Syntactic and Semantic Soundness of Structural Dataflow Analysis

Patrick Cousot
Courant Institute of Mathematical Sciences, New York University

Abstract. We show that the classical approach to the soundness of dataflow analysis is with respect to a syntactic path abstraction that may be problematic with respect to a semantics trace-based specification. The fix is a rigorous abstract interpretation based approach to formally construct dataflow analysis algorithms by calculational design.

Keywords: Abstract interpretation · Dataflow analysis · Model-checking · Soundness.

1 Introduction

The very first data flow analysis algorithms [1,2,3,4] were postulated: map the program to a control flow graph (CFG), derive binary vector fixpoint equations using transfer functions/transformers to abstract the actions in the CFG, solve iteratively or by elimination, the result is postutaled to be the abstract information available on the program semantics. We call this approach "syntactic" since the values of the variables are not taken into account at all by the transfer functions/transformers in the equations.

Gary Kildall proposed to reason on paths in the CFG [22]: define the abstract information available on any path in the CFG by composition of syntactic transfer functions/transformers along that path and then merge/join/meet the information on all paths. In general, this yields more precise results than the fixpoint equations (except for distributive frameworks where transformers preserve joins/meets and the results are the same). This is an abstract form of soundness since one can prove that the solution of the equations over-approximates the merge over all paths solution. [12, Section 9] showed that the merge over all paths solution is also the solution of fixpoint equations taken over the disjunctive completion [12,16] of the original abstract domain. So the imprecision is not due to the equations but to the abstract domain [17].

Bernhard Steffen observed that by considering the CFG as a transition system, the information along a path can be specified by a modal/temporal logic formula [28,29]. Model-checking over all paths yields the abstract information available about the program semantics. The specification is concise and an existing model-checker can be reused for the implementation. Fixpoint iterates convergence requires the abstract domain to be finite (which excludes *e.g.* Kildall's constant propagation [22] for which the model checker would not be guaranteed to terminate). The information on the program semantics is still defined with respect to a syntactic abstraction of the semantics, not the semantics itself.

To solve this problem, David Schmidt proposed to get the abstraction of the paths by abstract interpretation of a trace semantics [25]. Now the information

B.-Y. E. Chang (Ed.): SAS 2019, LNCS 11822, pp. 1–22, 2019.
https://doi.org/10.1007/978-3-030-32304-2_6

extracted from the program is related to the semantics, but indirectly, since it is postulated syntactically on abstract paths, not on the traces of the semantics itself.

David Schmidt used his model to explore "Why some flow analyzes are unsound?" and claimed that the live variable analysis is unsound [25, Section 7]. As shown in [13] this is because the analysis is about potential liveness while David Schmidt's counter-example is on definite liveness. David Schmidt claims that this is not a problem in practice since the information is used dually [25]. If a variable is not potentially live, it is definitely dead and its value need not be stored *e.g.* in a register. But if a data flow analysis were wrong, its dual would be wrong too. As shown by this erroneous reasoning, the syntactic modal/temporal specification on abstract paths but not directly on the semantics may be problematic.

In this paper, we explore the definition of dataflow analyses by direct abstraction of the trace semantics. So the abstract information extracted by the static analysis is directly related to the program trace semantics, not to an abstraction of this semantics. In this way, values of variables can be taken into account, which is not the case with temporal specifications on abstract paths. The analyzes should therefore be more precise and provably sound.

Surprisingly, this approach shows that the abstract syntactic definition of liveness is unsound with respect to its semantic definition. The problem is both for definite and potential liveness. The problem comes from the fact that the semantic definition takes values into account while the abstract definition hence the resulting dataflow analysis algorithm captures that incorrectly.

Example 1 For definite liveness, consider for example `if` ℓ_1 `(x==0)` ℓ_2 `x = x-x ;` where `x` is dead on exit. The syntactic equational and path-based definitions of definite liveness both yield `x` is live at ℓ_1 and ℓ_2. However, this program is equivalent to `if` ℓ_1 `(x==0)` ℓ_2 `x = 0 ;` so `x` is not live at ℓ_2. Moreover, this last program is itself equivalent to ℓ_1`;` (skip) so that no variable, in particular `x` is live at ℓ_1. Therefore the semantic definition of definite liveness at ℓ_1 and ℓ_2 in the original program `if` ℓ_1 `(x==0)` ℓ_2 `x = x-x ;` should be that `x` is not live, in contradiction with the syntactic equational and path-based definite liveness. □

Potential liveness or, dually, definite deadness is not better.

Example 2 For definite deadness, consider ℓ_1 `x = y-y ;`ℓ_2 where `x` is live at ℓ_2 on exit. Syntactically, `x` is not used in `y-y` and `x` is modified by the assignment so `x` is syntactically dead at ℓ_1. Semantically, `x` is not used in `y-y` since changing the value of `x` at ℓ_1 will not change the value of `y-y` which is always 0. However, assume `x` = 0 at ℓ_1 then the assignment ℓ_1 `x = y-y ;` does not modify this value. So in that case `x` is not modified by the assignment and therefore `x` is live at ℓ_1 *i.e.* if the precondition `x` = 0 is always true, the compiler is allowed to remove the assignment. For all other initial values `x` $\neq 0$ at ℓ_1, the assignment does modify this value by assigning 0 in which case `x` is dead at ℓ_1. So syntactically, `x` is definitely dead at ℓ_1 while, semantically, this is not always the case (*i.e.* when `x` is 0 at ℓ_1). □

To solve these soundness problems, we first define a structural fixpoint trace semantics in Section 2. Then, in Section 3, we first provide an intuitive semantic definition of liveness by abstraction of a trace semantics: "a variable is live at some point if its value may be read before the next time it is modified". The above examples 1 and 2 show that the classical syntactic liveness algorithm is unsound with respect to this definition. At that point we could change the algorithm or the liveness definition. We choose the second alternative (so as not to have to change compilers, but this choice is arbitrary!). This second definition "a variable is live at some point if its value may be read before the next time it is assigned to" mixes a syntactic (assignment) and a semantic (value) points of view (thus preventing meaningful program syntactic transformation such as useless assignment elimination). It specifies exactly in what sense the classical syntactic deadness/liveness algorithm [20,19,21] is sound. Then by a further purely syntactic abstraction "a variable is live at some point if its value may be used before the next time it is assigned to" (where use and assigned to are defined syntactically, thus preventing expression and assignment optimizations), we get, by calculational design [8], the classical syntactic potential liveness algorithm [20,19,21] in Section 4, and the dual definite deadness algorithm in Section 5. The definition of the trace semantics is structural, so we get the classical syntactic deadness/liveness algorithm in structural form. Surprisingly, there is no fixpoint iteration and the (implicit) equations are solved by elimination, which is more efficient. This is comparable to equation resolution by elimination for reducible flowcharts [27,24,26] but much simpler and efficient. In Section 6, we discuss whether liveness analysis is correctly used for code optimization. We conclude in Section 7.

2 Syntax and Trace Semantics

Programs are a subset of `C` with the following context-free syntax.

$\mathtt{x}, \mathtt{y}, \ldots \in \mathbb{V}$	variable ($\mathbb{V}$ not empty)
$\mathtt{A} \in \mathbb{A} ::= \mathtt{1} \mid \mathtt{x} \mid \mathtt{A}_1 - \mathtt{A}_2$	arithmetic expression
$\mathtt{B} \in \mathbb{B} ::= \mathtt{A}_1 < \mathtt{A}_2 \mid \mathtt{B}_1\ \mathtt{nand}\ \mathtt{B}_2$	boolean expression
$\mathtt{S} \in \mathbb{S} ::=$	statement
$\quad \mathtt{x = A ;}$	assignment
$\quad \mid \mathtt{;}$	skip
$\quad \mid \mathtt{if\ (B)\ S} \mid \mathtt{if\ (B)\ S\ else\ S}$	conditionals
$\quad \mid \mathtt{while\ (B)\ S} \mid \mathtt{break\ ;}$	iteration and break
$\quad \mid \mathtt{\{\ Sl\ \}}$	compound statement
$\mathtt{Sl} \in \mathbb{Sl} ::= \mathtt{Sl}\ \ \mathtt{S} \mid \epsilon$	statement list
$\mathtt{P} \in \mathbb{P} ::= \mathtt{Sl}$	program

A break exits the closest enclosing loop, if none this is a syntactic error. If `P` is a program then `int main () { P }` is a valid `C` program. We call "[program] component" $\mathtt{S} \in \mathbb{Pc} \triangleq \mathbb{S} \cup \mathbb{Sl} \cup \mathbb{P}$ either a statement, a statement list, or a program.

2.1 Program labels

Labels are not part of the language, but useful to discuss program points reached during execution. For each program component `S`, we define informally

$\mathsf{at}[\![\mathtt{S}]\!]$	the program point at which execution of `S` starts;
$\mathsf{aft}[\![\mathtt{S}]\!]$	the program exit point after `S`, at which execution of `S` is supposed to normally terminate, if ever;
$\mathsf{esc}[\![\mathtt{S}]\!]$	a boolean indicating whether or not the program component `S` contains a **`break ;`** statement escaping out of that component `S`;
$\mathsf{brk\text{-}to}[\![\mathtt{S}]\!]$	the program point at which execution of the program component `S` goes to when a **`break ;`** statement escapes out of that component `S`;
$\mathsf{brks\text{-}of}[\![\mathtt{S}]\!]$	the set of labels of all **`break ;`** statements that can escape out of `S`;
$\mathsf{in}[\![\mathtt{S}]\!]$	the set of program points inside `S` (including $\mathsf{at}[\![\mathtt{S}]\!]$ but excluding $\mathsf{aft}[\![\mathtt{S}]\!]$ and $\mathsf{brk\text{-}to}[\![\mathtt{S}]\!]$);
$\mathsf{labs}[\![\mathtt{S}]\!]$	the potentially reachable program points while executing `S` either at, in, or after the statement, or resulting from a break.

2.2 Traces

Because liveness analysis at a program point relates the past, present, and future of a computation, we use a trace semantics relating the past computation reaching that program point to the future computation continuing this past computation. For simplicity, the program point where liveness is calculated is the entry point $\mathsf{at}[\![\mathtt{S}]\!]$ at a program component `S`.

A trace $\pi \in \mathbb{T}^{+\infty}$ is a sequence of states separated by events. States are program labels designating the next action to be executed in the program. The events record the effect of this execution *i.e.* the value assigned to a variable, a test `B` which is true (marked (`B`)) or false (marked (¬`B`)), a **`break ;`** exiting from a loop, or a `skip` when execution goes on with no variable modification. For example, the program

$$\ell_1\ \mathtt{x = x + 1\ ;\ if}\ \ell_2\ \mathtt{(x < 0)}\ \ell_3\ \mathtt{x = 0\ ;}\ \ell_4 \tag{1}$$

executed with initial value 0 of `x` has execution trace $\ell_1 \xrightarrow{\mathtt{x = x + 1} = 1} \ell_2 \xrightarrow{\neg(\mathtt{x} < 0)} \ell_4$. A trace π can be finite $\pi \in \mathbb{T}^+$ or infinite $\pi \in \mathbb{T}^\infty$ (recording a non-terminating computation) so $\mathbb{T}^{+\infty} \triangleq \mathbb{T}^+ \cup \mathbb{T}^\infty$ [1]. Trace concatenation $\frown$ is defined as follows

$$\begin{array}{lll} \pi_1\ell_1 \frown \ell_2\pi_2 & \text{undefined if } \ell_1 \neq \ell_2 & \pi_1 \frown \ell_2\pi_2 \triangleq \pi_1 \quad \text{if } \pi_1 \in \mathbb{T}^\infty \text{ is infinite} \\ \pi_1\ell_1 \frown \ell_1\pi_2 \triangleq \pi_1\ell_1\pi_2 & \text{if } \pi_1 \in \mathbb{T}^+ \text{ is finite} & \end{array}$$

In pattern matching, we sometimes need the empty trace $\ni$. For example if $\ell\pi\ell' = \ell$ then $\pi = \ni$ and so $\ell = \ell'$.

States do not record the value of variables `x`. $\boldsymbol{\varrho}(\pi)\mathtt{x}$ is the last value assigned to `x` on trace π (or 0 at initialization).

$$\boldsymbol{\varrho}(\ell)\mathtt{x} \triangleq 0 \qquad \boldsymbol{\varrho}(\pi\ell \xrightarrow{\mathtt{x = A} = \nu} \ell')\mathtt{x} \triangleq \nu \qquad \boldsymbol{\varrho}(\pi\ell \xrightarrow{\cdots} \ell')\mathtt{x} \triangleq \boldsymbol{\varrho}(\pi\ell)\mathtt{x} \ \text{ otherwise} \tag{2}$$

[1] Abstracting program label states would yield Stephen Brookes trace semantics [6].

