Phd Thesis in Computer Science
Static Analysis of Embedded Multithreaded Programs

Jean-Loup Carré

Mai 2010

Jury

- Prof. HALBWACHS Nicolas (president)
- Prof. SEIDL Helmut (reviewer)
- Reviewer : Prof. BOUAJJANI Ahmed (reviewer)
- Dr. JEANNET Bertrand
- Dr. KUNCAK Viktor
- Dr. HYMANS Charles
- Pr. GOUBAULT-LARRECQ Jean (Phd Advisor)

Table of Contents

Part I

Introduction

CHAPTER 1

Introduction

1.1 Multithreading

The main feature of multithreading is to allow several threads to be executed concurrently. This enables the implementation of new features and improved speed. This is why multithreading is frequently used in practice, even in embedded software.

In sequential programs, some run-time errors may happen, e.g. array overflows (attempt to access in an array outside of its range), integer overflows (computes an integer greater that `INT_MAX`), invalid pointer dereferences, notably. These bugs can also happen in multithreaded programs. Worse than that, they are harder to detect due to possible interferences between threads.

In addition to this, multithreading comes with new kinds of bugs, e.g, data-races or deadlocks. A data-race occurs when two different threads attempt to access the same variable at the same time and at least one of these accesses is a "write". Data-race may lead to an unspecified behavior of the program, e.g., in C norm [ISO99].

A large variety of parallel execution models exists, some easier than others to analyze. The simplest kind of parallelisms has been well studied [FQ03, MPR06b, MPR06a, MPR07]: threads exist at the beginning of the execution of the program, and no new thread is even created.

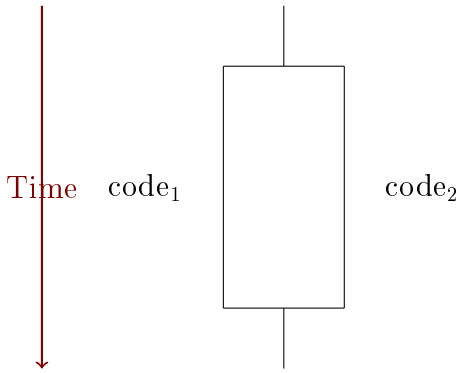


Figure 1.1: *par*{{ code$_1$ }{ code$_2$ }}

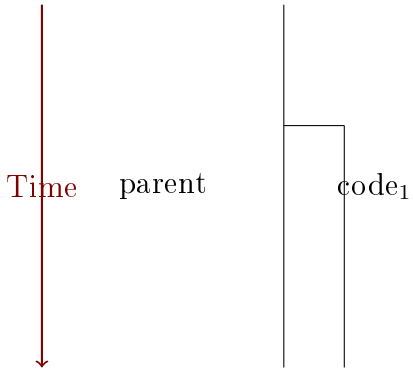


Figure 1.2: *create*{code$_1$}

A more general kind of parallelisms is thread creation using a *par* statement. The *par* statement [KSV96, RR99, RR03, SS00] executes in parallel two pieces of code: *par*(f_1, f_2) executes f_1 and f_2 in parallel and then returns. This kind of parallelisms is used in some API [Boa08]. It is illustrated in Figure 1.1 : the execution begins at the top of the figure, the main thread spawns two threads and waits until their termination. Using the *par* statement, we can encode a program where all threads are created at the beginning: e.g., a program where two threads execute in parallel f_1 and f_2 can be modeled by *par*(f_1, f_2).

The *create* statement has been less studied [LMO08, GBC+07, BMOT05], but is more used in practice. E.g., it is used in Java [GJSB05], in POSIX [IT04] and in Cilk[fCS98]. The *create* constructor spawns a new thread and immediately returns. The Figure 1.2 shows an execution of *create* statements. The *create* statement is known [LMO08] to be more complex to analyze than *par*. Furthermore, as explained by A. Bouajjani, M. Müller-Olm, and T. Touili [BMOT05], parallel calls cannot adequately model a command that spawns another thread and immediately returns.

Defining a semantics for multithreaded programs is not so easy. What is the meaning of *par*$(x = 1, x = x)$? Multithreaded programs should ideally be executed with *sequential consistency*, i.e., any run would be an interleaving of sequential runs. In the name of simplicity, a large number of analyses [FQ03, MPR06b, MPR06a, MPR07, KSV96, LMO08,

RR99, RR03, SS00] assume sequential consistency.

Nevertheless, as Lamport said [Lam79] : "*For some applications, achieving sequential consistency may not be worth the price of slowing down the processors.*" Memory models without sequential consistency, a.k.a. *weak memory models*, allow for speed increases in two ways: first, as in Lamport's quote, by lifting the constraints that multi-processors should ensure sequential consistency, and second, by allowing compilers to apply more aggressive optimizations, e.g., by reordering instructions as explicitly mandated in Java [GJSB05], and done in practice in any reasonable C compiler.

In a weak memory model, each thread has a temporary view of the memory. The shared memory and the temporary view of a thread are not necessarily consistent with each other: two threads that read the same variable simultaneously may obtain different values.

To our knowledge, in most standard thread models, e.g., Posix [IT04] or OpenMP [Boa08], the memory model is not specified accurately. In practice, their weak memory model is a combination of the processor's memory model and the need to allow for specific families of optimizations.

1.2 Program Verification

We do not recall here the well-known definition of Turing machines. Intuitively, a Turing machine is an abstract computer, which can use an arbitrary large amount of memory[1] and can run for an arbitrary long amount of time.

A set X is decidable, if there exists a Turing machine that, given an entry, answers[2] whether this entry is in X or not.

We recall the well-known Rice Theorem:

Theorem 1.1 (Rice Theorem)**.** *Given a non-constant predicate P :*

$$\{M \mid M \text{ is a Turing machine } \wedge P(L(M))\} \text{ is undecidable.}$$

where $L(M)$ is the language recognized by the Turing machine M.

The Rice theorem means that it is impossible to decide if the language recognized by a Turing machine satisfies a non-trivial predicate[3].

Most used programming languages are Turing powerful, i.e., all functions computable by a Turing machine may be written in these languages[4]. Moreover, these languages can simulate the execution of a Turing machine[5]. Therefore, most interesting properties are undecidable.

[1] No "*Out of memory*" will stop a Turing machine.

[2] In particular, this Turing machine always terminates.

[3] Obvioulsy, the problem is decidable if $P(X) = \mathit{true}$ for all X. Symmetrically, the problem is decidable if $P(X) = \mathit{false}$ for all X.

[4] Notice that, in practice, a computer have a finite memory. Therefore, it may raise an "Out of memory" when it computes some complex Turing-computable functions.

[5] "*To be able to simulate the execution of a Turing machine*" is a stronger property than "*to be Turing-powerful*", since we can define a Turing-powerful machine that is not able to simulate an arbitrary Turing

```
1  int v[3];
2
3  int f (void) {
4  int i;
5
6  ...
7  ...
8  ...
9
10 return i;
11 }
12
13 int main (void)
14 {
15   int i = f();
16   v[i]=5;
17   ...
18 }
```

Figure 1.3: Presence of an Array Overflow is Undecidable

For instance, let us consider the following problem:

ENTRY: A program P
QUESTION: Is the program P free of array overflows ?

Detecting array overflows has two main interests:

- For compilers, it allows them not to check during the execution, and therefore allows compilers to enhance execution speed significantly.
- It allows one to prove the absence of unwanted array overflows during runtime. An array overflow at run-time may corrupt silently corrupt the memory and lead to unwanted results.

An array overflow occurs when the program attempts to access an array out of its range. E.g., in Figure 1.3, if the function f returns a value that is not in $[0, 2]$, an array

machine. Let us define the n-small word machines. Such a machine is a pair (X, M) where X is a finite set of words and M a Turing machine. When we launch a n-small word machine $T = (X, M)$ on a word w, if w is length smaller or equal than n, then, T accepts w if and only if $w \in X$, else, T launches the Turing machine M on w, and recognizes w if and only if M recognizes w.

Obviously, n-small word machines are Turing powerful. Nevertheless, Rice theorem is false for n-small word machines, since given a n-small word machine T, the problem "*Does T recognize the empty word ?*" is decidable.

overflow occurs at line 16, because the array `v` only has three cells : `v[0]`, `v[1]` and `v[2]`. Attempting to access `v[42]` causes an array overflow.

Since the function f may simulate a Turing machine M on a random entry and return 0 if the word is rejected by M and 42 otherwise, therefore, deciding the absence of array overflow can be reduced to the problem of deciding if the language of a Turing machine contains an integer that is not in the array bounds. According to the Rice Theorem (Theorem 1.1), the problem to know whether the language of a Turing machine is a subset of $\{0, \ldots, n\}$ is undecidable. Hence, the problem of detecting array overflows is undecidable.

Due to Rice Theorem (Theorem 1.1), all interesting safety properties are undecidable, e.g., to detect array overflows, integer overflows, divisions by zero or data-races.

We may consider that, "*in a computer, everything is finite, therefore everything is decidable*" [Hym06]. Indeed, a computer has a finite memory, a finite hard disk, etc. Hence a computer is a finite machine. Nevertheless, a basic program, that uses 4 Mbytes ($= 8 \times 4 \times 2^{20}$ bits) of memory will lead to a large number c of configurations: $c = 2^{8 \times 4 \times 2^{20}} \simeq 10^{10^6}$, i.e, a one followed by one million zeroes.

It is physically impossible to explore the whole state space. Imagine a modern computer of $4GHz$ that computes a new state in only one clock cycle. If this computer started during the Big Bang (14 billions years ago), it would have explored only 10^{27} states. Space complexity is worse: the number of possible configurations, c, is larger than the number of atoms in the Universe ($\simeq 10^{80}$). Obviously, analyses [FQ03, MPR06b, MPR06a, MPR07] that are polynomial in time (or worse, in space) in the size of the state space will not scale up.

Hence, we need another approach. Instead of checking exactly whether there is an array overflow, we can instead design an approximation. This approximation should be computable with a low complexity.

- Under-approximations allow one to find errors, and, then, to enhance code quality.
- Over-approximations allow one to *prove* that some errors will never happen.

In an under-approximation, we have false negatives: the analysis may fail to detect an error than can happen in practice. In an over-approximation, we have false positives: the analysis may pretend that some bugs may happen although the program is correct.

In this thesis, we focus on over-approximations: our aim is to prove automatically that some embedded programs do not make errors at run-time.

CHAPTER 2

Mathematical Basis

2.1 Classical Notations

In this section, we recall classical notations. These notations are needed to understand the other chapters.

2.1.1 Logical Symbols

We use the classical notations for logical symbols:

- $\wedge$ represents conjunction "and".
- $\vee$ represents disjunction "or".
- $\Leftrightarrow$ represents equivalence.
- $\Rightarrow$ is implication. $A \Rightarrow B$ means "if A then B".

2.1.2 Sets

We assume the set theory, and recall here some classical notations that are used in this thesis:

- The empty set is written $\emptyset$.
- $\{x \mid \phi(x)\}$ represents the set of all elements x such that $\phi(x)$ holds (if such a set exists).
- The inclusion of two sets X and Y is written $X \subseteq Y$ or $Y \supseteq X$. Formally:

$$X \subseteq Y \overset{\text{def}}{\Leftrightarrow} \forall x \in X, x \in Y.$$

- The set of subsets of X is written $\mathcal{P}(X)$. Formally:

$$\mathcal{P}(X) = \{Y \mid Y \subseteq X\}.$$

- The intersection of two sets is written $\cap$. Formally:

$$X \cap Y = \{x \mid x \in X \wedge y \in Y\}.$$

- The union is written $\cup$. When the two sets are disjoint[6], we may stress this by using $\uplus$ instead of $\cup$.
- The Cartesian product between a set X and a set Y is written $X \times Y$.
- The set of functions from I to X is written X^I. To say that the function f is in X^I, we will write $f : I \to X$.

2.1.3 Functions

We use the lambda notation to define functions: $\lambda x.f(x)$ is the function that maps x to $f(x)$.

The composition of two functions f and g is written: $g \circ f$. Formally:

$$g \circ f \overset{\text{def}}{=} \lambda x.g(f(x)).$$

We can then define by induction the iteration of a function:

$$\begin{array}{ccccc} f^0 & \overset{\text{def}}{=} & id & \overset{\text{def}}{=} & \lambda x.x \\ f^n & \overset{\text{def}}{=} & f^n \circ f & \overset{\text{def}}{=} & f \circ f^n \end{array}$$

Given a partial function f, we write $Dom(f)$ the domain of f.

E.g, to define assignment, we need to modify a function on only one element. To this aim, we introduce the following notation. Given a partial function f, let $f[x_0 \mapsto v]$ be the partial function defined by

$$f(x) \overset{\text{def}}{=} \begin{cases} v & \text{if } x = x_0 \\ f(x) & \text{if } x \in Dom(f) \wedge x \neq x_0 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

[6] X and Y are disjoint if and only if $X \cap Y = \emptyset$

To handle fixpoints, we need the concept of stationary sequence. A sequence $s_1, s_2, \ldots$ is stationary if and only if there exists $N \in \mathbb{N}$ such that $\forall n \geqslant N, s_n = s_N$. This means that the sequence $s_1, s_2, \ldots$ reaches its limit after a finite number of steps.

2.2 Binary Relations

A binary relation R on a set Σ is a set of pairs of elements of Σ: $R \subseteq \Sigma \times \Sigma$.

Notations. For a binary relation R, there exists three well-known and equivalent notations:

- xRy,
- $R(x, y)$ (Predicate Notation),
- $(x, y) \in R$ (Set Notation).

Relations are, in some way, similar to functions. A relation on a set Σ can be applied to a subset of Σ:

Definition 2.1. $R\langle S\rangle = \{s' \mid \exists s \in S : (s, s') \in R\}$ be the *application* of R on S.

This definition means that a relation R on Σ induce a canonical function $f_R : \mathcal{P}(X) \to \mathcal{P}(X)$ such that:

$$f_R(S) \stackrel{\text{def}}{=} R\langle S\rangle.$$

Notice that each relation R defines a unique function on $\mathcal{P}(\Sigma)$, and, reversely, two distinct relations define to distinct functions. These functions may be composed:

Definition 2.2. Given two binary relations R and R' on a set Σ, $R; R' = \{(s, s'') \mid \exists s' \in \Sigma : (s, s') \in R \wedge (s', s'') \in R'\}$ is the *composition* of R and R'.

The composition of R and R' corresponds to the composition of their functions:

$$f_{R;R'} = f_{R'} \circ f_R.$$

As for function, a relation may be iterated, given a relation R on a set Σ we define:

$$\begin{array}{rcl} R^0 & \stackrel{\text{def}}{=} & \{(s, s) \mid s \in \Sigma\} \\ R^{k+1} & \stackrel{\text{def}}{=} & R; R^k \end{array}$$

There is a simple correspondence between function iterations and relation iterations:

$$f_{R^k} = f_R^k.$$

Now, we introduce a concept specific to relations, the reflexive-transitive closure:

Definition 2.3. Given a relation R on Σ, let $R^\star = \bigcup_{k\in\mathbb{N}} R^k$ where $R^0 = \{(s,s) \mid s \in \Sigma\}$ and $R^{k+1} = R; R^k$. $R^\star$ is called the reflexive-transitive closure of R.

On functions, the reflexive-transitive closure corresponds to:

$$f_{R^\star} = \lambda S. \bigcup_{n\in\mathbb{N}} f^n(S).$$

Now, we introduce the concept of *restriction*. There also exists a concept of restriction on functions, but we will not use it. Notice that a restriction of R has no link with any restriction of f_R.

Definition 2.4. Given a binary relation R on a set Σ and $S \subseteq \Sigma$, let $R_{|S} = \{(s,s') \in R \mid s \in S\}$ be the *restriction* of R to S.

The corresponding concept on function is:

$$f_{R_{|S}}(X) = R\langle X \cap S\rangle.$$

2.3 Ordering

In this subsection we will study binary relations that have some interesting properties, e.g., that may be used to order a set.

Definition 2.5. A binary relation R on a set Σ is a pre-ordering if and only if:

(Reflexivity) $\forall x, xRx$,

(Transitivity) $\forall x \forall y, \forall z, xRy \wedge yRz \Rightarrow xRz$.

Definition 2.6. A binary relation R on a set Σ is an ordering if and only if:

(Pre-ordering) R is a pre-ordering,

(Antisymmetry) $\forall x, \forall y, xRy \wedge yRx \Leftrightarrow x = y$.

An ordering $\leqslant$ on Σ is *total* if and only if $\forall x, y \in \Sigma, x \leqslant y \vee y \leqslant x$. A great majority of orderings used in this thesis are partial, i.e., not total.

Definition 2.7. A binary relation R on a set Σ is a strict ordering if and only if:

(Anti-Reflexivity) $\forall x, \neg(xRx)$,

(Transitivity) $\forall x \forall y, \forall z, xRy \wedge yRz \Rightarrow xRz$.

Orderings are often written $\leqslant$ and strict orderings $<$. There exists a link between orderings and strict orderings:

Claim 2.1.

- *If $\leqslant$ is an ordering, then the relation $<$ defined by $x < y \overset{def}{\Leftrightarrow} x \leqslant y \wedge x \neq y$ is a strict ordering.*
- *If $<$ is a strict ordering, then the relation $\leqslant$ defined by $x \leqslant y \overset{def}{\Leftrightarrow} x < y \vee x = y$ is an ordering.*

Definition 2.8. Given an ordering $\leqslant$, we define the reverse ordering $\geqslant$ by:

$$a \geqslant b \overset{\text{def}}{\Leftrightarrow} b \leqslant a.$$

Whenever an ordering is written $\leqslant$ we will write $\geqslant$ for the reverse ordering.

Examples. Let us give some examples of binary relations:

- The relation $\Sigma \times \Sigma$ is a preordering on Σ but is not an ordering.
- The relation "equal" (i.e, the relation $\{(x, x) \mid x \in \Sigma\}$) is an ordering on Σ.
- The reflexive-transitive closure $R^\star$ of a binary relation R is a pre-ordering.
- The inclusion $\subseteq$ on the set $\mathcal{P}(\Sigma)$ is an ordering.

An ordered set (also called poset[7]) $(\Sigma, \leqslant)$ is a pair composed of a set Σ and an ordering $\leqslant$ on Σ. The reversed ordered set of $(\Sigma, \leqslant)$ is $(\Sigma, \geqslant)$.

Definition 2.9 (Product Ordering)**.** If $\leqslant_1$ and $\leqslant_2$ are two orderings on Σ_1 and Σ_2 respectively. The *product ordering* $\leqslant_{1,2}$ on $\Sigma_1 \times \Sigma_2$ is defined by:

$$(x, y) \leqslant_{1,2} (x', y') \overset{\text{def}}{\Leftrightarrow} x \leqslant_1 x' \wedge y \leqslant_2 y'$$

Claim 2.2. *The product ordering is an ordering.*

The definition of product ordering is given only for a product of two sets. It is straightforward to generalize this definition to a product of an arbitrary number of sets. The pointwise ordering (defined below) is a product ordering on a potentially infinite product:

Definition 2.10 (Pointwise Ordering)**.** Given an ordered set $(\Sigma, \leqslant)$ and an arbitrary set Y, we define as follow the *pointwise ordering* $\leqslant_{\Sigma^X}$ on the set Σ^X:

$$f \leqslant_{\Sigma^X} g \overset{\text{def}}{\Leftrightarrow} \forall x, f(x) \leqslant g(x).$$

Claim 2.3. *The pointwise ordering is an ordering.*

Let us define two interesting properties for functions in a poset:

[7] Poset means "*partially ordered set*".

Definition 2.11. A function f on a poset $(\Sigma, \leqslant)$ is *monotone* if and only if:

$$\forall x, y \in \Sigma, x \leqslant y \Rightarrow f(x) \leqslant f(y)$$

We write $\mathbf{Mon}(X)$ the set of monotone functions from X to X.

Recall that, for a relation R, we have defined the reflexive-transitive closure $R^\star$. We define the corresponding concept, called ω-iteration, for functions:

$$f^{\uparrow\omega}(X) \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} f^n(X)$$

.

Definition 2.12. A function f on a poset $(\Sigma, \leqslant)$ is *reductive* if and only if:

$$\forall x \in \Sigma, f(x) \leqslant x$$

2.3.1 Bounds

Definition 2.13. Given a subset X of a poset $(\Sigma, \leqslant)$, a lower bound of X is an element $b \in \Sigma$ such that:

$$\forall x \in X, b \leqslant x$$

Definition 2.14. Given a subset X of a poset $(\Sigma, \leqslant)$, an upper bound of X is a lower bound of X in the reversed ordered set $(\Sigma, \geqslant)$.

Definition 2.15. The least element of a subset X of a poset $(\Sigma, \leqslant)$ is a lower bound b of X such that $b \in X$.

