, A) \sim C'} \quad \text{(wf-Opt)}$$

Well-formed Analysis:

$$\frac{\begin{array}{l} A = (\mathbb{L}_i, \mathbb{L}_o) \qquad \exists is.\, (\mathsf{Init}(ge), is) \models I(\mathbb{L}_i(\mathsf{f})) \\ \text{for all } l \in \mathrm{dom}(C): \\ \qquad I(\mathbb{L}_o(l)) \Rrightarrow I(\mathbb{L}_i(l_s)), \ \text{ for all } l_s \in \mathrm{succ}(C(l)) \\ \qquad \vdash \{I(\mathbb{L}_i(l))\}\ C(l)\ \{I(\mathbb{L}_o(l))\} \end{array}}{I, C \vdash_{ge,\mathsf{f}} A} \quad \text{(wf-Analysis)}$$

$$\frac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow p'}{\vdash \{p[e/r] \wedge \mathsf{liveE}(e)\}\ \langle r := e\rangle_l\ \{p'\}} \quad \text{(Asgn-Ac)}$$

$$\frac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow p'}{\vdash \{((e \mapsto e') \oplus p[e'/r]) \wedge \mathsf{liveE}(e) \wedge \mathsf{liveX}(e)\}\ \langle r := [e]\rangle_l\ \{p'\}} \quad \text{(Ld-Ac)}$$

$$\frac{\mathsf{ALStaX}((e_1 \mapsto - * p), e_1) \qquad (e_1 \mapsto e_2 * p) \wedge \mathsf{liveX}(e_1) \Rrightarrow p'}{\vdash \{((e_1 \mapsto -) \oplus (e_1 \mapsto - * p)) \wedge \mathsf{liveE}(e_1, e_2)\}\ \langle [e_1] := e_2\rangle_l\ \{p'\}} \quad \text{(St-Ac)}$$

$$\frac{p_1 \Rightarrow p_1' \qquad \vdash \{p_1'\}\ i\ \{p_2'\} \qquad p_2' \Rightarrow p_2}{\vdash \{p_1\}\ i\ \{p_2\}} \quad \text{(Consq-Ac)}$$

Well-formed Translater:

$$\frac{I, A \vdash C_1 \sim C_1' \qquad I, A \vdash C_2 \sim C_2'}{I, A \vdash (C_1 \uplus C_2) \sim (C_1' \uplus C_2')} \quad \text{(Link)} \qquad \frac{A = (\mathbb{L}_i, \mathbb{L}_o) \qquad I(\mathbb{L}_i(l)), I(\mathbb{L}_o(l)) \vdash i \sim i'}{I, A \vdash \{l \rightsquigarrow i\} \sim \{l \rightsquigarrow i'\}} \quad \text{(Ins-Trans)}$$

$$\frac{p \Rightarrow (e = e') \wedge \mathsf{liveE}(e')}{p, p' \vdash \langle r := e\rangle_l \sim \langle r := e'\rangle_l} \quad \text{(Asgn)} \qquad \frac{p' \Rightarrow \neg\mathsf{liveE}(r)}{p, p' \vdash \langle r := e\rangle_l \sim \langle \mathtt{skip}\rangle_l} \quad \text{(Asgn-Elim)}$$

$$\frac{p \Rightarrow (e = e') \wedge \mathsf{liveE}(e') \wedge \mathsf{liveX}(e')}{p, p' \vdash \langle r := [e]\rangle_l \sim \langle r := [e']\rangle_l} \quad \text{(Ld)} \qquad \frac{p' \Rightarrow \neg\mathsf{liveE}(r)}{p, p' \vdash \langle r := [e]\rangle_l \sim \langle \mathtt{skip}\rangle_l} \quad \text{(Ld-Elim)}$$

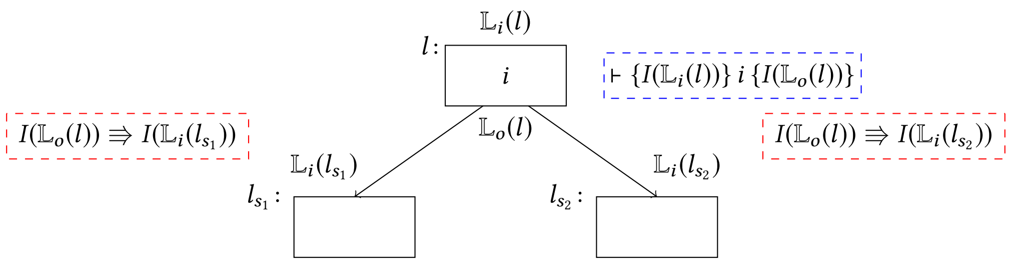
$$\frac{p \Rightarrow (e \mapsto e') \wedge \mathsf{liveE}(e')}{p, p' \vdash \langle r := [e]\rangle_l \sim \langle r := e'\rangle_l} \quad \text{(Ld-Elim2)}$$

$$\frac{p \Rightarrow (e_1 = e_1') \wedge (e_2 = e_2') \wedge \mathsf{liveE}(e_1', e_2')}{p, p' \vdash \langle [e_1] := e_2\rangle_l \sim \langle [e_1'] := e_2'\rangle_{l'}} \quad \text{(St)} \qquad \frac{p' \Rightarrow \neg\mathsf{liveX}(e_1)}{p, p' \vdash \langle [e_1] := e_2\rangle_l \sim \langle \mathtt{skip}\rangle_l} \quad \text{(St-Elim)}$$

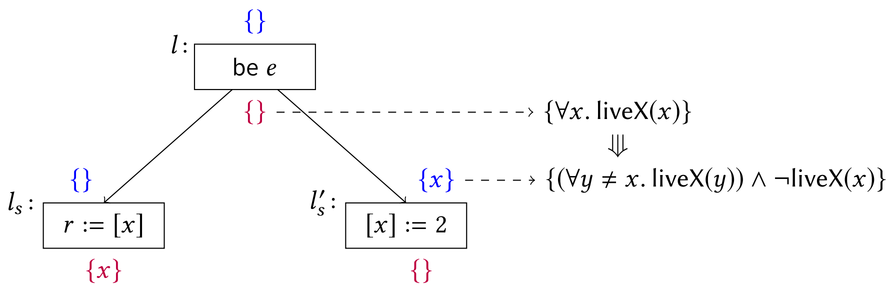
$$\frac{p \Rightarrow \mathsf{liveE}(r)}{p, p' \vdash \langle \mathtt{print}\ r\rangle_l \sim \langle \mathtt{print}\ r\rangle_l} \quad \text{(Print)}$$

Fig. 4. Transformation rules

the following digram to show the correctness checking of each node.


$\mathbb{L}_i(l)$
$l:$
i
$\vdash \{I(\mathbb{L}_i(l))\}\ i\ \{I(\mathbb{L}_o(l))\}$
$I(\mathbb{L}_o(l)) \Rrightarrow I(\mathbb{L}_i(l_{s_1}))$
$\mathbb{L}_o(l)$
$I(\mathbb{L}_o(l)) \Rrightarrow I(\mathbb{L}_i(l_{s_2}))$
$\mathbb{L}_i(l_{s_1})$
$\mathbb{L}_i(l_{s_2})$
$l_{s_1}:$
$l_{s_2}:$

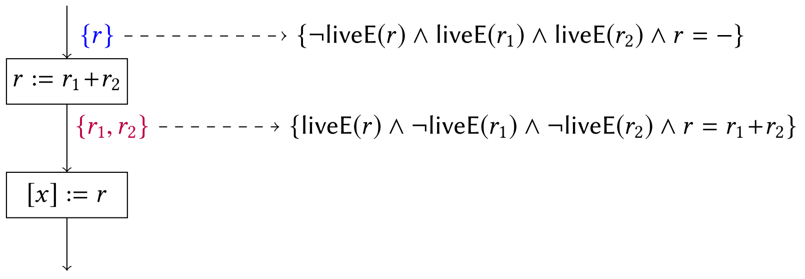
As for the implication of the assertion, we define two types of the implication in Fig. 3 shown as "$p \Rightarrow p'$" and "$p' \Rrightarrow p'$" respectively. The example shown below explains why we define "$p' \Rrightarrow p'$"


$\{\}$
$l:$
be e
$\{\}$
$\{\forall x.\ \mathsf{liveX}(x)\}$
$\Downarrow$
$\{\}$
$\{x\}$
$\{(\forall y \neq x.\ \mathsf{liveX}(y)) \wedge \neg\mathsf{liveX}(x)\}$
$l_s:$
$r := [x]$
$l'_s:$
$[x] := 2$
$\{x\}$
$\{\}$

By the meaning of the "$p \Rrightarrow p'$", we have

$$\mathsf{liveX}(e) \Rrightarrow \neg\mathsf{liveX}(e) \qquad \mathsf{liveE}(e) \Rrightarrow \neg\mathsf{liveE}(e)$$

The Asgn-Ac rule checks whether the analysis results for the assignment instruction is correct. ALStaE(p, r) says that the assertion p is stable if we remove the register r from the *is*, which records the identifiers that are not live. It is because after $r := e$ is executed, the register r may become live after being written as shown below.

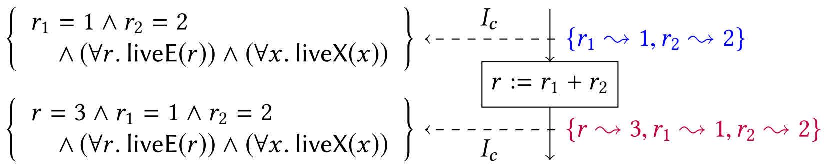

$\{r\}$
$\{\neg\mathsf{liveE}(r) \wedge \mathsf{liveE}(r_1) \wedge \mathsf{liveE}(r_2) \wedge r = -\}$
$r := r_1{+}r_2$
$\{r_1, r_2\}$
$\{\mathsf{liveE}(r) \wedge \neg\mathsf{liveE}(r_1) \wedge \neg\mathsf{liveE}(r_2) \wedge r = r_1{+}r_2\}$
$[x] := r$

$$\text{Let} \quad p = \{\mathsf{liveE}(r_1) \wedge \mathsf{liveE}(r_2) \wedge r = r_1{+}r_2\}, \quad \text{and} \quad p' = \{\neg\mathsf{liveE}(r) \wedge \mathsf{liveE}(r_1) \wedge \mathsf{liveE}(r_2) \wedge r = r_1{+}r_2\}$$

$$\text{We can prove that} \quad \mathsf{ALStaE}(p, r) \quad \text{and} \quad (p \wedge \mathsf{liveE}(r)) \Rrightarrow p'$$

$$\text{Thus, we have} \quad \vdash \{p[(r_1{+}r_2)/r] \wedge \mathsf{liveE}(r_1{+}r_2)\}\ \langle r := r_1{+}r_2\rangle\ \{p'\}$$

For the example shown above, the registers r_1 and r_2 may become not live after the execution of "$r := r_1{+}r_2$", since they will not be read before the next updating in the following execution. We give an example below to show the use of the Asgn-Ac rule.


$\left\{ \begin{array}{l} r_1 = 1 \land r_2 = 2 \\ \quad \land (\forall r.\, \mathsf{liveE}(r)) \land (\forall x.\, \mathsf{liveX}(x)) \end{array} \right\}$
I_c
$\{r_1 \rightsquigarrow 1, r_2 \rightsquigarrow 2\}$
$r := r_1 + r_2$
$\left\{ \begin{array}{l} r = 3 \land r_1 = 1 \land r_2 = 2 \\ \quad \land (\forall r.\, \mathsf{liveE}(r)) \land (\forall x.\, \mathsf{liveX}(x)) \end{array} \right\}$
$\{r \rightsquigarrow 3, r_1 \rightsquigarrow 1, r_2 \rightsquigarrow 2\}$
I_c

$$\text{let} \quad p = p' = \left\{ \begin{array}{l} r = 3 \land r_1 = 1 \land r_2 = 2 \\ \quad \land (\forall r.\, \mathsf{liveE}(r)) \land (\forall x.\, \mathsf{liveX}(x)) \end{array} \right\}$$

$$\text{Thus, we can prove} \quad \vdash \{p[(r_1 + r_2)/r]\}\ \langle r := r_1 + r_2 \rangle_l\ \{p'\}$$

Handling the liveness analysis. Below the definition of the *liveness* taken from "Data Flow Analysis: Theory and Practice" [?].

> *A pointer is live at a program point u if the address that it holds at u is read along some path starting at u.*

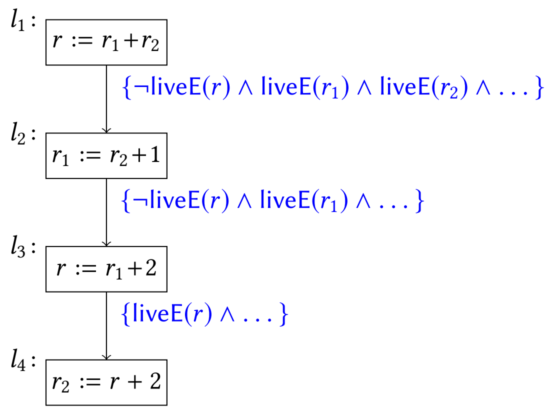
Similarily, in our work, a register or a memory location is live at a program point u, if it is read along some path starting at u.