2.3 Trace semantics

The trace semantics of a program component $\mathtt{S}$ is a relation between past traces reaching the entry point $\mathsf{at}[\![\mathtt{S}]\!]$ and future traces recording the computation of $\mathtt{S}$ from $\mathsf{at}[\![\mathtt{S}]\!]$. For example, program $\mathtt{S}$ in (1) has the following two pairs of traces in its trace semantics.

$$\langle \ell_0 \xrightarrow{\mathtt{x = 0} = 0} \ell_1,\ \ell_1 \xrightarrow{\mathtt{x = x + 1} = 1} \ell_2 \xrightarrow{\neg(\mathtt{x} < 0)} \ell_4\rangle \in \boldsymbol{\mathcal{S}}^{+\infty}[\![\mathtt{S}]\!]$$
$$\langle \ell_0 \xrightarrow{\mathtt{x = 1} = 1} \ell_1,\ \ell_1 \xrightarrow{\mathtt{x = x + 1} = 2} \ell_2 \xrightarrow{\neg(\mathtt{x} < 0)} \ell_4\rangle \in \boldsymbol{\mathcal{S}}^{+\infty}[\![\mathtt{S}]\!]$$

In the *maximal trace semantics* $\boldsymbol{\mathcal{S}}^{+\infty}[\![\mathtt{S}]\!]$, the observation of the future computation is maximal. It is finite when the program execution stops and infinite when the execution does not terminate. In the *prefix trace semantics* $\boldsymbol{\mathcal{S}}^{*}[\![\mathtt{S}]\!]$, the observation of the future computation is finite and can stop at any time during the execution (in particular just at the program entry). For example, program $\mathtt{S}$ in (1) has the following two pairs of traces in its prefix trace semantics.

$$\langle \ell_0 \xrightarrow{\mathtt{x = 0} = 0} \ell_1,\ \ell_1\rangle \in \boldsymbol{\mathcal{S}}^{*}[\![\mathtt{S}]\!] \qquad \langle \ell_0 \xrightarrow{\mathtt{x = 1} = 1} \ell_1,\ \ell_1 \xrightarrow{\mathtt{x = x + 1} = 2} \ell_2\rangle \in \boldsymbol{\mathcal{S}}^{*}[\![\mathtt{S}]\!]$$

It follows from this discussion that the prefix trace semantics is a relation between finite traces $\boldsymbol{\mathcal{S}}^{*}[\![\mathtt{S}]\!] \in \wp(\mathbb{T}^+ \times \mathbb{T}^+)$ while the maximal trace semantics is a relation between finite traces and finite or infinite traces $\boldsymbol{\mathcal{S}}^{+\infty}[\![\mathtt{S}]\!] \in \wp(\mathbb{T}^+ \times \mathbb{T}^{+\infty})$.

2.4 Formal definition of the prefix trace semantics

The prefix trace semantics is defined in fixpoint form by structural induction on the syntax of program components.

• A prefix future trace of an assignment $\mathtt{S} ::= \ell\ \mathtt{x = A ;}$ (where $\mathsf{at}[\![\mathtt{S}]\!] = \ell$) continuing some past trace $\pi\ell$ either stops at ℓ or is ℓ followed by the event $\mathtt{x = A} = \nu$ where $\nu \in \mathbb{V}$ is the value assigned to $\mathtt{x}$ (that is the value of the arithmetic expression $\mathtt{A}$ evaluated on $\pi\ell$) and finishing at the label $\mathsf{aft}[\![\mathtt{S}]\!]$ after the assignment.

$$\boldsymbol{\mathcal{S}}^{*}[\![\mathtt{S}]\!] \triangleq \{\langle \pi\ell,\ \ell\rangle, \langle \pi\ell,\ \ell \xrightarrow{\mathtt{x = A} = \nu} \mathsf{aft}[\![\mathtt{S}]\!]\rangle \mid \pi\ell \in \mathbb{T}^+ \wedge \nu = \boldsymbol{\mathcal{A}}[\![\mathtt{A}]\!]\boldsymbol{\varrho}(\pi\ell)\} \tag{3}$$

We often write $\ell \xrightarrow{\mathtt{x} = v} \ell'$ for $\ell \xrightarrow{\mathtt{x = A} = v} \ell'$ (since $\ell\ \mathtt{x = A ;}$ can be recovered from the program text and the unique program label ℓ). The value of an arithmetic expression $\mathtt{A}$ in environment $\rho \in \mathbb{E}\mathbb{v} \triangleq \mathbb{V} \to \mathbb{V}$ is $\boldsymbol{\mathcal{A}}[\![\mathtt{A}]\!]\rho \in \mathbb{V}$:

$$\boldsymbol{\mathcal{A}}[\![\mathtt{1}]\!]\rho \triangleq 1 \qquad \boldsymbol{\mathcal{A}}[\![\mathtt{x}]\!]\rho \triangleq \rho(\mathtt{x}) \qquad \boldsymbol{\mathcal{A}}[\![\mathtt{A}_1 - \mathtt{A}_2]\!]\rho \triangleq \boldsymbol{\mathcal{A}}[\![\mathtt{A}_1]\!]\rho - \boldsymbol{\mathcal{A}}[\![\mathtt{A}_2]\!]\rho \tag{4}$$

• A prefix trace of a break statement $\mathtt{S} ::= \ell\ \mathtt{\mathbf{break}\ ;}$ continuing some initial trace $\pi\ell$ either stops at ℓ or is the trace ℓ followed by the **break** event and ending at the break label $\mathsf{brk\text{-}to}[\![\mathtt{S}]\!]$ (which is defined as the exit label of the closest enclosing iteration).

$$\boldsymbol{\mathcal{S}}^{*}[\![\mathtt{S}]\!] \triangleq \{\langle \pi\ell,\ \ell\rangle, \langle \pi\ell,\ \ell \xrightarrow{\mathtt{break}} \mathsf{brk\text{-}to}[\![\mathtt{S}]\!]\rangle \mid \pi\ell \in \mathbb{T}^+\} \tag{5}$$

• A prefix trace of a conditional statement $\mathtt{S} ::= \mathtt{\mathbf{if}}\ \ell\ \mathtt{(B)}\ \mathtt{S}_t$ continuing some initial trace $\pi_1\ell$ is

- either ℓ when the observation of the execution stops on entry of the program component;
- or, when the value of the boolean expression $\mathtt{B}$ on $\pi_1\ell$ is ff, ℓ followed by the event $\neg(\mathtt{B})$ and finishing at the label $\mathsf{aft}[\![\mathtt{S}]\!]$ after the conditional statement;
- or finally, when the value of the boolean expression $\mathtt{B}$ on $\pi_1\ell$ is tt, ℓ followed by the test event $\mathtt{B}$ followed by a prefix trace of $\mathtt{S}_t$ continuing $\pi_1\ell \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_t]\!]$.

$$\begin{aligned}\mathcal{S}^*[\![\mathtt{S}]\!] \triangleq{}& \{\langle\pi_1\ell,\ \ell\rangle \mid \pi_1\ell \in \mathbb{T}^+\} \\ &\cup \{\langle\pi_1\ell,\ \ell \xrightarrow{\neg(\mathtt{B})} \mathsf{aft}[\![\mathtt{S}]\!]\rangle \mid \mathcal{B}[\![\mathtt{B}]\!]\varrho(\pi_1\ell) = \mathsf{ff} \wedge \pi_1\ell \in \mathbb{T}^+\} \\ &\cup \{\langle\pi_1\ell,\ \ell \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_t]\!] \mathbin{\widehat{}} \pi_2\rangle \mid \mathcal{B}[\![\mathtt{B}]\!]\varrho(\pi_1\ell) = \mathsf{tt} \wedge \langle\pi_1\ell \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_t]\!],\ \pi_2\rangle \in \mathcal{S}^*[\![\mathtt{S}_t]\!]\}\end{aligned} \tag{6}$$

Notice that if π_2 starting $\mathsf{at}[\![\mathtt{S}_t]\!]$ is a maximal trace of $\mathtt{S}_t$ terminating $\mathsf{aft}[\![\mathtt{S}_t]\!]$ then $\ell \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_t]\!] \mathbin{\widehat{}} \pi_2$ is also a maximal trace of $\mathtt{S}$ terminating $\mathsf{aft}[\![\mathtt{S}]\!]$ since $\mathsf{aft}[\![\mathtt{S}_t]\!] = \mathsf{aft}[\![\mathtt{S}]\!]$.

Observe also that definition (6) includes the case of a conditional within an iteration and containing a break statement in the true branch $\mathtt{S}_t$. Since $\mathsf{brk\text{-}to}[\![\mathtt{S}]\!] = \mathsf{brk\text{-}to}[\![\mathtt{S}_t]\!]$, from $\langle\pi_1\ell \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_t]\!],\ \pi_2 \xrightarrow{\mathtt{break}} \mathsf{brk\text{-}to}[\![\mathtt{S}_t]\!]\rangle \in \mathcal{S}^*[\![\mathtt{S}_t]\!]$, we infer that $\langle\pi_1\ell,\ \ell \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_t]\!] \mathbin{\widehat{}} \pi_2 \xrightarrow{\mathtt{break}} \mathsf{brk\text{-}to}[\![\mathtt{S}]\!]\rangle \in \mathcal{S}^*[\![\mathtt{S}]\!]$.

- A prefix trace π of the empty statement list $\mathtt{Sl} ::= \epsilon$ is reduced to the program label at that empty statement.

$$\mathcal{S}^*[\![\mathtt{Sl}]\!] \triangleq \{\langle\pi\mathsf{at}[\![\mathtt{Sl}]\!],\ \mathsf{at}[\![\mathtt{Sl}]\!]\rangle \mid \pi\mathsf{at}[\![\mathtt{Sl}]\!] \in \mathbb{T}^+\} \tag{7}$$

- A prefix trace of a statement list $\mathtt{Sl} ::= \mathtt{Sl}'\ \mathtt{S}$ continuing an initial trace π_1 can be a prefix trace of $\mathtt{Sl}'$ or a finite maximal trace of $\mathtt{Sl}'$ followed by a prefix trace of $\mathtt{S}$.

$$\begin{aligned}\mathcal{S}^*[\![\mathtt{Sl}]\!] \triangleq{}& \mathcal{S}^*[\![\mathtt{Sl}']\!] \\ &\cup \{\langle\pi_1,\ \pi_2 \mathbin{\widehat{}} \pi_3\rangle \mid \langle\pi_1,\ \pi_2\rangle \in \mathcal{S}^*[\![\mathtt{Sl}']\!] \wedge \langle\pi_1 \mathbin{\widehat{}} \pi_2,\ \pi_3\rangle \in \mathcal{S}^*[\![\mathtt{S}]\!]\}\end{aligned} \tag{8}$$

Notice that if $\langle\pi_1 \mathbin{\widehat{}} \pi_2,\ \pi_3\rangle \in \mathcal{S}^*[\![\mathtt{S}]\!]$ then trace π_3 starts $\mathsf{at}[\![\mathtt{S}]\!] = \mathsf{aft}[\![\mathtt{Sl}']\!]$ so the trace π_2 in $\langle\pi_1, \pi_2\rangle \in \mathcal{S}^*[\![\mathtt{Sl}']\!]$ must end $\mathsf{aft}[\![\mathtt{Sl}']\!]$. Therefore π_2 must be a maximal terminating execution of $\mathtt{Sl}'$ *i.e.* $\mathtt{S}$ is executed only if $\mathtt{Sl}'$ terminates.

- The prefix finite trace semantic definition $\mathcal{S}^*[\![\mathtt{S}]\!]$ (9) of an iteration statement of the form $\mathtt{S} ::= \mathtt{while}\ \ell\ (\mathtt{B})\ \mathtt{S}_b$ is the $\subseteq$-least solution $\mathsf{lfp}^{\subseteq}\,\mathcal{F}^*[\![\mathtt{S}]\!]$ to the equation $X = \mathcal{F}^*[\![\mathtt{S}]\!](X)$. Since $\mathcal{F}^*[\![\mathtt{S}]\!] \in \wp(\mathbb{T}^+ \times \mathbb{T}^+) \to \wp(\mathbb{T}^+ \times \mathbb{T}^+)$ is $\subseteq$- monotone (if $X \subseteq X'$ then $\mathcal{F}^*[\![\mathtt{S}]\!](X) \subseteq \mathcal{F}^*[\![\mathtt{S}]\!](X')$) and $\langle\wp(\mathbb{T}^+ \times \mathbb{T}^+), \subseteq, \varnothing, \mathbb{T}^+ \times \mathbb{T}^+, \cup, \cap\rangle$ is a complete lattice, $\mathsf{lfp}^{\subseteq}\,\mathcal{F}^*[\![\mathtt{S}]\!]$ exists by Tarski's fixpoint theorem [30] and can be defined as the limit of iterates [11], which is useful to abstract into iterative static analysis algorithms. In definition (9) of the transformer $\mathcal{F}^*[\![\mathtt{S}]\!]$, case (9.a) corresponds to a loop execution observation stopping on entry, (9.b) corresponds to an observation of a loop exiting after 0 or more iterations, and (9.c) corresponds to a loop execution observation that stops anywhere in the body $\mathtt{S}_b$ after 0 or more iterations. This last case covers the case of an iteration terminated by a break statement (to $\mathsf{aft}[\![\mathtt{S}]\!]$ after the iteration statement).

$$\boldsymbol{\mathcal{S}}^*[\![\mathtt{S}]\!] = \mathsf{lfp}^{\subseteq}\, \boldsymbol{\mathcal{F}}^*[\![\mathtt{S}]\!] \tag{9}$$

$$\boldsymbol{\mathcal{F}}^*[\![\mathtt{while}\;\ell\;(\mathtt{B})\;\mathtt{S}_b]\!](X) \triangleq \{\langle \pi_1\ell',\ \ell'\rangle \mid \pi_1\ell' \in \mathbb{T}^+ \wedge \ell' = \ell\}\ ^{2} \qquad \text{(a)}$$

$$\cup \{\langle \pi_1\ell',\ \ell'\pi_2\ell' \xrightarrow{\neg(\mathtt{B})} \mathsf{aft}[\![\mathtt{S}]\!]\rangle \mid \langle \pi_1\ell',\ \ell'\pi_2\ell'\rangle \in X \wedge$$
$$\boldsymbol{\mathscr{B}}[\![\mathtt{B}]\!]\boldsymbol{\varrho}(\pi_1\ell'\pi_2\ell') = \mathrm{ff} \wedge \ell' = \ell\} \qquad \text{(b)}$$

$$\cup \{\langle \pi_1\ell',\ \ell'\pi_2\ell' \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_b]\!] \frown \pi_3\rangle \mid \langle \pi_1\ell',\ \ell'\pi_2\ell'\rangle \in X \wedge$$
$$\boldsymbol{\mathscr{B}}[\![\mathtt{B}]\!]\boldsymbol{\varrho}(\pi_1\ell'\pi_2\ell') = \mathrm{tt} \wedge \langle \pi_1\ell'\pi_2\ell' \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_b]\!],\ \pi_3\rangle \in \boldsymbol{\mathcal{S}}^*[\![\mathtt{S}_b]\!] \wedge \ell' = \ell\} \qquad \text{(c)}$$