Symmetrically, we define a greatest element:

Definition 2.16. The greatest element of a subset X of a poset $(\Sigma, \leqslant)$ is a least element for the reverse ordered set $(\Sigma, \geqslant)$.

Definition 2.17. The greatest lower bound (glb) of a subset X of a poset $(\Sigma, \leqslant)$ is, if it exists, the one greatest element of the set $\{y \in \Sigma | \forall x \in X, x \leqslant y\}$ of the lower bounds of X

Definition 2.18. The least upper bound (lup) of a subset X of a poset $(\Sigma, \leqslant)$ is, if it exists, the greatest upper bound of X for the reverse ordering.

For convenience, in any poset $(\Sigma, \leqslant)$, we write $x_1 \sqcap x_2$ the greatest lower bound of the set $\{x_1, x_2\}$ if it exists. Furthermore, we write $\bigsqcap_{i \in I} x_i$ the greatest lower bound of the set $\{x_i \mid i \in I\}$. Symmetrically, we write $x_1 \sqcup x_2$ the least upper bound of the set $\{x_1, x_2\}$ and $\bigsqcup_{i \in I} x_i$ the least upper bound of the set $\{x_i \mid i \in I\}$.

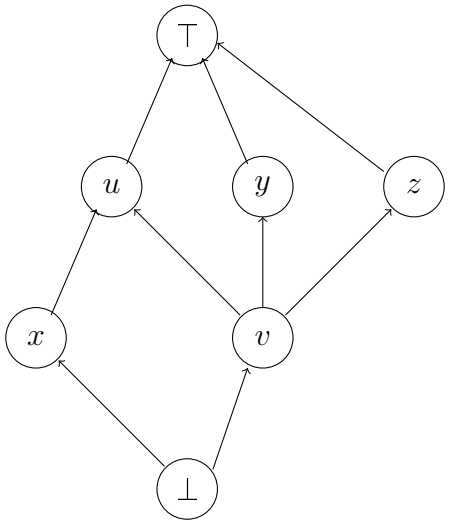


Figure 2.1: Example of Lattice

2.3.2 Lattices

2.3.2.a Definition and Examples

Definition 2.19. A *lattice* L is a poset such that for all x and y in L:

- x and y have a greatest lower bound,
- x and y have a least upper bound

Lattices arise frequently in practice. Let us give some examples:

- The most current example of lattices is the set of subsets $\mathcal{P}(X)$ of a set X for the inclusion $\subseteq$ ordering.
- The set $\mathbb{R}$ of reals is a lattice without greatest element.
- The set $\bar{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$ is a lattice.
- The set $\bar{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, +\infty\}$ is a lattice.
- The set $\mathbb{I} = \{x \in \mathbb{R} \mid 0 \leqslant 1\}$ of real numbers between 0 and 1 is a lattice.
- Figure 2.1 gives an example of lattice.
  Consider the relation $\rightarrow$:
  - $\bot \rightarrow x$,
  - $v \rightarrow y$,
  - etc.

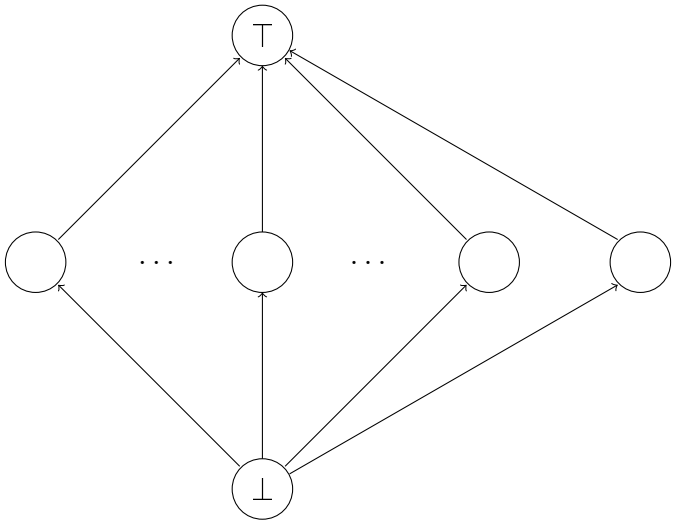


Figure 2.2: A Flat Lattice

It is straightforward to check[8] that $\leqslant$ is an ordering on the set $L = \{\bot, x, y, z, u, v, \top\}$. $(L, \leqslant)$ is a finite lattice. The lower bound of y and z is: $y \sqcap z = v$

- The lattice **Ranges** of integer ranges [CC04, CC77] is a sublattice of $\mathcal{P}(\mathbb{Z})$. It is the set of intervals of $\mathbb{Z}$ and is formally defined by:

$$\begin{aligned} \textbf{Ranges} &\stackrel{\text{def}}{=} \{X \subseteq \mathbb{Z} \mid \forall a, b \in X, \forall x \in \mathbb{Z}, a \leqslant x \leqslant b \Rightarrow x \in X\} \\ &= \{X \mid \exists a, b \in \bar{\mathbb{Z}} : X = \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}\}. \end{aligned}$$

We write $[a, b]$ the interval of integers[9] between a and b; formally: $[a, b] \stackrel{\text{def}}{=} \{x \in \mathbb{Z} \mid a \leqslant x \leqslant b\}$. Hence, we have a simpler definition of **Ranges**:

$$\textbf{Ranges} \stackrel{\text{def}}{=} \{[a, b] \mid a, b \in \bar{\mathbb{Z}}\}.$$

- Another classical lattice is the *flat lattice* on X. This lattice is the set $\Sigma = X \uplus \{\bot\} \uplus \{\top\}$ ordered by the ordering R defined by:

$$R = (\{\bot\} \times \Sigma) \cup (\Sigma \times \{\top\}) \cup \{(\bot, \top)\}.$$

Figure 2.2 represents such a lattice with the same conventions than Figure 2.1.

2.3.2.b Main Properties

[8] The reflexive-transitive closure of a relation is always a preordering, but it may not be an ordering. E.g., the reflexive-transitive closure of $\Sigma \times \Sigma$ is $\Sigma \times \Sigma$ and is not an ordering, since the "antisymmetry" property of Definition 2.6 is not satisfied.

[9] Notice that, in our definition, $[0, +\infty] = \{n \in \mathbb{Z} \mid 0 \leqslant n < +\infty\} = \mathbb{N} \neq \mathbb{N} \cup \{+\infty\}$.

Definition 2.20. A *bounded* lattice is a lattice with a greatest element and a smallest element.

In the name of simplicity, when it is clear due to the context, we write $\bot$ for the smallest element (bottom) of a bounded lattice and $\top$ for the greatest element (top) of a bounded lattice. Notice that some authors [GHK+98, GHK+03] call lattices what we call bounded lattices.

Let us gives some examples:

- $\mathbb{R}$ is a lattice but not a bounded lattice
- **Ranges**, the interval $\mathbb{I}$ of real numbers between 0 and 1, and the lattice of Figure 2.1 are bounded lattices.

Definition 2.21. A *complete* lattice L is a lattice such that for any subset $X \subseteq L$, X has a least upper bound and a greatest lower bound.

As a consequence, all complete lattices are bounded lattices; and all finite non-empty lattices are complete lattices. For instance, **Ranges** and $\mathcal{P}(\Sigma)$ are complete lattices.

Definition 2.22. A *distributive* lattice is a lattice for which the operations of join and meet distribute over each other. Formally:

$$\forall x \forall y \forall z, x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z) \wedge x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z).$$

All lattices are not distributive, e.g, the lattice of Figure 2.1 is not distributive since $x \sqcup (y \sqcap z) = u$ and $(x \sqcup y) \sqcap (x \sqcup z) = \top$.

Definition 2.23. A *complemented* lattice is a bounded lattice L such that each element $x \in L$ has a complement, i.e.:

$$\forall x \in L, \exists y \in L : x \sqcap y = \bot \wedge x \sqcup y = \top.$$

Notice that, in a distributed and complemented lattice, each element has a unique complement.

Definition 2.24. The height of a lattice $(\Sigma, \leqslant)$ is the largest $n \in \mathbb{N}$ such that there exists a sequence $x_0 \in \Sigma, \ldots, x_n \in \Sigma$ such that for every $k \in \{0, \ldots, n\}$, $x_k < x_{k+1}$.

If no such n exists[10], then we say that the lattice has infinite height.

In other words, the height of a lattice, is the length (minus 1) of the greatest strictly increasing chain. For instance, a flat lattice has height 2, the lattice of Figure 2.1 has height 3, and the lattice **Ranges** has infinite height.

[10] We should have defined the height as an ordinal or a cardinal. In this case, a lattice has an infinite height whenever its height is an infinite ordinal or cardinal.

2.3.3 Construction of Lattices

In this section, we give some ways to construct new lattices. Given two lattices $(L_1, \leqslant_1)$, $(L_2, \leqslant_2)$ we can define their product:

Definition 2.25. The Cartesian *product* of two lattices $(L_1, \leqslant_1)$ and $(L_2, \leqslant_2)$ is the set $L_1 \times L_2$ ordered by the product ordering.

In a similar way, we can construct a lattice of functions from a set X. This lattice may be seen as the product of $card(X)$ times the same lattice:

Definition 2.26. Given a lattice $(L, \leqslant)$ and an arbitrary set X, the *lattice of functions* from X to L is the set L^X ordered by the pointwise ordering.

Obviously, they are lattices:

Claim 2.4. *The Cartesian product of two lattices is a lattice.*
The lattice of functions from a set X to a lattice L is a lattice.

Another way to construct lattices is to consider the set of subsets $\mathcal{P}(X)$ of some set X. Here we consider a sublattice of the set of subsets of a poset:

Definition 2.27. A subset X of an poset $L, \leqslant$ is upper-closed if and only if

$$\forall x \in x, \forall y \in L, x \leqslant y \rightarrow y \in X.$$

The set of upper closed subset of L is written $\mathcal{P}^{\uparrow}(L)$.

Claim 2.5. *If $L, \leqslant$ is a poset, $\mathcal{P}^{\uparrow}(L)$ is a complete lattice for the inclusion ordering.*

Notice that, given a finite lattice $L, \leqslant$, any element X of $\mathcal{P}^{\uparrow}(L)$ can be represented a sequence $s_1, \ldots, s_n$ of elements of L such that $X = \bigcup_{k \in \{1,\ldots,n\}} \{x \in L \mid s_n \leqslant x\}$. There exist several sequences that gives the same set. Let us consider antichains:

Definition 2.28. An antichain is a set X such that $\forall x, y \in X, x \neq y \Rightarrow \neg(x \leqslant y) \wedge \neg(y \leqslant x)$.

An antichain is a set such that two distinct elements are uncomparable. Given a finite lattice $L, \leqslant$, each element of $\mathcal{P}^{\uparrow}(L)$ can be represented by a finite antichain. This antichain is unique, i.e., two distinct antichains represents two distinct subsets of L.

2.4 Words

Given an alphabet Σ, the elements of Σ are called letters, and a *word* is a finite sequence of letters. E.g., *aaabab* is a word on the alphabet $\{a, b\}$. We write $w = a_1 \ldots a_n$ to say that w is the finite sequence of letters a_1, ..., a_n.

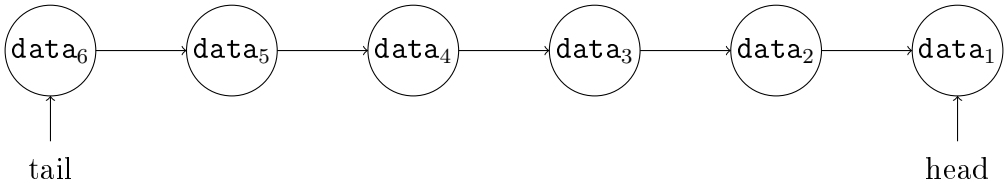


Figure 2.3: Example of FIFO

The concatenation of two words $u = a_1 \dots a_n$ and $v = b_1 \dots b_m$ is $u \cdot v \stackrel{\text{def}}{=} a_1 \dots a_n b_1 \dots b_m$. The empty word, i.e., the word with zero letters, is written ϵ.

A word $u = a_1 \dots a_n$ is a *subword* of a word v if there exists a sequence $w_1, \dots, w_{n+1}$ such that $v = w_1 \cdot a_1 \cdot w_2 \cdot \dots \cdot a_n \cdot w_{n+1}$. In other words, a word u is a subword of v is we can reach u by erasing letters in v. E.g., *aa* and *bb* are subwords of *baab* but *aba* is not a subword of *baab*.

A word u is a *prefix* of a word v if there exists a word w such that $u \cdot w = v$. The relation $\leqslant_{\text{prefix}}$ is defined by $u \leqslant_{\text{prefix}} v \stackrel{\text{def}}{\Leftrightarrow} \exists w : u \cdot w = v$.

The concatenation by an inverse word is defined by : $u^{-1} \cdot (u \cdot v) = v$. $u^{-1}w$ is undefined if u is not a prefix of w.

Claim 2.6. *The relation $\leqslant_{prefix}$ is an ordering on words.*

This relation is a total ordering on some set of words:

Claim 2.7. *Given a word w, the relation $\leqslant_{prefix}$ is a total ordering on the set of prefixes of w.*

2.4.1 FIFO

First-in first-out queues (FIFO) are an abstract data structure. In a FIFO, we can add some elements. The first element pushed on a FIFO will be the first element that will be extracted. Figure 2.3 gives an example of a FIFO containing $\texttt{data}_1, \dots, \texttt{data}_6$. If we want to add $\texttt{data}_7$, it will be added to the tail, after $\texttt{data}_6$. The first element that will be extracted from the FIFO is $\texttt{data}_1$.

Formally, we model FIFO as words: A FIFO on the alphabet Σ is represented by a word on Σ. Let $\mathbf{FIFO}_\Sigma$ be the set of FIFOs on the alphabet Σ.

We define on words the standard FIFO operations:

- $fst : \mathbf{FIFO}_\Sigma \to \Sigma$
- $deq : \mathbf{FIFO}_\Sigma \to \mathbf{FIFO}_\Sigma$
- $enq : \Sigma \times \mathbf{FIFO}_\Sigma \to \mathbf{FIFO}_\Sigma$

The partial function *fst* reads the first element of the FIFO; the partial function *deq* discards the first element of the FIFO, and the function *enq* adds an element at the end of a FIFO and ϵ is the empty FIFO.

Formally, for any letter a and any word u:

$$\begin{aligned} \mathit{fst}(u \cdot a) &\stackrel{\text{def}}{=} a \\ \mathit{deq}(u \cdot a) &\stackrel{\text{def}}{=} u \\ \mathit{enq}(a, u) &\stackrel{\text{def}}{=} a \cdot u \end{aligned}$$

In Figure 2.3, the function *fst* will return $\texttt{data}_1$, and the function *deq* will erase $\texttt{data}_1$. FIFO will be used in Chapter 9 to define buffers.

CHAPTER 3

Abstract Interpretation

3.1 Basic Principles

A semantics $[\![\cdot]\!] : \textbf{Programs} \to \mathtt{S}$ associates to each program a value, in a set $\mathtt{S}$.

For instance, a semantics can associate to each program a transition system. The transition system represents the possible behaviors of the program during an execution. This kind of semantics is called small-step semantics, because it describes each step of the execution of a program. In Part II we give a such semantics for our programs.

A semantics may be hard to compute, or even may be unrepresentable or uncomputable. To study the properties of a semantics $[\![\cdot]\!]$, an approach to abstract interpretation [Cou96] is to give an alternative semantics $(\!|\cdot|\!)$ to programs. Given an abstract domain $\mathscr{S}$, an abstract semantics $(\!|\cdot|\!) : \textbf{Programs} \to \mathtt{S}$ maps programs to $\mathscr{S}$. Programs then have two semantics, a semantics $[\![\cdot]\!]$, called *concrete semantics* and an abstract semantics $(\!|\cdot|\!)$.

An abstract semantics may be anything. Nevertheless, the main interest of an abstract semantics is its link with the concrete semantics. This is modeled by a soundness property σ that is hold for all programs. Formally, we want:

$$\forall p \in \textbf{Programs}, \sigma([\![p]\!], (\!|p|\!)).$$

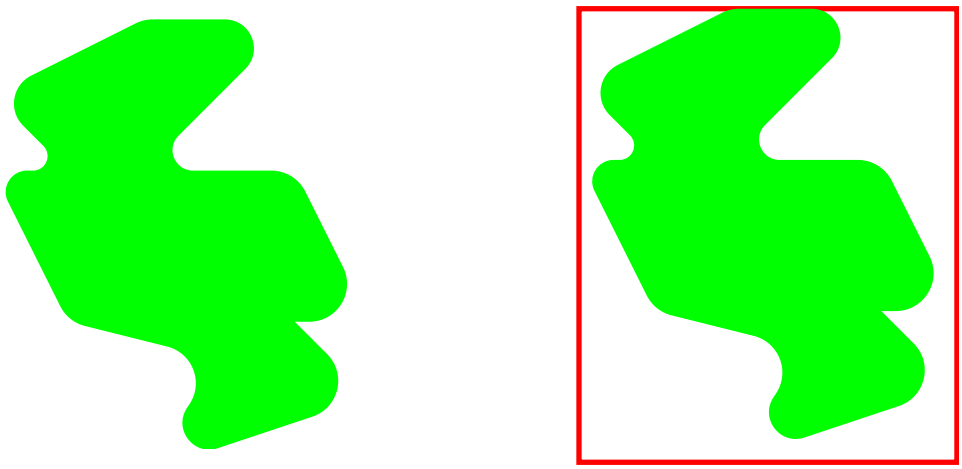


Figure 3.1: Overapproximation

3.2 Galois Connections

Here, we use abstract interpretation to overapproximate [CC04] the possible behaviors of a program. The abstract semantics $(\!|\cdot|\!)$ will overapproximate (in some sense) the concrete semantics $[\![\cdot]\!]$. To formally define "overapproximate", we use Galois connections:

Definition 3.1. A *Galois connection* [GHK+03, CC91, CC04] between a poset X and a poset Y is a pair of monotone functions $\alpha : X \to Y$ and $\gamma : Y \to X$ such that:

$$\forall x \in X, \forall y \in Y, \alpha(x) \leqslant y \Leftrightarrow x \leqslant \gamma(y).$$

Definition 3.2. A *domain* on a concrete complete lattice $\mathtt{D}$ is a tuple $(\mathscr{D}, \alpha, \gamma)$ where $\mathscr{D}$ is an abstract lattice, and α, γ is a Galois connection between $\mathtt{D}$ and $\mathscr{D}$.

The function α, called *abstraction function*, lose information. It overapproximates a complex concrete object by a simpler abstract one. In Figure 3.1, the green object, at the left, can be approximated by the rectangle (at the right). The function α is called the *abstraction* function and γ is the *concretization* function.

We use a particular instance of Galois connections. Let us consider two lattices $\mathtt{D}$ and $\mathscr{D}$. A concrete semantics $[\![\cdot]\!] : \textbf{Programs} \to (\mathtt{D} \to \mathtt{D})$ associates to each program, a monotone function (called "transfer function") from the concrete lattice to itself. Similarly, the abstract semantics $[\![\cdot]\!] : \textbf{Programs} \to (\mathscr{D} \to \mathscr{D})$ associates to each program a monotone function from the abstract lattice to itself. The soundness property is:

$$\forall p \in \textbf{Programs}, \forall X \in \mathtt{D}, \alpha \circ [\![p]\!] \circ \gamma(X) \leqslant (\!|p|\!)(X).$$

This means that the abstract semantics is an abstraction of the concrete semantics. Formally:

Definition 3.3. Given a Galois connection α, γ. A monotone function $f^\sharp$ is an *abstraction* of a monotone function $f^\natural$ if and only if $\alpha \circ f^\natural \circ \gamma \leqslant f^\sharp$.

There exists several equivalent definitions of abstractions:

Claim 3.1. *Given a Galois connection α, γ between $\mathtt{D}$ and $\mathscr{D}$ and two monotone functions $f^\natural : \mathtt{D} \to \mathtt{D}$ and $f^\sharp : \mathscr{D} \to \mathscr{D}$, the following properties are equivalent:*

1. *$f^\sharp$ is an* abstraction *of $f^\natural$,*
2. *$\alpha \circ f^\natural \circ \gamma \leqslant f^\sharp$,*
3. *$f^\natural \circ \gamma \leqslant \gamma \circ f^\sharp$,*
4. *$\alpha \circ f^\natural \leqslant f^\sharp \circ \alpha$,*
5. *$f^\natural \leqslant \gamma \circ f^\sharp \circ \alpha$.*

Proof. The equivalence between Points 1 and 2 is given by Definition 3.3.

Definition 3.1 gives the equivalence between Points 3 and 2 and the equivalence between Points 4 and 5.