An important use of the *liveness analysis* is in the *dead code elimination* optimization. The *liveness analysis* first caculates whether a register or a memory location may be live at some program point. Then, the code transformation in the *dead code elimination* eliminates the redundant assignment if its target is not live.

The *dead code elimination* optimization that we hope to prove uses the information that a register or a memory location is not live. So, in our work, as we have introduced, we define *is* representing the set of registers and memory locations that are not live. So, the registers and memory locations, which are recorded in *is*, are entirely unused in the following execution of the program before the next writes to them.

$$\textit{(IdSet)} \quad \textit{is} \quad \in \quad \mathcal{P}(\textit{Addr} \cup \textit{Reg})$$

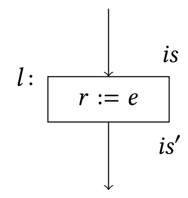
We give an example below to show how we describe that a register is not live after the execution of the instruction in the node labelled l_1.


$l_1:$
$r := r_1 + r_2$
$\{\neg\mathsf{liveE}(r) \land \mathsf{liveE}(r_1) \land \mathsf{liveE}(r_2) \land \dots\}$
$l_2:$
$r_1 := r_2 + 1$
$\{\neg\mathsf{liveE}(r) \land \mathsf{liveE}(r_1) \land \dots\}$
$l_3:$
$r := r_1 + 2$
$\{\mathsf{liveE}(r) \land \dots\}$
$l_4:$
$r_2 := r + 2$

We mark the register r is *not live* after the execution of the node l_1, shown as "$\neg\mathsf{liveE}(r)$". The assertion "$\neg\mathsf{liveE}(r)$" says that the register r is in the *not live* set *is*.

$$\forall M, R, is.\ (M, R, is) \models \neg\mathsf{liveE}(r) \implies r \in is.$$

The instruction $r_1 := r_2{+}1$ in the node labelled l_2 should not read the register r, according to the meaning the set *is*, which records the set of registers and memory locations that are not live. So, before the execution of the instruction "$r_1 := r_2{+}1$" in the node labelled l_2. We need to ensure that the register r_2 that will be read is *live* to ensure that the exeuction of such instruction will not read the registers in the set *is*. The register r can be removed from the set *is* after the execution of the instruction "$r := r_1{+}2$" in the node labelled l_3.



Relation between *is* and *is′* established in our work :

$$\frac{\forall r' \in is.\ r' \notin \mathsf{fv}(e) \wedge (r' \neq r \implies r' \in is')}{is \ltimes^{r} is'}$$

Relation between *is* and *is′* in the liveness analysis :

$$is = (is' \cup \{r\}) \backslash \mathsf{fv}(e)$$

Fig. 5. Handling the liveness analysis

We use Fig. 5 to illustrate how we handle the liveness analysis. The *liveness analysis* is a backward analysis, which caculates *is*, the set of not liveness identifiers, according to *is′*, the set of liveness identifiers. It caculates the set of not liveness identifiers before the execution of the instruction "$r := e$" based on three facts.

(1) the register r written by the instruction "$r := e$" is not live before its execution, if r is not read by this instruction;
(2) the registers read by the instruction "$r := e$" should be live before its execution.
(3) Except for the register r written, the registers that may be live after the execution the instruction "$r := e$" still may be live before the execution of this instruction, as shown below.

$$\forall r' \notin is' \implies (r' \notin is \vee r = r')$$

It explains why the caculation in the *liveness analysis* can only add the register r in the not live set.

We explain that how we present the three facts shown above in our (Asgn-Ac) rule below.

(1) We relax the fact of the item (1) shown above. we do not care that whether the register r is live or not before the exeuction of the instruction $r := e$, if r is not read by this instruction. It can be find from the precondition "$p[e/r] \wedge \mathsf{liveE}(e)$" and the requirement "$\mathsf{ALStaE}(p, r)$" (defined in Fig. 3) in our (Asgn-Ac) rule.
(2) As the for the item (2) shown above, the requirement that the register read by the instruction "$r := e$" is live before its execution is shown in the pre-condition "$\mathsf{liveE}(e)$".
(3) As for the item (3) shown above, we present it in its converse-negative form. Except for the register r written, the registers that are not live before the execution of the instruction "$r := e$" are still not live after its execution.

$$\forall r' \in (is \backslash \{r\}).\ r' \in is$$

In our (Asgn-Ac) rule, we show such requirement as "$p \wedge \mathsf{liveE}(r) \Rrightarrow p'$" ($p \Rrightarrow p'$ is defined in Fig. 3).

In mathmatically, we can find that the relation established between *is* and *is′* by the liveness analysis implies the relation between the *is* and *is′* ensured by the (Asgn-Ac) rule.

$$is = (is' \cup \{r\}) \backslash \mathsf{fv}(e) \implies is \ltimes^{r} is'$$

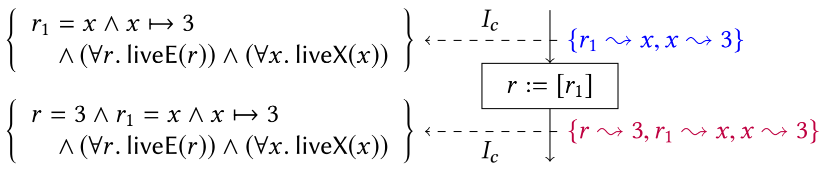
First, we can prove that any register in *is* is not read by the execution of the instruction "$r := e$".

$$\forall r' \in (is' \cup \{r\}) \backslash \mathsf{fv}(e).\ r' \notin \mathsf{fv}(e)$$

Second, registers that are in *is* and not written by the instruction "$r := e$" (not equal to r) are still in is'.

$$\forall r' \in (is' \cup \{r\}) \backslash \mathsf{fv}(e).\ r \neq r' \implies r' \in is'$$

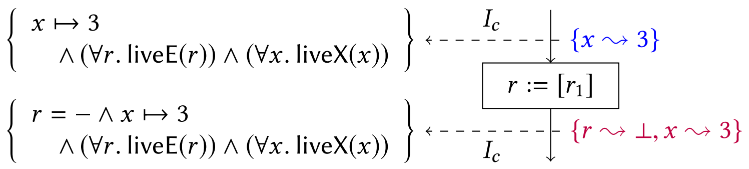
The Ld-Ac rule checks whether the analysis results for the memory location instruction is correct. The rule for the memory load is a little difficult, since we need to handle two different conditions. *First*, the precondition describes the specific memory location to load. We use the example below to show such condition.


$\left\{ \begin{array}{l} r_1 = x \wedge x \mapsto 3 \\ \quad \wedge (\forall r.\ \mathsf{liveE}(r)) \wedge (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$
I_c
$\{r_1 \rightsquigarrow x, x \rightsquigarrow 3\}$
$r := [r_1]$
$\left\{ \begin{array}{l} r = 3 \wedge r_1 = x \wedge x \mapsto 3 \\ \quad \wedge (\forall r.\ \mathsf{liveE}(r)) \wedge (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$
$\{r \rightsquigarrow 3, r_1 \rightsquigarrow x, x \rightsquigarrow 3\}$
I_c

$$\text{let} \quad p = p' = \left\{ \begin{array}{l} r = 3 \wedge r_1 = x \wedge x \mapsto 3 \\ \quad \wedge (\forall r.\ \mathsf{liveE}(r)) \wedge (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$$

Thus, we can prove $\vdash \{((r_1 \mapsto 3) \oplus p[3/r]) \wedge \mathsf{liveE}(r) \wedge \mathsf{liveX}(r_1)\}\ \langle r := [r_1] \rangle_l\ \{p'\}$

The other condition is that the precondition does not describe the specific memory location to load. We use the example below to show such condition.


$\left\{ \begin{array}{l} x \mapsto 3 \\ \quad \wedge (\forall r.\ \mathsf{liveE}(r)) \wedge (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$
I_c
$\{x \rightsquigarrow 3\}$
$r := [r_1]$
$\left\{ \begin{array}{l} r = - \wedge x \mapsto 3 \\ \quad \wedge (\forall r.\ \mathsf{liveE}(r)) \wedge (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$
$\{r \rightsquigarrow \bot, x \rightsquigarrow 3\}$
I_c

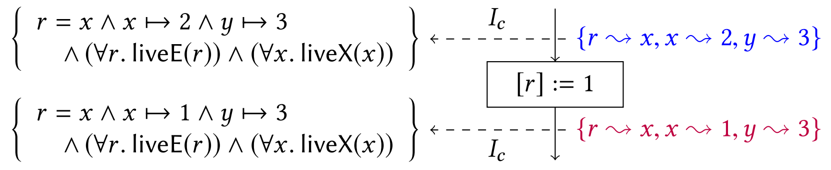
$$\text{let} \quad p = p' = \left\{ \begin{array}{l} r = - \wedge x \mapsto 3 \\ \quad \wedge (\forall r.\ \mathsf{liveE}(r)) \wedge (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$$

Thus, we can prove $\vdash \{((r_1 \mapsto -) \oplus p[-/r]) \wedge \mathsf{liveE}(r) \wedge \mathsf{liveX}(r_1)\}\ \langle r := [r_1] \rangle_l\ \{p'\}$

Here, the assertion "$(e_1 \mapsto e_2) \oplus p$" says that either "$(e_1 \mapsto e_2) \wedge p$" holds or "$(e_1 \mapsto e_2) \mathrel{-\!\!*} p$" holds. Sometimes, we do not know whether the memory location specified by the expression e_1 is depicted in the precondition. From the semantics of $p_1 \oplus p_2$, we have

$$\forall p, e. p \Rightarrow (e \mapsto -) \oplus p.$$

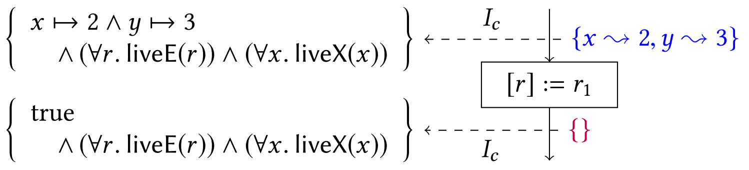
The St-Ac rule checks whether the analysis results for the memory store instruction is correct. It needs to handle two different conditions. *First*, the precondition describes the specific memory location to write to.


$r = x \land x \mapsto 2 \land y \mapsto 3 \land (\forall r.\ \mathsf{liveE}(r)) \land (\forall x.\ \mathsf{liveX}(x))$
I_c
$\{r \rightsquigarrow x, x \rightsquigarrow 2, y \rightsquigarrow 3\}$
$[r] := 1$
$r = x \land x \mapsto 1 \land y \mapsto 3 \land (\forall r.\ \mathsf{liveE}(r)) \land (\forall x.\ \mathsf{liveX}(x))$
$\{r \rightsquigarrow x, x \rightsquigarrow 1, y \rightsquigarrow 3\}$
I_c

$$\text{let}\quad p = \left\{ \begin{array}{l} r = x \land y \mapsto 3 \\ \quad \land (\forall r.\ \mathsf{liveE}(r)) \land (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\},\ p' = \left\{ \begin{array}{l} r = x \land x \mapsto 1 \land y \mapsto 3 \\ \quad \land (\forall r.\ \mathsf{liveE}(r)) \land (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$$

$$\text{Thus, we can prove}\quad \vdash \{((r \mapsto -) \oplus (r \mapsto - * p)) \land \mathsf{liveE}(r)\}\ \langle [r] := 1 \rangle_l\ \{p'\}$$

Second, the precondition does not describe the specific memory location to write to.