• The prefix trace semantics of the other program components is similar. It follows that for each program component $\mathtt{S}$, we have

$$\{\langle \pi_1\mathsf{at}[\![\mathtt{S}]\!],\ \mathsf{at}[\![\mathtt{S}]\!]\rangle \mid \pi_1\mathsf{at}[\![\mathtt{S}]\!] \in \mathbb{T}^+\} \subseteq \boldsymbol{\mathcal{S}}^*[\![\mathtt{S}]\!] \tag{10}$$

2.5 Definition of the maximal trace semantics

The maximal trace semantics $\boldsymbol{\mathcal{S}}^{+\infty}[\![\mathtt{S}]\!] = \boldsymbol{\mathcal{S}}^{+}[\![\mathtt{S}]\!] \cup \boldsymbol{\mathcal{S}}^{\infty}[\![\mathtt{S}]\!]$ is derived from the prefix trace semantics $\boldsymbol{\mathcal{S}}^*[\![\mathtt{S}]\!]$ by keeping the longest finite traces $\boldsymbol{\mathcal{S}}^{+}[\![\mathtt{S}]\!]$ and passing to the limit $\boldsymbol{\mathcal{S}}^{\infty}[\![\mathtt{S}]\!]$ of prefix-closed traces for infinite traces.

$$\boldsymbol{\mathcal{S}}^{+}[\![\mathtt{S}]\!] \triangleq \{\langle \pi_1,\ \pi_2\ell\rangle \in \boldsymbol{\mathcal{S}}^*[\![\mathtt{S}]\!] \mid (\ell = \mathsf{aft}[\![\mathtt{S}]\!]) \vee (\mathsf{esc}[\![\mathtt{S}]\!] \wedge \ell = \mathsf{brk\text{-}to}[\![\mathtt{S}]\!])\} \tag{11}$$
$$\boldsymbol{\mathcal{S}}^{\infty}[\![\mathtt{S}]\!] \triangleq \lim(\boldsymbol{\mathcal{S}}^*[\![\mathtt{S}]\!]) \tag{12}$$

where the limit is $\lim \mathcal{T} \triangleq \{\langle \pi,\ \pi'\rangle \mid \pi' \in \mathbb{T}^{\infty} \wedge \forall n \in \mathbb{N}\ .\ \langle \pi,\ \pi'[0..n]\rangle \in \mathcal{T}\}$. (13)

The intuition for (13) is the following. Let $\mathtt{S}$ be an iteration. $\langle \pi,\ \pi'\rangle \in \boldsymbol{\mathcal{S}}^{\infty}[\![\mathtt{S}]\!] = \lim \boldsymbol{\mathcal{S}}^*[\![\mathtt{S}]\!]$ where π' is infinite if and only if, whenever we take a prefix $\pi'[0..n]$ of π', it is a possible finite observation of the execution of $\mathtt{S}$ and so belongs to the prefix trace semantics $\langle \pi,\ \pi'[0..n]\rangle \in \boldsymbol{\mathcal{S}}^*[\![\mathtt{S}]\!]$.

3 The semantic and syntactic liveness/deadness abstractions

3.1 The generic liveness/deadness abstractions

Informally "a variable is (potentially/definitely) live at some point if it holds a value that may/must be used in the future before the next time the variable is modified". The liveness abstraction $\alpha^{l}_{use,mod}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle \pi_0,\ \pi\rangle$ of a program trace π continuing an initial trace π_0 of a program component $\mathtt{S}$ is parameterized by

[2] A definition of the form $d(\vec{x}) \triangleq \{f(\vec{x}') \mid P(\vec{x}',\vec{x})\}$ has the variables $\vec{x}'$ in $P(\vec{x}',\vec{x})$ bound to those of $f(\vec{x}')$ whereas $\vec{x}$ is free in $P(\vec{x}',\vec{x})$ since it appears neither in $f(\vec{x}')$ nor (by assumption) under quantifiers in $P(\vec{x}',\vec{x})$. The $\vec{x}$ of $P(\vec{x}',\vec{x})$ is therefore bound to the $\vec{x}$ of $d(\vec{x})$.

– *use* defining the set $use[\![a]\!]\rho$ of variables which value is used when executing action a in environment ρ;
– *mod* defining the set $mod[\![a]\!]\rho$ of variables which value is modified when executing action a in environment ρ.

Liveness depends on the set L_b of variables assumed to be live on exit of the program component S by a break statement and L_e by a normal exit after S. It is defined inductively on a finite trace (or co-inductively for an infinite trace) as follows

$$\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \ell\rangle \triangleq \{\mathsf{x} \in \mathbb{V} \mid (\ell = \mathsf{aft}[\![\mathsf{S}]\!] \wedge \mathsf{x} \in L_e) \vee (\mathsf{esc}[\![\mathsf{S}]\!] \wedge \ell = \mathsf{brk\text{-}to}[\![\mathsf{S}]\!] \wedge \mathsf{x} \in L_b)\} \quad \text{(a)} \quad (14)$$

$$\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \ell \xrightarrow{a} \ell'\pi_1\rangle \triangleq \{\mathsf{x} \in \mathbb{V} \mid \mathsf{x} \in use[\![a]\!]\varrho(\pi_0) \vee (\mathsf{x} \notin mod[\![a]\!]\varrho(\pi_0) \wedge \mathsf{x} \in \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0 \frown \ell \xrightarrow{a} \ell',\ \ell'\pi_1\rangle)\} \quad \text{(b)}$$

The potential and definite liveness are abstractions of the maximal trace semantics $\boldsymbol{\mathcal{S}} = \boldsymbol{\mathcal{S}}^{+\infty}[\![\mathsf{S}]\!]$ is by merge over all traces

$$\alpha^{\exists l}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ L_b, L_e = \bigcup_{\langle\pi_0,\,\pi\rangle\,\in\,\boldsymbol{\mathcal{S}}} \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi\rangle \qquad \text{potential liveness} \quad (15)$$

$$\alpha^{\forall l}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ L_b, L_e = \bigcap_{\langle\pi_0,\,\pi\rangle\,\in\,\boldsymbol{\mathcal{S}}} \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi\rangle \qquad \text{definite liveness} \quad (16)$$

Potential and definite deadness are defined dually.

$$\alpha^{\exists d}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ D_b, D_e = \neg\alpha^{\forall l}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ \neg D_b, \neg D_e \qquad \text{potential deadness} \quad (17)$$

$$\alpha^{\forall d}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ D_b, D_e = \neg\alpha^{\exists l}_{use,mod}[\![\mathsf{S}]\!]\ \boldsymbol{\mathcal{S}}\ \neg D_b, \neg D_e \qquad \text{definite deadness} \quad (18)$$

If S and S' have the same aft, esc, and $\mathsf{brk\text{-}to}$ labelling, they have the same $\alpha^{l}_{use,mod}$, $\alpha^{\exists l}_{use,mod}$, $\alpha^{\forall l}_{use,mod}$, $\alpha^{\exists d}_{use,mod}$, and $\alpha^{\exists l}_{use,mod}$.

Unfolding the recursive definition (14) , we get

Lemma 1 *If* $\pi_1 = \ell_1 \xrightarrow{a_1} \ell_2 \xrightarrow{a_2} \dots \xrightarrow{a_{n-1}} \ell_n$ *and* $\langle\pi_0,\ \pi_1\rangle \in \boldsymbol{\mathcal{S}}^*[\![\mathsf{S}]\!]$ *then*

$$\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle = \{\mathsf{x} \in \mathbb{V} \mid \exists i \in [1, n-1]\ .\ \forall j \in [1, i-1]\ .\ \mathsf{x} \notin mod[\![a_j]\!]\varrho(\pi_0 \frown \ell_1 \xrightarrow{a_1} \ell_2 \dots \xrightarrow{a_{j-1}} \ell_j) \wedge \mathsf{x} \in use[\![a_i]\!]\varrho(\pi_0 \frown \ell_1 \xrightarrow{a_1} \ell_2 \dots \xrightarrow{a_{i-1}} \ell_i)\} \cup (\!| \ell_n = \mathsf{aft}[\![\mathsf{S}]\!] \ ?\ L_e \ ⨾\ \varnothing |\!) \cup (\!| \mathsf{esc}[\![\mathsf{S}]\!] \wedge \ell_n = \mathsf{brk\text{-}to}[\![\mathsf{S}]\!] \ ?\ L_b \ ⨾\ \varnothing |\!). \quad \square$$

Proof (of Lem. 1) For the basis $n = 1$, only the first clause (a) of (14) is applicable with $\pi_1 = \ell_1$, $[1, n-1]$ is empty, and $\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle = (\!| \ell_1 = \mathsf{aft}[\![\mathsf{S}]\!] \ ?\ L_e \ ⨾\ \varnothing |\!) \cup (\!| \mathsf{esc}[\![\mathsf{S}]\!] \wedge \ell_1 = \mathsf{brk\text{-}to}[\![\mathsf{S}]\!] \ ?\ L_b \ ⨾\ \varnothing |\!)$ which is precisely what is given by Lem. 1 since $[1, n-1] = \varnothing$ so the first term is empty.

For the induction step $n+1 > 1$, we have $\pi_1 = \ell_1 \xrightarrow{a_1} \ell_2 \xrightarrow{a_2} \ell_3 \xrightarrow{a_3} \dots \xrightarrow{a_n} \ell_{n+1}$ and only the second clause (b) of (14) is applicable so we get

$$
\begin{aligned}
&\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \qquad \wr\text{assuming } n+1 \geqslant 2\wr\\
=\ &\{\mathsf{x}\in\mathbb{V} \mid \mathsf{x}\in use[\![a_1]\!]\boldsymbol{\varrho}(\pi_0) \vee (\mathsf{x}\notin mod[\![a_1]\!]\boldsymbol{\varrho}(\pi_0)) \wedge \mathsf{x}\in\alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0 \frown \\
&\ell_1\xrightarrow{a_1}\ell_2,\ \ell_2\xrightarrow{a_2}\ell_3\xrightarrow{a_3}\ldots\xrightarrow{a_n}\ell_{n+1}\rangle)\} \qquad \wr(14.\text{b}) \text{ when } n>1\wr\\
=\ &\{(\mathsf{x}\in\mathbb{V} \mid \mathsf{x}\in use[\![a_1]\!]\boldsymbol{\varrho}(\pi_0)) \vee (\mathsf{x}\notin mod[\![a_1]\!]\boldsymbol{\varrho}(\pi_0) \wedge \exists i\in[2,n]\ .\ \forall j\in[2,i-1]\ .\ \mathsf{x}\notin\\
&mod[\![a_j]\!]\boldsymbol{\varrho}(\pi_0\frown\ell_1\xrightarrow{a_1}\ell_2\ldots\xrightarrow{a_{j-1}}\ell_j)\wedge\mathsf{x}\in use[\![a_i]\!]\boldsymbol{\varrho}(\pi_0\frown\ell_1\xrightarrow{a_1}\ell_2\ldots\xrightarrow{a_{i-1}}\ell_i))\vee\\
&\llparenthesis\ \ell_{n+1}=\mathsf{aft}[\![\mathsf{S}]\!]\ ?\ \mathsf{x}\in L_e : \mathsf{ff}\ \rrparenthesis \vee \llparenthesis\ \mathsf{esc}[\![\mathsf{S}]\!]\wedge\ell_{n+1}=\mathsf{brk\text{-}to}[\![\mathsf{S}]\!]\ ?\ \mathsf{x}\in L_b : \mathsf{ff}\ \rrparenthesis\}\\
&\qquad\wr\text{since } \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0\frown\ell_1\xrightarrow{a_1},\ \ell_2\xrightarrow{a_2}\ \ldots\xrightarrow{a_n}\ell_{n+1}\rangle=\{\mathsf{x}\in\mathbb{V}\mid\\
&\qquad\exists i\in[2,n]\ .\ \forall j\in[2,i-1]\ .\ \mathsf{x}\notin mod[\![a_j]\!]\boldsymbol{\varrho}(\pi_0\frown\ell_1\xrightarrow{a_1}\ell_2\ldots\xrightarrow{a_{j-1}}\ell_j)\wedge\mathsf{x}\in\\
&\qquad use[\![a_i]\!]\boldsymbol{\varrho}(\pi_0\frown\ell_1\xrightarrow{a_1}\ell_2\ldots\xrightarrow{a_{i-1}}\ell_i)\}\cup\llparenthesis\ \ell_{n+1}=\mathsf{aft}[\![\mathsf{S}]\!]\ ?\ L_e : \varnothing\ \rrparenthesis\cup\llparenthesis\ \mathsf{esc}[\![\mathsf{S}]\!]\wedge\\
&\qquad\ell_{n+1}=\mathsf{brk\text{-}to}[\![\mathsf{S}]\!]\ ?\ L_b : \varnothing\ \rrparenthesis \text{ by ind. hyp. for Lem. 1}\wr\\
=\ &\{\mathsf{x}\in\mathbb{V}\mid\exists i\in[1,n]\ .\ \forall j\in[1,i-1]\ .\ \mathsf{x}\notin mod[\![a_j]\!]\boldsymbol{\varrho}(\pi_0\frown\ell_1\xrightarrow{a_1}\ell_2\ldots\xrightarrow{a_{j-1}}\ell_j)\wedge\mathsf{x}\in\\
&use[\![a_i]\!]\boldsymbol{\varrho}(\pi_0\frown\ell_1\xrightarrow{a_1}\ell_2\ldots\xrightarrow{a_{i-1}}\ell_i)\}\cup\llparenthesis\ \ell_{n+1}=\mathsf{aft}[\![\mathsf{S}]\!]\ ?\ L_e : \varnothing\ \rrparenthesis\cup\llparenthesis\ \mathsf{esc}[\![\mathsf{S}]\!]\wedge\ell_{n+1}=\\
&\mathsf{brk\text{-}to}[\![\mathsf{S}]\!]\ ?\ L_b : \varnothing\ \rrparenthesis\\
&\qquad\wr\text{incorporating } (\mathsf{x}\in\mathbb{V}\mid\mathsf{x}\in use[\![a_1]\!]\boldsymbol{\varrho}(\pi_0)) \text{ in the case } i=1 \text{ for which}\\
&\qquad[1,i-1]=\varnothing \text{ and } \boldsymbol{\varrho}(\pi_0\frown\ell_1\xrightarrow{a_1}\ell_2\ldots\xrightarrow{a_{i-1}}\ell_i)=\boldsymbol{\varrho}(\pi_0\frown\ell_1)=\boldsymbol{\varrho}(\pi_0).\wr
\end{aligned}
$$

This proves Lem. 1 for the induction step and we conclude by recurrence on n. ◻

We also observe that potential liveness (hence dually definite deadness) can be equivalently defined using maximal or prefix traces.