If $\forall Y \in \mathscr{D}, f^\natural \circ \gamma(Y) \leqslant \gamma \circ f^\sharp(Y)$, then $\forall X \in \mathtt{D}, f^\natural \circ \gamma(\alpha(X)) \leqslant \gamma \circ f^\sharp(\alpha(X))$, then, because $\forall X \in \mathtt{D}, X \leqslant \gamma \circ \alpha(X)$, $\forall X \in \mathtt{D}, f^\natural(X) \leqslant \gamma \circ f^\sharp \circ \alpha(X)$. The reverse inclusion is proven similarly using the fact that $\forall Y \in \mathscr{D}, \alpha \circ \gamma(Y) \leqslant Y$. □

We give as example a Galois connection $\alpha_{\mathbf{Ranges}}, \gamma_{\mathbf{Ranges}}$ between $\mathcal{P}(\mathbb{Z})$ and **Ranges**:

$$\begin{array}{rcl} \alpha_{\mathbf{Ranges}}(X) & \stackrel{\text{def}}{=} & [\mathrm{glb}(X), \mathrm{lup}(X)], \\ \gamma_{\mathbf{Ranges}}(X) & \stackrel{\text{def}}{=} & X. \end{array}$$

This Galois connection allows us to abstract the value of one integer variable. Let us give a second example, which allows us to represent several variables. Let $\mathcal{V}ar$ be the set of variables, the concrete lattice is $\mathcal{P}(\mathbb{Z}^{\mathcal{V}ar})$ and the abstract lattice is $\mathbf{Ranges}^{\mathcal{V}ar}$, ordered by the pointwise ordering[11]. The Galois connection is then:

$$\begin{array}{rcl} \alpha(\sigma) & \stackrel{\text{def}}{=} & \alpha_{\mathbf{Ranges}} \circ \sigma, \\ \gamma(\sigma^\sharp) & \stackrel{\text{def}}{=} & \{\sigma \in \mathbb{Z}^{\mathcal{V}ar} \mid \forall x \in \mathcal{V}ar, \sigma(x) \in \gamma_{\mathbf{Ranges}} \circ \sigma^\sharp(x)\}. \end{array}$$

Our semantics will be defined by induction on programs. Typically, a semantics is defined using function composition (e.g, for sequences), ω-iterations (e.g., for *while* loops), union etc.

For instance, let us consider the program[12] of Figure 3.2. We use for this example the concrete lattice $\mathtt{D} = \mathcal{P}(\mathbb{Z})$ (a set $X \in \mathtt{D}$ represents all possible values of i) and the abstract lattice **Ranges**. The transfer function $[\![i := 1]\!]$ associated to the "i := 1 ;" statement is:

[11] This ordering is defined in Definition 2.10.

[12] This program was given as an example by P. Cousot and R. Cousot [CC92].

```
1  i := 1;
2  while(i ⩽ 100)
3     { i:=i+1; };
```

Figure 3.2: Program Example

$\lambda X.\{i\}$. The function transfer $[\![i := i+1]\!]$ is defined by $[\![i := i+1]\!](X) = \{n+1 \mid n \in X\}$. The transfer function of the guard "i⩽ 100" is $[\![i \leqslant 100]\!](X) = X \cap [-\infty, 100]$. The transfer function of the while loop is then defined using composition and ω-iteration:

$$[\![\textit{while}(i \leqslant 100)\{i := i+1\}]\!] = [\![i > 100]\!] \circ ([\![i := i+1]\!] \circ [\![i \leqslant 100]\!])^{\uparrow\omega}.$$

Fortunately, abstractions can be composed, iterated, etc.

Proposition 3.1. *Let α, γ be a Galois connection between two complete lattices $\mathtt{D}$ and $\mathscr{D}$. Let us consider two monotone functions $f^\natural : \mathtt{D} \to \mathtt{D}$, $g^\natural : \mathtt{D} \to \mathtt{D}$ and their respective abstractions $f^\sharp : \mathscr{D} \to \mathscr{D}$, $g^\sharp :: \mathscr{D} \to \mathscr{D}$.*

- *$g^\sharp \circ f^\sharp$ is an abstraction of $g^\natural \circ f^\natural$.*
- *$(f^\sharp)^{\uparrow\omega}$ is an abstraction of $(f^\natural)^{\uparrow\omega}$*
- *$\lambda x. f^\sharp(x) \sqcup g^\sharp(x)$ is an abstraction of $\lambda x. f^\natural(x) \sqcup g^\natural(x)$.*

Hence, in the example of Figure 3.2, we only need to give an abstraction of basic statements. The abstract semantics of the *while* loop may be defined by:

$$(\!|\textit{while}(i \leqslant 100)\{i := i+1\}|\!) = (\!|i > 100|\!) \circ ((\!|i := i+1|\!) \circ (\!|i \leqslant 100|\!))^{\uparrow\omega}.$$

Galois connections satisfy properties that simplify their definition.

Proposition 3.2. *Let α, γ be a Galois connection between two complete lattices $\mathtt{D}$ and $\mathscr{D}$. Therefore:*

1. $\forall F \in \mathtt{D}^I, \alpha(\bigsqcup_{i\in I} F(i)) = \bigsqcup_{i\in I} \alpha(F(i))$,
2. $\forall G \in \mathscr{D}^I, \gamma(\bigsqcap_{i\in I} G(i)) = \bigsqcap_{i\in I} \gamma(G(i))$,
3. $\forall X \in \mathtt{D}, \alpha(X) = \bigsqcap_{Y\in\mathscr{D}\wedge X\leqslant\gamma(Y)} Y$,
4. $\forall Y \in \mathscr{D}, \gamma(Y) = \bigsqcup_{X\in\mathtt{D}\wedge\alpha(X)\leqslant Y} X$.

Point 1 allows us to simplify the definition of an abstraction function. Let us consider a subset S of $\mathtt{D}$ that generates $\mathtt{D}$, i.e., such that: $\forall X \in \mathtt{D}, \exists S' \subseteq S : X = \bigsqcup_{X'\in S'} X'$. The values of α on elements of S uniquely determine α. Hence, to define α, we may give

the definition of α only on S. For instance, if $\mathtt{D} = \mathcal{P}(\Sigma)$, then, we may define α only on singletons. The definition of $\alpha_{\mathbf{Ranges}}$ is then simplified:

$$\alpha(\{x\}) \stackrel{\text{def}}{=} \{x\}.$$

Indeed, if $\alpha(\{x\}) \stackrel{\text{def}}{=} \{x\}$, therefore, $\alpha(X) = \bigsqcup_{x\in X} \alpha(\{x\}) = \bigsqcup_{x\in X}\{x\}$. Notice that $\sqcup$ on **Ranges** is distinct from the union $\cup$, since $\{0\}\sqcup\{2\} = \{0,1,2\}$ and $\{0\}\sqcup\{2\} = \{0,2\}$. If X is finite non-empty, $\alpha(X) = \{\mathrm{glb}(X)\}\sqcup\{\mathrm{lub}(X)\}\sqcup\bigsqcup_{x\in X}\{x\} = [\mathrm{glb}(X), \mathrm{lup}(X)]\sqcup\bigsqcup_{x\in X}\{x\}$. Since **Ranges** are ordered by inclusion, $\alpha(X) = [\mathrm{glb}(X), \mathrm{lub}(X)]$. The case where X is infinite is similar. The case $X = \emptyset$ is trivial: $\alpha(\emptyset) = \emptyset = [+\infty, -\infty] = [\mathrm{glb}(\emptyset), \mathrm{lub}(\emptyset)]$.

The Points 3 and 4 mean that the abstraction function uniquely determines the concretization function and reciprocally. Hence, to define a Galois connection α, γ we just have to give α or to define γ. Finally, the Galois connection $\alpha_{\mathbf{Ranges}}, \gamma_{\mathbf{Ranges}}$ may be defined by the simple following equation: $\alpha(\{x\}) \stackrel{\text{def}}{=} \{x\}$.

Definition 3.4. Product of domains. We consider two concrete lattices $\mathtt{D}_1$ and $\mathtt{D}_2$ and two abstract lattices $\mathscr{D}_1$, $\mathscr{D}_2$. Let us assume two Galois connections α_1, γ_1 and α_2, γ_2 from $\mathtt{D}_1$ to $\mathscr{D}_1$ and from $\mathtt{D}_2$ to $\mathscr{D}_2$ respectively.

The *separate* product of domains is the domain $\mathscr{D}_{1|2}, \alpha_{1|2}, \gamma_{1|2}$ where:

- $\mathscr{D}_{1|2}$ is the Cartesian product of $\mathscr{D}_1$ and $\mathscr{D}_2$, ordered by the product ordering.
- $\alpha_{1|2}, \gamma_{1|2}$ is a Galois connection between $\mathtt{D}_1 \times \mathtt{D}_2$ (ordered by the product ordering) and $\mathscr{D}_{1|2}$ defined by:

$$\alpha_{1|2}(x_1, x_2) = (\alpha_1(x_1), \alpha_2(x_2)) \tag{3.1}$$
$$\gamma_{1|2}(y_1, y_2) = (\gamma_1(x_1), \gamma_2(y_2)) \tag{3.2}$$
$$\tag{3.3}$$

3.3 Widening and Narrowing

As seen before, if $f^\sharp$ is an abstraction of $f^\natural$, then $(f^\sharp)^{\uparrow\omega}$ is an abstraction of $(f^\natural)^{\uparrow\omega}$. Nevertheless, even though $f^\sharp$ is computable, $(f^\sharp)^{\uparrow\omega}$ may be uncomputable or may be hard to compute. In the example of Figure 3.2, computing $(\!|i := i + 1|\!) \circ (\!|i \leqslant 100|\!))^{\uparrow\omega}(\{1\})$ need 100 iterations! We need a new method to find an easily computable abstraction of $(f^\natural)^{\uparrow\omega}$.

P. Cousot and R. Cousot introduce the concept of widening [CC92, CC91].

Definition 3.5. A *simple widening operator* on an abstract lattice $\mathscr{D}$ is a binary operator $\triangledown : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$ such that:

1. $\forall x, y \in \mathscr{D}, x \sqcup y \leqslant x \triangledown y$.
2. For every infinite increasing chain x_1, x_2, ..., the sequence y_n inductively defined by $y_0 = x_0$ and $y_n + 1 = y_n \triangledown x_{n+1}$ is stationary.

The first point means that the widening operator overapproximates the least upper bound. The second point ensures termination when computing inductively y (y_n is an overapproximation of x_n). Notice that, on a lattice of finite height, the least upper bound $\sqcup$ is a widening.

Let us give an example. We define the widening operator $\triangledown^{\mathbf{Ranges}}$ by:

- $\emptyset \triangledown^{\mathbf{Ranges}} X = X$
- $X \triangledown^{\mathbf{Ranges}} \emptyset = X$
- $[a,b] \triangledown^{\mathbf{Ranges}} [a',b'] \stackrel{\text{def}}{=} [c,d]$ where $a \leqslant b$, $a' \leqslant b'$,
  $$c \stackrel{\text{def}}{=} \begin{cases} a & \text{if } a \leqslant a' \\ -\infty & \text{otherwise} \end{cases} \quad \text{and} \quad d \stackrel{\text{def}}{=} \begin{cases} b & \text{if } b \geqslant b' \\ +\infty & \text{otherwise} \end{cases}$$

Notice that the $\triangledown$ operator overapproximate $\sqcup$ that is commutative, but, $\triangledown$ may not be commutative. For instance, $[0,1] \triangledown^{\mathbf{Ranges}} [0,2] = [0,+\infty]$ and $[0,2] \triangledown^{\mathbf{Ranges}} [0,1] = [0,2]$.

A widening operator allows us to compute an overapproximation of $(f^\sharp)^{\uparrow\omega}$. Indeed, given an abstract function $f^\sharp$, let:

$$\begin{aligned} (f^\sharp)^{\uparrow\triangledown} &= \textstyle\bigsqcup_{n\in\mathbb{N}} ((f^\sharp)^\triangledown)^n = ((f^\sharp)^\triangledown)^{\uparrow\omega} \\ &\text{where } (f^\sharp)^\triangledown = \lambda X. X \triangledown f^\sharp(X). \end{aligned}$$

By construction, $(f^\sharp)^{\uparrow\omega} \leqslant (f^\sharp)^{\uparrow\triangledown}$. If $f^\sharp$ is an abstraction of $f^\natural$, then, according to Proposition 3.1, $(f^\sharp)^{\uparrow\omega}$ is an abstraction of $(f^\natural)^{\uparrow\omega}$ and therefore $(f^\sharp)^{\uparrow\triangledown}$ is an abstraction of $(f^\natural)^{\uparrow\omega}$.

For instance, $(\langle\!|i := i+1|\!\rangle \circ \langle\!|i \leqslant 100|\!\rangle)^{\uparrow\triangledown^{\mathbf{Ranges}}}$ is an abstraction of $(\llbracket i := i+1 \rrbracket \circ \llbracket i \leqslant 100 \rrbracket)^{\uparrow\omega}$.

Definition 3.6. A *general widening operator* on an abstract lattice $\mathscr{D}$ is a sequence of binary operators $\triangledown_n : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$ such that:

1. $\forall n \in \mathbb{N}, \forall x, y \in \mathscr{D}, x \sqcup y \leqslant x \triangledown_n y$.
2. For every infinite increasing chain x_1, x_2, ..., the sequence y_n inductively defined by $y_0 = x_0$ and $y_n + 1 = y_n \triangledown_n x_{n+1}$ is stationary.

As for simple widening operators, general widening operators overapproximate the least upper bound. The main difference is in the infinite chain condition. Point 2 allows us to change the overapproximation of the lest upper bound during the fixpoint computation. As for simple widenings, we define an overapproximation of the ω-iteration: $(f^\sharp)^{\uparrow\triangledown} = \bigsqcup_{n\in\mathbb{N}} f^\sharp_n$ where $f^\sharp_0 = f^\sharp$ and $f^\sharp_{n+1} = \lambda X. f^\sharp_n(X) \triangledown_n f^\sharp \circ f^\sharp_n(X)$.

E.g., on **Ranges** we may define the following widening operator:

- $\triangledown_0 = \triangledown_1 = \triangledown_2 = \sqcup$,
- For $n \geqslant 3$, $\triangledown_n = \triangledown^{\mathbf{Ranges}}$.

```
1  i := 1;
2  while(i ⩽ 3)
3     { i:=i+1; };
```

Figure 3.3: Program Example

In practice, this widening operator "unrolls" three times a ***while*** loop. Hence, using this widening: $(\!|i := i+1|\!) \circ (\!|i \leqslant 100|\!))^{\uparrow\triangledown} = (\!|i := i+1|\!) \circ (\!|i \leqslant 100|\!))^{\uparrow\triangledown^{\mathbf{Ranges}}} \circ (\!|i := i+1|\!) \circ (\!|i \leqslant 100|\!))^3$.

This widening is not very precise, because $(\!|i := i+1|\!) \circ (\!|i \leqslant 100|\!))^{\uparrow\triangledown}\{1\} = [1, +\infty]$. Nevertheless this widening is more precise than $\triangledown^{\mathbf{Ranges}}$: consider the program of Figure 3.3. With this general widening operator : $(\!|i := i+1|\!) \circ (\!|i \leqslant 3|\!))^{\uparrow\triangledown}\{1\} = [1, 3]$, but with $\triangledown^{\mathbf{Ranges}}$: $(\!|i := i+1|\!) \circ (\!|i \leqslant 3|\!))^{\uparrow\triangledown^{\mathbf{Ranges}}}\{1\} = [1, +\infty]$.

To enhance precision, P. Cousot and R. Cousot introduce narrowing operators:

Definition 3.7. A *simple narrowing operator* on an abstract lattice $\mathscr{D}$ is a binary operator $\Delta : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$ such that:

1. $\forall x, y \in \mathscr{D}, y \leqslant x \Rightarrow y \leqslant x \Delta y \leqslant x$.

2. For each infinite decreasing chain x_1, x_2, ..., the sequence y_n inductively defined by $y_0 = x_0$ and $y_n + 1 = y_n \Delta x_{n+1}$ is stationary.

A narrowing operator is used after a widening. It allows to enhance precision. We define $(f^\sharp)^{\downarrow\Delta}$ in the same way as $(f^\sharp)^{\uparrow\triangledown}$:

$$(f^\sharp)^{\downarrow\Delta} = \bigsqcap_{n\in\mathbb{N}}((f^\sharp)^\Delta)^n \quad \text{where } (f^\sharp)^\Delta = \lambda X. X \Delta f^\sharp(X).$$

Notice that $((f^\sharp)^{\uparrow\triangledown})^{\downarrow\Delta}$ is still an abstraction of $(f^\sharp)^{\uparrow\omega}$. Nevertheless, $((f^\sharp)^{\uparrow\triangledown})^{\downarrow\Delta}$ is a more precise abstraction than $(f^\sharp)^{\uparrow\triangledown}$, in the sense that $((f^\sharp)^{\uparrow\triangledown})^{\downarrow\Delta} \leqslant (f^\sharp)^{\uparrow\triangledown}$.

Notice that on a lattice without any infinite decreasing chain, the greatest lower bound $\sqcap$ is a narrowing.

Let us recall the Cousot and Cousot [CC92, CC77] narrowing on **Ranges**:

- $\emptyset \Delta^{\mathbf{Ranges}} X = \emptyset$
- $X \Delta^{\mathbf{Ranges}} \emptyset = \emptyset$
- $[a, b] \triangledown^{\mathbf{Ranges}} [a', b'] \stackrel{\text{def}}{=} [c, d]$ where $a \leqslant b$, $a' \leqslant b'$,
  $$c \stackrel{\text{def}}{=} \begin{cases} a & \text{if } -\infty < a \\ a' & \text{otherwise} \end{cases} \quad \text{and} \quad d \stackrel{\text{def}}{=} \begin{cases} b & \text{if } b < +\infty \\ b' & \text{otherwise} \end{cases}$$

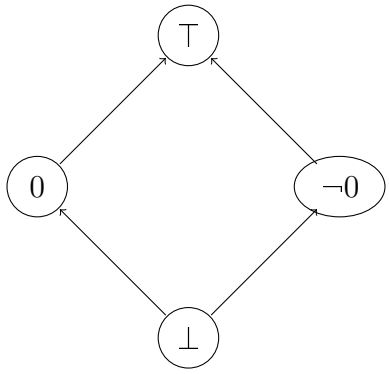


Figure 3.4: The Lattice **NotZero**.

Using this narrowing, we obtain with the example of Figure 3.2:

$$((\!|i := i + 1|\!) \circ (\!|i \leqslant 100|\!))^{\uparrow\triangledown^{\mathbf{Ranges}}})^{\downarrow\Delta^{\mathbf{Ranges}}}\{1\} = [1, 100].$$

As for widening, narrowing can change during a fixpoint computation:

Definition 3.8. A *general widening operator* on an abstract lattice $\mathscr{D}$ is a sequence of binary operators $\triangledown_n : \mathscr{D} \times \mathscr{D} \to \mathscr{D}$ such that:

1. $\forall n \in \mathbb{N}, \forall x, y \in \mathscr{D}, y \leqslant x \Rightarrow y \leqslant x\Delta_n y \leqslant x$.
2. For each infinite decreasing chain x_1, x_2, ..., the sequence y_n inductively defined by $y_0 = x_0$ and $y_n + 1 = y_n\Delta_n x_{n+1}$ is stationary.

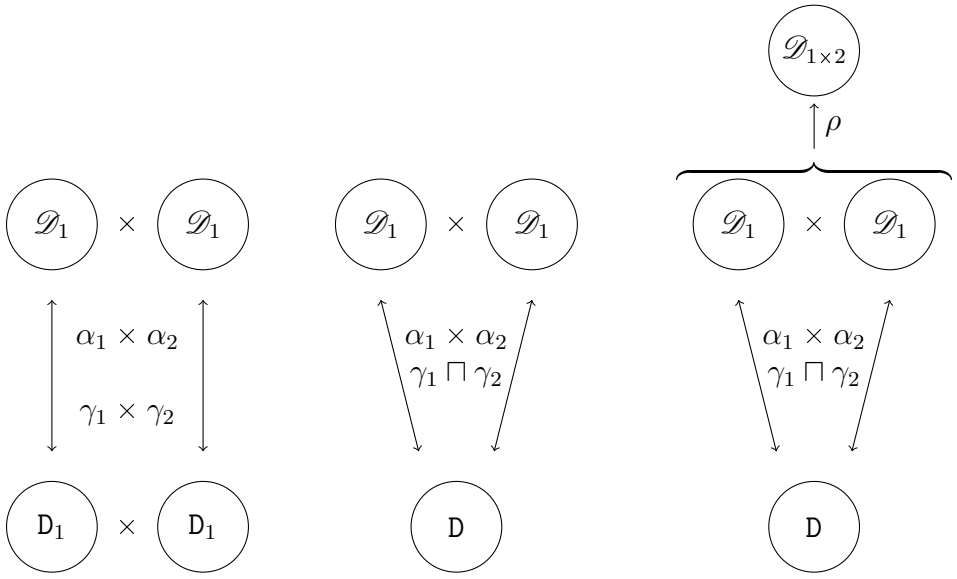
Hence, we can use as narrowing:

- $\Delta_0 = \sqcap$,
- For $n \geqslant 1$, $\Delta_n = \Delta^{\mathbf{Ranges}}$.