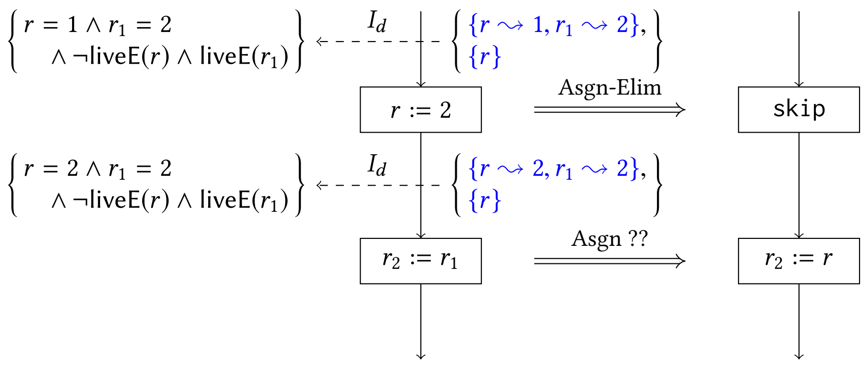

$x \mapsto 2 \land y \mapsto 3 \land (\forall r.\ \mathsf{liveE}(r)) \land (\forall x.\ \mathsf{liveX}(x))$
I_c
$\{x \rightsquigarrow 2, y \rightsquigarrow 3\}$
$[r] := r_1$
true $\land (\forall r.\ \mathsf{liveE}(r)) \land (\forall x.\ \mathsf{liveX}(x))$
$\{\}$
I_c

$$\text{let}\quad p = p' = \left\{ \begin{array}{l} \text{true} \\ \quad \land (\forall r.\ \mathsf{liveE}(r)) \land (\forall x.\ \mathsf{liveX}(x)) \end{array} \right\}$$

$$\text{Thus, we can prove}\quad \vdash \{((r \mapsto -) \oplus (r \mapsto - * p)) \land \mathsf{liveE}(r)\}\ \langle [r] := r_1 \rangle_l\ \{p'\}$$

Well-formed Translater. We can use the Link rule to divide the correctness proof of the code transformation to two disjoint parts, and use the Ins-Trans rule to check the correctness of tranfering an instruction in the source code to the target code.

The Asgn rule is used to simplify the expression e. The reason, why we require that the expression e' is live ($\mathsf{liveE}(e')$), is that registers marked not live may have different values in the source and target programs. Because the elimination rules (Asgn-Elim, Ld-Elim) will eliminate the write operations to registers, which is not live after the modification. We use an example below to show such condition.


$r = 1 \land r_1 = 2 \land \neg\mathsf{liveE}(r) \land \mathsf{liveE}(r_1)$
I_d
$\{r \rightsquigarrow 1, r_1 \rightsquigarrow 2\}, \{r\}$
$r := 2$
Asgn-Elim
skip
$r = 2 \land r_1 = 2 \land \neg\mathsf{liveE}(r) \land \mathsf{liveE}(r_1)$
I_d
$\{r \rightsquigarrow 2, r_1 \rightsquigarrow 2\}, \{r\}$
$r_2 := r_1$
Asgn ??
$r_2 := r$

Assuming that before the source executing "$r := 2$" and the target executing "$r_2 := r_1$", the same registers in the source and target programs have the same values. Since the register r is not live after "$r := 2$", we can use the Asgn-Elim rule to eliminate "$r := 2$" and cause r in the source and target programs having different values. Although we know that "$r = r_1$" holds in the source program before executing "$r_2 := r_1$", we can not replace "$r_2 := r_1$" with $r_2 := r$.

The other transformation rules are similar with the Asgn rule and the Asgn-Elim rule. The Ld-Elim2 rule is used to handle the elimination of the redundant reads in the *constant propagation* and the *common subexpression elimination*. The Ld-Elim rule and the St-Elim rule are used to handle the elimination of the redundant reads and the redundant writes respectively in the *dead code elimination*.

5.1 Soundness of the Transformation Rules

Definition 5.1 (Semantics of the well-formed analysis).

$$\begin{array}{rcl} I, C \models_{ge,\mathsf{f}} A & ::= & (\forall M, R, \mathsf{pc}, M', R', \mathsf{pc}', is. \\ & & \quad (M, R, is) \models I(\mathbb{L}_i(\mathsf{pc})) \land C \vdash (M, R, \mathsf{pc}) \longrightarrow (M', R', \mathsf{pc}') \\ & & \quad \Longrightarrow \exists is'. \, is \ltimes_R^{C(\mathsf{pc})} is' \land (M', R', is') \models I(\mathbb{L}_i(\mathsf{pc}'))) \\ & & \land (\exists is. \, \mathsf{Init}(ge), is \models I(\mathbb{L}_i(\mathsf{f}))) \\ & & \text{where } \; A(\mathsf{pc}) = (\mathbb{L}_i, \mathbb{L}_o) \end{array}$$

Here, "$is \ltimes_R^i is'$" describes how the live identifier set is changes. We give its formal definition below.

$$\frac{\begin{array}{c} \forall r \in is. \, r \notin \mathsf{usedR}(i) \land (r \notin \mathsf{updR}(i) \implies r \in is') \\ \forall x \in is. \, x \notin \mathsf{ldM}(i, R) \land (x \notin \mathsf{stM}(i, R) \implies x \in is') \end{array}}{is \ltimes_R^i is'}$$

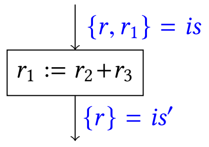
$$\mathsf{usedR}(i) \; ::= \; \begin{cases} \mathsf{fv}(e) & \textit{if } i \in \{\langle r := e\rangle_l, \langle r := [e]\rangle_l, \langle \mathsf{be}\ e\rangle_{l_1,l_2}\} \\ \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) & \textit{if } i = \langle [e_1] := e_2\rangle_l \\ \{r\} & \textit{if } i = \langle \mathtt{print}\ r\rangle \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\mathsf{updR}(i) \; ::= \; \begin{cases} \{r\} & \textit{if } i \in \{\langle r := e\rangle_l, \langle r := [e]\rangle_l\} \\ \emptyset & \textit{otherwise} \end{cases}$$

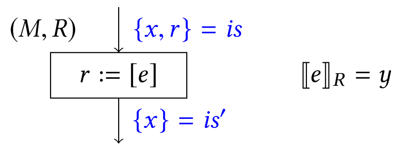
$$\mathsf{ldM}(i, R) \; ::= \; \begin{cases} \{[\![e]\!]_R\} & \textit{if } i = \langle r := [e]\rangle_l \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\mathsf{stM}(i, R) \; ::= \; \begin{cases} \{[\![e_1]\!]_R\} & \textit{if } i = \langle [e_1] := e_2\rangle_l \\ \emptyset & \textit{otherwise} \end{cases}$$

We use the following example to illustrate the meaning of the "$is \ltimes_R^i is'$". Here, the is is the subset of the registers and memory locations that are not live.



As the example shown above, if the register r is in is before the execution of the instruction "$r_1 := r_1 + r_2$", the execution of the instruction "$r_1 := r_1 + r_2$" does not read r and it is still not live after the execution of such instruction.



Similarly, if a memory location x is in is before the execution of the instruction "$r := [e]$", the execution of the instruction "$r := [e]$" does not read the memory location x and it is still not live after the execution of such instruction.

Then, we define the semantics of the well-formed translater.

Definition 5.2 (Semantics of the well-formed translater).

$$I, A \models C \sim C' \ ::= \ \forall ge, \mathsf{f}.\ I, C \models_{ge,\mathsf{f}} A \implies C' \preccurlyeq^{A,I} C$$

We define the simulation relation "$C' \preccurlyeq^{A,I} C$" below. Before we give the simulation relation, we first define the state relation between the source and target program states.

Definition 5.3 (State relation).

$$p, is \models (M, R) \sim (M', R') \quad ::= \quad (M, R, is) \models p \wedge (\forall r \notin is.\ R(r) = R'(r)) \wedge (\forall x \notin is.\ M(x) = M'(x))$$

Definition 5.4 (Simulation relation). $C' \preccurlyeq^{A,I} C$ holds, where $A = (\mathbb{L}_i, \mathbb{L}_o)$, if and only if the followings are true: for all $M, R, \mathsf{pc}, M_1, R_1, M_1', R_1', \mathsf{pc}'$ and is, if $I(\mathbb{L}_i(\mathsf{pc}))$, $is \models (M_1, R_1) \sim (M, R)$ and $C' \vdash (M_1, R_1, \mathsf{pc}) \longrightarrow (M_1', R_1', \mathsf{pc}')$, then

- either, there exists M', R' and is', such that
$$C \vdash (M, R, \mathsf{pc}) \longrightarrow (M', R', \mathsf{pc}') \ \text{ and } \ I(\mathbb{L}_i(\mathsf{pc}')), is' \models (M', R') \sim (M_1', R_1');$$
- or, $C \vdash (M, R, \mathsf{pc}) \longrightarrow$ **abort**.

Lemma 5.5 (Soundness of the well-formed analysis).

$$I, C \vdash_{ge,\mathsf{f}} A \implies I, C \models_{ge,\mathsf{f}} A$$

Lemma 5.6 (Soundness of the well-formed translater).

$$I, A \vdash C \sim C' \implies I, A \models C \sim C'$$

Proof. Prove by induction on "$I, A \vdash C \sim C'$".

- If $C = \{l \rightsquigarrow i\}$ and $C' = \{l \rightsquigarrow i'\}$, we have
$$I, A \vdash \{l \rightsquigarrow i\} \sim \{l \rightsquigarrow i'\} \tag{a1}$$
Let $A = (\mathbb{L}_i, \mathbb{L}_o)$ and we unfold (a1). We have
$$I(\mathbb{L}_i(l)), I(\mathbb{L}_o(l)) \vdash i \sim i' \tag{a1.1}$$
We do inversion on (a1.1) and discuss each case. Here, we show that proof of the two cases: (1) $i = \langle r := e \rangle_l$ and $i' = \langle r := e' \rangle_l$; (2) $i = \langle r := e \rangle_l$ and $i' = \langle \mathtt{skip} \rangle_l$.

(1) We have "$p, p' \vdash \langle r := e \rangle_l \sim \langle r := e' \rangle_l$", where $p = I(\mathbb{L}_i(l))$ and $p' = I(\mathbb{L}_o(l))$. From the (Asgn) rule, we have

$$p \Rightarrow (e = e') \wedge \mathsf{liveE}(e') \tag{a1.2}$$

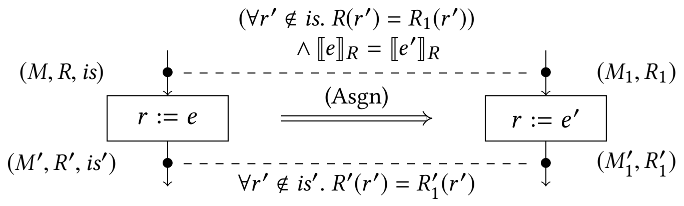
And we need to prove the following holds

$$I, (\mathbb{L}_i, \mathbb{L}_o) \models \{l \rightsquigarrow i\} \sim \{l \rightsquigarrow i'\} \tag{g1.1}$$

We unfold (g1.1) by Def. 5.2. We need to prove the following holds:

$$I, \{l \rightsquigarrow i\} \models (\mathbb{L}_i, \mathbb{L}_o) \implies \{l \rightsquigarrow i'\} \preccurlyeq^{(\mathbb{L}_i, \mathbb{L}_o), I} \{l \rightsquigarrow i\} \tag{g1.2}$$

We can prove that (g1.2) holds, since the results of the expression e' in source and target program states equal, and, except for the register r assigned, the registers that are in is are also in is'. We use the following figure to illustrate the fact that the values of the register r assigned equal between the source and target program states, and, except the register r, the other registers that do not equal between the source and target program states before exeuction also do not equal after execution.



$(\forall r' \notin is.\ R(r') = R_1(r'))$
$\wedge [\![e]\!]_R = [\![e']\!]_R$
(M, R, is)
(M_1, R_1)
$r := e$
(Asgn)
$r := e'$
(M', R', is')
(M'_1, R'_1)
$\forall r' \notin is'.\ R'(r') = R'_1(r')$

$(is \backslash r) \subseteq is'$

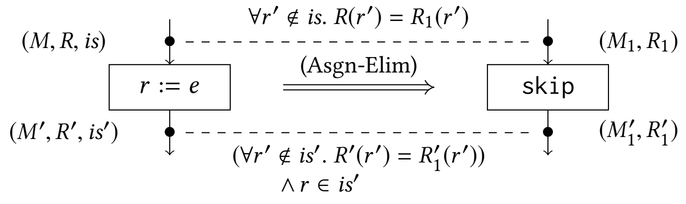
(2) We have "$p, p' \vdash \langle r := e \rangle_l \sim \langle \mathtt{skip} \rangle_l$", where $p = I(\mathbb{L}_i(l))$ and $p' = I(\mathbb{L}_o(l))$. From the (Asgn-Elim) rule, we have

$$p' \Rightarrow \neg\mathsf{liveE}(r) \tag{a1.3}$$

We need to prove the following holds:

$$I, \{l \rightsquigarrow i\} \models (\mathbb{L}_i, \mathbb{L}_o) \implies \{l \rightsquigarrow i'\} \preccurlyeq^{(\mathbb{L}_i, \mathbb{L}_o), I} \{l \rightsquigarrow i\} \tag{g1.3}$$

We can prove that (g1.3) holds, since the registers that are in is are also in is' and the register r assigned in the source program is also in is'.



$(is \backslash r) \subseteq is' \wedge r \in is' \implies is \subseteq is'$

It means that the registers that do not equal between the source and target program states before the execution also do not equal after execution. The register r also do not require to equal after execution.