Lemma 2 $\alpha^{\exists l}_{use,mod}[\![\mathsf{S}]\!]\ (\boldsymbol{\mathcal{S}}^{+\infty}[\![\mathsf{S}]\!]) = \alpha^{\exists l}_{use,mod}[\![\mathsf{S}]\!]\ (\boldsymbol{\mathcal{S}}^{*}[\![\mathsf{S}]\!])$. ◻

Proof of Lem. 2. To show that $\alpha^{\exists l}_{use,mod}[\![\mathsf{S}]\!]\ (\boldsymbol{\mathcal{S}}^{+\infty}[\![\mathsf{S}]\!]) = \alpha^{\exists l}_{use,mod}[\![\mathsf{S}]\!]\ (\boldsymbol{\mathcal{S}}^{*}[\![\mathsf{S}]\!])$ we must, by (15), prove that

$$
A = \bigcup_{\langle\pi_0,\pi\rangle\in\boldsymbol{\mathcal{S}}^{+\infty}[\![\mathsf{S}]\!]} \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\langle\pi_0,\pi\rangle = \bigcup_{\langle\pi_0,\pi'\rangle\in\boldsymbol{\mathcal{S}}^{*}[\![\mathsf{S}]\!]} \alpha^{l}_{use,mod}[\![\mathsf{S}]\!]\ L_b, L_e\langle\pi_0,\pi'\rangle = B.
$$

– Assume $\mathsf{x}\in A$ because of some $\langle\pi_0,\ \pi\rangle\ \in\ \boldsymbol{\mathcal{S}}^{+\infty}[\![\mathsf{S}]\!]$. There are two cases.
  - Either $\mathsf{x}\in A$ follows from (14.a) and so the second alternative in (14.b) has always been chosen before reaching the end of the trace π with a label $\ell=\mathsf{aft}[\![\mathsf{S}]\!]$ or $\mathsf{esc}[\![\mathsf{S}]\!]=\mathsf{tt}$ and $\ell=\mathsf{brk\text{-}to}[\![\mathsf{S}]\!]$. In both cases, π is maximal by (11), $\langle\pi_0,\ \pi\rangle\ \in\ \boldsymbol{\mathcal{S}}^{*}[\![\mathsf{S}]\!]$, and so $\mathsf{x}\in B$ by (14).
  - Otherwise, $\mathsf{x}\in A$ follows from (14.b) where the second alternative has been chosen finitely many times (so x is unmodified) until the first alternative is chosen because x is used. Consider the prefix of π up to that point of use. By (13), it is, or an extension of it, π' is in the prefix semantics $\langle\pi_0,\ \pi'\rangle\in\boldsymbol{\mathcal{S}}^{*}[\![\mathsf{S}]\!]$ and so from this trace we derive from (14.b) that $\mathsf{x}\in B$.

It follows that $A \subseteq B$.

– Conversely, assume $\mathtt{x} \in B$. Then there exists $\langle\pi_0,\ \pi'\rangle \in \boldsymbol{\mathcal{S}}^*[\![\mathtt{S}]\!]$ such that $\mathtt{x} \in \alpha^l_{use,mod}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi'\rangle$. Consider a maximal extension of π' so that there exists π'' with $\langle\pi_0,\ \pi'\cdot\pi''\rangle \in \boldsymbol{\mathcal{S}}^{+\infty}[\![\mathtt{S}]\!]$. There are two cases, depending of whether $\mathtt{x} \in B$ in (14.a) or (14.b).
  - If $\mathtt{x} \in B$ because of (14.a) then the π' ends at $\mathsf{aft}[\![\mathtt{S}]\!]$ or at $\mathsf{brk\text{-}to}[\![\mathtt{S}]\!]$ and so π' is maximal that is $\langle\pi_0,\ \pi'\rangle \in \boldsymbol{\mathcal{S}}^{+\infty}[\![\mathtt{S}]\!]$ and so $\mathtt{x} \in A$.
  - If $\mathtt{x} \in B$ because of (14.b) then $\mathtt{x} \in B$ is used in π' without being modified before and so this is also the case in $\langle\pi_0,\ \pi'\cdot\pi''\rangle \in \boldsymbol{\mathcal{S}}^{+\infty}[\![\mathtt{S}]\!]$, $\pi'' = \ni$, and then $\mathtt{x} \in A$ by (14).

  In both cases, $B \subseteq A$.

– By antisymmetry, $A = B$. □

3.2 The semantic liveness/deadness abstractions

Semantically, an action a uses variable $\mathtt{y}$ in a given environment ρ if and only if it is possible to change the value of $\mathtt{y}$ so as to change the effect of action a on program execution. For an assignment, the assigned value will be changed. For a test, which has no side effect, the branch taken will be different. For example, $\mathtt{y} \notin \mathsf{use}[\![\mathtt{x = y - y}]\!]\,\rho$ and $\mathtt{x} \notin \mathsf{use}[\![\mathtt{x = x}]\!]\,\rho$. Formally,

$$\begin{aligned}
\mathsf{use}[\![\mathtt{skip}]\!]\,\rho &\triangleq \emptyset \\
\mathsf{use}[\![\mathtt{x = A}]\!]\,\rho &\triangleq \{\mathtt{y} \mid \exists \nu \in \mathbb{V}\ .\ \boldsymbol{\mathcal{A}}[\![\mathtt{A}]\!]\,\rho \neq \boldsymbol{\mathcal{A}}[\![\mathtt{A}]\!]\,\rho[\mathtt{y} \leftarrow \nu] \wedge \rho(\mathtt{x}) \neq \boldsymbol{\mathcal{A}}[\![\mathtt{A}]\!]\,\rho\} \\
\mathsf{use}[\![a]\!]\,\rho &\triangleq \{\mathtt{y} \mid \exists \nu \in \mathbb{V}\ .\ \boldsymbol{\mathcal{B}}[\![a]\!]\,\rho \neq \boldsymbol{\mathcal{B}}[\![a]\!]\,\rho[\mathtt{y} \leftarrow \nu]\} \qquad a \in \{\mathtt{B}, \neg(\mathtt{B})\}
\end{aligned} \tag{19}$$

Notice that $\mathtt{x} \in \mathsf{use}[\![a]\!]$ in (19) compares two executions of action a in different environments so that (14) is a dependency analysis involving a trace and the abstraction of another one by a different environment [10]. An action a modifies variable $\mathtt{x}$ in an environment ρ if and only the execution of action a in environment ρ changes the value of $\mathtt{x}$. This corresponds to

$$\mathsf{mod}[\![a]\!]\,\rho \triangleq \{\mathtt{x} \mid a = (\mathtt{x = A}) \wedge (\rho(\mathtt{x}) \neq \boldsymbol{\mathcal{A}}[\![\mathtt{A}]\!]\,\rho)\}$$

So the semantic potential liveness abstract semantics is

$$\boldsymbol{\mathcal{S}}^{\exists l}[\![\mathtt{S}]\!] \triangleq \alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}[\![\mathtt{S}]\!]\ (\boldsymbol{\mathcal{S}}^{+\infty}[\![\mathtt{S}]\!]) \tag{20}$$

instantiating (15) with *use* as use and *mod* as mod (and similarly for the other cases).

3.3 The classical syntactic liveness/deadness abstractions

Classical dataflow analysis as considered in [25] is purely syntactic *i.e.* approximates semantic properties by coarser syntactic ones based on the program syntax only. The set $\mathbb{use}[\![a]\!]$ of variables used and the set $\mathbb{mod}[\![a]\!]$ of variables assigned to/modified in an action $a \in \mathbb{A}$ are postulated to be as follows (the parameter ρ is useless but added for consistency with (14)).

$$\begin{array}{llll} \mathbb{use}[\![\mathsf{x = A}]\!]\,\rho \triangleq \mathbb{vars}[\![\mathsf{A}]\!] & \mathbb{use}[\![\mathsf{skip}]\!]\,\rho \triangleq \varnothing & \mathbb{use}[\![\mathsf{B}]\!]\,\rho \triangleq & \mathbb{use}[\![\neg(\mathsf{B})]\!]\,\rho \triangleq \mathbb{vars}[\![\mathsf{B}]\!] \\ \mathbb{mod}[\![\mathsf{x = A}]\!]\,\rho \triangleq \{\mathsf{x}\} & \mathbb{mod}[\![\mathsf{skip}]\!]\,\rho \triangleq \varnothing & \mathbb{mod}[\![\mathsf{B}]\!]\,\rho \triangleq & \mathbb{mod}[\![\neg(\mathsf{B})]\!]\,\rho \triangleq \varnothing \end{array} \tag{21}$$

where $\mathbb{vars}[\![\mathsf{E}]\!]$ is the set of program variables occurring in arithmetic or boolean expression E.

So the classical syntactic potential liveness abstract semantics is

$$\mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!] \triangleq \alpha^{\exists l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\;(\mathcal{S}^{+\infty}[\![\mathsf{S}]\!]) \tag{22}$$

instantiating (15) with *use* as $\mathbb{use}$ and *mod* as $\mathbb{mod}$ (and similarly for the other cases).

3.4 Unsoundness of the syntactic liveness/deadness abstractions

One would expect soundness that is the potentially live variables determined syntactically by [25] is a pointwise over-approximation of the potentially live variables determined semantically but this is wrong $\mathcal{S}^{\exists l}[\![\mathsf{S}]\!] \dot{\nsubseteq} \mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$, as shown by Ex. 2. The problem is that

$$\exists\rho \in \mathbb{Ev} \,.\, \mathsf{y} \in \mathsf{use}[\![a]\!]\,\rho \Rightarrow \forall\rho \in \mathbb{Ev} \,.\, \mathsf{y} \in \mathbb{use}[\![a]\!]\,\rho \tag{23}$$

but in general, as shown by Ex. 2, $\exists\rho \in \mathbb{Ev} \,.\, \mathsf{x} \in \mathbb{mod}[\![a]\!]\,\rho \wedge \mathsf{x} \notin \mathsf{mod}[\![a]\!]\,\rho$.

Proof of (23). Let us first remark that if $\mathsf{x} \notin \mathbb{vars}[\![\mathsf{B}]\!]$ and $\forall \mathsf{y} \in \mathbb{V} \setminus \{\mathsf{x}\} \,.\, \rho'(\mathsf{y}) = \rho(\mathsf{y})$ then $\mathcal{B}[\![\mathsf{B}]\!]\rho = \mathcal{B}[\![\mathsf{B}]\!]\rho'$ and similarly for arithmetic expressions.

(23) is trivial for skip since $\mathsf{use}[\![\mathsf{skip}]\!]\,\rho\;\mathsf{y} = \mathit{ff}$ in (19). Otherwise, by contraposition, assume that $\mathsf{y} \notin \mathbb{use}[\![a]\!]\rho$.

— If $a = \mathsf{x = A}$ then $\mathsf{y} \notin \mathbb{vars}[\![\mathsf{A}]\!]$ by (21) so $\forall v \in \mathbb{V} \,.\, \mathcal{A}[\![\mathsf{A}]\!]\,\rho = \mathcal{A}[\![\mathsf{A}]\!]\,\rho[\mathsf{y} \leftarrow v]$, proving $\neg(\mathsf{use}[\![\mathsf{x = A}]\!]\,\rho\;\mathsf{y})$ by (19).

— Similarly if $a = \mathsf{B}$ or $a = \neg(\mathsf{B})$ then changing y does not change the value of the boolean expression so y is not semantically used by (19). □

3.5 Soundness of the syntactic liveness/deadness abstractions with respect to revised syntactic/semantic liveness/deadness abstractions

To fix the problem $\mathcal{S}^{\exists l}[\![\mathsf{S}]\!] \dot{\nsubseteq} \mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$, we can either change $\alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}$ or $\alpha^{\exists l}_{\mathbb{use},\mathbb{mod}}$. Changing $\alpha^{\exists l}_{\mathbb{use},\mathbb{mod}}$ would mean changing the classical potential live variable algorithm [20,19,21] and all compilers using it. So we change $\alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}$ so as to explain exactly in what sense the unchanged classical potential live variable algorithm is sound (even if this is not the most semantically intuitive one). We remark that we have $\alpha^{\exists l}_{\mathsf{use},\mathbb{mod}} \dot{\subseteq} \alpha^{\exists l}_{\mathbb{use},\mathbb{mod}}$ so the classical potential live variable algorithm $\mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$ which over-approximates $\alpha^{\exists l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\;(\mathcal{S}^{+\infty}[\![\mathsf{S}]\!])$ is sound. However, the program transformations that preserve mod but not $\mathbb{mod}$ may change the liveness analysis. Therefore we define

$$\mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!] \triangleq \alpha^{\exists l}_{\mathsf{use},\mathbb{mod}}\;(\mathcal{S}^{+\infty}[\![\mathsf{S}]\!]) \tag{24}$$

Theorem 1 *If* $\alpha^{\exists l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ (\mathcal{S}^{+\infty}[\![\mathsf{S}]\!]) \mathrel{\dot{\subseteq}} \mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$ *then* $\mathcal{S}^{\exists l}[\![\mathsf{S}]\!] \mathrel{\dot{\subseteq}} \mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$.