3.4 Reduced Product

Let us consider the Euclides algorithm (See Figure 3.6). This algorithm computes the greatest common divisor between two integers `a` and `b`. This algorithm uses the modulo operator "%" that uses a division. Then, a division by 0 may occur, if at line 6, the value of b is zero. The domain of ranges will not be sufficient to prove this piece of code.

Since $\alpha_{\mathbf{Ranges}}(\mathbb{Z} \setminus \{0\}) = [-\infty, +\infty])$, the condition "`b` $\neq 0$" at line 4 of Figure 3.6 does not gives us any information: the domain **Ranges** loses all precision. After this condition, the real value of `b` is not zero, but the abstract value of `b` is still the range $[-\infty, +\infty]$. Hence, for the domain **Ranges**, the value of `b` may be zero at lines 5 to 9. Therefore, at line 6, for the domain **Ranges** a division by zero may occur. This is a false positive, since the real value of `b` cannot be 0 at line 6.



(a) Separated Product (b) Simple Product (c) Reduced Product

Figure 3.5: Products

The absence of division by zero may be proved by the domain **NotZero** $= \{\bot, 0, \neg 0, \top\}$ whose ordering is given by Figure 3.4. The Galois connection between $\mathcal{P}(\mathbb{Z})$ and **NotZero** is[13]:

$$\begin{aligned} \gamma_{\neg 0}(\bot) &\stackrel{\text{def}}{=} \emptyset \\ \gamma_{\neg 0}(0) &\stackrel{\text{def}}{=} \{0\} \\ \gamma_{\neg 0}(\neg 0) &\stackrel{\text{def}}{=} \mathbb{Z} \setminus \{0\} \\ \gamma_{\neg 0}(\top) &\stackrel{\text{def}}{=} \mathbb{Z}. \end{aligned}$$

Using the domain **NotZero**, the guard "$\mathtt{b} \neq 0$" at line four implies that the abstract value of b is $\neg 0$ at lines 6, 7 and 8. Therefore, the modulo operation at line 6 is correct.

In the same program, we may find both:

- functions like `euclide` of Fig 3.6 that need the domain **NotZero**,
- and array access (e.g., Figure 1.3) that need the domain of ranges.

In this case, we need both domains: we use the product domain [CC79, Theorem 10.1.0.1].

Definition 3.9. Given a concrete complete lattice $\mathtt{D}$, two abstract lattices $\mathscr{D}_1$ and $\mathscr{D}_2$ and two Galois connections α_1, γ_1 and α_2, γ_2 from $\mathtt{D}$ to $\mathscr{D}_1$ and $\mathscr{D}_2$ respectively, we define the

[13] Recall Section 3.2. In this Section, we show that a Galois connection is uniquely defined by its concretization function.

```
1  int euclide(int a, int b)
2  {
3   int r;
4   while(b ≠ 0)
5     {
6       r := a % b;
7       a := b;
8       b := r;
9     }
10  return a;
11 }
```

Figure 3.6: Euclides Algorithm

simple product domain $(\mathscr{D}_{1,2}, \alpha_{1,2}, \gamma_{1,2})$:

$$\begin{aligned} \mathscr{D}_{1,2} &\stackrel{\text{def}}{=} \mathscr{D}_1 \times \mathscr{D}_2 \\ \alpha_{1,2}(X) &\stackrel{\text{def}}{=} (\alpha_1(X), \alpha_2(X)) \\ \gamma_{1,2}(Y_1, Y_2) &\stackrel{\text{def}}{=} \gamma_1(Y_1) \sqcap \gamma_2(Y_2) \end{aligned}$$

This simple product is not totally satisfactory, since $\gamma_{1,2}$ may not be injective, and $\alpha_{1,2}$ may not be surjective. This means that two distinct abstract elements may represent the same concrete element. For instance, let us consider the product between the domain of ranges **Ranges** and the domain **NotZero**. The empty set $\emptyset$ is represented, in the abstract, by $(\emptyset, \bot)$ and by $([0,0], \neg 0)$, i.e.:

$$\gamma_{1,2}(\emptyset, \bot) = \gamma_{1,2}([0,0], \neg 0) = \emptyset.$$

The function $\alpha_{1,2}$ is not injective, since there exists no set $X \subseteq \mathbb{Z}$ such that $\alpha_{1,2}(X) = ([0,0], \neg 0)$. The set of abstract elements representing $\emptyset$ (i.e., the preimage of $\emptyset$ under f, formally $\{Y \in \mathscr{D}_{1,2} \mid \gamma_{1,2}(Y) = \emptyset\}$) is:

$$\{(\emptyset, Y) \mid Y \in \mathscr{D}_2\} \cup \{(Y, \bot) \mid Y \in \mathscr{D}_1\} \cup \{([0,0], \neg 0)\} \cup \{([a,b], 0) \mid b < 0 \vee a > 0\}.$$

Notice that, the bottom element of $\mathscr{D}_{1,2}$ represents the same concrete set than a tuple $(y_1, y_2) \in \mathscr{D}_1 \times \mathscr{D}_2$ where y_1 or y_2 is the bottom element of its lattice. But, this is not the unique case, since $([0,0], \neg 0)$ represents $\emptyset$, but neither $[0,0]$, neither $\neg 0$ is the bottom element of its lattice.

As a consequence, we lose precision when we analyze a program with a product domain. Consider the program given in Figure 3.7. Consider that the statements represented by "..." do not modify the value of `b`. At the beginning of the program, we consider that the value of `b` is unknown, i.e., the abstract value of `b` is $([-\infty, +\infty], \top)$. At line 1, after applying the guard "`b` $\neq 0$", the abstract value of `b` is $([-\infty, +\infty], \neg 0)$. The abstract

```
1  if (b ≠ 0)
2    { ...
3    if (b ⩾ 0)
4        { ...
5        if (b ⩽ 0)
6          {Dead Code}
7        }
8    }
```

Figure 3.7: The Naive Product Fails

domain of ranges knows nothing about `b`, but the domain **NotZero** detects that the value of `b` is not zero. At line 3, the abstract value of `b` is $([0, +\infty], \neg 0)$ and at line 5, the abstract value of `b` is $([0, 0], \neg 0)$. Nevertheless, in reality, line 6 is dead code, i.e., code that is never executed. Therefore, no run-time error can occur due to line 6. The domain **Ranges** $\times$ **NotZero** does not detect that is dead code, since the abstract value is not bottom ($\bot$) at line 6.

We need a method to reduce the abstract domain $\mathscr{D}_1 \times \mathscr{D}_2$, a method that allows some kind of communication between the two domains. It is standard to use the reduced product [Cou05, CC79, CFR+97, CMB+95, GT06], getting a more precise domain than both domains separately.

Recall that the main problem is that there exists in the product domain $\mathscr{D}_{1,2}$ two elements that have the same concretization, i.e., there exists y_1 and y_2 such that $\gamma_{1,2}(y_1) = \gamma_{1,2}(y_2)$. We introduce a lower closure operator ρ:

Definition 3.10. A *lower closure operator* ρ is a reductive[14] and monotone function such that $\rho \circ \rho = \rho$.

We define the following lower closure operator:

$$\rho_{1,2}(y) = \bigcap_{x \in \mathscr{D}_{1,2} \wedge \gamma_{1,2}(x) = \gamma_{1,2}(y)} x.$$

By construction, if $\gamma(y_1) = \gamma(y_2)$, then $\rho(y_1) = \rho(y_2)$. This allows us to define a domain in which the concretization function will be injective. Noticing that $\rho_{1,2} = \alpha_{1,2} \circ \gamma_{1,2}$, we define the *reduced product:*

Definition 3.11. Given a concrete complete lattice `D`, two abstract lattices $\mathscr{D}_1$ and $\mathscr{D}_2$ and two Galois connections α_1, γ_1 and α_2, γ_2 from `D` to $\mathscr{D}_1$ and $\mathscr{D}_2$ respectively, we define the following lower closure operator:

$$\rho_{1,2} \stackrel{\text{def}}{=} \alpha_{1,2} \circ \gamma_{1,2}$$

[14] See Definition 2.12.

This operator is used to define the *reduced product* domain $(\mathscr{D}_{1\times 2}, \alpha_{1\times 2}, \gamma_{1\times 2})$:

$$\begin{aligned} \mathscr{D}_{1\times 2} &\stackrel{\text{def}}{=} \rho(\mathscr{D}_{1,2}) \\ \alpha_{1\times 2}(X) &\stackrel{\text{def}}{=} \rho \circ \alpha_{1,2} \\ \gamma_{1\times 2}(Y_1, Y_2) &\stackrel{\text{def}}{=} \gamma_{1,2}(Y_1, Y_2) \end{aligned}$$

where $\mathscr{D}_{1,2}, \alpha_{1,2}, \gamma_{1,2}$ is the product domain, and $\rho(\mathscr{D}_{1,2}) \stackrel{\text{def}}{=} \{\rho(x) \mid x \in \mathscr{D}_{1,2}\}$.

The Galois connection $\alpha_{1,2}, \gamma_{1,2}$ gives a natural lower closure operator:

$$\rho_{1,2} \stackrel{\text{def}}{=} \alpha_{1,2} \circ \gamma 1, 2$$

Figure 3.5 summarizes the different kinds of product defined on domains. Figure 3.5a represents the separate product (Recall Definition 3.4), and is constructed on the product of two concrete lattices. Figure 3.5b represents the simple product: two domains abstract the same concrete lattice. Figure 3.5c represents the reduced product.

Notice that the we can similarly define a closure operator $\rho = \alpha \circ \gamma$ for the separate products. In this case:

$$\rho(x, y) = \begin{cases} (x, y) & \text{if } x \neq \bot \wedge y \neq \bot \\ \bot & \text{if } x = \bot \vee y = \bot \end{cases}$$

Notice that, in the general case, the reduced product must be implemented from scratch, it is not possible to automatically generate an implementation for the reduced product given an implementation of two arbitrary domains. Gulwani and Tiwari [GT06] construct a fourth kind of product : the *logical product*. This product can be, under some hypotheses, constructed automatically.

3.5 Conditional Soundness/Blocking semantics

Another way to combine analyses is conditional soundness introduced by Conway *et al.* [CDNB08]. A program is modeled by a transition system, and we want to check a safety property. The semantics $[\![program]\!]$ of a program is the set of states reachable by this transition system.

The main idea is to introduce a new semantics, called blocking semantics. The transition system is restricted according to a predicate θ: : no transition can be fired from a state that satisfies θ. Hence, a state that satisfies θ may be reachable, but no state is reachable from a state satisfying θ. The θ-blocking semantics $[\![program]\!]_\theta$ of a program is the set of states reachable using the restricted transition system. I.e., $[\![program]\!]_\theta$ are the set of states that are reachable without going through a state that satisfies θ.

Let us give a practical example. Recall Figure 1.3. At line 16, an array overflow may occur. An array overflow may update *any* variable of the program. Consider a variable x

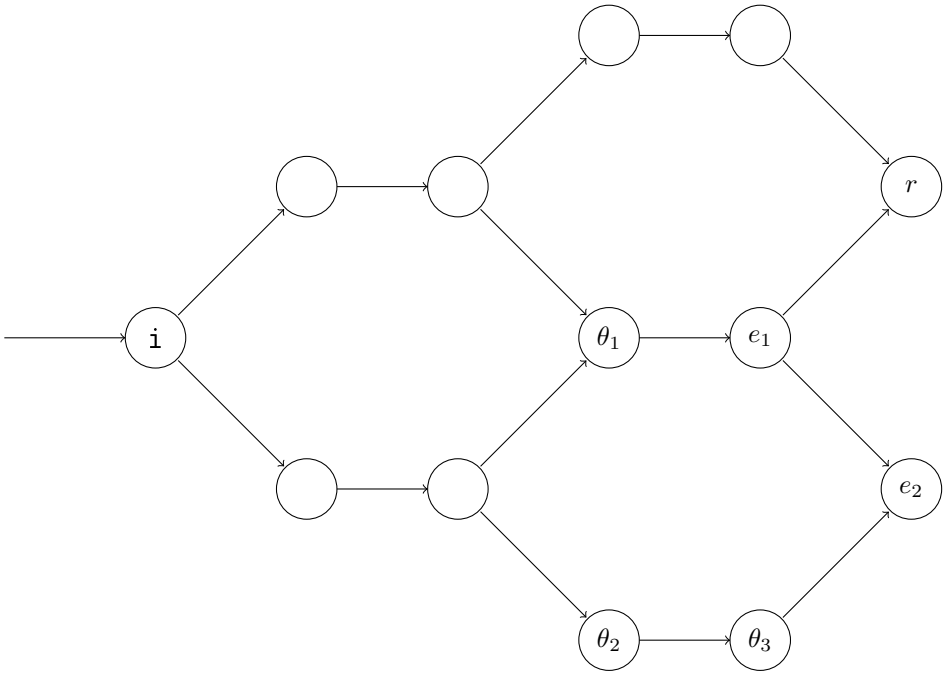


Figure 3.8: Example of Blocking Semantics

that appears somewhere in the program. Hence, after line 16, x may have been updated to 5. Without a blocking semantics, we have to update the abstract value of x and to resume the analysis with this new value.

Here, we can use the predicate $\theta \overset{\text{def}}{\Leftrightarrow}$ "No array overflow occurs". Hence, at line 16, the analysis raises an alarm, notifying that an array overflow may occur. Hence, the analysis assumes that this array overflow has not occurred, and resumes the analysis of this program using this hypothesis.

We give an example in Figure 3.8. Each circle represents a reachable state. The states θ_1, θ_2 and θ_3 are the only states that satisfies θ. Therefore, the θ-blocking semantics excludes the states θ_2, e_1 and e_2 since these states are only reachable through a state that satisfy θ. Notice that the state r is still reachable, since there exists a path from the initial state i to r without any state satisfying θ. The states θ_1 and θ_2 are reachable in the θ-restricted semantics, but not the state θ_3, since θ_3 is reachable only through the state θ_2.

CHAPTER 4

Existing analyses

4.1 Introduction

In this chapter, we dealt with some existing analyses for multithreaded programs. First, in Section 4.2 we recall what are controll flow graphs. They are program representation used by most static analyses, including analyses of multithreaded programs.

In Section 4.3, we recall what are locations sets. These locations sets are used by R. Rugina and M. C. Rinard analysis described in Section 4.4.

In Section 4.5, we dealt with thread modular model checking. Malkis *et al.* show that this model checking is a kind of abstraction.

In Section 4.6 we dealt with multithreaded Gen/Kill analysis. We want to generalize such an analysis (See Section 14.3 and Section 17.3.1.a).

Even if our main interest is array-overflows, invalid pointer dereference and NULL pointer dereference, we recall in Section 4.7 some data-race analysis.

4.2 Control Flow Graph

A program can be represented by a grammar or by a *Control Flow Graph*. A control flow graph is a graph whose edges are labeled by program instructions or by guards. The nodes

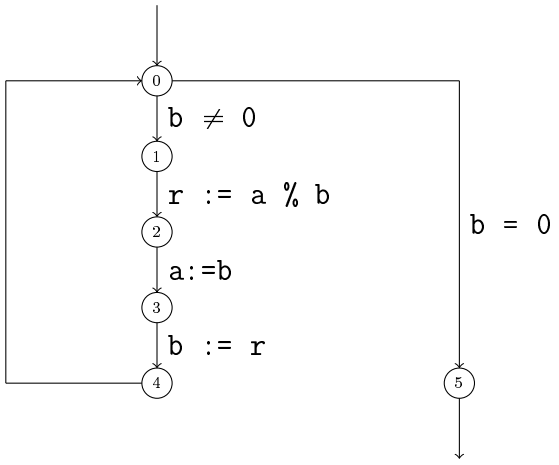


Figure 4.1: Control Flow of Euclides Program

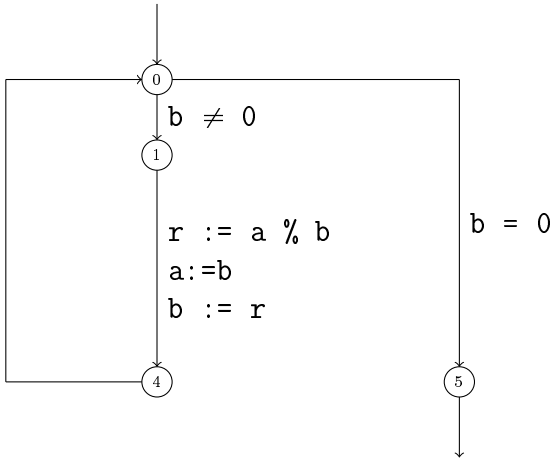


Figure 4.2: Simplified Control Flow of Euclides Program

of the control flow graph are the control points.

For instance, Figure 4.1 gives the control flow of the Euclides algorithm of Figure 3.6. Notice that, if the program is single-threaded, the control flow graph may be simplified (See Figure 4.2), since after going in control point 2, we always go in control point 3 and then to control point 4. In the multithreaded context, this is not true anymore. Actually, when a thread executes the Euclides algorithm, another thread may modify the variables `a` or `b`. For instance, when our thread is at control point 2 the value of `b` may be updated by another thread. Hence, there is a fundamental difference between the two control flow graphs of Figure 3.6 and of Figure 4.2.

4.3 Location Set

In a program, a variable denotes a memory location. The program can access the memory location of the variable, e.g, $x = 3$ assigns value 3 to the memory location of x. An instruction of the program may also access the memory location of a variable plus an offset, e.g, $t[i] = 3$ assigns the value 3 the third slot of the array t. Furthermore, a dynamic memory allocation can create a new memory block, which can be accessed through pointers.

The memory of a C program can be divided into blocks of continuous storage. The relative position of blocks is undefined. Each memory address can be represented by a pair $< name, offset >$. The *name* represents the name of the block, i.e, the name of the variable, or a fresh name for dynamically allocated blocks. Let us call **Locations** the set of memory locations.

According to the C norm [ISO99], array overflows lead to an unspecified result. A pointer that points to the n^{th} slot of an array that has $m < n$ slots is invalid. Therefore, two distinct pairs $< name, offset >$ represent distinct memory addresses (or invalid addresses).

In particular, this means:

- It is impossible to use an array overflow on an array t to write into another memory block
- A pointer to a deallocated memory block will never point to a new allocated memory block
- A pointer to a local variable x will never point to another local variable when this local variable is statically deallocated

On some computers[15], the C program given in Figure 4.3 will answer: $x = 0; y = 5$. Actually, the semantics of this program is undefined, since $v[-1]$ represents an invalid address. To handle this kind of programs, we use a blocking semantics (see Section 3.5). This means that we consider that the program stops with an error when it attempts to do the statement of line 10: $\mathtt{v}[-1] = 5$.

Wilson and Lam [WL95] introduce *location sets* to represent the memory address to which a pointer may legally points.

A *location set* is a tuple $< name, offset, stride >$. The *name* is the name of the memory block, e.g, a variable name, and *offset* and *stride* are integers. A tuple $< name, offset, stride >$ represents all locations $offset + i \times stride$ within the block *name*. Let **LocationSets** be the set of location sets.

A variable v is represented by $< v, 0, 0 >$. The field f in a structure s is represented by $< s, of, 0 >$ where of is the offset of the field f in the structure s. An array t is represented by $< t, 0, size >$ where *size* is the size of an element of the array.

An access to the field f of the element of an array t is represented by $< s, of, 0 >$ where of is the offset of the field f in an element of the array t.

[15] E.g., on my laptop, using `gcc`.

```
1  #include <stdio.h>
2
3  int x=0;
4  int v[3];
5  int y=0;
6
7
8  int main (void)
9  {
10   v[-1]=5;
11   printf("x=%d;y=%d\n",x,y);
12   return 0;
13 }
```

Figure 4.3: Array Overflow

Each dynamically allocated memory site s has a variable name. $< s, 0, i >$ represents *any* array of elements of size i allocated in the site s.

The location sets are some kind of abstraction. We consider that the name of memory block allocated in a site s is $s\#id$ where $\#$ is a separator and id an arbitrary identifier. We also consider a predicate `heap` that, given a name, decides whether it is the name of a dynamically allocated block or not. This allows us to recognize the memory blocks allocated in a given site.