- If $C = (C_1 \uplus C_2)$ and $C' = (C'_1 \uplus C'_2)$, we have

$$I, A \vdash (C_1 \uplus C_2) \sim (C'_1 \uplus C'_2) \tag{a2}$$

The below are the inductive hypotheses.

$$I, A \vdash C_1 \sim C'_1 \implies I, A \models C_1 \sim C'_1 \tag{a2.1}$$

$$I, A \vdash C_2 \sim C'_2 \implies I, A \models C_2 \sim C'_2 \tag{a2.2}$$

We unfold (a2) according to the (Link) rule, we have

$$I, A \vdash C_1 \sim C'_1 \tag{a2.3}$$

$$I, A \vdash C_2 \sim C'_2 \tag{a2.4}$$

By applying (a2.1) on (a2.3) and (a2.2) on (a2.4), we have

$$I, A \models C_1 \sim C'_1 \tag{a2.5}$$

$$I, A \models C_2 \sim C'_2 \tag{a2.6}$$

We need to prove

$$I, A \models (C_1 \uplus C'_1) \sim (C_2 \sim C'_2) \tag{g2}$$

Since we have the following lemmas

$$\forall I, C, C', A, ge, \mathsf{f}.\ (I, C \models_{ge,\mathsf{f}} A \land C' \subseteq C) \implies I, C' \models_{ge,\mathsf{f}} A$$

and

$$\forall C_1, C'_1, C_2, C'_2, A, I.\ (C'_1 \preccurlyeq^{A,I} C_1 \land C'_2 \preccurlyeq^{A,I} C_2) \implies (C'_1 \uplus C'_2) \preccurlyeq^{A,I} (C_1 \uplus C_2)$$

hold, we finish the proof of (g2).

□

Lemma 5.7 (Simulation ensures refinement).

$$(I, C \models_{ge,\mathsf{f}} A \land I, A \models C \sim C') \implies C' \subseteq_{ge,\mathsf{f}} C$$

Thus, we can prove the Lemma. 2.4 by applying Lemma. 5.5, Lemma. 5.6 and Lemma. 5.7.

6 PROOF OF THE OPTIMIZATION CORRECTNESS

6.1 Correctness proof: Constant propagation

Analyzer. The implementation of the *value analysis* is shown below.

```
Val_Analyzer($C$, f) :
1.    for ($\forall l \in \mathrm{dom}(C)$) :
2.        $\mathbb{L}_o(l) = \emptyset$
3.    while (any $\mathbb{L}_o(l)$ changes occur) :
4.        for ($\forall l \in \mathrm{dom}(C)$) :
5.            $\mathbb{L}_i(l) = \bigcap_{l_p \in \mathsf{pred}(C,l)} \mathbb{L}_o(l_p)$
6.            $\mathbb{L}_o(l) = f(\mathbb{L}_i(l), C(l))$
7.    return $(\mathbb{L}_i, \mathbb{L}_o)$
```

$$\mathsf{Analyzer}(C, ge, \mathsf{f}) = \mathsf{Val\_Analyzer}(C, \mathsf{f})$$

Below, we give the implememtation of the transfer function in the value analysis.

$$\begin{aligned}
f(\langle r := e \rangle_l, a_v) \quad &::= \quad a_v\{r \rightsquigarrow [\![e]\!]_{a_v}\} \\
f(\langle r := [e] \rangle_l, a_v) \quad &::= \quad \begin{cases} a_v\{r \rightsquigarrow \bot\} & \textit{if } [\![e]\!]_{a_v} = \bot \\ a_v\{r \rightsquigarrow a_v(x)\} & \textit{if } [\![e]\!]_{a_v} = x \\ a_v\{r \rightsquigarrow \top\} & \textit{otherwise} \end{cases} \\
f(\langle [e] := e' \rangle_l, a_v) \quad &::= \quad \begin{cases} a_v\{x \rightsquigarrow [\![e']\!]_{a_v}\} & \textit{if } [\![e]\!]_{a_v} = x \\ a_v \backslash Addr & \textit{otherwise} \end{cases}
\end{aligned}$$

Then, we define the merging of two abstract interpretations in the value analysis.

$$\begin{aligned}
a_v \cap a'_v \quad &::= \quad \{x \rightsquigarrow \hat{v} \mid x \in (\mathrm{dom}(a_v) \cap \mathrm{dom}(a'_v)) \wedge \hat{v} = (a_v(x) \cap a'_v(x))\} \\
&\qquad \cup \{r \rightsquigarrow \hat{v} \mid r \in (\mathrm{dom}(a_v) \cap \mathrm{dom}(a'_v)) \wedge \hat{v} = (a_v(r) \cap a'_v(r))\} \\
\hat{v}_1 \cap \hat{v}_2 \quad &::= \quad \begin{cases} \hat{v}_1 & \textit{if } \hat{v}_1 = \hat{v}_2 \\ \top & \textit{otherwise} \end{cases}
\end{aligned}$$

Translater. Below, we give an implememtation of the translater in *constant propagation* optimizer.

$$\mathsf{Translater}(C, A) \quad ::= \quad \begin{cases} \{l_1 \rightsquigarrow \langle r := v \rangle_{l_2}\} \uplus \mathsf{Translater}(C', A) & \textit{if } C = \{l_1 \rightsquigarrow \langle r := e \rangle_{l_2}\} \uplus C',\ [\![e]\!]_{\mathbb{L}_i(l_1)} = v \\ \{l_1 \rightsquigarrow \langle r := v \rangle_{l_2}\} \uplus \mathsf{Translater}(C', A) & \textit{elif } C = \{l_1 \rightsquigarrow \langle r := [e] \rangle_{l_2}\} \uplus C', \\ & \qquad [\![e]\!]_{\mathbb{L}_i(l_1)} = x,\ \mathbb{L}_i(l_1)(x) = v \\ \{l \rightsquigarrow i\} \uplus \mathsf{Translater}(C', A) & \textit{elif } C = \{l \rightsquigarrow i\} \uplus C' \\ \emptyset & \textit{elif } C = \emptyset \\ \mathsf{undef} & \textit{otherwise} \end{cases}$$

where $A = (\mathbb{L}_i, \mathbb{L}_o)$

Correctness proof. First, we define a converter I_c to convert the result of the value analysis to the assertion form.

$$\begin{aligned} I_c(a_v) \quad ::= \quad & (\forall r \in \mathrm{dom}(a_v).\,(\exists v.\, r = v \wedge a_v(r) = v) \vee (r = - \wedge a_v(r) \in \{\bot, \top\})) \\ & \quad \wedge (\forall x \in \mathrm{dom}(a_v).\,(\exists v.\, x \mapsto v \wedge a_v(x) = v) \vee (x \mapsto - \wedge a_v(x) \in \{\bot, \top\})) \\ & \wedge (\forall r.\, \mathsf{liveE}(r)) \wedge (\forall x.\, \mathsf{liveX}(x)) \end{aligned}$$

Then, we proof the correctness of the Lemma. 6.1.

Lemma 6.1 (Optimizer Correctness: Constant propagation).

$$\forall C, ge, \mathsf{f}, A, C'.\ (\mathsf{Analyzer}(C, ge, \mathsf{f}) = A \wedge \mathsf{Translater}(C, A) = C') \implies I_c \vdash_{ge,\mathsf{f}} (C, A) \sim C'$$

We divide the correctness proof the Lemma. 6.1 into two lemmas, shown as Lemma. 6.2 and Lemma. 6.3, respectively.

Lemma 6.2. $\forall C, ge, \mathsf{f}.\ \mathsf{Analyzer}(C, ge, \mathsf{f}) = A \implies I_c, C \vdash_{ge,\mathsf{f}} A$.

Proof. Let $A = (\mathbb{L}_i, \mathbb{L}_o)$ and we apply the (wf-Analysis) rule on the proof goal. Then, we need to prove the following subgoals.

$$\exists is.\ (\mathsf{Init}(ge), is) \models I_c(\mathbb{L}_i(\mathsf{f})) \tag{g1}$$

$$\begin{aligned} & \text{for all } l \in \mathrm{dom}(C): \\ & \quad I_c(\mathbb{L}_o(l)) \Rrightarrow I_c(\mathbb{L}_i(l_s)),\ \text{ for all } l_s \in \mathrm{succ}(C(l)) \\ & \quad \vdash \{I_c(\mathbb{L}_i(l))\}\ C(l)\ \{I_c(\mathbb{L}_o(l))\} \end{aligned} \tag{g2}$$

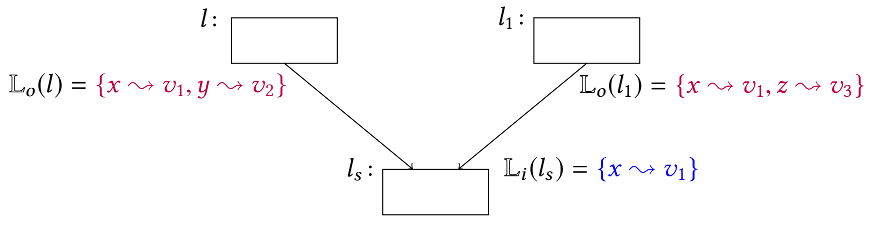
According to $\mathbb{L}_i(\mathsf{f}) = \emptyset$, we can prove that $(\mathsf{Init}(ge), \emptyset) \models (\forall r.\mathsf{liveE}(r)) \wedge (\forall x.\mathsf{liveX}(x))$. Thus, we prove the subgoal (g1).

For the subgoal (g2), we need to prove that for all $l \in \mathrm{dom}(C)$, the following subgoals hold:

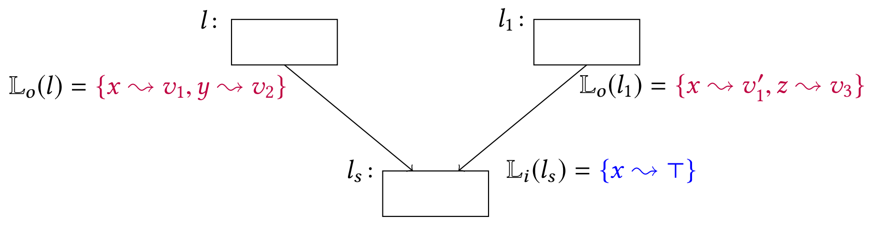
$$I_c(\mathbb{L}_o(l)) \Rrightarrow I_c(\mathbb{L}_i(l_s)),\ \text{for all } l_s \in \mathrm{succ}(C(l)) \tag{g2-1}$$

$$\vdash \{I_c(\mathbb{L}_i(l))\}\ C(l)\ \{I_c(\mathbb{L}_o(l))\} \tag{g2-2}$$

The correctness of the subgoal (g2-1) is straight-forward. Here, we omit the definition of merging two abstract interpretations. We use the figures shown below to illustrate the meaning of merging two abstract interpretations.


$l:$
$l_1:$
$\mathbb{L}_o(l) = \{x \rightsquigarrow v_1, y \rightsquigarrow v_2\}$
$\mathbb{L}_o(l_1) = \{x \rightsquigarrow v_1, z \rightsquigarrow v_3\}$
$l_s:$
$\mathbb{L}_i(l_s) = \{x \rightsquigarrow v_1\}$

$$\{x \rightsquigarrow v_1, y \rightsquigarrow v_2\} \cap \{x \rightsquigarrow v_1, z \rightsquigarrow v_2\} = \{x \rightsquigarrow v_1\}$$


$l:$
$l_1:$
$\mathbb{L}_o(l) = \{x \rightsquigarrow v_1, y \rightsquigarrow v_2\}$
$\mathbb{L}_o(l_1) = \{x \rightsquigarrow v_1', z \rightsquigarrow v_3\}$
$l_s:$
$\mathbb{L}_i(l_s) = \{x \rightsquigarrow \top\}$

$$\{x \rightsquigarrow v_1, y \rightsquigarrow v_2\} \cap \{x \rightsquigarrow v_1', z \rightsquigarrow v_2\} = \{x \rightsquigarrow \top\}$$

According to the definition of merging two abstract interpretations in the value analysis, we have, where $l_s \in \mathrm{succ}(C(l))$,

$$\forall r \in \mathrm{dom}(\mathbb{L}_i(l_s)).\ \mathbb{L}_o(l)(r) = \mathbb{L}_i(l_s)(r) \vee \mathbb{L}_i(l_s)(r) = \top$$
$$\mathrm{dom}(\mathbb{L}_i(l_s)) \subseteq \mathrm{dom}(\mathbb{L}_o(l))$$

Thus, we can prove that $I_c(\mathbb{L}_o(l)) \Rrightarrow I_c(\mathbb{L}_i(l_s))$ holds, and we finish the proof of the subgoal (g2-1).