Proof of Th. 1. We have to prove that $\alpha^{\exists l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!] \mathrel{\dot{\subseteq}} \alpha^{\exists l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]$, pointwise. We first prove that $\alpha^{l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!] \mathrel{\dot{\subseteq}} \alpha^{l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]$. We proceed by induction (more precisely bi-induction [15] to account for infinite traces).

— For the basis

$$\begin{aligned}
&\alpha^{l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \ell\rangle\\
=\ &\{\mathsf{x} \in \mathbb{V} \mid (\ell = \mathsf{aft}[\![\mathsf{S}]\!] \wedge \mathsf{x} \in L_e) \vee (\mathsf{esc}[\![\mathsf{S}]\!] \wedge \ell = \mathsf{brk\text{-}to}[\![\mathsf{S}]\!] \wedge \mathsf{x} \in L_b)\} && \wr(14.\mathrm{a})\int\\
\subseteq\ &\alpha^{l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \ell\rangle && \wr(14.\mathrm{a})\text{ and }\subseteq\text{ reflexive}\int
\end{aligned}$$

— For the induction step

$$\begin{aligned}
&\alpha^{l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \ell \xrightarrow{a} \ell'\pi_1\rangle\\
=\ &\{\mathsf{x} \in \mathbb{V} \mid \mathsf{x} \in \mathsf{use}[\![a]\!]\varrho(\pi_0) \vee (\mathsf{x} \notin \mathbb{mod}[\![a]\!]\varrho(\pi_0) \wedge \mathsf{x} \in \alpha^{l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0 \frown \ell \xrightarrow{a} \ell',\\
&\ell'\pi_1\rangle)\} && \wr(14.\mathrm{b})\int\\
\subseteq\ &\{\mathsf{x} \in \mathbb{V} \mid \mathsf{x} \in \mathbb{use}[\![a]\!]\varrho(\pi_0) \vee (\mathsf{x} \notin \mathbb{mod}[\![a]\!]\varrho(\pi_0) \wedge \mathsf{x} \in \alpha^{l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0 \frown \ell \xrightarrow{a} \ell',\\
&\ell'\pi_1\rangle)\} && \wr(23)\int\\
\subseteq\ &\{\mathsf{x} \in \mathbb{V} \mid \mathsf{x} \in \mathbb{use}[\![a]\!]\varrho(\pi_0) \vee (\mathsf{x} \notin \mathbb{mod}[\![a]\!]\varrho(\pi_0) \wedge \mathsf{x} \in \alpha^{l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0 \frown \ell \xrightarrow{a} \ell',\\
&\ell'\pi_1\rangle)\} && \wr\text{ind. hyp.}\int\\
=\ &\alpha^{l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \ell \xrightarrow{a} \ell'\pi_1\rangle && \wr(14.\mathrm{b})\int
\end{aligned}$$

It follows that

$$\begin{aligned}
&\alpha^{\exists l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ \mathcal{S}\ L_b, L_e\\
=\ &\bigcup_{\langle\pi_0,\ \pi\rangle \in \mathcal{S}} \alpha^{l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi\rangle && \wr(15)\int\\
\subseteq\ &\bigcup_{\langle\pi_0,\ \pi\rangle \in \mathcal{S}} \alpha^{l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi\rangle && \wr\alpha^{l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!] \mathrel{\dot{\subseteq}} \alpha^{l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\int\\
=\ &\alpha^{\exists l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ \mathcal{S}\ L_b, L_e && \wr(15)\int
\end{aligned}$$

If $\alpha^{\exists l}_{\mathbb{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ (\mathcal{S}^{+\infty}[\![\mathsf{S}]\!]) \mathrel{\dot{\subseteq}} \mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$ then $\alpha^{\exists l}_{\mathsf{use},\mathbb{mod}}[\![\mathsf{S}]\!]\ (\mathcal{S}^{+\infty}[\![\mathsf{S}]\!]) \mathrel{\dot{\subseteq}} \mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$ and therefore, by (24), $\mathcal{S}^{\exists l}[\![\mathsf{S}]\!] \triangleq \alpha^{\exists l}_{\mathsf{use},\mathbb{mod}}\ (\mathcal{S}^{+\infty}[\![\mathsf{S}]\!]) \mathrel{\dot{\subseteq}} \mathcal{S}^{\exists\mathbb{l}}[\![\mathsf{S}]\!]$.

The other cases $\mathcal{S}^{\forall\mathbb{l}}[\![\mathsf{S}]\!]$, $\mathcal{S}^{\exists\mathbb{d}}[\![\mathsf{S}]\!]$, and $\mathcal{S}^{\forall\mathbb{d}}[\![\mathsf{S}]\!]$ are similar.

4 Calculational design of the structural syntactic potential liveness static analysis

By Th. 1, a liveness inference algorithm $\mathcal{S}^{\exists l}[\![\mathtt{S}]\!]$ is sound whenever

$$\alpha^{\exists l}_{\mathtt{use},\mathtt{mod}}[\![\mathtt{S}]\!]\ (\mathcal{S}^{+\infty}[\![\mathtt{S}]\!]) \mathrel{\dot\subseteq} \mathcal{S}^{\exists l}[\![\mathtt{S}]\!],$$

equivalently

$$\alpha^{\exists l}_{\mathtt{use},\mathtt{mod}}[\![\mathtt{S}]\!]\ (\mathcal{S}^{*}[\![\mathtt{S}]\!]) \mathrel{\dot\subseteq} \mathcal{S}^{\exists l}[\![\mathtt{S}]\!]$$

by Lem. 2. So we can construct this algorithm $\mathcal{S}^{\exists l}[\![\mathtt{S}]\!]$ by a calculus that simplifies the term $\alpha^{\exists l}_{\mathtt{use},\mathtt{mod}}[\![\mathtt{S}]\!]\ (\mathcal{S}^{*}[\![\mathtt{S}]\!])$. Since the semantics $\mathcal{S}^{*}[\![\mathtt{S}]\!]$ is structural, we get a structural algorithm which proceeds by elimination, without any fixpoint iteration. We first give the result in Figure 1 and then show the systematic calculational design [8]. Notice that although the semantics is forward, the analysis is backward (see *e.g.* the statement list and iteration). We omit the unused environment parameter of $\mathtt{use}$ and $\mathtt{mod}$.

Structural syntactic potential liveness analysis

$$\begin{aligned}
\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{Sl}\ \ell]\!]\ L_e &\triangleq \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{Sl}\ \ell]\!]\ \varnothing, L_e &(25)\\
\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{x = A ;}]\!]\ L_b, L_e &\triangleq \mathtt{use}[\![\mathtt{x = A}]\!] \cup (L_e \setminus \mathtt{mod}[\![\mathtt{x = A}]\!])\\
\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{;}]\!]\ L_b, L_e &\triangleq L_e\\
\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{Sl'\ S}]\!]\ L_b, L_e &\triangleq \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{Sl'}]\!]\ L_b, (\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}]\!]\ L_b, L_e)\\
\widehat{\mathcal{S}}^{\exists l}[\![\ \epsilon\ ]\!]\ L_b, L_e &\triangleq L_e\\
\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{if\ (B)\ S}_t]\!]\ L_b, L_e &\triangleq \mathtt{use}[\![\mathtt{B}]\!] \cup L_e \cup \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}_t]\!]\ L_b, L_e\\
\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_f]\!]\ L_b, L_e &\triangleq \mathtt{use}[\![\mathtt{B}]\!] \cup \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}_t]\!]\ L_b, L_e \cup \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}_f]\!]\ L_b, L_e\\
\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{while\ (B)\ S}_b]\!]\ L_b, L_e &\triangleq \mathtt{use}[\![\mathtt{B}]\!] \cup L_e \cup \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}_b]\!]\ L_b, L_e\\
\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{break\ ;}]\!]\ L_b, L_e &\triangleq L_b\\
\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{\{\ Sl\ \}}]\!]\ L_b, L_e &\triangleq \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{Sl}]\!]\ L_b, L_e &\square
\end{aligned}$$

Fig. 1. Potential liveness

Theorem 2 *$\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}]\!]$ defined by (25) is syntactically sound that is $\mathcal{S}^{\exists l}[\![\mathtt{S}]\!] = \alpha^{\exists l}_{\mathtt{use},\mathtt{mod}}[\![\mathtt{S}]\!]\ (\mathcal{S}^{*}[\![\mathtt{S}]\!]) \mathrel{\dot\subseteq} \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}]\!]$.*

Proof of Th. 2. By structural induction on $\mathtt{S}$. We provide an example of a base case (assignment) and an inductive case (iteration), all other cases are similar.

– For the ***assignment*** $\mathtt{S} ::= \ell\ \mathtt{x = A ;}$, let us calculate $\mathcal{S}^{\exists l}[\![\mathtt{S}]\!]\ L_b, L_e$

$$= \alpha^{\exists l}_{\mathtt{use},\mathtt{mod}}[\![\mathtt{S}]\!](\mathcal{S}^{*}[\![\mathtt{S}]\!])\ L_b, L_e \qquad \text{⦃(22) and Lem. 2⦄}$$

$$= \bigcup\{\alpha^{l}_{\mathtt{use},\mathtt{mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \mid \langle\pi_0,\ \pi_1\rangle\ \in\ \widehat{\mathcal{S}}^{*}[\![\mathtt{S}]\!]\} \qquad \text{⦃def. (15) of } \alpha^{\exists l}_{\mathtt{use},\mathtt{mod}}[\![\mathtt{S}]\!]\text{⦄}$$

$= \bigcup\{\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ \ L_b, L_e\ \ \langle\pi_0\mathsf{at}[\![\mathtt{S}]\!],\ \ \mathsf{at}[\![\mathtt{S}]\!]\rangle\}\ \cup\ \bigcup\{\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ \ L_b, L_e\ \ \langle\pi_0\mathsf{at}[\![\mathtt{S}]\!],$
$\mathsf{at}[\![\mathtt{S}]\!] \xrightarrow{\mathtt{x\ =\ A} = \mathscr{A}[\![\mathtt{A}]\!]\varrho(\pi_0\mathsf{at}[\![\mathtt{S}]\!])} \mathsf{aft}[\![\mathtt{S}]\!]\rangle\}$ $\wr$def. (3) of $\mathscr{S}^*[\![\mathtt{S}]\!]\int$

$= \bigcup\{\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0\mathsf{at}[\![\mathtt{S}]\!],\ \mathsf{at}[\![\mathtt{S}]\!] \xrightarrow{\mathtt{x\ =\ A} = \mathscr{A}[\![\mathtt{A}]\!]\varrho(\pi_0\mathsf{at}[\![\mathtt{S}]\!])} \mathsf{aft}[\![\mathtt{S}]\!]\rangle\}$
$\wr$def. (14.a) of $\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0\mathsf{at}[\![\mathtt{S}]\!],\ \mathsf{at}[\![\mathtt{S}]\!]\rangle = \varnothing\int$

$= \bigcup\{\mathtt{y} \in \mathbb{V} \mid \mathtt{y} \in \mathtt{use}[\![\mathtt{x\ =\ A}]\!]\varrho(\pi_0\mathsf{at}[\![\mathtt{S}]\!]) \vee (\mathtt{y} \notin \mathtt{mod}[\![\mathtt{x\ =\ A}]\!]\varrho(\pi_0\mathsf{at}[\![\mathtt{S}]\!]) \wedge \mathtt{y} \in$
$\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0\mathsf{at}[\![\mathtt{S}]\!] \smallfrown \mathsf{at}[\![\mathtt{S}]\!] \xrightarrow{\mathtt{x\ =\ A} = \mathscr{A}[\![\mathtt{A}]\!]\varrho(\pi_0\mathsf{at}[\![\mathtt{S}]\!])} \mathsf{aft}[\![\mathtt{S}]\!],\ \mathsf{aft}[\![\mathtt{S}]\!]\rangle)\}$
$\wr$def. (14.b) of $\alpha^{l}_{\mathtt{use,mod}}\ L_b, L_e\ \langle\pi_0\mathsf{at}[\![\mathtt{S}]\!],\ \mathsf{at}[\![\mathtt{S}]\!] \xrightarrow{\mathtt{x\ =\ A} = \mathscr{A}[\![\mathtt{A}]\!]\varrho(\pi_0\mathsf{at}[\![\mathtt{S}]\!])} \mathsf{aft}[\![\mathtt{S}]\!]\rangle\int$

$= \{\mathtt{y} \in \mathbb{V} \mid \mathtt{y} \in \mathtt{use}[\![\mathtt{x\ =\ A}]\!] \vee (\mathtt{y} \notin \mathtt{mod}[\![\mathtt{x\ =\ A}]\!] \wedge \mathtt{y} \in L_e)\}$
$\wr$def. (14.a) of $\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \mathsf{aft}[\![\mathtt{S}]\!]\rangle \triangleq \{\mathtt{x} \in \mathbb{V} \mid \mathtt{x} \in L_e\} = L_e$ since $\mathsf{esc}[\![\mathtt{S}]\!] = \mathsf{ff}$ and omitting the useless parameters of $\mathtt{use}$ and $\mathtt{mod}\int$

$= \mathtt{use}[\![\mathtt{x\ =\ A}]\!] \cup (L_e \setminus \mathtt{mod}[\![\mathtt{x\ =\ A}]\!])$ $\wr$def. $\in\int$

$= \widehat{\mathscr{S}}^{\,\exists l}[\![\mathtt{x\ =\ A\ ;}]\!]\ L_b, L_e$ $\wr$(25), Q.E.D.$\int$

$\subseteq$ is never used in this derivation so $\widehat{\mathscr{S}}^{\,\exists l}[\![\mathtt{x\ =\ A\ ;}]\!]\ L_b, L_e\ =\ \mathscr{S}^{\exists l}[\![\mathtt{x\ =\ A\ ;}]\!]\ L_b, L_e$ is the best (most precise) abstraction for the assignment.