The location sets are then an abstraction of locations. The Galois connection between $\mathcal{P}$(**Locations**) and **LocationSets** is given by:

$$\gamma(< name, offset, stride >) \stackrel{\text{def}}{=} \begin{cases} \{< name, offset + istride > | \ i \in \mathbb{N}\} & \text{if } \neg\texttt{heap}(name) \\ \{< name\#id, offset + istride > | \ i \in \mathbb{N} \wedge \\ \qquad id \text{ an arbitrary identifier}\} & \text{if } \texttt{heap}(name) \end{cases}$$

Notice that a location set may represent one or several memory locations. A location set $< name, offset, stride >$ represents a single location if and only if $name$ is not the name of a site of dynamic allocation, and $stride = 0$. We define the predicate `unique` by:

$$\texttt{unique} < name, offset, stride > \stackrel{\text{def}}{\Leftrightarrow} \neg\texttt{heap}(name) \wedge stride = 0.$$

4.4 R. Rugina and M. C. Rinard Analysis

4.4.1 Points-to Graph

R. Rugina and M. C. Rinard [RR99, RR03] introduce a flow-sensitive and context-sensitive pointer analysis for multithreaded programs. The analysis of R. Rugina and M. C. Rinard uses location sets. They do not check the absence of array overflows or invalid pointer dereferences and assume the programs they analyze are free of these bugs. They add a special location set called `unk` to represent the unknown memory location.

Their algorithm computes a *points-to* graph for each program point. A points-to graph $G \subseteq \textbf{LocationSets} \times \textbf{LocationSets}$ is a set of edges (x, y). An edge (x, y) means that x *may* point to y. Furthermore, x *must* point to some z such that (x, z) is in the points-to graph.

Let $\textbf{Point-to\_Graphs} \stackrel{\text{def}}{=} \textbf{LocationSets} \times \textbf{LocationSets}$ be the set of points-to graphs. Given a points-to graph σ, they introduce the function $\texttt{deref}_\sigma(x)$ that maps x to the set of variables y such that x may point to y:

$$\texttt{deref}_\sigma(x) = \{y \mid (x, y) \in \sigma\}.$$

Notice that, each location x must point to some y such that $y \in \texttt{deref}_\sigma(y)$. The function `deref` naturally extends to sets of variables:

$$\texttt{deref}_\sigma(X) = \bigcup_{x \in X} \texttt{deref}_\sigma(x).$$

Let us give an example: $G = \{(x, \texttt{unk}), (x, y), (y, x)\}$. In this example, y must point to x, but x may point to y or to anywhere.

The set of points-to graphs is a lattice for the inclusion ordering.

4.4.2 Gen/Kill

More formally, given a points-to graph σ, each assignment ***assign*** determines a set $\texttt{gen}_{\text{ptr}}(\textit{assign}, \sigma)$ a set $\texttt{kill}_{\text{ptr}}(\textit{assign}, \sigma)$ and a boolean flag $\texttt{strong}_{\text{ptr}}(\textit{assign}, \sigma)$. Figure 4.4 represents R. Rugina and M. C. Rinard's sets $\texttt{gen}_{\text{ptr}}$ and $\texttt{kill}_{\text{ptr}}$ and the $\texttt{strong}_{\text{ptr}}$ flag [RR99, RR03]. The $\texttt{strong}(\textit{assign}, \sigma)$ flag is true if the assignment ***assign*** assigns a new value to a location set that represents a unique memory address. E.g., in the case $t[i] = \&x$ the boolean flag $\texttt{strong}_{\text{ptr}}$ is *false*, because t is an array, and the location set $< t, 0, \textit{size} >$ represents several memory locations.

4.4.3 Multithreading

The parallel model considered by R. Rugina and M. C. Rinard is based on ***par*** constructor; $\textit{par}\{\textit{stmt}_1 \mid \textit{stmt}_2\}$ executes in parallel the statements $\textit{stmt}_1$ and $\textit{stmt}_2$. Programs are modeled by *parallel flow graphs*. A parallel flow graph is a flow graph generated by a program using the ***par*** constructor. A parallel flow graph has two kinds of vertices:

Cases	Definitions	
$x := y$	$\texttt{gen}_{\text{ptr}}(x := \&y, \sigma)$	$\stackrel{\text{def}}{=} \{x\} \times \texttt{deref}_\sigma(y)$
	$\texttt{kill}_{\text{ptr}}(x := \&y, \sigma)$	$\stackrel{\text{def}}{=} \{x\} \times \texttt{deref}_\sigma(x)$
	$\texttt{strong}_{\text{ptr}}(x := \&y, \sigma)$	$\stackrel{\text{def}}{=} \texttt{unique}(x)$
$x := \&y$	$\texttt{gen}_{\text{ptr}}(x := \&y, \sigma)$	$\stackrel{\text{def}}{=} \{(x, y)\}$
	$\texttt{kill}_{\text{ptr}}(x := \&y, \sigma)$	$\stackrel{\text{def}}{=} \{x\} \times \texttt{deref}_\sigma(x)$
	$\texttt{strong}_{\text{ptr}}(x := \&y, \sigma)$	$\stackrel{\text{def}}{=} \texttt{unique}(x)$
$*x := y$	$\texttt{gen}_{\text{ptr}}(*x := y, \sigma)$	$\stackrel{\text{def}}{=} \texttt{deref}_\sigma(x) \times \{y\}$
	$\texttt{kill}_{\text{ptr}}(*x := y, \sigma)$	$\stackrel{\text{def}}{=} \texttt{deref}_\sigma(x) \times (\texttt{deref}_\sigma(\texttt{deref}_\sigma(x)))$
	$\texttt{strong}_{\text{ptr}}(*x := y, \sigma)$	$\stackrel{\text{def}}{=} \begin{cases} \textit{true} & \text{if } \texttt{deref}_\sigma(x) \text{ is a singleton } \{z\} \\ & \text{such that } \texttt{unique}(z) \\ \textit{false} & \text{else} \end{cases}$
$x = *y$	$\texttt{gen}_{\text{ptr}}(x = *y, \sigma)$	$= \{x\} \times \texttt{deref}_\sigma(\texttt{deref}_\sigma(y))$
	$\texttt{kill}_{\text{ptr}}(x = *y, \sigma)$	$= \{x\} \times \texttt{deref}_\sigma(x)$
	$\texttt{strong}_{\text{ptr}}(x = *y, \sigma)$	$= \texttt{unique}(x)$

Figure 4.4: *Gen* and *kill* sets for Point-to Graphs

- Statement Vertices that represent a pointer assignment : $x = y$, $x = \&y$, $x = *y$ or $*x = y$.
- Parbegin/parend and begin/end vertices that model the ***par*** statement. They come in corresponding pairs. Parbegin/Parend vertices represent the beginning and the end of a ***par*** statement, and begin/end vertices represent the beginning and the end of a thread (created by a ***par*** statement).

All conditions (e.g., for ***if*** and ***while*** statements) are non-deterministic, i.e., the value of the boolean expression tested is disregarded. R. Rugina and M. C. Rinard use a sequentially consistent semantics, i.e, an execution of ***par***$(stmt_1, stmt_2)$ is an interleaving of executions of $stmt_1$ and $stmt_2$.

The ***par*** constructor can handle some kinds of multithreaded programs (e.g., OpenMP programs [Boa08]). Nevertheless, as explained by A. Bouajjani, M. Müller-Olm, and T. Touili [BMOT05], parallel calls cannot adequately model a command that spawns another thread and immediately returns, e.g, in Java [GJSB05] or in C [Boa08]. We will explain in Part III how to handle such commands.

R. Rugina and M. C. Rinard [RR99, RR03] use a semantics that derives tuples containing points-to graphs information about current states, transitions of the current thread, and interferences from other threads. They define the MTI, the ***m**ulti**t**hreaded points-to **i**nformation*.

A *multithreaded points-to information* is a tuple:
$\langle C, I, E \rangle \in$ **Point-to_Graphs** $\times$ **Point-to_Graphs** $\times$ **Point-to_Graphs**.

- C represents the points-to graph at a control point of the program,
- I represents edges that may be created by other threads,
- E represents edges that may be created by the current thread.

The set of MTI is a lattice for the product ordering.

The idea of R. Rugina and M. C. Rinard is to associate to each control point of a program an MTI.

They define the semantics of basic statements as follow:

$$(\!|\mathit{assign}|\!)\langle C, I, E\rangle = \langle C', I, E \cup gen\rangle$$
$$\text{where } C' = \begin{cases} (C \setminus \mathtt{kill}_{\text{ptr}}(\mathit{assign}, C)) \cup gen_{\text{ptr}}(\mathit{assign}, C) & \text{if } \mathtt{strong}_{\text{ptr}}(\mathit{assign}, C)) \\ C \cup gen_{\text{ptr}}(\mathit{assign}, C) & \text{otherwise} \end{cases}$$

They handle the *par* constructor, its semantics is given as a fixpoint of the following equations:

$$\langle C', I', E'\rangle = (\!|\mathit{par}(stmt_1, stmt_2)|\!)\langle C, I, E\rangle$$

where:

$$\begin{aligned} C' &= C'_1 \cap C'_2 \\ E' &= E'_1 \cup E'_2 \\ I' &= I \\ C_1 &= C \cup E_2 \\ C_2 &= C \cup E_1 \\ I_1 &= I \cup E_2 \\ I_2 &= I \cup E_1 \end{aligned}$$

and:

$$\begin{aligned} \langle C'_1, I_1, E'_1\rangle &= (\!|stmt_1|\!)\langle C_1, I_1, \emptyset\rangle \\ \langle C'_2, I_1, E'_2\rangle &= (\!|stmt_1|\!)\langle C_2, I_2, \emptyset\rangle \end{aligned}$$

Notice that the semantics of a statement never changes the I-component. R. Rugina and M. C. Rinard compute the fixpoint on the whole Parallel Flow Graph.

R. Rugina and M. C. Rinard also describes how their work extends to *par* with an arbitrary number of threads, using a $\mathit{parfor}(body)$ constructor that executes *body* in parallel an unbounded number of times.

4.5 Thread-Modular Model-Checking

4.5.1 Model Checking

Flanagan and Qadeer [FQ03] use a model-checking approach to verify multi-threaded programs. Their main idea is to use thread-modular reasoning.

They separate the global and the local part. A Global store contains all variables that are shared between threads. A Local store contains the program counter and all variables specific to a thread. Let *GlobalStore* and *LocalStore* be the sets of global stores and local stores respectively.

The number of threads is fixed at the beginning of the program. The set of thread identifiers is then finite: $\mathbf{Ids} = \{1, \ldots, n\}$.

Each thread needs a local store. Hence, they define *LocalStores*, the set of mappings from **Ids** to *LocalStore*. A state $st \in \mathbf{States}$ is then a pair of $\mathit{GlobalStore} \times \mathit{LocalStores}$.

The behaviors of threads are modeled by a transition relation $T \subseteq \mathbf{Ids} \times (\mathit{GlobalStore} \times \mathit{LocalStore}) \times (\mathit{GlobalStore} \times \mathit{LocalStore})$.

Flanagan and Qadeer explain this relation: "*The relation $T(t, g, l, g, l)$ holds if the thread t can take a step from a state with global store g and where thread t has local store l, yielding a new state with global and local stores g and l , respectively.*"

In particular, this means that any execution of the system is an interleaving of executions of the threads. Furthermore, if a thread updates the shared memory (the global store) then, this update is instantaneously visible for the other threads. Hence, Flanagan and Qadeer strongly rely on sequential consistency.

The naive model-checking approach will explore all states and has space complexity $O(GL^n)$. The objective of Flanagan and Qadeer is to reach a polynomial complexity in n, G and L and no more exponential in n. Nevertheless G and L, are still exponential in the number of variables.

To this aim, Flanagan and Qadeer separate the local and global parts in their analysis. Instead of computing T, they compute two relations:

- $\mathcal{R} \subseteq \mathbf{Ids} \times \mathit{GlobalStore} \times \mathit{LocalStore}$
- $\mathcal{G} \subseteq \mathbf{Ids} \times \mathit{GlobalStore} \times \mathit{GlobalStore}$

The relation $\mathcal{R}(t, g, l)$ holds when the system can reach some state (g, ls) such that $ls(t) = l$. The relation $\mathcal{G}(t, g_1, g_2)$ hold when for some local stores l_1 and l_2 the relation $T(t, (g_1, l_1), (g_2, l_2))$.

The idea of Flanagan and Qadeer is to compute $\mathcal{R}$ and $\mathcal{G}$ instead of all reachable states.

4.5.2 Abstract Interpretation

We easily notice that $\mathcal{R}$ and $\mathcal{G}$ forget information with respect to the semantics of the system. Actually, they are abstractions. Malkis *et al.* [MPR06b, MPR06a] show that Flanagan and Qadeer analysis is a cartesian abstraction. Malkis *et al.* [MPR06b, MPR06a]

```
1  int x = 1;
2  mutex mx;
3
4  void p() {
5    lock(mx);
6    x = 0;
7    x = x + 1;
8    assert x > 0;
9    unlock(mx);
10 }
```

Figure 4.5: Flanagan and Qadeer Example

give the Galois connection in the case of two threads, The concrete lattice is $\mathcal{P}(\mathbf{States}) = \mathcal{P}(\mathit{GlobalStore} \times \mathit{LocalStore}^{\mathbf{Ids}})$, the set of states. And the abstract lattice is $\mathcal{P}(\mathit{GlobalStore} \times \mathit{LocalStore})^{\mathbf{Ids}}$. This explains why the Flanagan and Qadeer algorithm is polynomial in $\mathit{card}(\mathbf{Ids})$ and not exponential in $\mathit{card}(\mathbf{Ids})$. In the case where $\mathbf{Ids} = \{1, 2\}$, the cartesian Galois connection $\alpha_{\text{cart}}, \gamma_{\text{cart}}$ between the concrete lattice $\mathcal{P}(\mathbf{States})$ and the abstract lattice $\mathcal{P}(\mathit{GlobalStore} \times \mathit{LocalStore}) \times \mathcal{P}(\mathit{GlobalStore} \times \mathit{LocalStore})$ is defined by:

$$\begin{array}{rcl} \alpha_{\text{cart}}(S) & = & \big(\{(g, l_1) \mid \exists l_2 : (g, l_1, l_2) \in S\}, \{(g, l_2) \mid \exists l_1 : (g, l_1, l_2) \in S\}\big) \\ \gamma_{\text{cart}}(T_1, T_2) & = & \{(g, l_1, l_2) \mid (g, l_1) \in T_1 \wedge (g, l_2) \in T_2\} \end{array}$$

In the general case, the cartesian Galois connection $\alpha_{\text{cart}}, \gamma_{\text{cart}}$ between the concrete and the abstract lattice is defined by:

$$\begin{array}{rcl} \alpha_{\text{cart}}(S) & = & \lambda i.\{(g, ls(i)) \mid (g, ls) \in S\} \\ \gamma_{\text{cart}}(T) & = & \{(g, ls) \mid \forall i, (g, ls(i)) \in T(i)\} \end{array}$$

Malkis *et al.* show [MPR06b, Proposition 2] that $\alpha_{\text{cart}}, \gamma_{\text{cart}}$ is a Galois connection, and that [MPR06b, Theorem 3] the Flanagan and Qadeer algorithm computes the abstract semantics derived by $\alpha_{\text{cart}}, \gamma_{\text{cart}}$. Furthermore [MPR06b, Theorem 5] prove that the final results of the Flanagan and Qadeer algorithm is exactly the abstraction (without other loss of precision) of the concrete semantics.

Malkis *et al.* improve the precision using human specified [MPR07] "exception sets". They compose their Cartesian abstraction $\alpha_{\text{cart}}, \gamma_{\text{cart}}$ with another Galois connection. This new Galois connection allows them to correlate the local stores of distinct thread, therefore the precision is enhanced. This definetely rests on sequential consistency.

4.5.3 Mutexes

Flanagan and Qadeer [FQ03] give two ways to model mutexes:

```
1  int x = 1;
2  mutex mx;
3  int y = 1;
4  mutex my;
5
6  void p() {
7    if(rnd()){
8      lock(mx);
9      x = 0;
10     x = x + 1;
11     unlock(mx);
12   }
13   else
14   {
15     lock(my);
16     y = 0;
17     y = x + 1;
18     unlock(my);
19   }
20 }
```

Figure 4.6: Modified Flanagan and Qadeer Example

- First, a mutex is a boolean variable : when it is free, its value is true, and when it is locked its value is false. In particular, this means that a thread can unlock a mutex locked by another thread.

- Second, a mutex is associated to the thread that owns it, or the special value **none**. The value of a mutex is **none** whenever it is free.

Since, according to Posix Norm [IT04], the behavior of the program is undefined when a thread attempts to unlock a mutex owned by another thread, these two behaviors are acceptable. They are both in fact observed in practice.

Flanagan and Qadeer give a simple example of program with mutexes: n threads execute the function p of Figure 4.5. In this example, a variable `x` is protected by a mutex `mx`. Therefore, there is no data race.

To prove the absence of data-race on the variable x, Flanagan and Qadeer have to use the second model of mutexes. Therefore, a mutex may have $n+1$ distinct values where n is the number of threads. Hence, adding a mutex to the global store increases G by a factor n. The cost of the analysis of this example, in the number of threads, is then $O(n^2)$. The cost of the analysis of an example with two mutexes (E.g. as given in Figure 4.6) will be $O(n^3)$ and so on. Finally, $G \geqslant n^{card(\mathbf{Locks})}$ where **Locks** is the set of all mutexes.

4.6 Pure Gen/Kill Analyses

Gen/kill analyses are a family of abstractions.

A *pure gen/kill analysis on sets* is parametrized by a lattice $\mathcal{V} = \mathcal{P}(X)$ of subsets of some set X. Each basic statement *stmt* of a program is abstracted using two elements of $\mathcal{V}$ $\texttt{gen}(stmt)$ and $\texttt{kill}(stmt)$. In such an analysis, abstract stores are elements of $\mathcal{V}$ and the effect of the statement *stmt* is abstracted by the function

$$\lambda E.(E \setminus \texttt{kill}(stmt)) \cup \texttt{gen}(stmt).$$

The elements of $\texttt{kill}(stmt)$ are withdrawn, and elements in $\texttt{gen}(stmt)$ are added to the abstract store.

Gen/kill analyzes are generalizable to handle a lattice instead of a set of subsets. A *pure gen/kill analysis* use a lattice $\mathcal{V}$. Each basic statement *stmt* is mapped to two sets: $\texttt{gen}(stmt)$ and $\texttt{keep}(stmt)$. The effect of a the statement *stmt* is abstracted by a function $\lambda E.(E \sqcap \texttt{keep}(stmt)) \sqcup \texttt{gen}(stmt)$. The main difference is the use of a set `keep` instead of `kill`. In the lattice of subsets of X, theses definitions are equivalent according to the following claim (Claim 4.1). More generally, these approaches are equivalent in a complemented lattice. But in a non-complemented lattice, we do not have the operation $\setminus$ needed for `kill`; this is why we use `keep` instead of `kill` .

Claim 4.1.

$$\lambda E.(E \setminus \texttt{kill}(stmt)) \cup \texttt{gen}(stmt) = \lambda E.(E \cap \texttt{keep}(stmt)) \cup \texttt{gen}(stmt)$$

where $\texttt{keep}(stmt)$ *is the complementary of* $\texttt{kill}(stmt)$ *in* X.

Pure Gen/kill analyzes encompass several kinds of analyzes, e.g.:

1. bitvector analysis, e.g, [KSV96],
2. strong copy constant propagation
3. determination of live variables,
4. available expressions
5. and potentially uninitialized variables

For bitvector analysis, we may use the lattice $\{0,1\}^n$ with the pointwise ordering and $0<1$.

The main advantage of pure gen/kill analyses is that `kill` or `keep` and `gen` sets do not depend on the context. Notice that the R. Rugina and M. C. Rinard gen/kill analysis (See Section 4.4.2) is not a pure gen/kill analysis given that $\texttt{gen}_{\text{ptr}}$ and $\texttt{kill}_{\text{ptr}}$ depend not only on the statement, but also of the current abstract store. In Section 8.2.3 we give a general definition of gen/kill analyses that encompasses pure gen/kill analyses and R. Rugina and M. C. Rinard analysis.

H. Seidl and B. Steffen [SS00] use the advantage afforded by pure gen/kill analyses. They use the lattice $\mathbb{F}$ of functions $\mathcal{V} \to \mathcal{V}$ of the form $\lambda E.(E \sqcap \texttt{keep}(stmt)) \sqcup \texttt{gen}(stmt)$. The idea is to abstract the effect of several basic statements by one element of the lattice $\mathbb{F}$, this is possible due to the following claim:

Claim 4.2. *Let $\mathcal{V}$ be a distributive lattice. Each function of $\mathbb{F}$ is monotone and $\mathbb{F}$ is stable by composition*[16]*.*

Proof. Given $f = \lambda x.(x \sqcap a_1) \sqcup b_1$ and $g = \lambda x.(x \sqcap a_2) \sqcup b_2$, $g \circ f = \lambda x.(x \sqcap (a_1 \sqcap a_2)) \sqcup ((b_1 \sqcap a_2) \sqcup b_2)$. □

H. Seidl and B. Steffen [SS00] give an inter-procedural analysis for the primitive *par* and P. Lammich and M. Müller-Olm [LMO08] generalize this approach to the *create* primitive, which spawns a new thread and immediately returns. Programs are represented by a parallel flow graph, like [RR99, RR03], and *if* statements are abstracted as non-deterministic choices. The semantics is an interleaving semantics. They assume that the height[17] of the domain $\mathbb{F}$ is finite, but this is not a true restriction, since widening and narrowing [CC92] (See section 3.3) allow to bypass this limitation.