We prove the subgoal (g2-2) according to the following conclusion.

$$\forall a_v, a'_v, i.\ f(i, a_v) = a'_v \implies \vdash \{I_c(a_v)\}\ i\ \{I_c(a'_v)\}$$

We prove the conclusion shown above by discussing each case of the instruction i:

- $i = \langle r := e\rangle_l$, and $a'_v = a_v\{r \rightsquigarrow [\![e]\!]_{a_v}\}$. We discuss the result of the $[\![e]\!]_{a_v}$.
  - there exists v, such that $[\![e]\!]_{a_v} = v$. We have the following conclusion.

$$\forall v.\ [\![e]\!]_{a_v} = v \implies (I_c(a_v) \Rightarrow e = v)$$

Let $p = (I_c(a_v\backslash r) \wedge r = v)$, then we have

$$\frac{I_c(a_v) \Rightarrow p[e/r] \wedge \mathsf{liveE}(e) \qquad \dfrac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow I_c(a'_v)}{\vdash \{p[e/r] \wedge \mathsf{liveE}(e)\}\ \langle r := e\rangle_l\ \{I_c(a'_v)\}}}{\vdash \{I_c(a_v)\}\ \langle r := e\rangle_l\ \{I_c(a'_v)\}}$$

We prove the "$p \wedge \mathsf{liveE}(r) \Rrightarrow I_c(a'_v)$" by the following steps.

$$\begin{array}{ll} & I_c(a_v\backslash r) \wedge r = e \wedge e = v \wedge \mathsf{liveE}(r) \\ \Rightarrow & I_c(a_v\backslash r) \wedge r = v \wedge \mathsf{liveE}(r) \\ \Rightarrow & I_c(a_v\{r \rightsquigarrow [\![e]\!]_{a_v}\}) \qquad (\text{* where } [\![e]\!]_{a_v} = v \text{ *}) \end{array}$$

  - $[\![e]\!]_{a_v} \in \{\bot, \top\}$. Let $p = (I_c(a_v\backslash r) \wedge r = -)$, then we have

$$\frac{I_c(a_v) \Rightarrow p[e/r] \wedge \mathsf{liveE}(e) \qquad \dfrac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow I_c(a'_v)}{\{p[e/r] \wedge \mathsf{liveE}(e)\}\ \langle r := e\rangle_l\ \{I_c(a'_v)\}}}{\vdash \{I_c(a_v)\}\ \langle r := e\rangle_l\ \{I_c(a'_v)\}}$$

- $i = \langle r := [e]\rangle_l$. We discuss the result of the $[\![e]\!]_{a_v}$.
  - there exists x, such that $[\![e]\!]_{a_v} = x$. We have $a'_v = a_v\{r \rightsquigarrow a_v(x)\}$. Here, we consider that condition that there exists v such that $a_v(x) = v$. The condition that $a_v(x) \in \{\bot, \top\}$ is similar. We have the following conclusion.

$$\forall x, v.\ ([\![e]\!]_{a_v} = x \wedge a_v(x) = v) \implies (I_c(a_v) \Rightarrow e \mapsto v)$$

Let $p = (I_c(a_v\backslash r) \wedge r = v)$, $p_1 = ((e \mapsto v) \oplus p[v/r]) \wedge \mathsf{liveE}(e) \wedge \mathsf{liveX}(e)$, then we have

$$\frac{I_c(a_v) \Rightarrow p_1 \qquad \dfrac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow I_c(a'_v)}{\vdash \{p_1\}\ \langle r := [e]\rangle_l\ \{I_c(a'_v)\}}}{\vdash \{I_c(a_v)\}\ \langle r := [e]\rangle_l\ \{I_c(a'_v)\}}$$

We prove the "$p \wedge \mathsf{liveE}(r) \Rrightarrow I_c(a'_v)$" by the following steps.

$$\begin{array}{ll} & (I_c(a_v\backslash r) \wedge r = v) \wedge \mathsf{liveE}(r) \\ \Rightarrow & I_c(a'_v) \qquad (\text{* where } a'_v = a_v\{r \rightsquigarrow v\} \text{ *}) \end{array}$$

- $[\![e]\!]_{a_v} = \top$. We have $a_v' = a_v\{r \rightsquigarrow \top\}$. The condition that $[\![e]\!]_{a_v} = \bot$ is similar.
  Let $p = (I_c(a_v \backslash r) \wedge r = -)$, $p_1 = ((e \mapsto -) \oplus p[-/r]) \wedge \mathsf{liveE}(e) \wedge \mathsf{liveX}(e)$, then we have

$$\frac{I_c(a_v) \Rightarrow p_1 \qquad \dfrac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow I_c(a_v')}{\vdash \{p_1\}\ \langle r := [e]\rangle_l\ \{I_c(a_v')\}}}{\vdash \{I_c(a_v)\}\ \langle r := [e]\rangle_l\ \{I_c(a_v')\}}$$

  We prove the "$p \wedge \mathsf{liveE}(r) \Rrightarrow I_c(a_v')$" by the following steps.

$$\begin{array}{ll} & (I_c(a_v \backslash r) \wedge r = -) \wedge \mathsf{liveE}(r) \\ \Rightarrow & (I_c(a_v \backslash r) \wedge r = -) \wedge \mathsf{liveE}(r) \\ \Rightarrow & I_c(a_v') \qquad (\text{* where } a_v' = a_v\{r \rightsquigarrow \top\} \text{ *}) \end{array}$$

- $i = \langle [e] := e' \rangle_l$. We discuss the result of the $[\![e]\!]_{a_v}$.
  - there exists x, such that $[\![e]\!]_{a_v} = x$. We have $a_v' = a_v\{x \rightsquigarrow [\![e']\!]_{a_v}\}$. We consider the condition that there exists v such that $[\![e']\!]_{a_v} = v$. The condition that $[\![e']\!]_{a_v} \in \{\bot, \top\}$ is similar.
    Let $p = (I_c(a_v \backslash x) \wedge e = x \wedge e' = v)$, and $p_1 = ((e \mapsto -) \oplus (e \mapsto - * p) \wedge \mathsf{liveE}(e_1, e_2))$. Since we can prove the following conclusion

$$\forall p.\ p \Rightarrow ((e \mapsto -) \oplus (e \mapsto - * p)),$$

    we have

$$\frac{I_c(a_v) \Rightarrow p_1 \qquad \dfrac{\mathsf{ALStaE}(p, e) \qquad (e \mapsto e' * p) \wedge \mathsf{liveX}(e) \Rrightarrow I_c(a_v')}{\{p_1\}\ \langle [e] := e' \rangle_l\ \{I_c(a_v')\}}}{\vdash \{I_c(a_v)\}\ \langle [e] := e' \rangle_l\ \{I_c(a_v')\}}$$

    We prove "$(e \mapsto e' * p) \wedge \mathsf{liveX}(e) \Rrightarrow I_c(a_v')$" by the following steps.

$$\begin{array}{ll} & (e \mapsto e' * (I_c(a_v \backslash x) \wedge e = x \wedge e' = v)) \wedge \mathsf{liveX}(e) \\ \Rightarrow & I_c(a_v \backslash x) \wedge e \mapsto e' \wedge e = x \wedge e' = v \wedge \mathsf{liveX}(e) \\ \Rightarrow & I_c(a_v \backslash x) \wedge x \mapsto v \wedge \mathsf{liveX}(x) \\ \Rightarrow & I_c(a_v) \end{array}$$

  - $[\![e]\!]_{a_v} \in \{\bot, \top\}$. We have $a_v' = a_v \backslash Addr$.
    Let $p = I_c(a_v \backslash Addr)$, and $p_1 = ((e \mapsto -) \oplus (e \mapsto - * p) \wedge \mathsf{liveE}(e_1, e_2))$. And we have

$$\frac{I_c(a_v) \Rightarrow p_1 \qquad \dfrac{\mathsf{ALStaE}(p, e) \qquad (e \mapsto e' * p) \wedge \mathsf{liveX}(e) \Rrightarrow I_c(a_v')}{\{p_1\}\ \langle [e] := e' \rangle_l\ \{I_c(a_v')\}}}{\vdash \{I_c(a_v)\}\ \langle [e] := e' \rangle_l\ \{I_c(a_v')\}}$$

    We prove the "$(e \mapsto e' * p) \wedge \mathsf{liveX}(e) \Rrightarrow I_c(a_v')$" by the following steps.

$$\begin{array}{ll} & (e \mapsto e' * I_c(a_v \backslash Addr)) \wedge \mathsf{liveX}(e) \\ \Rightarrow & I_c(a_v \backslash Addr) \\ \Rightarrow & I_c(a_v') \end{array}$$

□

Lemma 6.3. $\forall C, A, C'.\ \mathsf{Translater}(C, A) = C' \implies I_c, A \vdash C \sim C'$.

Proof. We assume that the source code C is not an empty set. We prove this lemma by induction on C.

- If $C = \{l \rightsquigarrow i\}$, we have the following holds.
$$\mathsf{Translater}(\{l \rightsquigarrow i\}, A) = C' \tag{1}$$
We unfold (1) according to the definition of the translater. We discuss each case respectively and assume $A = (\mathbb{L}_i, \mathbb{L}_o)$.
  - $i = \langle r := e \rangle_{l_1}$ and there exists v such that $[\![e]\!]_{\mathbb{L}_i(l)} = v$. We get $C' = \{l \rightsquigarrow \langle r := v \rangle_{l_1}\}$. And we need to prove the following holds.
$$I_c, A \vdash \{l \rightsquigarrow \langle r := e \rangle_{l_1}\} \sim \{l \rightsquigarrow \langle r := v \rangle_{l_1}\} \tag{g3-1}$$
By applying (Ins-Trans) on (g3-1), we get
$$I_c(\mathbb{L}_i(l)), I_c(\mathbb{L}_o(l)) \vdash \langle r := e \rangle_{l_1} \sim \langle r := v \rangle_{l_1} \tag{g3-1.1}$$
By applying (Asgn) rule on (g3-1.1), we need to prove
$$I_c(\mathbb{L}_i(l)) \Rightarrow (e = v) \wedge \mathsf{liveE}(v) \tag{g3-1.2}$$
Since, we have the following conclusion
$$\forall v.\ [\![e]\!]_{a_v} = v \implies (I_c(a_v) \Rightarrow e = v),$$
we finish the proof of this case.
  - $i = \langle r := [e] \rangle_{l_1}$ and there exists x, v such that $[\![e]\!]_{\mathbb{L}_i(l)} = x$ and $\mathbb{L}_i(x) = v$. We get $C' = \{l \rightsquigarrow \langle r := v \rangle_{l_1}\}$. And we need to prove the following holds.
$$I_c, A \vdash \{l \rightsquigarrow \langle r := [e] \rangle_{l_1}\} \sim \{l \rightsquigarrow \langle r := v \rangle_{l_1}\} \tag{g3-2}$$
By applying (Ins-Trans) on (g3-2), we get
$$I_c(\mathbb{L}_i(l)), I_c(\mathbb{L}_o(l)) \vdash \langle r := [e] \rangle_{l_1} \sim \langle r := v \rangle_{l_1} \tag{g3-2.1}$$
By applying (Ld-Elim2) rule on (g3-2.1), we need to prove
$$I_c(\mathbb{L}_i(l)) \Rightarrow (e \mapsto v) \wedge \mathsf{liveE}(v) \tag{g3-2.2}$$
Since, we have the following conclusion
$$\forall x, v.\ ([\![e]\!]_{a_v} = x \wedge a_v(x) = v) \implies (I_c(a_v) \Rightarrow e \mapsto v),$$
we finish the proof of this case.
  - $C' = \{l \rightsquigarrow i\}$. And we need to prove the following holds.
$$I_c, A \vdash \{l \rightsquigarrow i\} \sim \{l \rightsquigarrow i\} \tag{g3-3}$$
The subgoal (g3-3) can be proved by discuss each case of the instruction i.
- If $C = \{l \rightsquigarrow i\} \uplus C_1$, we have the following holds.
$$\mathsf{Translater}(\{l \rightsquigarrow i\} \uplus C_1, A) = C' \tag{2}$$
And the inductive hypothesis is shown below.
$$\forall A, C_1'.\ \mathsf{Translater}(C_1, A) = C_1' \implies I_c, A \vdash C_1 \sim C_1' \tag{3}$$
We unfold (2) according to the definition of the translater. We prove by discussing each case respectively and assume $A = (\mathbb{L}_i, \mathbb{L}_o)$. Here, we only show the proof of the first case. The proof of the other cases are similar to the first one.

– $i = \langle r := e \rangle_{l_1}$ and there exists v such that $[\![e]\!]_{\mathbb{L}_i(l)} = v$. We get $C' = \{l \rightsquigarrow \langle r := v \rangle_{l_1}\} \uplus \mathsf{Translater}(C_1, A)$. Let $C'_1 = \mathsf{Translater}(C_1, A)$. We need to prove the following holds.

$$I_c, A \vdash (\{l \rightsquigarrow \langle r := e \rangle_{l_1}\} \uplus C_1) \sim (\{l \rightsquigarrow \langle r := v \rangle_{l_1}\} \uplus C'_1) \tag{g4}$$

By applying (Link) rule on (g4), we need to prove

$$I_c, A \vdash \{l \rightsquigarrow \langle r := e \rangle_{l_1}\} \sim \{l \rightsquigarrow \langle r := v \rangle_{l_1}\} \tag{g4-1}$$

$$I_c, A \vdash C_1 \sim C'_1 \tag{g4-2}$$

The subgoal (g4-1) can be proved by applying the (Ins-Trans) rule, and the subgoal (g4-2) can be proved by applying the inductive hypothesis (3).