– For the ***iteration*** $\mathtt{S} ::= \mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b$, we apply the semi-commutation fixpoint approximation Theorem 1 of [9] to the fixpoint definition (9) of the prefix trace semantics of the iteration. For the semi-commutation where we can assume that X is an iterate of $\mathscr{F}^*[\![\mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b]\!]$ from $\varnothing$ and therefore $X \subseteq \mathscr{S}^*[\![\mathtt{S}]\!]$, we have

$\alpha^{\exists l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ (\mathscr{F}^*[\![\mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b]\!](X))\ L_b, L_e$

$= \bigcup\{\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \mid \langle\pi_0,\ \pi_1\rangle\ \in\ \mathscr{F}^*[\![\mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b]\!](X)\}$ $\wr$(15)$\int$

$= \bigcup\{\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \mid \langle\pi_0,\ \pi_1\rangle\ \in\ \{\langle\pi_1\ell',\ \ell'\rangle \mid \pi_1\ell' \in \mathbb{T}^+ \wedge \ell' = \ell\}\} \cup$ (a)
$\bigcup\{\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \mid \langle\pi_0,\ \pi_1\rangle\ \in\ \{\langle\pi_1\ell',\ \ell'\pi_2\ell' \xrightarrow{\neg(\mathtt{B})} \mathsf{aft}[\![\mathtt{S}]\!]\rangle \mid \langle\pi_1\ell',$
$\ell'\pi_2\ell'\rangle \in X \wedge \mathscr{B}[\![\mathtt{B}]\!]\varrho(\pi_1\ell'\pi_2\ell') = \mathsf{ff} \wedge \ell' = \ell\}\} \cup$ (b)
$\bigcup\{\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \mid \langle\pi_0,\ \pi_1\rangle\ \in\ \{\langle\pi_1\ell',\ \ell'\pi_2\ell' \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_b]\!] \smallfrown \pi_3\rangle \mid \langle\pi_1\ell',$
$\ell'\pi_2\ell'\rangle \in X \wedge \mathscr{B}[\![\mathtt{B}]\!]\varrho(\pi_1\ell'\pi_2\ell') = \mathsf{tt} \wedge \langle\pi_1\ell'\pi_2\ell' \xrightarrow{\mathtt{B}} \mathsf{at}[\![\mathtt{S}_b]\!],\ \pi_3\rangle \in \mathscr{S}^*[\![\mathtt{S}_b]\!] \wedge \ell' = \ell\}\}$ (c)
$\wr$(9)$\int$

We go on by cases.

- For the case (a), we have

  $\bigcup\{\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \mid \langle\pi_0,\ \pi_1\rangle\ \in\ \{\langle\pi_1\ell',\ \ell'\rangle \mid \pi_1\ell' \in \mathbb{T}^+ \wedge \ell' = \ell\}\}$
  $\wr$(a)$\int$

  $= \bigcup\{\alpha^{l}_{\mathtt{use,mod}}[\![\mathtt{S}]\!]\ L_b, L_e\ \langle\pi_1\ell,\ \ell\rangle \mid \pi_1\ell \in \mathbb{T}^+\}$ $\wr$where $\ell = \mathsf{at}[\![\mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b]\!]\int$

  $= \{\mathtt{x} \in \mathbb{V} \mid (\ell = \mathsf{aft}[\![\mathtt{S}]\!] \wedge \mathtt{x} \in L_e) \vee (\mathsf{esc}[\![\mathtt{S}]\!] \wedge \ell = \mathsf{brk\text{-}to}[\![\mathtt{S}]\!] \wedge \mathtt{x} \in L_b)\}$ $\wr$(14.a)$\int$

  $= \varnothing$ $\wr\ell = \mathsf{at}[\![\mathtt{S}]\!] \neq \mathsf{aft}[\![\mathtt{S}]\!]$ and $\ell = \mathsf{at}[\![\mathtt{S}]\!] \neq \mathsf{brk\text{-}to}[\![\mathtt{S}]\!]$ for iteration$\int$

- For the case (b) where $X \subseteq \mathcal{S}^*[\![\mathsf{S}]\!]$ is a subset of the iterates, we have

$$\bigcup\{\alpha^{l}_{\mathsf{use},\mathsf{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \mid \langle\pi_0,\ \pi_1\rangle \in \{\langle\pi_1\ell',\ \ell'\pi_2\ell' \xrightarrow{\neg(\mathsf{B})} \mathsf{aft}[\![\mathsf{S}]\!]\rangle \mid \langle\pi_1\ell',\ \ell'\pi_2\ell'\rangle \in X \wedge \mathcal{B}[\![\mathsf{B}]\!]\varrho(\pi_1\ell'\pi_2\ell') = \mathsf{ff} \wedge \ell' = \ell\}\}$$

$$= \bigcup\{\alpha^{l}_{\mathsf{use},\mathsf{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_1\ell,\ \ell\pi_2\ell \xrightarrow{\neg(\mathsf{B})} \mathsf{aft}[\![\mathsf{S}]\!]\rangle \mid \langle\pi_1\ell,\ \ell\pi_2\ell\rangle \in X \wedge \mathcal{B}[\![\mathsf{B}]\!]\varrho(\pi_1\ell\pi_2\ell) = \mathsf{ff}\}$$

$\wr$def. $\in$ and $\ell' = \ell = \mathsf{at}[\![\mathsf{S}]\!]\wr$

$$= \bigcup\{\{\mathsf{x} \in \mathbb{V} \mid \exists i \in [1, n{-}1]\ .\ \forall j \in [1, i{-}1]\ .\ \mathsf{x} \notin \mathbb{mod}[\![a_j]\!] \wedge \mathsf{x} \in \mathbb{use}[\![a_i]\!]\} \cup L_e \cup \mid \langle\pi_1\ell,\ \ell\pi_2\ell\rangle \in X \wedge \mathcal{B}[\![\mathsf{B}]\!]\varrho(\pi_1\ell\pi_2\ell) = \mathsf{ff}\}$$

$\wr$by Lem. 1 where $\ell\pi_2\ell \xrightarrow{\neg(\mathsf{B})} \mathsf{aft}[\![\mathsf{S}]\!] = \ell_1 \xrightarrow{a_1} \ell_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-2}} \ell_{n-1} = \ell \xrightarrow{a_{n-1} = \neg(\mathsf{B})} \ell_n$ where $\ell = \ell_1$ and $\ell_n = \mathsf{aft}[\![\mathsf{S}]\!]$, $\langle\pi_1\ell,\ \ell\pi_2\ell\rangle \in X \subseteq \mathcal{S}^*[\![\mathsf{S}]\!]$ so $\langle\pi_1\ell,\ \ell\pi_2\ell \xrightarrow{\neg(\mathsf{B})} \mathsf{aft}[\![\mathsf{S}]\!]\rangle \in \mathcal{S}^*[\![\mathsf{S}]\!]$, and $\mathsf{esc}[\![\mathsf{S}]\!] = \mathsf{ff}\wr$

$$= \bigcup\{\{\mathsf{x} \in \mathbb{V} \mid \exists i \in [1, n-2]\ .\ \forall j \in [1, i-1]\ .\ \mathsf{x} \notin \mathbb{mod}[\![a_j]\!] \wedge \mathsf{x} \in \mathbb{use}[\![a_i]\!]\} \cup \{\mathsf{x} \in \mathbb{V} \mid \forall j \in [1, n{-}2]\ .\ \mathsf{x} \notin \mathbb{mod}[\![a_j]\!] \wedge \mathsf{x} \in \mathbb{use}[\![\mathsf{B}]\!]\} \cup L_e \mid \langle\pi_1\ell, \ell\pi_2\ell\rangle \in X \wedge \mathcal{B}[\![\mathsf{B}]\!]\varrho(\pi_1\ell\pi_2\ell) = \mathsf{ff}\}$$

$\wr[1, n-1] = [1, n-2] \cup \{n-1\}$, $a_{n-1} = \neg(\mathsf{B})$, and $\mathbb{use}[\![\neg(\mathsf{B})]\!] = \mathbb{use}[\![\mathsf{B}]\!]\wr$

$$\subseteq \bigcup\{\{\mathsf{x} \in \mathbb{V} \mid \exists i \in [1, n-2]\ .\ \forall j \in [1, i-1]\ .\ \mathsf{x} \notin \mathbb{mod}[\![a_j]\!] \wedge \mathsf{x} \in \mathbb{use}[\![a_i]\!] \mid \langle\pi_1\ell,\ \ell\pi_2\ell\rangle \in X\}\} \cup \mathbb{use}[\![\mathsf{B}]\!] \cup L_e$$

$\wr$ignoring the check $\forall j \in [1, n-2]\ .\ \mathsf{x} \notin \mathbb{mod}[\![a_j]\!]$ that x has not been modified before its use in $\neg(\mathsf{B})$, that the test B is false, and $\ell\pi_2\ell \triangleq \ell_1 \xrightarrow{a_1} \ell_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-2}} \ell_{n-1}$ with $\ell = \ell_1$ and $\ell_{n-1} = \ell\wr$

$$\subseteq \bigcup\{\alpha^{l}_{\mathsf{use},\mathsf{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \mid \langle\pi_0,\ \pi_1\rangle \in X\} \cup \mathbb{use}[\![\mathsf{B}]\!] \cup L_e \qquad \wr\text{Lem. } 1\wr$$

$$= \alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}[\![\mathsf{S}]\!]\ (X)\ L_b, L_e \cup \mathbb{use}[\![\mathsf{B}]\!] \cup L_e \qquad \wr(15)\wr$$

- For the case (c) where $X \subseteq \mathcal{S}^*[\![\mathsf{S}]\!]$ is a subset of the iterates, we have

$$\bigcup\{\alpha^{l}_{\mathsf{use},\mathsf{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_0,\ \pi_1\rangle \mid \langle\pi_0,\ \pi_1\rangle \in \{\langle\pi_1\ell',\ \ell'\pi_2\ell' \xrightarrow{\mathsf{B}} \mathsf{at}[\![\mathsf{S}_b]\!] \frown \pi_3\rangle \mid \langle\pi_1\ell',\ \ell'\pi_2\ell'\rangle \in X \wedge \mathcal{B}[\![\mathsf{B}]\!]\varrho(\pi_1\ell'\pi_2\ell') = \mathsf{tt} \wedge \langle\pi_1\ell'\pi_2\ell' \xrightarrow{\mathsf{B}} \mathsf{at}[\![\mathsf{S}_b]\!],\ \pi_3\rangle \in \mathcal{S}^*[\![\mathsf{S}_b]\!] \wedge \ell' = \ell\}\}$$

$$= \bigcup\{\alpha^{l}_{\mathsf{use},\mathsf{mod}}[\![\mathsf{S}]\!]\ L_b, L_e\ \langle\pi_1\ell,\ \ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}[\![\mathsf{S}_b]\!] \frown \pi_3\rangle \mid \langle\pi_1\ell,\ \ell\pi_2\ell\rangle \in X \wedge \mathcal{B}[\![\mathsf{B}]\!]\varrho(\pi_1\ell\pi_2\ell) = \mathsf{tt} \wedge \langle\pi_3,\ \pi_1\ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}[\![\mathsf{S}_b]\!]\rangle \in \mathcal{S}^*[\![\mathsf{S}_b]\!]\}$$

$\wr$def. $\in$ and $\ell' = \ell = \mathsf{at}[\![\mathsf{S}]\!]\wr$

$$= \bigcup\{\{\mathsf{x} \in \mathbb{V} \mid \exists i \in [1, n-1]\ .\ \forall j \in [1, i-1]\ .\ \mathsf{x} \notin \mathbb{mod}[\![a_j]\!] \wedge \mathsf{x} \in \mathbb{use}[\![a_i]\!]\} \cup (\!|\ \ell_n = \mathsf{aft}[\![\mathsf{S}]\!]\ ?\ L_e\ \text{:}\ \varnothing\ |\!) \cup (\!|\ \mathsf{esc}[\![\mathsf{S}]\!] \wedge \ell_n = \mathsf{brk\text{-}to}[\![\mathsf{S}]\!]\ ?\ L_b\ \text{:}\ \varnothing\ |\!) \mid \langle\pi_1\ell,\ \ell\pi_2\ell\rangle \in X \wedge \mathcal{B}[\![\mathsf{B}]\!]\varrho(\pi_1\ell\pi_2\ell) = \mathsf{tt} \wedge \langle\pi_3,\ \pi_1\ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}[\![\mathsf{S}_b]\!]\rangle \in \mathcal{S}^*[\![\mathsf{S}_b]\!] \wedge \ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}[\![\mathsf{S}_b]\!] \frown \pi_3 = \ell_1 \xrightarrow{a_1} \ell_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} \ell_n\}$$

$\wr$by Lem. 1 $\wr$

$= \bigcup\{\{\mathsf{x} \in \mathbb{V} \mid \exists i \in [1, n-1] \,.\, \forall j \in [1, i-1] \,.\, \mathsf{x} \notin \mathbb{mod}\llbracket a_j \rrbracket \wedge \mathsf{x} \in \mathbb{use}\llbracket a_i \rrbracket\} \mid \langle \pi_1\ell, \ell\pi_2\ell\rangle \in X \wedge \mathcal{B}\llbracket \mathsf{B} \rrbracket \varrho(\pi_1\ell\pi_2\ell) = \mathtt{tt} \wedge \langle \pi_3,\ \pi_1\ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}\llbracket \mathsf{S}_b \rrbracket\rangle \in \boldsymbol{\mathcal{S}}^*\llbracket \mathsf{S}_b \rrbracket \wedge \ell\pi_2\ell = \ell_1 \xrightarrow{a_1} \ell_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{m-1}} \ell_m \wedge \ell \xrightarrow{\mathsf{B}} \mathsf{at}\llbracket \mathsf{S}_b \rrbracket = \ell_m \xrightarrow{a_m = \mathsf{B}} \ell_{m+1} \wedge \pi_3 = \ell_{m+1} \xrightarrow{a_{m+1}} \ldots \xrightarrow{a_{n-1}} \ell_n\}$