4.7 Data-races

A Data-race is a run-time error that may occur due to multithreading. Recalling Section 1.1, a data-race occurs when a thread write into a memory location, and another thread accesses (for reading or writing) the same location.

To avoid data-races, several multithreaded libraries [IT04, But06, Bar10, Boa08] give locks/mutexes to the programmer. The two basic and standard functions on mutexes are *lock* and *unlock*. A lock/mutex may be free or owned by a thread. Whenever a thread calls the primitive *lock*(μ), it tries to acquire the mutex μ. If μ is free, then the thread acquires it, else, the thread waits until the mutex becomes free. Whenever a thread calls the function *unlock*(μ), it releases the mutex, i.e., the mutex becomes free. Notice that a thread is allowed to released a mutex only if it owns it, else, the behavior is unspecified. Locks can be used to "protect" a variable. E.g, in Flanagan and Qadeer's example (See Figure 4.5), the variable `x` is protected by the mutex `mx`. A thread may write into `x` if and only if it owns the mutex `mx`. Since the mutex `mx` cannot be owned by two threads at the same time, two distinct threads cannot access to the variable `x` at the same time.

A good programming practice is to use nested locks: locks are released in the same order than they have been acquired. Some languages, like Java [GJSB05] or Visual Basic [Vic07] syntactically enforce the use of nested locks. These languages provide a constructor *sync*(μ)$\{stmt\}$ that executes the statement *stmt* under the protection of the lock μ. In other words *sync*(μ)$\{stmt\}$ locks the mutex μ, executes *stmt* and then releases μ.

[16] I.e., if $f \in \mathbb{F}$ and $g \in \mathbb{F}$ then $g \circ f \in \mathbb{F}$.
[17] Recall Definition 2.24

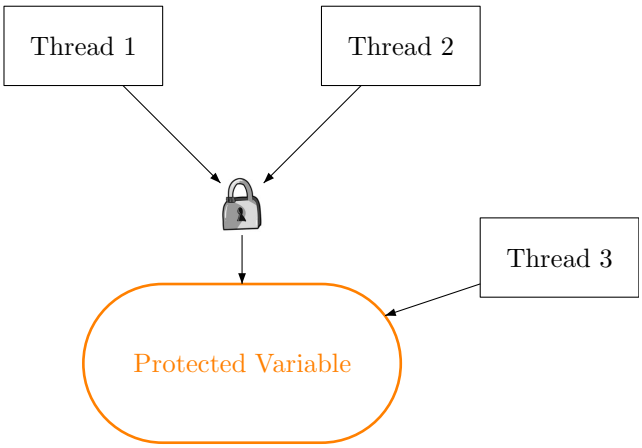


Figure 4.7: Mutexes Protect Variables

4.7.1 Types

The Locksmith tool [PFH06] uses a typing method to prove the absence of data-races.

Their analysis is based on the fact that mutexes are commonly used to protect variables. A variable v is protected by a mutex μ if every thread locks μ before accessing to v. In Figure 4.7, Thread 1 and Thread 2 access the protected variable after locking the mutex. Nevertheless, Thread 3 may access the variable without owning the lock, hence, a data-race may occurs. Hence, we have to check that whenever a thread accesses to a variable, this thread owns the mutex that protect this variable.

The main idea of Locksmith tool is to infer the link between a variable and the mutex that protects it. If the Locksmith tool guesses the right relation, then it propagates it, using type inference. If all variables are protected by a mutex, then the tool is sure that no data-race may occur.

4.7.2 The Goblint Tool

The Goblint tool [VV07] is based on theoretical works done by Seidl *et al.* [VMo03]. Based on abstract interpretation, this tool overapproximates all possible behaviors of the program and it is specialized in detecting data-races. Goblint analyzes each thread in turn, and computes a global fixpoint: it considers that any thread may interfere with any other thread at any time.

To enhance precision, the Goblint tool distinguishes an initialization phase, where only the ***main*** thread is executed, and a second phase, it which all threads may interfere.

Let us consider the program execution represented on Figure 4.8. The program is executed from the top of the figure to the bottom of the figure; moreover, horizontal lines represent thread creation. The Goblint Tool considers the execution of the thread ***main*** alone, and then, it considers that all threads may interfere. For instance, look at the bullet on thread j_2. When j_2 is in the bullet, thread j_6 has not yet been created. But

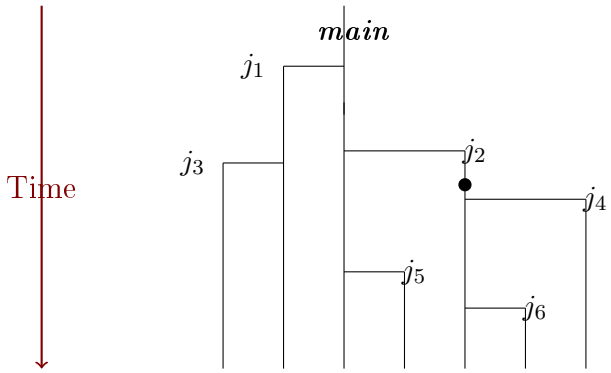


Figure 4.8: A Program Execution

```
1 void f (int b) {
2   sync(μ){ if (b==1) {g()} else h();}
3 }
4
5 void g () { sync(μ){...} }
6
7 void h () {...}
```

Figure 4.9: Reentrant Monitors

the Goblint tool considers that the action of the thread j_2 at the bullet may interfere with thread j_6: this is a safe overapproximation, but this approximation loses precision. Our analysis improves the Goblint method, by introducing an pre-ordering $\leqslant_{after}$ that will tell the analysis that the actions of j_2 done before the creation of j_6 cannot interact with j_6.

Notice that Goblint is one of the rare analysis tool that is able to handle guards. Most other analyses abstract if statements by non-deterministic choices.

4.7.3 Reentrant Monitors

P. Lammich and M. Müller-Olm [LMO08] analyze programs with reentrant monitors. Monitors are locks that are used in a structured way. It corresponds to the use of a primitive $\mathsf{sync}(\mu)\{stmt\}$, as explained at the beginning of this section.

The monitors studied by P. Lammich and M. Müller-Olm are reentrant. This means that the same thread can lock the same monitor several times. E.g., in Figure 4.9, the function `f` calls `g` or `h` depending of the value of its argument. With non-reentrant monitors, there will be a deadlock when `f` call `g`, because the thread that executes `f` still owns the monitor μ. With reentrant monitors, the thread will own the monitor μ a second time.

P. Lammich and M. Müller-Olm model programs b control-flow graphs. They abstract all guards by non-deterministic choices and they ignore information on data. Hence, they need another definition of data-races. The user specifies two sets of nodes of the control-

```
1  void f (int b) {
2    sync(μ1){
3         sync(μ2){ ... };
4         U;
5    }
6  }
7
8  void g (int b) {
9    sync(μ2){
10        sync(μ1){ ... }
11        V;
12    }
13 }
```

Figure 4.10: No Data-Race but a Deadlock

flow graph U and V. A data-race occurs in their model if and only if at the same time a thread reaches a control point in U and a distinct thread reaches a control point in V.

To detects data-races P. Lammich and M. Müller-Olm use acquisition histories introduced by Kahlon *et al.* [KIG05]. An acquisition history is a function from the set of monitors **Locks** to the set $\mathcal{P}(\mathbf{Locks})$. Intuitively, an acquisition history h maps each monitor μ to the set of monitors that will be acquired a time where μ is held.

Two acquisition histories h_1 and h_2 may be interleaved if during an execution, a thread may have the acquisition history h_1 and another an acquisition history h_2. P. Lammich and M. Müller-Olm detect if two distinct threads can reach U and V with interleavable acquisition histories. Formally, they introduce a predicate $h_1 \otimes h_2$ that means that h_1 and h_2 may be interleaved:

$$h_1 \otimes h_2 \overset{\text{def}}{\Leftrightarrow} \nexists \mu_1, \mu_2 : \mu_1 \in h_1(\mu_2) \wedge \mu_2 \in h_2(\mu_1).$$

Acquisition histories allow one to detect some spurious alarms. For instance consider Figure 4.10. Consider that a first thread executes `f` and a second thread executes `g`. The function `f` locks the monitor μ_1, locks the monitor μ_2, releases the monitor μ_2 and then goes to a control state U. The function `f` locks the monitors in the reverse order: first it locks the monitor μ_2, and second, it locks the monitor μ_1 and releases it. After, it goes to a control point in V.

No data-race can occur, instead a deadlock can occur. Nevertheless, tools like Locksmith [PFH06] will detect a possible data-race, since the control points U and V are not protected by the same lock (U is protected by μ_1 and V by μ_2). P. Lammich and M. Müller-Olm's analysis detects that no data race can occur, due to acquisition histories.

CHAPTER 5

Semantics Hierarchy

In this thesis we use several semantics. In Part II, we define a concrete semantics to models the behavior of real programs. In Part III we define two intermediates semantics. These semantics are used in Part IV to prove the soundness of an abstract semantics. This abstract semantics gives an efficient algorithm to check multithreaded programs.

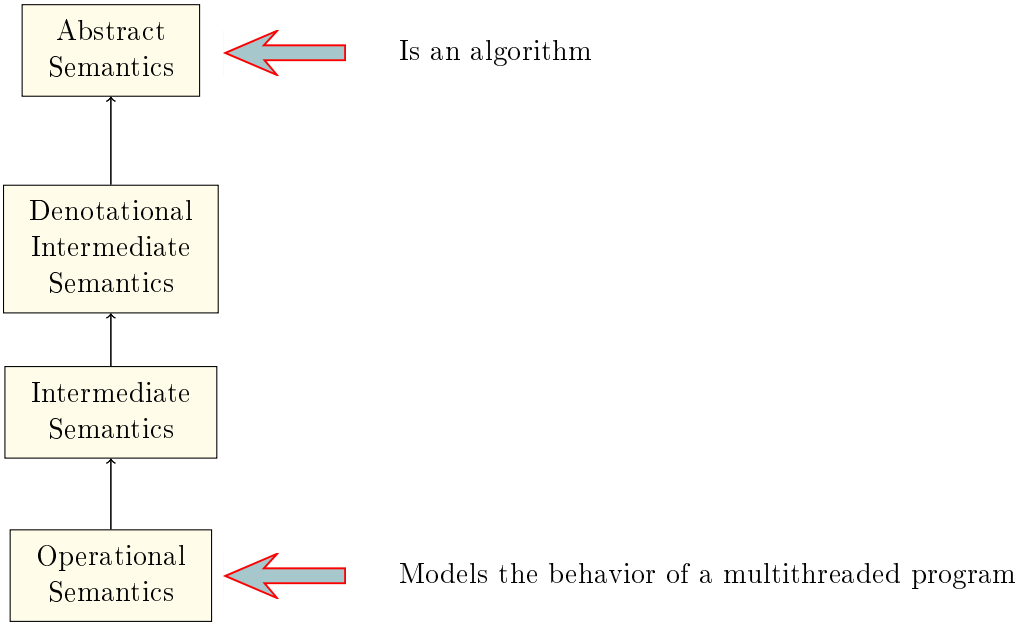


Figure 5.1: Semantics Hierarchy

Part II

Concrete Models

CHAPTER 6

Language

The syntax of our language is given in Fig. 6.1. Statements (*stmt*) are labeled; we denote by **Labels** the set of labels. Labels represent the control flow: the statement ${}^{\ell}stmt, \ell'$ begins at label ℓ and terminates at label ℓ', e.g., in Fig. 6.2a, a thread at label ℓ_2 will execute the assignment $p := \&y$ and go to label ℓ_3. It is assumed that in a given command or statement each label appears only once. Furthermore, to represent the end of the execution, we assume a special label ℓ_∞ which does not appear in a command, but may appear as the return label of a statement. Intuitively, this label represents the termination of a thread: a thread in this label will not be able to execute any statement.

Notice that sequences $cmd_1; cmd_2$ are not labeled. Indeed, the label of a sequence is implicitly the label of the first command, cmd_1.We write ${}^{\ell}cmd$ when the label of cmd is ℓ and we write ${}^{\ell}stmt, \ell'$ the statement $stmt$ labeled by ℓ and ℓ'. A program is represented by a statement of the form ${}^{\ell}cmd, \ell_\infty$. Other statements represent a partial execution of a program.

The *action*s represent store modifications. We call *basic statements* the statements of the form ${}^{\ell}action, \ell'$ or ${}^{\ell_1}guard(cond), \ell_2$ or ${}^{\ell_1}spawn(\ell_3), \ell_2$.

The statements *spawn* and *guard* will be useful in decomposing the steps taken in executing *create*, *while* and *if* statement. To make our presentation simpler, we consider all our variables to be global. The consideration of local variables is an orthogonal concern, and induces no additional complexity. Nevertheless, local variables have been implemented

$stmt$	$::=$		statement
		cmd, ℓ'	command
	$\mid$	${}^{\ell}\mathsf{guard}(cond), \ell'$	guard
	$\mid$	${}^{\ell}\mathsf{spawn}(\ell''), \ell'$	new thread
cmd	$::=$		command
		${}^{\ell}\mathsf{action}$	modify store
	$\mid$	$cmd_1; cmd_2$	sequence
	$\mid$	$\mathsf{if}\,(cond)\,\mathsf{then}\{cmd_1\}\mathsf{else}\{cmd_2\}$	if
	$\mid$	${}^{\ell}\mathsf{while}(cond)\{cmd\}$	while
	$\mid$	${}^{\ell}\mathsf{create}(cmd)$	new thread
action	$::=$		basic action
		$lv := e$	assignment
	$\mid$	$\mathsf{lock}(\mu)$	lock a mutex
	$\mid$	$\mathsf{unlock}(\mu)$	unlock a mutex
lv	$::=$		left value
		x	variable
	$\mid$	${}^{*}e$	pointer deref
e	$::=$		expression
		c	constant
	$\mid$	lv	left value
	$\mid$	$o(e_1, e_2)$	operator
	$\mid$	$\&x$	address
$cond$	$::=$		condition
		x	variable
	$\mid$	$\neg cond$	negation

Figure 6.1: Syntax

(See Section 18.2) as a stack.

Let $Labs({}^{\ell}cmd, \ell_\infty)$ be the set of labels of the statement ${}^{\ell}cmd, \ell_\infty$.

We also define by induction on commands, the set of labels of subthreads $Labs_{child}(\cdot)$ by:

$$
\begin{aligned}
Labs_{child}({}^{\ell_1}\mathsf{create}({}^{\ell_2}cmd), \ell_3) &\stackrel{\text{def}}{=} Labs({}^{\ell_2}cmd, \ell_\infty) \\
Labs_{child}({}^{\ell_1}cmd_1, {}^{\ell_2}cmd_2, \ell_3) &\stackrel{\text{def}}{=} Labs_{child}({}^{\ell_1}cmd_1, \ell_2) \cup Labs_{child}({}^{\ell_2}cmd_2, \ell_3) \\
Labs_{child}\left(\begin{array}{c}{}^{\ell_1}\mathsf{if}\,(cond)\,\mathsf{then}\{{}^{\ell_2}cmd_1\} \\ \mathsf{else}\{{}^{\ell_3}cmd_2\}, \ell_4\end{array}\right) &\stackrel{\text{def}}{=} Labs_{child}({}^{\ell_2}cmd_1, \ell_4) \cup Labs_{child}({}^{\ell_3}cmd_2, \ell_4) \\
Labs_{child}({}^{\ell_1}\mathsf{while}(cond)\{{}^{\ell_2}cmd\}, \ell_3) &\stackrel{\text{def}}{=} Labs_{child}({}^{\ell_2}cmd_1, \ell_1) \\
Labs_{child}({}^{\ell_1}basic, \ell_2) &\stackrel{\text{def}}{=} \emptyset \text{ if } {}^{\ell_1}basic, \ell_2 \text{ is a basic command.}
\end{aligned}
$$

This language contains all primitives that are difficult to analyze. We dealt with some extensions of this language in Chapter 17

$$\ell_1 p := \&x; \ell_2 p := \&y;$$
$$\ell_3 \mathsf{create}(\ell_4 \star p := 2);$$
$$\ell_5 x := 3, \ell_\infty$$

(a) Pointers

$$\ell_6 \mathsf{create}(\ell_7 y := y + z);$$
$$\ell_8 z := 3, \ell_\infty$$

(b) Interference on z

$$\ell_9 x := 0; \ell_{10} y := 0;$$
$$\ell_{11} \mathsf{create}(\ell_{12} x = x + y);$$
$$\ell_{13} y := 3, \ell_\infty$$

(c) $^{\ell_9} example_2, \ell_\infty$

$$\ell_{14} y := 0; \ell_{15} z := 0;$$
$$\ell_{16} \mathsf{create}(\ell_{17} y := 3);$$
$$\ell_{18} y := 1; \ell_{19} z := y, \ell_\infty$$

(d) $^{\ell_{14}} example_4, \ell_\infty$

Figure 6.2: Program Examples

CHAPTER 7

Operational Semantics

7.1 Introduction

An operational semantics describes how a program is executed. An execution of a program is a sequence of transition.

Several operational semantics are given here. They assume a set $\mathcal{V}ar$ of variables and a set $\mathcal{V}$ of values. Some variable are mutexes or locks. We call **Locks** the set of locks and assume that $\mathbf{Locks} \subseteq \mathcal{V}$.

In this chapter, we define a generic operational semantics. In Chapter 8 and Chapter 9, we will instantiate this semantics.

7.2 Description of the System.

To give semantics to threads, we use a set **Ids** of *thread identifiers*. During program execution, each thread is represented by a distinct identifier. We assume a distinguished identifier $\boldsymbol{main} \in \mathbf{Ids}$, and take it to denote the initial thread.

When a program is executed, threads go from a label to another one independently. A *control point* is a partial function P that maps thread identifiers to labels, and such that

$$\frac{lockable(i,\mu,\sigma) \wedge \sigma' = elem_{\mathsf{lock}(\mu)}(i,\sigma)}{{}^{\ell_1}\mathsf{lock}(\mu), \ell_2 \vdash_i (\ell_1,\sigma) \to (\ell_2,\sigma')}\,\text{lock} \qquad \frac{unlockable(i,\mu,\sigma) \wedge \sigma' = elem_{\mathsf{unlock}}(i,\mu)}{{}^{\ell_1}\mathsf{unlock}(\mu), \ell_2 \vdash_i (\ell_1,\sigma) \to (\ell_2,\sigma)}\,\text{unlock}$$

$$\frac{\sigma' = elem_{lv:=e}(i,\sigma)}{{}^{\ell_1} lv := e, \ell_2 \vdash_i (\ell_1,\sigma) \to (\ell_2,\sigma')}\,\text{assign} \qquad \frac{bool(i,\sigma,cond) = true}{{}^{\ell_1}\mathsf{guard}(cond), \ell_2 \vdash_i (\ell_1,\sigma) \to (\ell_2,\sigma)}\,\text{guard}$$

$$\frac{{}^{\ell_1}\mathsf{guard}(cond), \ell_2 \vdash_i t}{{}^{\ell_1}\mathsf{while}(cond)\{{}^{\ell_2} cmd\}, \ell_3 \vdash_i t}\,\text{while entry} \qquad \frac{{}^{\ell_1}\mathsf{guard}(\neg cond), \ell_3 \vdash_i t}{{}^{\ell_1}\mathsf{while}(cond)\{{}^{\ell_2} cmd\}, \ell_3 \vdash_i t}\,\text{while exit}$$

$$\frac{{}^{\ell_1}\mathsf{guard}(cond), \ell_2 \vdash_i t}{{}^{\ell_1}\mathsf{if}(cond)\mathsf{then}\{{}^{\ell_2} cmd_1\}\mathsf{else}\{{}^{\ell_3} cmd_2\}, \ell_4 \vdash_i t}\,\text{then}$$

$$\frac{{}^{\ell_1}\mathsf{guard}(\neg cond), \ell_3 \vdash_i t}{{}^{\ell_1}\mathsf{if}(cond)\mathsf{then}\{{}^{\ell_2} cmd_1\}\mathsf{else}\{{}^{\ell_3} cmd_2\}, \ell_4 \vdash_i t}\,\text{else}$$

Figure 7.1: Local Semantics Rules

$P(\boldsymbol{main})$ is defined. A control point associates each thread with its current label. The domain $Dom(P)$ of P is the set of created threads. Let $\mathbb{P}$ be the set of control points.