□

6.2 Correctness proof: Dead code elimination

Analyzer. The implememtation of the *liveness analysis* is shown below.

```
Lv_Analyzer($C, (\mathbb{L}_{vi}, \mathbb{L}_{vo}), \mathsf{f}$) :
1.    if($\forall l \in \mathrm{dom}(C)$) :
2.        $\mathbb{L}_i(l) = \emptyset$
3.    while(any$\mathbb{L}_i(l)$changes occur) :
4.        for($\forall l \in \mathrm{dom}(C)$) :
5.            $\mathbb{L}_o(l) = \bigcap_{l_s \in \mathsf{succ}(C(l))} \mathbb{L}_i(l_s)$
6.            $\mathbb{L}_i(l) = f_{\mathrm{L}}(\mathbb{L}_{vi}(l), \mathbb{L}_o(l), C(l))$
7.    return ($\mathbb{L}_i, \mathbb{L}_o$)
```

Below, we give the implememtation of the transfer function in the liveness analysis.

$$f_{\mathrm{L}}(\langle r := e \rangle_l, a_v, a_l) \quad ::= \quad (a_l \cup \{r\}) \backslash \mathsf{fv}(e)$$

$$f_{\mathrm{L}}(\langle r := [e] \rangle_l, a_v, a_l) \quad ::= \quad \begin{cases} (a_l \cup \{r\}) \backslash (\mathsf{fv}(e) \cup \{x\}) & \textit{if } [\![e]\!]_{a_v} = x \\ (a_l \cup \{r\}) \backslash (\mathsf{fv}(e) \cup \textit{Addr}) & \textit{otherwise} \end{cases}$$

$$f_{\mathrm{L}}(\langle [e_1] := e_2 \rangle_l, a_v, a_l) \quad ::= \quad \begin{cases} (a_l \cup \{x\}) \backslash (\mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)) & \textit{if } [\![e_1]\!]_{a_v} = x \\ a_l \backslash (\mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)) & \textit{otherwise} \end{cases}$$

The merging of two abstract interpretations in the liveness analysis is just the intersection of two the sets.

$$a_v \cap a'_v \quad ::= \quad \{x \mid x \in a_v \wedge x \in a'_v\} \cup \{r \mid r \in a_v \wedge r \in a'_v\}$$

The Analyzer is defined below. It includes the value analysis and the liveness analysis.

$$\begin{aligned} \mathsf{Analyzer}(C, ge, \mathsf{f}) \quad ::= \quad & \{l \rightsquigarrow (L_i, L_o) \mid l \in \mathrm{dom}(C) \wedge L_i = (\mathbb{L}_{vi}(l), \mathbb{L}_{li}(l)) \wedge L_o = (\mathbb{L}_{vo}(l), \mathbb{L}_{lo}(l))\} \\ & \text{where } (L_{vi}, L_{vo}) = \mathtt{Val\_Analyzer}(C, \mathsf{f}),\ (L_{li}, L_{lo}) = \mathtt{Lv\_Analyzer}(C, (L_{vi}, L_{vo}), \mathsf{f}) \end{aligned}$$

Translater. Below, we give the implememtation of the translater in the *dead code elimination* optimizer.

$$\mathsf{Translater}(C, A) \quad ::= \quad \begin{cases} \{l_1 \rightsquigarrow \langle \mathtt{skip} \rangle_{l_2}\} \uplus \mathsf{Translater}(C', A) & \textit{if } C = \{l_1 \rightsquigarrow \langle r := e \rangle_{l_2}\} \uplus C', r \in a'_l, \\ & \quad \mathbb{L}_o(l) = (a'_v, a'_l) \\ \{l_1 \rightsquigarrow \langle \mathtt{skip} \rangle_{l_2}\} \uplus \mathsf{Translater}(C', A) & \textit{elif } C = \{l_1 \rightsquigarrow \langle r := [e] \rangle_{l_2}\} \uplus C', r \in a'_l, \\ & \quad \mathbb{L}_o(l) = (a'_v, a'_l) \\ \{l_1 \rightsquigarrow \langle \mathtt{skip} \rangle_{l_2}\} \uplus \mathsf{Translater}(C', A) & \textit{elif } C = \{l_1 \rightsquigarrow \langle [e_1] := e_2 \rangle_{l_2}\} \uplus C', x \in a'_l, \\ & \quad [\![e_1]\!]_{a_v} = x, \mathbb{L}_i = (a_v, a_l), \mathbb{L}_o = (a'_v, a'_l) \\ \{l \rightsquigarrow i\} \uplus \mathsf{Translater}(C', A) & \textit{elif } C = \{l_1 \rightsquigarrow i\} \uplus C' \\ \emptyset & \textit{elif } C = \emptyset \\ \mathrm{undef} & \textit{otherwise} \end{cases}$$
$$\text{where } A = (\mathbb{L}_i, \mathbb{L}_o)$$

Correctness proof. First, we define a converter I_l to convert the result of the analysis to the assertion form.

$$\begin{aligned} I'_v(a_v) \quad &::= \quad (\forall r \in \mathrm{dom}(a_v).\, (\exists v.\, r = v \wedge a_v(r) = v) \wedge (r = - \wedge a_v(r) \in \{\bot, \top\})) \\ &\qquad \wedge (\forall x \in \mathrm{dom}(a_v).\, (\exists x.\, x \mapsto v \wedge a_v(x) = v) \wedge (x \mapsto - \wedge a_v(x) \in \{\bot, \top\})) \\ I'_l(a_l) \quad &::= \quad (\forall r \notin a_l.\, \mathsf{liveE}(r)) \wedge (\forall r \in a_l.\, \neg\mathsf{liveE}(r)) \\ &\qquad \wedge (\forall x \notin a_l.\, \mathsf{liveX}(x)) \wedge (\forall x \in a_l.\, \neg\mathsf{liveX}(x)) \\ I_d(\{a_v, a_l\}) \quad &::= \quad I'_v(a_v) \wedge I'_l(a_l) \end{aligned}$$

Then, we prove the correctness of the Lemma 6.4.

LEMMA 6.4 (OPTIMIZER CORRECTNESS: DEAD CODE ELIMINATION).

$$\forall C, ge, \mathsf{f}, A.\ (\mathsf{Analyzer}(C, ge, \mathsf{f}) = A \wedge \mathsf{Translater}(C, A) = C') \implies I_d \vdash_{ge,\mathsf{f}} (C, A) \sim C'$$

We divide the correctness proof of Lemma. 6.4 into two lemmas, shown as Lemma. 6.5 and Lemma. 6.6 respectively.

LEMMA 6.5. $\forall C, ge, \mathsf{f}.\ \mathsf{Analyzer}(C, ge, \mathsf{f}) = A \implies I_d, C \vdash_{ge,\mathsf{f}} A.$

PROOF. Let $A = (\mathbb{L}_i, \mathbb{L}_o)$ and we apply the (wf-Analysis) rule on the proof goal. Then, we need to prove the following subgoals.

$$\exists is.\ (\mathsf{Init}(ge), is) \models I_d(\mathbb{L}_i(\mathsf{f})) \tag{g5}$$

$$\begin{aligned} &\text{for all } l \in \mathrm{dom}(C): \\ &\qquad I_d(\mathbb{L}_o(l)) \Rrightarrow I_d(\mathbb{L}_i(l_s)), \quad \text{for all } l_s \in \mathrm{succ}(C(l)) \\ &\qquad \vdash \{I_d(\mathbb{L}_i(l))\}\ C(l)\ \{I_d(\mathbb{L}_o(l))\} \end{aligned} \tag{g6}$$

Let $\mathbb{L}_i(\mathsf{f}) = \{a_{v_1}, a_{l_1}\}$. Since $a_{v_1} = \emptyset$, we can prove that $(\mathsf{Init}(ge), a_{l_1}) \models I'_v(\emptyset) \wedge I'_l(a_{l_1})$ holds. Thus, we prove the subgoal (g5).

For the subgoal (g6), we need to prove that for all $l \in \mathrm{dom}(C)$, the following subgoals hold:

$$I_d(\mathbb{L}_o(l)) \Rrightarrow I_d(\mathbb{L}_i(l_s)), \quad \text{for all } l_s \in \mathrm{succ}(C(l)) \tag{g6-1}$$
$$\vdash \{I_d(\mathbb{L}_i(l))\}\ C(l)\ \{I_d(\mathbb{L}_o(l))\} \tag{g6-2}$$

Let $\mathbb{L}_o(l) = (a_v, a_l)$ and $\mathbb{L}_i(l_s) = (a'_v, a'_l)$. We first prove that subgoal (g6-1). Since we can prove that "$I'_v(a_v) \Rrightarrow I'_v(a'_v)$" and $I'_l(a_l) \Rrightarrow I'_l(a'_l)$ hold, we have

$$\begin{array}{rcl} I_d(\{a_v, a_l\}) & \Rightarrow & I'_v(a_v) \wedge I'_l(a_l) \\ & \Rrightarrow & I'_v(a'_v) \wedge I'_l(a'_l) \\ & \Rightarrow & I_d(\{a'_v, a'_l\}), \end{array}$$

and we finish the proof of the subgoal (g6-1).

We define

$$\begin{array}{rcl} I'_{l-r}(a_l, r') & ::= & (\forall r \notin (a_l \cup \{r'\}).\, \mathsf{liveE}(r)) \wedge (\forall r \in (a_l \backslash r').\, \neg\mathsf{liveE}(r)) \\ & & \wedge(\forall x \notin a_l.\, \mathsf{liveX}(x)) \wedge (\forall x \in a_l.\, \neg\mathsf{liveX}(x)) \\ I'_{l-x}(a_l, x') & ::= & (\forall r \notin a_l.\, \mathsf{liveE}(r)) \wedge (\forall r \in a_l.\, \neg\mathsf{liveE}(r)) \\ & & \wedge(\forall x \notin (a_l \cup \{x'\}).\, \mathsf{liveX}(x)) \wedge (\forall x \in (a_l \backslash x').\, \neg\mathsf{liveX}(x)) \\ I'_{l-nlA}(a_l) & ::= & (\forall r \notin a_l.\, \mathsf{liveE}(r)) \wedge (\forall r \in a_l.\, \neg\mathsf{liveE}(r)) \\ & & \wedge(\forall x \notin a_l.\, \mathsf{liveX}(x)) \end{array}$$

as the auxiliary definitions, and prove the subgoal (g6-2) according to the following conclusion.

$$\begin{array}{l} \forall a_v, a'_v, a_l, a'_l, i. \\ \quad (f(i, a_v) = a'_v \wedge f_{\mathbb{L}}(i, a_v, a'_l) = a_l) \implies \vdash \{I_l(\{a_v, a_l\})\}\ i\ \{I_l(\{a'_v, a'_l\})\} \end{array}$$

We prove the conclusion shown above by discussing each case of the instruction i:

- $i = \langle r := e\rangle_l$, $a'_v = a_v\{r \rightsquigarrow [\![e]\!]_{a_v}\}$ and $a_l = (a'_l \backslash \mathsf{fv}(e)) \cup \{r\}$. We discuss the result of the $[\![e]\!]_{a_v}$.
  - there exists v, such that $[\![e]\!]_{a_v} = v$. We have the following conclusions.

$$\forall v, e, a_v.\ [\![e]\!]_{a_v} = v \implies (I'_c(a_v) \Rightarrow e = v)$$
$$\forall e, a_l.\ (\mathsf{fv}(e) \cap a_l = \emptyset) \implies (I'_v(a_l) \Rightarrow \mathsf{liveE}(e))$$
$$\forall a_l, a'_l.\ (a_l \subseteq a'_l) \implies (I'_l(a_l) \Rrightarrow I'_l(a'_l))$$

Let $p = (I'_v(a_v \backslash r) \wedge I'_{l-r}(a_l, r) \wedge r = v)$, then we have

$$\frac{I_d(\{a_v, a_l\}) \Rightarrow p[e/r] \wedge \mathsf{liveE}(e) \qquad \dfrac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow I_d(\{a'_v, a'_l\})}{\vdash \{p[e/r] \wedge \mathsf{liveE}(e)\}\ \langle r := e\rangle_l\ \{I_d(\{a'_v, a'_l\})\}}}{\vdash \{I_d(\{a_l, a_v\})\}\ \langle r := e\rangle_l\ \{I_d(\{a'_l, a'_v\})\}}$$