(by decomposing the trace according to its pattern, $\langle \pi_3,\ \pi_1\ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}\llbracket \mathsf{S}_b \rrbracket\rangle \in \boldsymbol{\mathcal{S}}^*\llbracket \mathsf{S}_b \rrbracket$ so $\ell_n \neq \mathsf{aft}\llbracket \mathsf{S} \rrbracket$, and $\mathsf{esc}\llbracket \mathsf{S} \rrbracket = \mathtt{ff}$)

$= \bigcup\{\{\mathsf{x} \in \mathbb{V} \mid \exists i \in [1, m-1] \,.\, \forall j \in [1, i-1] \,.\, \mathsf{x} \notin \mathbb{mod}\llbracket a_j \rrbracket \wedge \mathsf{x} \in \mathbb{use}\llbracket a_i \rrbracket\} \cup \{\mathsf{x} \in \mathbb{V} \mid \forall j \in [1, m-1] \,.\, \mathsf{x} \notin \mathbb{mod}\llbracket a_j \rrbracket \wedge \mathsf{x} \in \mathbb{use}\llbracket a_m \rrbracket\} \cup \{\mathsf{x} \in \mathbb{V} \mid \exists i \in [m+1, n-1] \,.\, \forall j \in [1, i-1] \,.\, \mathsf{x} \notin \mathbb{mod}\llbracket a_j \rrbracket \wedge \mathsf{x} \in \mathbb{use}\llbracket a_i \rrbracket\} \mid \langle \pi_1\ell, \ell\pi_2\ell\rangle \in X \wedge \mathcal{B}\llbracket \mathsf{B} \rrbracket \varrho(\pi_1\ell\pi_2\ell) = \mathtt{tt} \wedge \langle \pi_3,\ \pi_1\ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}\llbracket \mathsf{S}_b \rrbracket\rangle \in \boldsymbol{\mathcal{S}}^*\llbracket \mathsf{S}_b \rrbracket \wedge \ell\pi_2\ell = \ell_1 \xrightarrow{a_1} \ell_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{m-1}} \ell_m \wedge \ell \xrightarrow{\mathsf{B}} \mathsf{at}\llbracket \mathsf{S}_b \rrbracket = \ell_m \xrightarrow{a_m = \mathsf{B}} \ell_{m+1} \wedge \pi_3 = \ell_{m+1} \xrightarrow{a_{m+1}} \ldots \xrightarrow{a_{n-1}} \ell_n\}$

(by decomposing $[1, n-1] = [1, m-1] \cup \{m\} \cup [m+1, n-1]$)

$\subseteq \bigcup\{\{\mathsf{x} \in \mathbb{V} \mid \exists i \in [1, m-1] \,.\, \forall j \in [1, i-1] \,.\, \mathsf{x} \notin \mathbb{mod}\llbracket a_j \rrbracket \wedge \mathsf{x} \in \mathbb{use}\llbracket a_i \rrbracket\} \mid \langle \pi_1\ell, \ell\pi_2\ell\rangle \in X \wedge \ell\pi_2\ell = \ell_1 \xrightarrow{a_1} \ell_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{m-1}} \ell_m\} \cup \mathbb{use}\llbracket \mathsf{B} \rrbracket \cup \bigcup\{\{\mathsf{x} \in \mathbb{V} \mid \exists i \in [m+1, n-1] \,.\, \forall j \in [1, i-1] \,.\, \mathsf{x} \notin \mathbb{mod}\llbracket a_j \rrbracket \wedge \mathsf{x} \in \mathbb{use}\llbracket a_i \rrbracket\} \mid \langle \pi_3,\ \pi_1\ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}\llbracket \mathsf{S}_b \rrbracket\rangle \in \boldsymbol{\mathcal{S}}^*\llbracket \mathsf{S}_b \rrbracket \wedge \pi_3 = \ell_{m+1} \xrightarrow{a_{m+1}} \ldots \xrightarrow{a_{n-1}} \ell_n\}$

(def. $\cup$, ignoring the check $\forall j \in [1, m-1] \,.\, \mathsf{x} \notin \mathbb{mod}\llbracket a_j \rrbracket$ that x has not been modified before its use in $a_m = \mathsf{B}$, ignoring the value of $\mathcal{B}\llbracket \mathsf{B} \rrbracket \varrho(\pi_1\ell\pi_2\ell) = \mathtt{tt}$)

$\subseteq \bigcup\{\alpha^{l}_{\mathsf{use,mod}}\llbracket \mathsf{S} \rrbracket\ L_b, L_e\ \langle \pi_1\ell,\ \ell\pi_2\ell\rangle \mid \langle \pi_1\ell,\ \ell\pi_2\ell\rangle \in X\} \cup \mathbb{use}\llbracket \mathsf{B} \rrbracket \cup \bigcup\{\{\mathsf{x} \in \mathbb{V} \mid \exists i \in [m+1, n-1] \,.\, \forall j \in [1, i-1] \,.\, \mathsf{x} \notin \mathbb{mod}\llbracket a_j \rrbracket \wedge \mathsf{x} \in \mathbb{use}\llbracket a_i \rrbracket\} \cup \llparenthesis\, \ell_n = \mathsf{aft}\llbracket \mathsf{S}_b \rrbracket \mathbin{?} L_e \mathbin{⁏} \varnothing \,\rrparenthesis \cup \llparenthesis\, \mathsf{esc}\llbracket \mathsf{S}_b \rrbracket \wedge \ell_n = \mathsf{brk\text{-}to}\llbracket \mathsf{S}_b \rrbracket \mathbin{?} L_b \mathbin{⁏} \varnothing \,\rrparenthesis \mid \langle \pi_3,\ \pi_1\ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}\llbracket \mathsf{S}_b \rrbracket\rangle \in \boldsymbol{\mathcal{S}}^*\llbracket \mathsf{S}_b \rrbracket \wedge \pi_3 = \ell_{m+1} \xrightarrow{a_{m+1}} \ldots \xrightarrow{a_{n-1}} \ell_n\}$

(by Lem. 1 for the first term since $\mathsf{aft}\llbracket \mathsf{S} \rrbracket \neq \ell$ and $\mathsf{brk\text{-}to}\llbracket \mathsf{S} \rrbracket \neq \ell$ and over-approximating the third term)

$\subseteq \bigcup\{\alpha^{l}_{\mathsf{use,mod}}\llbracket \mathsf{S} \rrbracket\ L_b, L_e\ \langle \pi_1\ell,\ \ell\pi_2\ell\rangle \mid \langle \pi_1\ell,\ \ell\pi_2\ell\rangle \in X\} \cup \mathbb{use}\llbracket \mathsf{B} \rrbracket \cup \bigcup\{\alpha^{l}_{\mathsf{use,mod}}\llbracket \mathsf{S}_b \rrbracket\ L_b, L_e\ \langle \pi_1\ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}\llbracket \mathsf{S}_b \rrbracket,\ \pi_3\rangle \mid \langle \pi_1\ell\pi_2\ell \xrightarrow{\mathsf{B}} \mathsf{at}\llbracket \mathsf{S}_b \rrbracket,\ \pi_3\rangle \in \boldsymbol{\mathcal{S}}^*\llbracket \mathsf{S}_b \rrbracket\}$ (by Lem. 1)

$\subseteq \bigcup\{\alpha^{l}_{\mathsf{use,mod}}\llbracket \mathsf{S} \rrbracket\ L_b, L_e \langle \pi_0, \pi_1\rangle \mid \langle \pi_0, \pi_1\rangle \in X\} \cup \mathbb{use}\llbracket \mathsf{B} \rrbracket \cup \bigcup\{\alpha^{l}_{\mathsf{use,mod}}\llbracket \mathsf{S}_b \rrbracket\ L_b, L_e \langle \pi_0, \pi_1\rangle \mid \langle \pi_0,\ \pi_1\rangle \in \boldsymbol{\mathcal{S}}^*\llbracket \mathsf{S}_b \rrbracket\}$ (over-approximating the semantics X and $\boldsymbol{\mathcal{S}}^*\llbracket \mathsf{S}_b \rrbracket$)

$\subseteq (\alpha^{\exists l}_{\mathsf{use,mod}}\llbracket \mathsf{S} \rrbracket\ (X)\ L_b, L_e) \cup \mathbb{use}\llbracket \mathsf{B} \rrbracket \cup (\alpha^{\exists l}_{\mathsf{use,mod}} \mathsf{S}_b]\ (\boldsymbol{\mathcal{S}}^*\llbracket \mathsf{S}_b \rrbracket)\ L_b, L_e)$ ((15))

$\subseteq (\alpha^{\exists l}_{\mathsf{use,mod}}\llbracket \mathsf{S} \rrbracket\ (X)\ L_b, L_e) \cup \mathbb{use}\llbracket \mathsf{B} \rrbracket \cup \widehat{\boldsymbol{\mathcal{S}}}^{\exists l}\llbracket \mathsf{S}_b \rrbracket\ L_b, L_e$

≀structural induction hypothesis of Th. 2≀

- Gathering the three cases *(a)*, *(b)*, and *(c)*, we have proved the semi-commutation condition

  $\alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}[\![\mathtt{S}]\!]\ (\mathcal{F}^*[\![\mathtt{while}\ \ell\ \mathtt{(B)}\ \mathtt{S}_b]\!](X))\ L_b, L_e \subseteq$
  $L_e \cup (\alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}[\![\mathtt{S}]\!]\ (X)\ L_b, L_e \cup \mathbb{use}[\![\mathtt{B}]\!] \cup L_e) \cup (\alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}[\![\mathtt{S}]\!]\ (X)\ L_b, L_e) \cup \mathbb{use}[\![\mathtt{B}]\!] \cup$
  $\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}_b]\!]\ L_b, L_e$

  So we define

  $$\mathcal{B}^{\exists l}[\![\mathtt{while}\ \mathtt{(B)}\ \mathtt{S}_b]\!]\ L_b, L_e\ X \triangleq L_e \cup X \cup \mathbb{use}[\![\mathtt{B}]\!] \cup \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}_b]\!]\ L_b, L_e$$

  to get $\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{while}\ \mathtt{(B)}\ \mathtt{S}_b]\!]\ L_b, L_e \triangleq \mathsf{lfp}^{\subseteq}\ \mathcal{B}^{\exists l}[\![\mathtt{while}\ \mathtt{(B)}\ \mathtt{S}_b]\!]\ L_b, L_e$. The iterates are
  - $X^0 = \varnothing$
  - $X^1 = \mathcal{B}^{\exists l}[\![\mathtt{while}\ \mathtt{(B)}\ \mathtt{S}_b]\!]\ L_b, L_e\ X^0 = L_e \cup \mathbb{use}[\![\mathtt{B}]\!] \cup \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}_b]\!]\ L_b, L_e$
  - $X^2 = \mathcal{B}^{\exists l}[\![\mathtt{while}\ \mathtt{(B)}\ \mathtt{S}_b]\!]\ L_b, L_e\ X^2 = L_e \cup \mathbb{use}[\![\mathtt{B}]\!] \cup \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}_b]\!]\ L_b, L_e = X^1$

  Therefore the least fixpoint is the constant

  $$\widehat{\mathcal{S}}^{\exists l}[\![\mathtt{while}\ \mathtt{(B)}\ \mathtt{S}_b]\!]\ L_b, L_e = L_e \cup \mathbb{use}[\![\mathtt{B}]\!] \cup \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}_b]\!]\ L_b, L_e$$

  as stated in (25), Q.E.D. □

We conclude that algorithm (25) is sound with respect to the revised syntactic/semantic definition $\mathcal{S}^{\exists l}[\![\mathtt{S}]\!]$ of liveness in (24).

Theorem 3 $\mathcal{S}^{\exists l}[\![\mathtt{S}]\!] = \alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}\ (\mathcal{S}^{+\infty}[\![\mathtt{S}]\!]) \dot{\subseteq} \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}]\!]$.

Proof (of Th. 3)

$$\begin{aligned}
& \alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}\ (\mathcal{S}^{+\infty}[\![\mathtt{S}]\!]) \\
= {} & \alpha^{\exists l}_{\mathsf{use},\mathsf{mod}}\ (\mathcal{S}^{*}[\![\mathtt{S}]\!]) && \text{≀Lem. 2≀} \\
\dot{\subseteq} {} & \widehat{\mathcal{S}}^{\exists l}[\![\mathtt{S}]\!] && \text{≀Th. 2 and Th. 1≀}
\end{aligned}$$
□

5 Calculational design of the syntactic structural deadness static analysis

By complement duality we obtain the syntactic definite deadness analysis which is the information actually needed in compilers.