Furthermore, threads may create other threads at any time. A *genealogy* of threads is a finite sequence[1] of tuples $(i,\ell,j) \in \mathbf{Ids} \times \mathbf{Labels} \times \mathbf{Ids}$ such that the transitive closure $\prec_g$ of the binary relation $i \leftarrow_g j$ if and only if $(i,\ell,j) \in g$ is a strict ordering and $\boldsymbol{main}$ is a minimal element for this ordering. Intuitively, $i \prec_g j$ means that the thread i is an ancestor of j. We call $\preceq_g$ the reflexive closure of $\prec_g$. A genealogy g is *well formed*, if each thread j is created only once, and a thread j never creates another thread i_1 before having been created. Formally g is *well formed* if for all threads identifiers i_1, i_2, j, for all labels ℓ, ℓ', neither $(i_1,\ell,j)\cdot(i_2,\ell',j)$ nor $(j,\ell,i_1)\cdot(i_2,\ell',j)$ is a subword of g.

We leave the precise semantics of stores undefined for now, and only require four primitives:

- $elem_{\mathsf{action}} : \mathbf{Ids} \times \mathbf{Stores} \to \mathbf{Stores}$,
- $bool : \mathbf{Ids} \times \mathbf{Stores} \to \{true, false\}$,
- $lockable : \mathbf{Ids} \times \mathbf{Locks} \times \mathbf{Stores} \to \{true, false\}$
- and $unlockable : \mathbf{Ids} \times \mathbf{Locks} \times \mathbf{Stores} \to \{true, false\}$.

We also assume a set of initial stores **StoresInit**, e.g., all stores, or a store that maps all variables to 0 as required for global variables by the C norm [ISO99, Section 6.7.9 item 10].

Intuitively, $elem_{\mathsf{action}}(i,\sigma)$ returns the store after i executes the basic operation action (see Fig. 6.1) on the store σ. The function $bool(i,\sigma,cond)$ checks if the condition $cond$ is true in the context σ when the current thread[2] is i. A mutex may be locked or unlocked

[1] I.e., a word, see Chapter 2.

[2] To know which is the current thread will be an important point for weak memory models. See Chapter 9

$$\frac{P(i) = \ell \qquad {}^{\ell_1}stmt, \ell_2 \vdash_i (\ell, \sigma) \to (\ell', \sigma')}{{}^{\ell_1}stmt, \ell_2 \Vdash (i, P, \sigma, g) \to (i, P[i \mapsto \ell'], \sigma', g)}\ \text{parallel}$$

$$\frac{P(i) = \ell_1 \qquad j \text{ is fresh in } (i, P, \sigma, g) \qquad P' = P[i \mapsto \ell_3][j \mapsto \ell_2] \qquad spawnable(i, \sigma)}{{}^{\ell_1}\mathsf{spawn}(\ell_2), \ell_3 \Vdash (i, P, \sigma, g) \to (i, P', \sigma, g \cdot (i, \ell_2, j))}\ \text{spawn}$$

$$\frac{{}^{\ell_2}cmd_1, \ell_4 \Vdash \tau}{{}^{\ell_1}\mathsf{if}\,(cond)\,\mathsf{then}\{{}^{\ell_2}cmd_1\}\mathsf{else}\{{}^{\ell_3}cmd_2\}, \ell_4 \Vdash \tau}\ \text{then body}$$

$$\frac{{}^{\ell_3}cmd_2, \ell_4 \Vdash \tau}{{}^{\ell_1}\mathsf{if}\,(cond)\,\mathsf{then}\{{}^{\ell_2}cmd_1\}\mathsf{else}\{{}^{\ell_3}cmd_2\}, \ell_4 \Vdash \tau}\ \text{else body}$$

$$\frac{{}^{\ell_1}\mathsf{spawn}(\ell_2), \ell_3 \Vdash \tau}{{}^{\ell_1}\mathsf{create}({}^{\ell_2}cmd), \ell_3 \Vdash \tau}\ \text{create} \qquad \frac{{}^{\ell_2}cmd, \ell_1 \Vdash \tau}{{}^{\ell_1}\mathsf{while}(cond)\{{}^{\ell_2}cmd\}, \ell_3 \Vdash \tau}\ \text{while body}$$

$$\frac{{}^{\ell_1}cmd_1, \ell_2 \Vdash \tau}{{}^{\ell_1}cmd_1; {}^{\ell_2}cmd_2, \ell_3 \Vdash \tau}\ \text{sequence 1} \qquad \frac{{}^{\ell_2}cmd_2, \ell_3 \Vdash \tau}{{}^{\ell_1}cmd_1; {}^{\ell_2}cmd_2, \ell_3 \Vdash \tau}\ \text{sequence 2}$$

$$\frac{{}^{\ell_2}cmd, \ell_\infty \Vdash \tau}{{}^{\ell_1}\mathsf{create}({}^{\ell_2}cmd), \ell_3 \Vdash \tau}\ \text{child} \qquad \frac{\tau \in System}{{}^{\ell}stmt, \ell' \Vdash \tau}\ \text{system}$$

Figure 7.2: Global Semantics Rules

only under some assumptions, e.g, a lock may be acquired only it it is free. The predicates *lockable* and *unlockable* model these conditions.

Similarly, a thread cannot necessary spawn another thread at any time. Then we introduce the predicate $spawnable : \mathbf{Ids} \times \mathbf{Stores} \to \{true, false\}$.

A tuple $(i, P, \sigma, g) \in \mathbf{Ids} \times \mathbb{P} \times \mathbf{Stores} \times \mathbf{Genealogies}$ is a *state* if:

(a) $i \in Dom(P)$,

(b) $Dom(P)$ is the disjoint union between $\{\boldsymbol{main}\}$ and the set of threads created in g,

(c) and h is a well formed genealogy.

Let **States** be the set of states. A state is a tuple (i, P, σ, g) where:

- i is the currently running thread,
- P describes where each thread is in the control flow,
- σ is the current store
- and g is the genealogy of thread creations.

$Dom(P)$ is the set of existing threads. The constraint (a) means that the current thread exists, the constraint (b) means that the only threads that exist are the initial threads and the thread created in the past.

Given a program ${}^{\ell_0}cmd, \ell_\infty$ the set *Init* of initial states is the set of tuples $(\boldsymbol{main}, P_0, \sigma, \epsilon)$ where:

- $Dom(P_0) = \{\textbf{\textit{main}}\}$, $P_0(\textbf{\textit{main}}) = \ell_0$,
- σ is an initial store (i.e., $\sigma \in$ **StoresInit**)
- and ϵ is the empty genealogy.

A *transition* is a pair of states $\tau = \big((i, P, \sigma, g), (i', P', \sigma', g \cdot g')\big)$ such that:

(a) for every $j \in Dom(P) \setminus \{i\}, P(j) = P'(j)$

(b) The set of letters of g' is exactely $\{(i, P'(j), j) \mid j \in Dom(P') \setminus Dom(P)\}$

We denote by **Transitions** the set of all transitions. The point (a) means that a transition may change the label of the current thread, but cannot change the label of any other thread. The point (b) means that all new threads are added to the genealogy. Notice that, while we will use a *create* statement that create only one thread, for all transition either $g' = \epsilon$ or $g' = (i, P'(j), j)$ for some $j \in$ **Ids**. When we will use *par* statements (See Section 17.2), we will use transitions that may spawn several threads at the same time.

Notice that, the genealogy increase when transitions are applied. Formally:

Claim 7.1. *Let $s = (i, P, \sigma, g)$ and $s' = (i', P', \sigma', g')$ be two states.*
*If $(s, s') \in$ **Transitions**$^\star \Leftrightarrow g \leqslant_{prefix} g'$.*

7.2.1 Program execution

We use a small step semantics: each statement gives rise to an infinite transition system over states where edges $s_1 \to s_2$ correspond to elementary computation steps from state s_1 to s_2. We define the judgment ${}^{\ell_1}stmt, \ell_2 \Vdash s_1 \to s_2$ to state that $s_1 \to s_2$ is one of these global computation steps that arise when *cmd* is executed.

To simplify the semantic rules, we use an auxiliary judgment ${}^{\ell_1}stmt, \ell_2 \vdash_i (\ell, \sigma) \to (\ell', \sigma')$ to describe evolutions that are local to a given thread i. A *local state* $(\ell, \sigma) \in$ **Labels** $\times$ **Stores** is a pair. The label represents the program pointer of the current thread, and the store σ represents the current store. The judgment ${}^{\ell_1}stmt, \ell_2 \vdash_i (\ell, \sigma) \to (\ell', \sigma')$ means that the thread i can fire a local transition, and go from ℓ to ℓ', modifying the store σ into σ'.

Judgments are derived using the rules of Fig. 7.1 and Fig. 7.2.

The rules "lock" and "unlock" check is the mutex is lockable (respectively unlockable), and update the store if the condition is satisfied. The rule "assign" changes the value of a variable. The rule "guard" allows to fire a transition only if the condition is true.

The rules "while entry" and "while exit" give the guard necessary to enter or to exit the while loop. Rules "then" and "else" respectively give the transitions to enter into the "then" (respectively the "else") branch of the *if* statement.

The rule "parallel" transform a local transition into a global one. The label of the current thread and the store are updated.

In the rule "spawn", the expression "j is fresh in (i, P, σ, g)" means that $i \neq j$ and $P(j)$ is not defined, i.e., thread j does not exists yet. The transitions generated by this rule does

Name	Threads	Control point	Store	Genealogy
s_1	$\underline{\boldsymbol{main}}$	ℓ_1	σ_0	ϵ
s_2	$\underline{\boldsymbol{main}}$	ℓ_2	σ_0	ϵ
s_3	$\underline{\boldsymbol{main}}$ i	ℓ_1 ℓ_3	σ_0	$(\boldsymbol{main}, \ell_2, i)$
s_4	$\boldsymbol{main}$ $\underline{i}$	ℓ_1 ℓ_3	σ_0	$(\boldsymbol{main}, \ell_2, i)$
s_5	$\boldsymbol{main}$ $\underline{i}$	ℓ_1 ℓ_∞	$\sigma_1 \stackrel{\text{def}}{=} elem_{x=1}(\sigma_0)$	$(\boldsymbol{main}, \ell_2, i)$
s_6	$\underline{\boldsymbol{main}}$ i	ℓ_1 ℓ_∞	σ_1	$(\boldsymbol{main}, \ell_2, i)$
s_7	$\underline{\boldsymbol{main}}$ i	ℓ_2 ℓ_∞	σ_1	$(\boldsymbol{main}, \ell_2, i)$
s_8	$\underline{\boldsymbol{main}}$ i j	ℓ_1 ℓ_∞ ℓ_3	σ_1	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
s_9	$\boldsymbol{main}$ i $\underline{j}$	ℓ_1 ℓ_∞ ℓ_3	σ_1	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$

Figure 7.3: Example of Program Execution

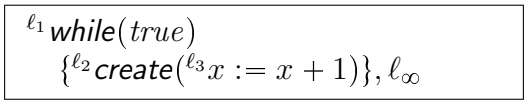


Figure 7.4: Thread Creation in a While Loop

not change the store but creates a new thread and therefore updates the control point and the genealogy.

The rules "then body", "else body", "while body", "sequence 1" and "sequence 2" say that a statement generates all transitions generated by its substatements. The rule "create" say that the statement *create* spawns a thread.

For the rule "system" we define the set of *schedule* transitions by:
$Schedule \stackrel{\text{def}}{=} \{((i, P, \sigma, g), (j, P, \sigma, g)) \mid j \neq i\}$.
Furthermore, we assume a set of transitions *System* such that:

$$\forall((i, P, \sigma, g), (i', P', \sigma', g')) \in System, P = P' \wedge g = g' \text{ and } Schedule \subseteq System.$$

The set *System* contains all transitions common to all programs, e.g., transitions that switch the current thread.

The set of transitions generated by statement $^{\ell}stmt, \ell'$ is $\mathcal{T}r_{^{\ell}stmt,\ell'} = \{(s, s') \mid {}^{\ell}stmt, \ell' \Vdash s \rightarrow s'\}$. Furthermore, let $\mathcal{T}r^{-}_{^{\ell}stmt,\ell'} = \mathcal{T}r_{^{\ell}stmt,\ell'} \setminus System$ be the set of transitions *specific* to the statement $^{\ell}stmt, \ell'$.

Figure 7.3 gives the beginning of one possible execution of the program of Fig. 7.4. The first colum gives the name of the states. The second column indicates created threads, the current thread is underlined. The third column gives the label of each thread, i.e., the control point P. The fourth column gives the store and the last column gives the genealogy.

Hence, in Figure 7.3:

- $s_0 = (\boldsymbol{main}, P_1, \sigma_0, \epsilon)$ where $P_1(\boldsymbol{main}) = \ell_1$
- $s_9 = (j, P_9, \sigma_1, (\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j))$ where $P_9(\boldsymbol{main}) = \ell_1$, $P_9(i) = \ell_\infty$ and $P_9(j) = \ell_3$

The store σ_0 is assumed to be an initial store, i.e., $\sigma_0 \in \textbf{StoresInit}$. In that figure, (s_1, s_2), (s_5, s_6) and (s_8, s_9) are in *System*, but $(s_1, s_2) \notin System$.

7.3 Descendants

Figure 7.6 illustrates the execution of a whole program. Each vertical line represents the execution of a thread from top to bottom, and each horizontal line represents the creation of a thread. At the beginning (top of the figure), there is only the thread $\boldsymbol{main} = j_0$.

During execution, each thread may execute transitions. At state s_0, $thread(s_0)$ denotes the *currently running thread* (or *current thread*), see Fig. 7.5. On Fig. 7.6, the current thread of s_0 is j_0 and the current thread of s is j_2.

For any set of states S, let $\overline{S} = \mathbf{States} \setminus S$ be the *complement* of S.
$thread(i, P, \sigma, g) \stackrel{\text{def}}{=} i$
$label(i, P, \sigma, g) \stackrel{\text{def}}{=} P(i)$
$desc_g(i) = \{j \mid i \preceq_g j\}$
$desc_g(X) = \bigcup_{i \in X} desc_g(i)$

Figure 7.5: Auxiliary definitions

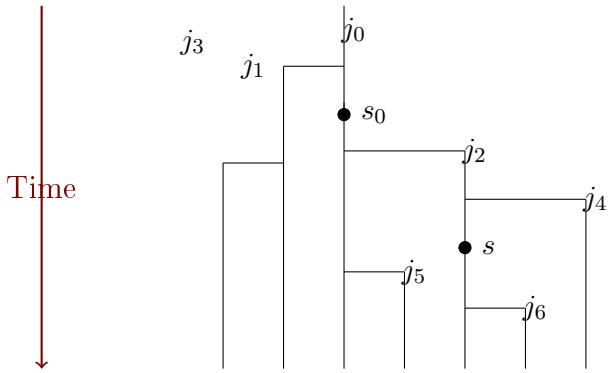


Figure 7.6: A thread Execution

During the program execution given in Fig. 7.6, j_0 creates j_1. We say that j_1 is a *child* of j_0 and j_0 is the *parent* of j_1. Furthermore, j_1 creates j_3. We then introduce the concept of *descendant*: the thread j_3 is a descendant of j_0 because it has been created by j_1 which has been created by j_0. More precisely, descendants depend on genealogies. Consider the state $s_0 = (j_0, P_0, \sigma_0, g_0)$ with $g_0 = [(j_0, \ell_1, j_1)]$: the set of descendants of j_0 from g_0 (written $desc_{g_0}(\{j_0\})$, see Fig. 7.5) is just $\{j_0, j_1\}$. The set of descendants of a given thread increases during the execution of the program. In Fig. 7.6, the genealogy of s is of the form $g_0 \cdot g$ for some g, here $g = [(j_0, \ell_2, j_2), (j_1, \ell_3, j_3), (j_2, \ell_4, j_4)]$. When the execution of the program reaches the state s, the set of descendants of j_0 from $g_0 \cdot g$ is $desc_{g_0 \cdot g}(j_0) = \{j_0, j_1, j_2, j_3, j_4\}$.

In a genealogy, there are two important pieces of information. First, there is a tree structure: a thread creates children that may create children and so on... Second, there is a global time, e.g., in g, the thread j_2 has been created before the thread j_3.

Lemma 7.2. *If $g \cdot g'$ is a well formed genealogy therefore $desc_{g \cdot g'}(X) = desc_{g'}(desc_g(X))$.*

Proof. Let $j \in desc_{g \cdot g'}$. Therefore, by definition of $\prec_g$, there exists $i_0, \ldots, i_n$ such that:

- For all $k \in \{0, \ldots, n-1\}$, $i_k \leftarrow_{g \cdot g'} i_{k+1}$
- and $i_n = j$.

Notice that, by definition, $i_k \leftarrow_{g \cdot g'} i_{k+1}$ is equivalent to $i_k \leftarrow_g i_{k+1}$ or $i_k \leftarrow_{g'} i_{k+1}$.

- First case: for all k, $i_k \leftarrow_g i_{k+1}$. Therefore $j \in desc_g(X) \subseteq desc_{g'}(desc_g(X))$.

- Second case: there exists k such that $i_k \leftarrow_g i_{k+1}$. Let k_0 the smallest such k. By definition, there exists ℓ_0 such that $(i_{k_0}, \ell_0, i_{k_0+1}) \in g'$.

  By minimality of k_0, $i_{k_0} \in desc_g(X)$.

  Let $k > k_0$. Assume by contradiction that $i_k \leftarrow_g i_{k+1}$. Therefore, there exists ℓ such that $(i_{k_0}, \ell_0, i_{k_0+1}) \in g$. Hence $(i_{k_0}, \ell_0, i_{k_0+1}) \cdot (i_{k_0}, \ell_0, i_{k_0+1})$ is a subword of $g \cdot g'$ and therefore $g \cdot g'$ is not weel formed. Hence for all $k > k_0$, $i_k \leftarrow_{g'} i_{k+1}$.

  We conclude that $j \in desc_{g'}(i_{k_0}) \subseteq desc_{g'}(desc_g(X))$

□

During the execution of a program, each thread may only be created once:

Lemma 7.3 (Unique Parent)**.** *Let g a well formed genealogy.*
If $i_1 \leftarrow_g j$ and $i_2 \leftarrow_g j$ then $i_1 = i_2$.

Proof. Because $i_1 \leftarrow_g j$, there exists ℓ such that (i_1, ℓ, j). Because $i_2 \leftarrow_g j$, there exists ℓ such that (i_2, ℓ, j).

The genealogy g is well formed. Hence, neither $(i_1, \ell, j) \cdot (i_2, \ell', j)$ nor $(i_2, \ell', j) \cdot (i_1, \ell, j)$ is a subword of g.

We conclude that$(i_1, \ell, j) = (i_2, \ell', j)$ and then $i_1 = i_2$. □

Lemma 7.4. *Let $g \cdot g'$ a well formed genealogy and i, j which are not created in g'. Therefore, either $desc_{g'}(j) \subseteq desc_{g \cdot g'}(i)$ or $desc_{g'}(j) \cap desc_{g \cdot g'}(i) = \emptyset$.*

Proof. We consider the case where $desc_{g'}(j) \cap desc_{g \cdot g'}(i) \neq \emptyset$. Let $i' \in desc_{g'}(j) \cap desc_{g \cdot g'}(i)$.

Therefore, there exists to sequences of threads identifiers $i_1, \ldots, i_n$ and $j_1, \ldots, j_m$ such that

- $i_n = i$
- For all $k \in \{0, \ldots, n-1\}$, $i_{k+1} \leftarrow_{g \cdot g'} i_k$
- $i_1 = i'$
- $j_1 = j$
- For all $k \in \{0, \ldots, m-1\}$, $j_{k+1} \leftarrow_{g'} j_k$
- $j_1 = i'$

Given that g' is a subword of $g \cdot g'$, we conclude that for all $k \in \{0, \ldots, m-1\}$, $j_k \leftarrow_{g \cdot g'} j_{k+1}$. We apply by induction the Lemma 7.3 and state that for all $k \in \{1, \ldots, \min(n, m)\}$, $i_k = j_k$.

- First case: $n < m$. Therefore $j_{n+1} \leftarrow_{g'} i$. This is in contradiction with the fact that i have not been created in g'.

- Second case: $m > n$. Therefore $i_{m+1} \leftarrow_{g \cdot g'} j$. Because j have not been created in g', therefore $i_{m+1} \leftarrow_g j$.

  Assume by contradiction that, for some $k > n$, $i_{k+1} \leftarrow_{g'} i_k$. Let k_0 the smallest such k. Therefore, there exists ℓ' such that $(i_{k+1}, \ell', i_k) \in g'$ and there exists ℓ'' such that $(i_k, \ell', i_{k-1}) \in g$. Hence $(i_k, \ell'', i_{k--}) \cdot (i_{k+1}, \ell', i_k)$ is a subword of $g \cdot g'$; this is impossible because $g \cdot g'$ is well formed.