We prove "$p \wedge \mathsf{liveE}(r) \Rrightarrow I_l(\{a'_v, a'_l\})$" by the following steps.

$$\begin{array}{rl} & I'_v(a_v \backslash r) \wedge I'_{l-r}(a_l, r) \wedge r = v \wedge \mathsf{liveE}(r) \\ \Rightarrow & (I'_v(a_v \backslash r) \wedge r = v) \wedge (I'_{l-r}(a_l, r) \wedge \mathsf{liveE}(r)) \\ \Rightarrow & I'_v(a'_v) \wedge (I'_{l-r}(a_l, r) \wedge \mathsf{liveE}(r)) \qquad (*\text{ where }\ a'_v = a_v\{r \rightsquigarrow v\}\ *) \\ \Rightarrow & I'_v(a'_v) \wedge I'_l(a_l \backslash r) \\ \Rrightarrow & I'_v(a'_v) \wedge I'_l(a'_l) \qquad (*\text{ Since }\ (a_l \backslash r) \subseteq a'_l\ *) \\ \Rightarrow & I_l(\{a'_v, a'_l\}) \end{array}$$

  - $[\![e]\!]_{a_v} \in \{\bot, \top\}$. Let $p = (I_v(a_v \backslash r) \wedge I'_{l-r}(a_l, r) \wedge r = -)$, then we have

$$\frac{I_l(\{a_v, a_l\}) \Rightarrow p[e/r] \wedge \mathsf{liveE}(e) \qquad \dfrac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow I_l(\{a'_v, a'_l\})}{\vdash \{p[e/r] \wedge \mathsf{liveE}(e)\}\ \langle r := e\rangle_l\ \{I_l(\{a'_v, a'_l\})\}}}{\vdash \{I_l(\{a_l, a_v\})\}\ \langle r := e\rangle_l\ \{I_l(\{a'_l, a'_v\})\}}$$

- $i = \langle r := [e] \rangle_l$. We discuss the result of the $[\![e]\!]_{a_v}$.
  - there exists x, such that $[\![e]\!]_{a_v} = x$. We have $a'_v = a_v\{x \rightsquigarrow [\![e']\!]_{a_v}\}$ and $a_l = (a'_l \cup \{r\})\backslash(\mathsf{fv}(e) \cup Addr)$. We consider the condition that there exists v such that $[\![e']\!]_{a_v} = v$. The condition that $[\![e']\!]_{a_v} \in \{\bot, \top\}$ is similar. We have the following conclusion.

$$\forall x, v, a_v.\, ([\![e]\!]_{a_v} = x \wedge a_v(x) = v) \implies (I'_v(a_v) \Rightarrow e \mapsto v)$$

Let $p = (I'_v(a_v\backslash r) \wedge I'_{l-r}(a_l, r) \wedge r = v)$, $p_1 = ((e \mapsto v) \oplus p[v/r]) \wedge \mathsf{liveE}(e) \wedge \mathsf{liveX}(e)$, then we have

$$\frac{I_d(\{a_v, a_l\}) \Rightarrow p_1 \qquad \dfrac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow I_d(\{a'_v, a'_l\})}{\vdash \{p_1\}\ \langle r := [e]\rangle_l\ \{I_d(\{a'_v, a'_l\})\}}}{\vdash \{I_d(\{a_v, a_l\})\}\ \langle r := [e]\rangle_l \{I_d(\{a'_v, a'_l\})\}}$$

We prove the "$p \wedge \mathsf{liveE}(r) \Rrightarrow I_d(\{a'_v, a'_l\})$" by the following steps.

$$\begin{array}{ll} & I'_v(a_v\backslash r) \wedge I'_{l-r}(a_l, r) \wedge r = v \wedge \mathsf{liveE}(r) \\ \Rightarrow & (I'_v(a_v\backslash r) \wedge r = v) \wedge (I'_{l-r}(a_l, r) \wedge \mathsf{liveE}(r)) \\ \Rightarrow & I'_v(a'_v) \wedge (I'_{l-r}(a_l, r) \wedge \mathsf{liveE}(r)) \qquad (* \text{ where } \ a'_v = a_v\{r \rightsquigarrow v\}\ *) \\ \Rightarrow & I'_v(a'_v) \wedge I'_l(a_l\backslash r) \qquad (* \text{ where } \ (a_l\backslash r) \subseteq a'_l\ *) \\ \Rrightarrow & I'_v(a'_v) \wedge I'_l(a'_l) \\ \Rightarrow & I_d(\{a'_v, a'_l\}) \end{array}$$

  - $[\![e]\!]_{a_v} = \top$. We have $a'_v = a_v\{r \rightsquigarrow \top\}$ and $a_l = (a'_l \cup r)\backslash(\mathsf{fv}(e) \cup Addr)$. The condition that $[\![e]\!]_{a_v} = \bot$ is similar.
    Let $p = (I'_v(a_v\backslash r) \wedge I'_{l-r}(a_l, r) \wedge r = -)$, $p_1 = ((e \mapsto -) \oplus p[-/r]) \wedge \mathsf{liveE}(e) \wedge \mathsf{liveX}(e)$, then we have

$$\frac{I_d(\{a_v, a_l\}) \Rightarrow p_1 \qquad \dfrac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow I_d(\{a'_v, a'_l\})}{\vdash \{p_1\}\ \langle r := [e]\rangle_l\ \{I_d(\{a'_v, a'_l\})\}}}{\vdash \{I_d(\{a_v, a_l\})\}\ \langle r := [e]\rangle_l\ \{I_d(\{a'_v, a'_l\})\}}$$

We prove the "$p \wedge \mathsf{liveE}(r) \Rrightarrow I_d(\{a'_v, a'_l\})$" by the following steps.

$$\begin{array}{ll} & I'_v(a_v\backslash r) \wedge I'_{l-r}(a_l, r) \wedge r = - \wedge \mathsf{liveE}(r) \\ \Rightarrow & (I'_v(a_v\backslash r) \wedge r = -) \wedge I'_{l-r}(a_l, r) \wedge \mathsf{liveE}(r) \\ \Rightarrow & I'_v(a'_v) \wedge I'_{l-r}(a_l, r) \wedge \mathsf{liveE}(r) \qquad (* \text{ where } a'_v = a_v\{r \rightsquigarrow \top\}\ *) \\ \Rightarrow & I'_v(a'_v) \wedge I'_l(a_l\backslash r) \\ \Rrightarrow & I'_v(a'_v) \wedge I'_l(a'_l) \qquad (* \text{ where } (a_l\backslash r) \subseteq a'_l\ *) \\ \Rightarrow & I_d(\{a'_v, a'_l\}) \end{array}$$

- $i = \langle [e_1] := e_2 \rangle_l$. We discuss the result of the $[\![e]\!]_{a_v}$.
  - there exists x, such that $[\![e_1]\!]_{a_v} = x$. We have $a'_v = a_v\{x \rightsquigarrow [\![e_2]\!]_{a_v}\}$ and $a_l = (a'_l \cup \{x\})\backslash(\mathsf{fv}(e_1) \cup \mathsf{fv}(e_2))$. We consider the condition that there exists v such that $[\![e_2]\!]_{a_v} = v$. The condition that $[\![e'_2]\!]_{a_v} \in \{\bot, \top\}$ is similar.
    Let $p = I'_v(a_v\backslash x) \wedge I'_{l-x}(a_l, x) \wedge e_1 = x \wedge e_2 = v$, and $p_1 = ((e_1 \mapsto -) \oplus (e_1 \mapsto - * p) \wedge \mathsf{liveE}(e_1, e_2))$. Then, we have

$$\frac{I_d(\{a_v, a_l\}) \Rightarrow p_1 \qquad \dfrac{\mathsf{ALStaE}(p, e_1) \qquad (e_1 \mapsto e_2 * p) \wedge \mathsf{liveX}(e_1) \Rrightarrow I_d(\{a'_v, a'_l\})}{\vdash \{p_1\}\ \langle [e_1] := e_2\rangle\ \{I_d(\{a'_v, a'_l\})\}}}{\vdash \{I_d(\{a_v, a_l\})\}\ \langle [e_1] := e_2\rangle\ \{I_d(\{a'_v, a'_l\})\}}$$

We prove the "$(e_1 \mapsto e_2 * p) \wedge \mathsf{liveX}(e) \Rrightarrow I_d(\{a'_v, a'_l\})$" by the following steps.

$$\begin{array}{rl}
& (e_1 \mapsto e_2 * (I'_v(a_v \backslash x) \wedge I'_{l-x}(a_l, x) \wedge e_1 = x \wedge e_2 = v)) \wedge \mathsf{liveX}(e) \\
\Rightarrow & (I'_v(a_v \backslash x) \wedge e_1 \mapsto e_2) \wedge e_1 = x \wedge e_2 = v \wedge I'_{l-x}(a_l, x) \wedge \mathsf{liveX}(e_1) \\
\Rightarrow & I'_v(a'_v) \wedge (e_1 = x \wedge I'_{l-x}(a_l, x)) \wedge \mathsf{liveX}(e_1) \qquad (* \text{ where } \ a'_v = a_v\{x \rightsquigarrow v\} *) \\
\Rightarrow & I'_v(a'_v) \wedge I'_l(a_l \backslash x) \\
\Rrightarrow & I'_v(a'_v) \wedge I'_l(a'_l) \qquad (* \text{ since } \ (a_l \backslash x) \subseteq a'_l \, *) \\
\Rightarrow & I_d(\{a'_v, a'_l\})
\end{array}$$

– $[\![e_1]\!]_{a_v} \in \{\bot, \top\}$. We have $a'_v = a_v \backslash Addr$ and $a_l = a'_l \backslash (\mathsf{fv}(e_1) \cup \mathsf{fv}(e_2))$.
Let $p = I'_v(a_v \backslash Addr) \wedge I'_{l-lnA}(a_l)$ and $p_1 = ((e_1 \mapsto -) \oplus (e_1 \mapsto - * p)) \wedge \mathsf{liveE}(e_1, e_2)$. And we have

$$\dfrac{I_d(\{a_v, a_l\}) \Rightarrow p_1 \qquad \dfrac{\mathsf{ALStaE}(p, e_1) \qquad (e_1 \mapsto e_2 * p) \wedge \mathsf{liveX}(e_1) \Rrightarrow I_d(\{a'_v, a'_l\})}{\vdash \{p_1\} \ \langle [e_1] := e_2 \rangle \ \{I_d(\{a'_v, a'_l\})\}}}{\vdash \{I_d(\{a_v, a_l\})\} \ \langle [e_1] := e_2 \rangle \ \{I_d(\{a'_v, a'_l\})\}}$$

We prove the "$(e_1 \mapsto e_2 * p) \wedge \mathsf{liveX}(e) \Rrightarrow I_d(\{a'_v, a'_l\})$" by the following steps.

$$\begin{array}{rl}
& ((e_1 \mapsto e_2) * (I'_v(a_v \backslash Addr) \wedge I_{l-\mathrm{lnA}}(a_l))) \wedge \mathsf{liveE}(e) \\
\Rightarrow & I'_v(a'_v) \wedge I'_{l-lnA}(a_l) \qquad (* \text{ where } \ a'_v = a_v \backslash Addr \, *) \\
\Rrightarrow & I'_r(a'_v) \wedge I'_l(a_l) \qquad (* \text{ Since } \ I_{l-lnA}(a_l) \Rightarrow I'_l(a_l) \, *) \\
\Rrightarrow & I'_r(a'_v) \wedge I'_l(a'_l) \qquad (* \text{ Since } \ a_l \subseteq a'_l \, *) \\
\Rightarrow & I_d(\{a'_v, a'_l\})
\end{array}$$

□

Lemma 6.6. $\forall C, A, C'. \ \mathsf{Translater}(C, A) = C' \implies I_d, A \vdash C \sim C'$.

Proof. We assume that the source code C is not an empty set. We prove this lemma by induction on C.