Structural syntactic definite deadness analysis

$$\widehat{\mathcal{S}}^{\forall d}[\![\texttt{Sl}\ \ell]\!]\ D_e = \widehat{\mathcal{S}}^{\forall d}[\![\texttt{Sl}\ \ell]\!]\ \mathbb{V}, D_e \tag{26}$$
$$\widehat{\mathcal{S}}^{\forall d}[\![\texttt{x = A ;}]\!]\ D_b, D_e = \neg\,\mathbb{use}[\![\texttt{x = A}]\!] \cap (D_e \cup \mathbb{mod}[\![\texttt{x = A}]\!])$$
$$\widehat{\mathcal{S}}^{\forall d}[\![\texttt{;}]\!]\ D_b, D_e = D_e$$
$$\widehat{\mathcal{S}}^{\forall d}[\![\texttt{Sl}'\ \texttt{S}]\!]\ D_b, D_e = \widehat{\mathcal{S}}^{\forall d}[\![\texttt{Sl}']\!]\ D_b, (\widehat{\mathcal{S}}^{\forall d}[\![\texttt{S}]\!]\ D_b, D_e)$$
$$\widehat{\mathcal{S}}^{\forall d}[\![\ \epsilon\ ]\!]\ D_b, D_e = D_e$$
$$\widehat{\mathcal{S}}^{\forall d}[\![\texttt{if (B) S}_t]\!]\ D_b, D_e = \neg\,\mathbb{use}[\![\texttt{B}]\!] \cap D_e \cap \widehat{\mathcal{S}}^{\forall d}[\![\texttt{S}_t]\!]\ D_b, D_e$$
$$\widehat{\mathcal{S}}^{\forall d}[\![\texttt{if (B) S}_t\ \texttt{else S}_f]\!]\ D_b, D_e = \neg\,\mathbb{use}[\![\texttt{B}]\!] \cap \widehat{\mathcal{S}}^{\forall d}[\![\texttt{S}_t]\!]\ D_b, D_e \cap \widehat{\mathcal{S}}^{\forall d}[\![\texttt{S}_f]\!]\ D_b, D_e$$
$$\widehat{\mathcal{S}}^{\forall d}[\![\texttt{while (B) S}_b]\!]\ D_b, D_e = \neg\,\mathbb{use}[\![\texttt{B}]\!] \cap D_e \cap \widehat{\mathcal{S}}^{\forall d}[\![\texttt{S}_b]\!]\ D_b, D_e$$
$$\widehat{\mathcal{S}}^{\forall d}[\![\texttt{break ;}]\!]\ D_b, D_e = D_b$$
$$\widehat{\mathcal{S}}^{\forall d}[\![\texttt{\{ Sl \}}]\!]\ D_b, D_e = \widehat{\mathcal{S}}^{\forall d}[\![\texttt{Sl}]\!]\ D_b, D_e \qquad \square$$

Theorem 4 (Structural syntactic definite deadness analysis) *For all program components* `S`*, define* $\mathcal{S}^{\forall d}[\![\texttt{S}]\!]\ D_b, D_e \triangleq \neg\mathcal{S}^{\exists l}[\![\texttt{S}]\!]\ \neg D_b, \neg D_e$*.* $\mathcal{S}^{\forall d}$ *is equivalently defined by* $\widehat{\mathcal{S}}^{\forall d}$ *in (26).*

Proof of Th. 4. The proof is by structural induction and essentially consists in applying De Morgan laws for complement. For example,

$$\begin{aligned}
&\mathcal{S}^{\forall d}[\![\texttt{if (B) S}_t]\!]\ D_b, D_e\\
=\ &\neg\mathcal{S}^{\exists l}[\![\texttt{if (B) S}_t]\!]\ \neg D_b, \neg D_e && \text{(definition of } \mathcal{S}^{\forall d}[\![\texttt{S}]\!] \text{ as dual of } \mathcal{S}^{\exists l}[\![\texttt{S}]\!]\text{)}\\
=\ &\neg(\mathbb{use}[\![\texttt{B}]\!] \cup \neg D_e \cup \mathcal{S}^{\exists l}[\![\texttt{S}_t]\!]\ \neg D_b, \neg D_e) && \text{((25))}\\
=\ &\neg\,\mathbb{use}[\![\texttt{B}]\!] \cap \neg\neg D_e \cap \neg\mathcal{S}^{\exists l}[\![\texttt{S}_t]\!]\ \neg D_b, \neg D_e) && \text{(De Morgan laws)}\\
=\ &\neg\,\mathbb{use}[\![\texttt{B}]\!] \cap D_e \cap \mathcal{S}^{\forall d}[\![\texttt{S}_t]\!]\ D_b, D_e && \text{(structural induction hypothesis)}
\end{aligned}$$

All other cases are similar. □

6 Is liveness analysis correctly used for code optimization?

6.1 Liveness specification

We have considered three possible specifications of liveness. A purely semantic one $\mathcal{S}^{\exists l}$ in (20) with respect to which the liveness analysis algorithm (25) is unsound and a syntactic one $\mathcal{S}^{\exists\!\!\exists l}$ in (22) as well as a revised syntactic/semantic liveness specification $\mathcal{S}^{\exists l}$ in (24) for which, by Th. 1 and 2, the liveness analysis algorithm (25) is sound. The problem is that, as shown in Section **3.4**, the syntactic specification of liveness $\mathcal{S}^{\exists\!\!\exists l}$ in (22) is unsound with respect to the purely semantic specification $\mathcal{S}^{\exists l}$ in (20). This is problematic since applications of the liveness analysis algorithm (25) are not designed with respect to what the algorithm does, but with respect to the specification of what it is supposed to do. Therefore, a potential problem is in the use of the liveness analysis algorithm (25) with a semantic definition $\mathcal{S}^{\exists l}$ in (20) of soundness for which it is incorrect.

6.2 What could go wrong when optimizing programs?

Consider a compiler that successively performs

1. a (syntactic) liveness analysis $\mathcal{S}^{\exists\mathbb{l}}$;
2. next, a code optimization by removal
   (a) of assignments to variables that are dead after this assignment,
   (b) of assignments to variables that do not change the value of this variable (using Kildall's constancy analysis [22] or a more precise symbolic constancy analysis [18,31]);
3. next, a register allocation such that
   (a) simultaneously live variables are stored in different registers,
   (b) when no register is left and one is needed, one of those containing the value of a dead variable is preferred (to avoid saving the value of the variable to its memory location as would be needed for live variables).

For the following program (where all variables are dead on exit)

		semantically		syntactically	
		live	dead	live	dead
`x=0; scanf(y);`					
`if (x==0){`					
ℓ_1 `...` x and y neither used nor modified `...`	ℓ_1	{x}	{y}	{y}	{x}
ℓ_2 `x = y - y; }`	ℓ_2	{x}	{y}	{y}	{x}
`else {`					
`x=42;`					
`}`					
ℓ_3 `print(x);`	ℓ_3	{x}	{y}	{x}	{y}

x is semantically live at ℓ_1, ℓ_2, and ℓ_3 since it is never modified (in particular not modified at ℓ_2) before being used at ℓ_3. However it is syntactically dead at ℓ_1 and ℓ_2 since it is not used before being assigned at ℓ_2. Code elimination (2b) will suppress the assignment at ℓ_2 since the value of x is unchanged. Assume x is in a register at ℓ_1 and a fresh register is needed but none is left available. By (3b) the register containing x may be selected since its value need not be saved to memory because x is syntactically dead at ℓ_1. Then the value of x is lost at ℓ_3, a compilation bug. The problem is the notion of modification assimilated to an assignment in (21) and syntactic liveness $\mathcal{S}^{\exists\mathbb{l}}$ in (22) when this assignment is redundant and may be eliminated from the object program.

This error does not occur with semantic liveness $\mathcal{S}^{\exists l}$ in (20) which declares x live at ℓ_1 so the register containing its value will be saved to memory (and reloaded at ℓ_3).

6.3 Why does it not go wrong?

One solution is to prevent program transformations (such as (2b) and (3b) above) that do not preserve the soundness of the semantic liveness $\boldsymbol{\mathcal{S}}^{\exists\mathsf{l}}$ in (20). Since (2b) does not depend on the liveness analysis, it can be moved before. Another solution is to redo the liveness analysis after any program transformation that does not preserve the information. A better solution is adopted in CompCert [23]: the liveness analysis and code elimination are performed simultaneously and the liveness analysis is designed to be valid *after* code elimination. The soundness of the liveness analysis is stated and proved as "after code elimination, the program execution does not depend on the values of the variables declared dead by the analysis". More generally, a program transformation based on a sound static program analysis must be formally proved to be correct. This can be done in the framework of abstract interpretation [14].

7 Conclusion

We have shown that Gary Kildall approach to data flow analysis by abstraction over a path and merge over all paths [22] as well as Bernhard Steffen's approach "Data Flow Analysis is Model Checking" [28,29] (requiring finite abstract domains) formalized by David Schmidt as "Data Flow Analysis is Model Checking of Abstract Interpretations" [25], (including its recent reformulation [5]), hide subtleties in the definition of soundness, which may lead to incorrect semantics-based compiler optimizations.

Moreover, the use of transition systems in model checking forgets about the program structure and so cannot be used directly to formally derive structural elimination algorithms which may be more efficient than fixpoint algorithms. Of course elimination would not be necessarily feasible in presence of arbitrary branching in or out of loops. But nevertheless, by the chaotic iteration theorem [7], the result remains valid for all loops with forward branching only.

We have argued that "Data Flow Analysis is an Abstract Interpretation of a Trace Semantics", as first propounded by [12, Section 7.2.0.6.3] solves the soundness and design problems thanks to a not so natural replacement of "semantically modified" by "syntactically assigned to". Therefore liveness analysis must be performed after program assignment transformations.

Since the program cannot be modified after the classical syntactic liveness analysis since the analysis can become wrong after the transformation, an alternative, à la CompCert [23], is to use dependency: the soundness of the liveness analysis is stated and proved as "the program execution does not depend on the values of the variables declared dead by the analysis".

More generally, this is another illustration that program property specification is better performed directly on a semantics rather than, as is the case in dataflow analysis, on any of its abstractions.

Let us leave the conclusion to an anonymous reviewer. "It is an old story that the dataflow analysis framework ("syntactic" dataflow analysis in paper's characterization) is way too weak. For modern programming languages, control flow is not syntactic but a part of semantics. Dataflow analysis assumes the control flow to be available before the analysis hence a stalemate for modern languages with higher order functions, dynamic bindings, or dynamic gotos; dataflow analysis has neither a systematic guide to prove the correctness of an analysis nor systematic approach to manage the precision of the analysis. On the other hand, the semantics-based design theory (abstract interpretation) is general enough to handle any kind of source languages and powerful enough to prove the correctness and to manage its precision."

Acknowledgement. I thank Sandrine Blazy, Xavier Leroy, and Francesco Ranzato for lively discussions. I thank the reviewers for their livable comments. This work was supported in part by NSF Grant CNS-1446511. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

1. Allen, F.E.: Control flow analysis. SIGPLAN Not. **5**(7), 1–19 (1970)
2. Allen, F.E.: A basis for program optimization. In: IFIP Congress (1). pp. 385–390 (1971)
3. Allen, F.E.: Interprocedural data flow analysis. In: Rosenfeld, J.L. (ed.) Information Processing 74. pp. 398–402. North-Holland Pub. Co. (1974)
4. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Commun. ACM **19**(3), 137–147 (1976)
5. Beyer, D., Gulwani, S., Schmidt, D.A.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018)
6. Brookes, S.D.: Traces, pomsets, fairness and full abstraction for communicating processes. In: CONCUR. Lecture Notes in Computer Science, vol. 2421, pp. 466–482. Springer (2002)
7. Cousot, P.: Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Res. rep. R.R. 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France (Sep 1977), 15 p.
8. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
9. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. **277**(1-2), 47–103 (2002)
10. Cousot, P.: Abstract semantic dependency. In: SAS. Lecture Notes in Computer Science, this volume. Springer (2019)
11. Cousot, P., Cousot, R.: Constructive versions of Tarski's fixed point theorems. Pacific Journal of Mathematics **81**(1), 43–57 (1979)
12. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282. ACM Press (1979)
13. Cousot, P., Cousot, R.: Temporal abstract interpretation. In: POPL. pp. 12–25. ACM (2000)

14. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: POPL. pp. 178–190. ACM (2002)
15. Cousot, P., Cousot, R.: Bi-inductive structural semantics. Inf. Comput. **207**(2), 258–283 (2009)
16. Filé, G., Ranzato, F.: The powerset operator on abstract interpretations. Theor. Comput. Sci. **222**(1-2), 77–111 (1999)
17. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM **47**(2), 361–416 (2000)
18. Haghighat, M.R., Polychronopoulos, C.D.: Symbolic analysis for parallelizing compilers. ACM Trans. Program. Lang. Syst. **18**(4), 477–518 (1996)
19. Kennedy, K.: Node listings applied to data flow analysis. In: POPL. pp. 10–21. ACM Press (1975)
20. Kennedy, K.: A comparison of two algorithms for global data flow analysis. Int. J. of Comp. Math. **Section A, Volume 3**, 5–15 (1976)
21. Kennedy, K.: A comparison of two algorithms for global data flow analysis. SIAM J. Comput. **5**(1), 158–180 (Mar 1976)
22. Kildall, G.A.: A unified approach to global program optimization. In: POPL. pp. 194–206. ACM Press (1973)
23. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
24. Ryder, B.G., Paull, M.C.: Elimination algorithms for data flow analysis. ACM Comput. Surv. **18**(3), 277–316 (1986)
25. Schmidt, D.A.: Data flow analysis is model checking of abstract interpretations. In: POPL. pp. 38–48. ACM (1998)
26. Scholz, B., Blieberger, J.: A new elimination-based data flow analysis framework using annotated decomposition trees. In: CC. Lecture Notes in Computer Science, vol. 4420, pp. 202–217. Springer (2007)
27. Sharir, M.: Structural analysis: A new approch to flow analysis in optimizing compilers. Comput. Lang. **5**(3), 141–153 (1980)
28. Steffen, B.: Data flow analysis as model checking. In: TACS. Lecture Notes in Computer Science, vol. 526, pp. 346–365. Springer (1991)
29. Steffen, B.: Generating data flow analysis algorithms from modal specifications. Sci. Comput. Program. **21**(2), 115–139 (1993)
30. Tarski, A.: A lattice theoretical fixpoint theorem and its applications. Pacific J. of Math. **5**, 285–310 (1955)
31. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. **13**(2), 181–210 (1991)