- If $C = \{l \rightsquigarrow i\}$, we have the following holds.

$$\mathsf{Translater}(\{l \rightsquigarrow i\}, A) = C' \tag{4}$$

We unfold (4) according to the definition of the translater. We discuss each case respectively and assume $A = (\mathbb{L}_i, \mathbb{L}_o)$, $\mathbb{L}_i(l) = (a_v, a_l)$ and $\mathbb{L}_i(l) = (a'_v, a'_l)$.
– $i = \langle r := e \rangle_{l_1}$ and $r \in a'_l$. We get $C' = \{l \rightsquigarrow \langle \mathsf{skip} \rangle_{l_1}\}$. And we need to prove the following holds.

$$I_d, A \vdash \{l \rightsquigarrow \langle r := e \rangle_{l_1}\} \sim \{l \rightsquigarrow \langle \mathsf{skip} \rangle_{l_1}\} \tag{g7-1}$$

By applying (Ins-Trans) rule on (g7-1), we get

$$I_d(\{a_v, a_l\}), I_d(\{a'_v, a'_l\}) \vdash \langle r := e \rangle_{l_1} \sim \langle \mathsf{skip} \rangle_{l_1} \tag{g7-1.1}$$

By applying (Assgn-Elim) on (g7-1.1), we need to prove that

$$I_d(\{a'_v, a'_l\}) \Rightarrow \neg \mathsf{liveE}(r) \tag{g7-1.2}$$

Since $r \in a'_l$, we prove (g7-1.2) according to the definition of I_d.
– $i = \langle r := [e] \rangle_{l_1}$ and $r \in a'_l$. We get $C' = \{l \rightsquigarrow \langle \mathsf{skip} \rangle_{l_1}\}$. And we need to prove the following holds.

$$I_d, A \vdash \{l \rightsquigarrow \langle r := [e] \rangle_{l_1}\} \sim \{l \rightsquigarrow \langle \mathsf{skip} \rangle_{l_1}\} \tag{g7-2}$$

By applying (Ins-Trans) rule on (g7-2), we get

$$I_d(\{a_v, a_l\}), I_d(\{a'_v, a'_l\}) \vdash \langle r := [e] \rangle_{l_1} \sim \langle \mathsf{skip} \rangle_{l_1} \tag{g7-2.1}$$

By applying (Ld-Elim) on (g7-3), we need to prove that

$$I_d(\{a'_v, a'_l\}) \Rightarrow \neg\mathsf{liveE}(r) \tag{g7-2.2}$$

Since $r \in a'_l$, we prove (g7-2.2) according to the definition of I_d.

– $i = \langle [e_1] := e_2 \rangle_{l_1}$, and there exists x and v such that $x \in a'_l$ and $[\![e_1]\!]_{a_v} = x$. We get $C' = \{l \rightsquigarrow \langle \mathtt{skip} \rangle_{l_1}\}$. And we need to prove the followig holds.

$$I_d, A \vdash \{l \rightsquigarrow \langle [e_1] := e_2 \rangle_{l_1}\} \sim \{l \rightsquigarrow \langle \mathtt{skip} \rangle_{l_1}\} \tag{g7-3}$$

By applying (Ins-Trans) rule on (g7-3), we get

$$I_d(\{a_v, a_l\}), I_d(\{a'_v, a'_l\}) \vdash \langle [e_1] := e_2 \rangle_{l_1} \sim \langle \mathtt{skip} \rangle_{l_1} \tag{g7-3.1}$$

By applying (St-Elim) rule on (g7-3.1), we need to prove that

$$I_d(\{a'_v, a'_l\}) \Rightarrow \neg\mathsf{liveX}(e_1) \tag{g7-3.2}$$

Since $x \in a'_l$ and $[\![e_1]\!]_{a_v} = x$, we get

$$I_d(\{a'_v, a'_l\}) \Rightarrow (e_1 = x) \wedge \neg\mathsf{liveX}(x) \Rightarrow \neg\mathsf{liveX}(e_1).$$

We finish the proof of (g7-3.2).

– $C' = \{l \rightsquigarrow i\}$. And we need to prove the following holds.

$$I_d, A \vdash \{l \rightsquigarrow i\} \sim \{l \rightsquigarrow i\} \tag{g7-4}$$

The subgoal (g7-4) can be proved by discuss each case of the instruction i.

- If $C = \{l \rightsquigarrow i\} \uplus C_1$, we have the following holds.

$$\mathsf{Translater}(\{l \rightsquigarrow i\} \uplus C_1, A) = C' \tag{5}$$

And the inductive hypothesis is shown below.

$$\forall A, C'_1.\ \mathsf{Translater}(C_1, A) = C'_1 \implies I_d, A \vdash C_1 \sim C'_1 \tag{6}$$

We unfold (5) according to the definition of the translater. We prove by discussing each case respectively and assume $A = (\mathbb{L}_i, \mathbb{L}_o)$, $\mathbb{L}_i(l) = (a_v, a_l)$ and $\mathbb{L}_o(l) = (a'_v, a'_l)$. Here, we only show the proof of the first case. The proof of the other cases are similar to the first one.

– $i = \langle r := e \rangle_{l_1}$ and $r \in a'_l$. We get $C' = \{l \rightsquigarrow \langle \mathtt{skip} \rangle_{l_1}\} \uplus \mathsf{Translater}(C_1, A)$. Let $C'_1 = \mathsf{Translater}(C_1, A)$. We need to prove the following holds.

$$I_d, A \vdash (\{l \rightsquigarrow \langle r := e \rangle_{l_1}\} \uplus C_1) \sim (\{l \rightsquigarrow \langle \mathtt{skip} \rangle_{l_1}\} \uplus C'_1) \tag{g8}$$

By applying (Link) rule on (g4), we need to prove

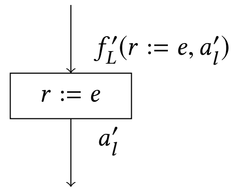
$$I_d, A \vdash \{l \rightsquigarrow \langle r := e \rangle_{l_1}\} \sim \{l \rightsquigarrow \langle \mathtt{skip} \rangle_{l_1}\} \tag{g8-1}$$

$$I_d, A \vdash C_1 \sim C'_1 \tag{g8-2}$$

The subgoal (g8-1) can be proved by applying the (Ins-Trans) rule and (Asgn-Elim) rule, and the subgoal (g8-2) can be proved by applying the inductive hypothesis (6).

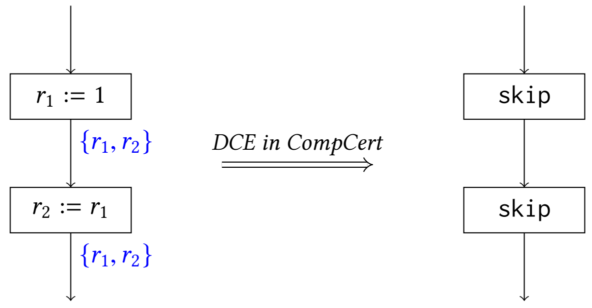
□

Shortcommings of the current work. The current work can not handle the correctness proof of the *dead code elimination* in CompCert. We consider the condition when meeting the instruction for the assignement ($r := e$). The transfer function (f'_L) for "$r := e$" in CompCert is:



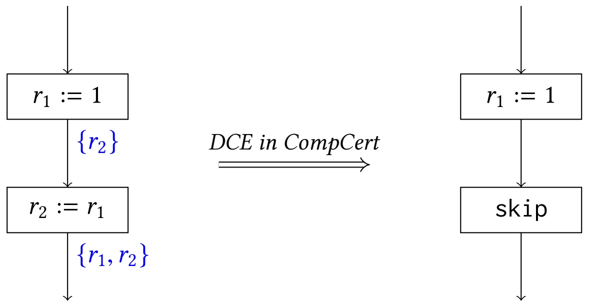
$$f'_L(r := e, a'_l) \quad ::= \quad \begin{cases} a'_l & \textit{if } r \in a'_l \\ (a'_l \cup \{r\}) \backslash \mathsf{fv}(e) & \textit{otherwise} \end{cases}$$

The *dead code elimination* pass in CompCert can perform the following optimizations shown below.



However, the optimization shown above can not be performed in the *dead code elimination* that we have implememted in this subsection. The transfer function for "$r := e$" in our work does not deal specially with the situation when $r \in a'_l$. The *dead code elimination* pass that we have implememted can only perform the following optimization.

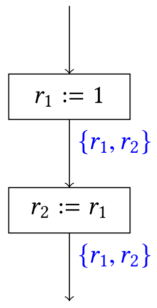
$$f_L(r := e, a'_l) \ ::= \ (a'_l \cup \{r\}) \backslash \mathsf{fv}(e)$$



Our current work can not handle the correctness proof of the *dead code elimination* in CompCert, since the following conclusion does not hold (for simplicity, we omit the parameter a_v of I_d).

$$f'_L(\langle r := e\rangle_l, a'_l) = a_l \implies \vdash \{I_d(a_l)\} \ \langle r := e\rangle_l \ \{I_d(a'_l)\}$$

The problem here is that the following analysis result is permitted in CompCert's liveness analysis. Our (Asgn-Ac) rule can not handle such condition, since such rule requires that the register r_1 is live when starting to execute the instruction $r_2 := r_1$.



We try to solve the problem shown above by relaxing our (Asgn-Ac) rule as the following form.

$$\frac{\begin{array}{c}(p[e/r] \Rightarrow \mathsf{liveE}(e)) \vee (p' \Rightarrow \neg\mathsf{liveE}(r)) \\ \mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow p'\end{array}}{\vdash \{p[e/r]\}\ \langle r := e\rangle_l\ \{p'\}}$$

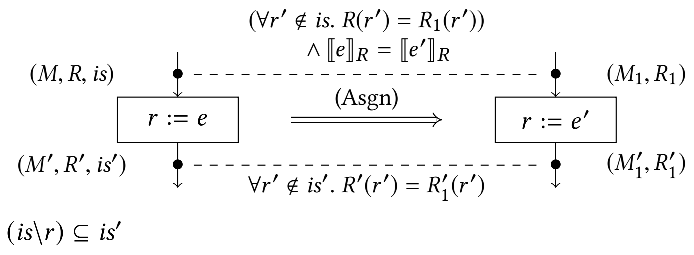
However, the relaxed (Asgn-Ac) rule is not sound. The semantics of the well-formed analysis requires that the registers in *is*, which is used to interpret the pre-condition, can not be read by the instruction $\langle r := e\rangle_l$. So, the condition "$\mathsf{liveE}(e)$" is essential in the precondition.

The role of the not live set. We give more discussion about why we need the not live set *is*. The main role of *is* is to define the simulation relation. We define the state relation between source and target programs in Def. 5.3 and the simulation relation in Def. 5.4. We can find that we permit that the registers or memory locations, which are not in *is*, do not equal. So, a register r or a memory location x that is describe as "$\mathsf{liveE}(r)$" or $\mathsf{liveX}(x)$ in assertion has the same value in the source and target program states. For example, in the (Asgn) rule, it requires that the registers in the expression e' are all live, since the registers that are live have the same values in the source and target program states. It also explains why the (Asgn-Elim) rule permits to eliminate the assignement to the register r, which is not live in the following execution.

When proving Lemma. 5.6, we find that "$is \ltimes^i_R is'$" can be relaxed. "$is \ltimes^i_R is'$" includes two restrictions: 1. the instruction i can only read the registers that are live; 2. except the register r assigned, we can not set other registers that are not live to live. We find that we only need the property (2). So, the definition of "$is \ltimes^i_R is'$" can be relaxed to the following form.

$$\frac{\begin{array}{c}\forall r \in is.\ r \notin \mathsf{updR}(i) \implies r \in is' \\ \forall x \in is.\ x \notin \mathsf{stM}(i, R) \implies x \in is'\end{array}}{is \ltimes^i_R is'}$$

The following figure illustrates that the property (2) is sufficient.


$(\forall r' \notin is.\ R(r') = R_1(r'))$
$\wedge [\![e]\!]_R = [\![e']\!]_R$
(M, R, is)
(M_1, R_1)
(Asgn)
$r := e$
$r := e'$
(M', R', is')
(M'_1, R'_1)
$\forall r' \notin is'.\ R'(r') = R'_1(r')$
$(is \backslash r) \subseteq is'$

From the precondition, we can prove that the value of the register r has the same value in the source and target program state. And from the property (2), we have "$(is \backslash r) \subseteq is'$". We know that except the register r, the registers that are not in is' are also not in is. Thus, we can prove that the after the source program executing "$r := e$" and the target program exeucting "$r := e'$", the registers that are not in is' have the same values in the source and target program states.

After using the new definition of $is \ltimes is'$, we can redefine the (Asgn-Ac) rule, (Ld-Ac) rule and (St-Ac) rule as the following form.

$$\frac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow p'}{\vdash \{p[e/r]\}\ \langle r := e\rangle_l\ \{p'\}} \quad \text{(Asgn-Ac)}$$

$$\frac{\mathsf{ALStaE}(p, r) \qquad p \wedge \mathsf{liveE}(r) \Rrightarrow p'}{\vdash \{(e \mapsto e') \oplus p[e'/r]\}\ \langle r := [e]\rangle_l\ \{p'\}} \quad \text{(Ld-Ac)}$$

$$\frac{\mathsf{ALStaX}((e_1 \mapsto - * p), e_1) \qquad (e_1 \mapsto e_2 * p) \wedge \mathsf{liveX}(e_1) \Rrightarrow p'}{\vdash \{(e_1 \mapsto -) \oplus (e_1 \mapsto - * p)\}\ \langle [e_1] := e_2\rangle_l\ \{p'\}} \quad \text{(St-Ac)}$$

Thus, the modified program logic can handle the correctness proof of the *dead code elimination* in CompCert, since we do not require that the registers and memory locations read by the current instruction are live.