  Therefore, for every $k > n$, $i_{k+1} \leftarrow_g i_k$. Hence $j \in desc_g(i)$. Therefore $desc_{g'}(j) \subseteq desc_{g'}(desc_g(i))$.

- Third case: $n = m$. Therefore $i = j$ and $desc_{g'}(j) = desc_{g'}(i) \subseteq desc(desc_{g'}(i))$ by Lemma 7.2.

□

We also need to consider sub-genealogies such as g. In this partial genealogy, j_1 has not been created by j_0. Hence $desc_g(\{j_0\}) = \{j_0, j_2, j_4\}$. Notice that $j_3 \notin desc_g(\{j_0\})$ even though the creation of j_3 is in the genealogy g.

We say that a set of transitions T is *conservative* if and only if for all transitions: $((i, P, \sigma, g), (i', P', \sigma', g')) \in T, g = g'$. The following lemma exhibits some conservative sets:

Lemma 7.5. *The following sets of transitions are conservative:*

- *System*
- *Schedule*
- $\mathcal{T}r_{\ell_1 basic, \ell_2}$ *where* ${}^{\ell_1}basic, \ell_2$ *is an arbitrary basic statement.*

7.4 Properties of the language

In this section, we give some useful properties on the transitions generated by the statements of our language.

7.4.1 Labels

A transition generated by a basic statement go from the initial label of the statement to the final label. This is not true for non-basic statements (e.g., composition). Formally:

Lemma 7.6. *Let* ${}^{\ell_1}basic, \ell_2$ *be a basic statement.*
If $(s, s') = ((i, P, \sigma, g), (i', P', \sigma', g')) \in \mathcal{T}r^{-}_{\ell_1 basic, \ell_2}$ *then*

1. $label(s) = \ell_1$

2. $label(s') = \ell_2$
3. $thread(s) = thread(s')$
4. *s and s' has the same genealogy.*

Proof. This lemma is a consequence of rules of Fig. 7.1 and rule "parallel" of Fig. 7.2. □

A statement generates only transitions from its labels and to its labels, e.g., the statement of Figure 6.2a generates transitions from the label ℓ_2, this is formalized by the following lemma:

Lemma 7.7. *If $(s, s') \in \mathcal{T}r^{-}_{{}^{\ell}stmt,\ell'}$ then:*

1. $label(s) \in Labs({}^{\ell}stmt, \ell') \setminus \{\ell'\}$
2. $label(s') \in Labs({}^{\ell}stmt, \ell') \cup \{\ell_\infty\}$
3. $thread(s) = thread(s')$

Proof. This lemma is true for basic statement according to Lemma 7.6. We conclude by induction. □

The contrapositive gives the following lemma:

Lemma 7.8. *If $label(s) \notin Labs({}^{\ell}stmt, \ell') \setminus \{\ell'\}$ then for all state s', $(s, s') \notin \mathcal{T}r^{-}_{{}^{\ell}stmt,\ell'}$*

Whenever a statement ${}^{\ell}stmt, \ell'$ generates a transition that creates a new thread j, this new thread j is in a label of $Labs_{child}({}^{\ell}stmt, \ell')$. Formally:

Lemma 7.9. *If $(s, s') = ((i, P, \sigma, g), (i', P', \sigma', g \cdot g')) \in \mathcal{T}r_{{}^{\ell}stmt,\ell'}$ and $i \prec_{g'} j$ then $P'(j) \in Labs_{child}({}^{\ell}stmt, \ell') \subseteq Labs({}^{\ell}stmt, \ell')$.*

Lemma 7.10. *If $(s, s') \in \mathcal{T}r^{-}_{{}^{\ell}stmt,\ell'}$ and $label(s) \in Labs_{child}({}^{\ell}stmt, \ell')$ then $label(s') \in Labs_{child}({}^{\ell}stmt, \ell')$.*

Furthermore $\ell \notin Labs_{child}({}^{\ell}stmt, \ell')$ and $\ell' \notin Labs_{child}({}^{\ell}stmt, \ell')$.

7.5 Conclusion

We define an operational semantics, assuming only the following sets and functions:

- **Stores**
- **StoresInit**
- $elem_{\mathsf{action}} : \mathbf{Ids} \times \mathbf{Stores} \to \mathbf{Stores}$

- $bool(\sigma, cond) : \mathbf{Stores} \times \mathbf{Conditions} \rightarrow \{true, false\}$
  where **Conditions** is the set of conditions generated by the rules of Figure 6.1.
- $lockable : \mathbf{Ids} \times \mathbf{Locks} \times \mathbf{Stores} \rightarrow \{true, false\}$
- $unlockable : \mathbf{Ids} \times \mathbf{Locks} \times \mathbf{Stores} \rightarrow \{true, false\}$
- $spawnable : \mathbf{Ids} \times \mathbf{Stores} \rightarrow \{true, false\}$
- *System*

Hence, we can instantiate an operational semantics, giving only these sets and functions. In Chapter 8 and Chapter 9 we give three different instantiations.

CHAPTER 8

Interleaving Semantics

In this semantics, threads execute their code with respect to sequential consistency. The principle has been summarized by Lamport [Lam79]: "*... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"

A large number of multithread analyses uses sequential consistency [LMO07, MPR07, FQ03].

In this semantics *System* = *Schedule* = $\{\tau \in \textbf{Transitions} \mid \Vdash_{\text{SC}} \tau\}$ (See Figure 8.2). Sequential consistency is used with several kinds of store. In this chapter, we describe two kinds of stores : Maps and Gen/Kill stores.

8.1 Maps

Concrete stores are maps from the set of variables $\mathcal{V}ar$ to the set $\mathcal{V}$ of *concrete values*.

StoresInit = **Stores** or **StoresInit** = $\lambda x.0$.

To define $elem_{lv:=e}$ we need to evaluate a left value and an expression. We assume a function $\texttt{addr}_\sigma(lv)$ that, given a left value, returns the name of the corresponding variable. E.g, $\texttt{addr}_\sigma(x) = x$, $\texttt{addr}_\sigma(*x) = y$ if $\sigma(x) = \&y$. We also assume the classical function

Name	Threads	Control point	Store	Genealogy
s_1	$\underline{\boldsymbol{main}}$	ℓ_1	$x = 0$	ϵ
s_2	$\underline{\boldsymbol{main}}$	ℓ_2	$x = 0$	ϵ
s_3	$\underline{\boldsymbol{main}}$ i	ℓ_1 ℓ_3	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s_4	$\boldsymbol{main}$ $\underline{i}$	ℓ_1 ℓ_3	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s_5	$\boldsymbol{main}$ $\underline{i}$	ℓ_1 ℓ_∞	$x = 1$	$(\boldsymbol{main}, \ell_2, i)$
s_6	$\underline{\boldsymbol{main}}$ i	ℓ_1 ℓ_∞	$x = 1$	$(\boldsymbol{main}, \ell_2, i)$
s_7	$\underline{\boldsymbol{main}}$ i	ℓ_2 ℓ_∞	$x = 1$	$(\boldsymbol{main}, \ell_2, i)$
s_8	$\underline{\boldsymbol{main}}$ i j	ℓ_1 ℓ_∞ ℓ_3	$x = 1$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
s_9	$\boldsymbol{main}$ i $\underline{j}$	ℓ_1 ℓ_∞ ℓ_3	$x = 1$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$

Figure 8.1: Interleaving Semantics Example

$$\frac{P(j) \text{ is defined} \qquad i \neq j}{\Vdash_{\mathrm{SC}} (i, P, \sigma, g) \to (j, P, \sigma, g)} \text{schedule}$$

Figure 8.2: System Transitions for Interleaving Semantics

$\texttt{val}_\sigma(e)$ that gives the value of an expression. E.g., $\texttt{val}_\sigma(2) = 2$, $\texttt{val}_\sigma(x) = \sigma(x)$, $\texttt{val}_\sigma(x + y) = \sigma(x) + \sigma(y)$,... Finally:

$$elem_{lv:=e} = \sigma[\texttt{addr}_\sigma(lv) \mapsto \texttt{val}_\sigma(e)].$$

The boolean evaluation is defined as follow:

$$bool(\sigma, x) = \begin{cases} true & \text{if } \sigma(x) \neq 0 \\ false & \text{if } \sigma(x) = 0 \end{cases}$$

The value of the mutex variable μ is the identifier of the thread that owns it, or the special symbol **none**. The special symbol **none** means that the mutex is free. Formally:

$$\begin{aligned} elem_{\mathsf{lock}(\mu)}(i, \sigma)) &= \sigma[\mu \mapsto i] \\ elem_{\mathsf{unlock}(\mu)}(i, \sigma) &= \sigma[\mu \mapsto \textbf{none}] \\ lockable(i, \mu, \sigma) &= \begin{cases} true & \text{if } \sigma(\mu) = \textbf{none} \\ false & \text{if } \sigma(\mu) \neq \textbf{none} \end{cases} \\ unlockable(i, \mu, \sigma) &= \begin{cases} true & \text{if } \sigma(\mu) = i \\ false & \text{if } \sigma(\mu) \neq i \end{cases} \end{aligned}$$

Threads can spawn another thread at any time, hence $spawnable(i, \sigma) = true$ for all i and σ.

The Figure 8.1 gives an example of the execution of program of Figure 7.4. This is the same example than Figure 7.3, instanced in the case of an interleaving semantics.

8.2 Gen/Kill

In Section 4.4.2 and Section 4.6, we have discussed Gen/Kill analyses. Here, we adapt Gen/Kill analyses to our concrete model.

8.2.1 Pure Gen/Kill

In such analyses [SS00, LMO07], stores are values in a lattice $\mathcal{V}$, e.g., $\mathcal{V}$ is a set of uninitialized variable, i.e., $\textbf{Stores} = \mathcal{V}$.

Each gen/kill analysis gives, for each action, two sets:

- gen(*action*)
- kill(*action*) (if the lattice $\mathcal{V}$ is a complemented lattice) or keep(*action*) (if $\mathcal{V}$ is not a complemented lattice).

The function *elem* is defined by:

$$elem_{\mathit{action}}(\sigma) = (\sigma \setminus \texttt{kill}(\mathit{action})) \sqcup \texttt{gen}(\mathit{action}).$$

or by:

$$elem_{\mathit{action}}(\sigma) = (\sigma \sqcap \texttt{keep}(\mathit{action})) \sqcup \texttt{gen}(\mathit{action}).$$

8.2.2 Points-to Graph

Rugina and Rinard [RR99, RR03] present a pointer analysis for parallel programs. The concrete stores $s \in \mathbf{Stores}$ are points-to graphs (See Section 4.4.1).

The definitions of functions $elem_{lv:=e}$ is implicitly given in Fig. 3 and Fig. 4 of their paper [RR99]. More formally, given a concrete store σ each assignment $lv := e$ determines a set $\texttt{gen}_{\text{ptr}}(lv := e, \sigma)$ a set $\texttt{kill}_{\text{ptr}}(lv := e, \sigma)$ and a boolean flag $\texttt{strong}(lv := e, \sigma)$. Figure 4.4 represents Rugina and Rinard's sets $\texttt{gen}_{\text{ptr}}$ and $\texttt{kill}_{\text{ptr}}$ [RR99, RR03].

Given these sets and this flag, the primitive $elem_{lv:=e}$ is defined by:

$$elem_{lv:=e}(X) = \begin{cases} (\sigma \setminus \texttt{kill}_{\text{ptr}}(lv := e, \sigma) \cup \texttt{gen}_{\text{ptr}}(lv := e, \sigma) & \text{if } \texttt{strong}(lv := e, \sigma) \\ \sigma \cup \texttt{gen}_{\text{ptr}}(lv := e, \sigma) & \text{if not } \texttt{strong}(lv := e, \sigma) \end{cases}$$

8.2.3 General Gen/Kill Analysis

As for Section 8.2.1, $\mathcal{V}$ is a lattice and $\mathbf{Stores} = \mathcal{V}$ and each gen/kill analysis gives, for each action and for each store σ, two sets: $\texttt{gen}(\mathit{action}, \sigma)$ and $\texttt{keep}(\mathit{action}, \sigma)$. We assume that `gen` and `keep` are monotonic[3] in σ.

The main difference with Section 8.2.1 is that *gen* and *kill* sets may depend on the current store (e.g, strong flag of Section 8.2.2).

$$elem_{\mathit{action}}(\sigma) = (\sigma \setminus \texttt{kill}(\mathit{action}, \sigma)) \cup \texttt{gen}(\mathit{action}, \sigma)$$

The analysis of Rugina and Rinart is a particular case of Gen/Kill analysis where:

$$\texttt{kill}(lv := e, \sigma) = \begin{cases} \texttt{kill}_{\text{ptr}}(lv := e, \sigma) & \text{if } \texttt{strong}(lv := e, \sigma) \\ \emptyset & \text{otherwise} \end{cases}$$

[3] An analysis that use the set $\texttt{kill}(\mathit{action}, \sigma)$ need that `kill` is decreasing in σ.

CHAPTER 9

Weak Memory Model

9.1 Introduction

There exists several kinds of weak memory models. In a weak memory model, each thread as its own view of the memory, but two distinct threads may have two distinct views at the same time. As explained in introduction, weak memory models are used in practice, to allows compilers for optimisations and to enhance processor speed.

Nevertheless, M. F. Atig and A. Bouajjani [ABBM10] recall that, in most languages, for data-race free programs, there is no difference between strong and weak memory models: the execution of a data-race free program p in a weak memory model is always equivalent to sequentially consistent execution of p.

This is not true in all languages, e.g, the language C# [ISO06, Section 17.4.3] allows programmers to access simultaneously to several "volatile" variables, and the semantics of this accesses is a weak memory model.

Moreover, in practice, due to human errors or in the name of efficiency, a large number of programs are not data-race free. Microsoft guidelines for .NET [Mic10] advise to keep some data-races : "*Sometimes the algorithm can be adjusted to tolerate race conditions rather than eliminate them.*"

Here, we focus on two weak memory models : TSO and PSO.

9.2 TSO

TSO is a suitable model of the behavior of modern Intel processors [OSS09]. TSO is the "write to read" relaxation, i.e., when reading a value from memory, a thread may pretend to ignore some past writes from other threads. An adequate semantics for TSO is Atig *et al.*'s *operational model* [ABBM10].

Threads share a memory, but do not write instantaneously in it. Each thread has a write buffer. Instead of writing into a variable, a thread writes into its write buffer, modifying its own view of the memory, but leaving the shared memory untouched. At any time, some of the writes may be dequeued from the write buffer and the shared memory is updated accordingly.

We assume a set **Memories** of *memories* and a set **WriteOp** of *write operation* and a function *update-memory* : $\mathbf{WriteOp} \times \mathbf{Memories} \to \mathbf{Memories}$ that updates a memory according to a write operation.

A *write buffer* w is a FIFO queue of write operations. The set of buffer is define as follow: $\mathbf{Buffers} \stackrel{\text{def}}{=} \mathbf{FIFO}_{\mathbf{WriteOp}}$.

A *store* is a pair (m, b) where $m \in \mathbf{Memories}$ is a memory and $b : \mathbf{Ids} \to \mathbf{Buffers}$ is a map from threads identifiers to buffers. Let *memory-action* the partial function that updates the shared memory with a write buffer. Given a store $\sigma = (m, b)$ and a thread i such that $b(i)$ is not an empty FIFO, $\mathit{memory\text{-}action}(i, \sigma)$ extracts the first write operation (x, v) of the buffer $b(i)$ and applies it to the memory, formally:

$$\begin{array}{l}\mathit{memory\text{-}action}(i, (m, b)) = (\mathit{update\text{-}memory}(op, m), b[i \mapsto w]) \\ \text{where } op = \mathit{fst}(b(i)) \text{ and } w = \mathit{deq}(b(i)).\end{array}$$

Whenever a process reads a variable x it does as though it reads the memory after all pending updates have been effected by its buffer. Given a store $\sigma = (m, b)$ the *view* of the thread i is m modified by $b(i)$, the write buffer of i. This view is written $\mathit{view}(i, \sigma)$. Notice that $\mathit{view}(i, \sigma) \in \mathbf{Memories}$.

Formally, given a store $\sigma = (m, b)$, $\mathit{view}(i, \sigma)$ is defined by induction:

$$\mathit{view}(i, \sigma) = \begin{cases} m & \text{if } b(i) = \epsilon, \text{ i.e., } b(i) \text{ is an empty FIFO} \\ \mathit{view}(i, \mathit{memory\text{-}action}(i, \sigma)) & \text{otherwise.} \end{cases}$$

Expressions e are always evaluated in such a view. We leave the formal definition of evaluation as an exercise. We Shall only need it through two primitives $\mathit{elem}_{lv:=e}(i, m)$, $\mathit{bool}(m, \mathit{cond})$.

Given a thread i and a view m, $\mathit{elem}_{lv:=e}(i, m)$ evaluates lv and e in the view m and returns the corresponding write operation. E.g, $\mathit{elem}_{x=3}$ returns the write operation $(x, 3)$ that puts the value 3 in x. The function $\mathit{bool}(m, \mathit{cond})$ evaluates the condition *cond* in the view m, returning *true* or *false*.

We define the function $\mathit{elem}_{\mathit{action}}(i, \sigma)$, as required for our semantics, See Section 7.5. When a thread i makes an assignment $lv := e$ on a store $\sigma = (m, b)$, the thread evaluates

Name	Threads	Control point	Buffers	Memory	Genealogy
s_1	$\underline{\boldsymbol{main}}$	ℓ_1	$\emptyset$	$x = 0$	ϵ
s_2	$\underline{\boldsymbol{main}}$	ℓ_2	$\emptyset$	$x = 0$	ϵ
s_3	$\underline{\boldsymbol{main}}$ i	ℓ_1 ℓ_3	$\emptyset$ $\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s_4	$\boldsymbol{main}$ $\underline{i}$	ℓ_1 ℓ_3	$\emptyset$ $\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s_5	$\boldsymbol{main}$ $\underline{i}$	ℓ_1 ℓ_∞	$\emptyset$ $(x, 1)$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s_6	$\underline{\boldsymbol{main}}$ i	ℓ_1 ℓ_∞	$\emptyset$ $(x, 1)$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s_7	$\underline{\boldsymbol{main}}$ i	ℓ_2 ℓ_∞	$\emptyset$ $(x, 1)$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s_8	$\underline{\boldsymbol{main}}$ i j	ℓ_1 ℓ_∞ ℓ_3	$\emptyset$ $(x, 1)$ $\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
s_9	$\boldsymbol{main}$ i $\underline{j}$	ℓ_1 ℓ_∞ ℓ_3	$\emptyset$ $(x, 1)$ $\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$

Figure 9.1: TSO Example

lv and e from its view of the memory; computes the write operation and then adds it to the write buffer. Formally, given a thread i and a store $\sigma = (m, b)$:

$$elem_{lv:=x}(i, \sigma) \stackrel{\text{def}}{=} (m, b[i \mapsto enq(op, b(i))])$$
$$\text{where } op = \mathit{elem}_{lv:=e}(i, view(i, \sigma)).$$

Locks and unlocks do not use write buffers, but alter memory. We assume two functions $lock : \mathbf{Ids} \times \mathbf{Locks} \times \mathbf{Memories} \to \mathbf{Memories}$ and $unlock : \mathbf{Ids} \times \mathbf{Locks} \times \mathbf{Memories} \to \mathbf{Memories}$. Intuitively $lock(i, \mu, m)$ locks the mutex μ for the thread i in the memory m. Formally: $elem_{\mathsf{lock}(\mu)}(i, (m, b)) = (lock(i, \mu, m), b)$ and $elem_{\mathsf{unlock}(\mu)}(i, (m, b)) = (unlock(i, \mu, m), b)$.

We assume a set of initial memories **MemsInit**, e.g., **MemsInit** = **Memories** or **MemsInit** = $\{\lambda x.0\}$. An initial store σ ($\sigma \in$ **StoresInit**) is a pair (m, b) such that $m \in$ **MemsInit** and b maps all threads to the empty FIFO.

The set *System* is the defined by the rules of Fig. 9.2 : $System \stackrel{\text{def}}{=} \{\tau \mid \Vdash_{\text{TSO}} \tau\}$. The rule "schedule" switches current threads and the rule "memory" executes some pending write operation, can be triggered at any time.

Fig. 9.1 gives the beginning of one possible execution of the program of Fig. 7.4. This is the same example as Figures 7.3 and 8.1, but in the TSO Model.

$$\frac{P(j) \text{ is defined} \qquad i \neq j}{\Vdash_{\text{TSO}} (i, P, \sigma, g) \to (j, P, \sigma, g)}\text{schedule} \qquad \frac{\sigma = (m, b) \wedge b(i) \neq \epsilon}{\Vdash_{\text{TSO}} (i, P, \sigma, g) \to (i, P, \mathit{memory\text{-}action}(i, \sigma), g)}\text{memory}$$

Figure 9.2: System Transitions for TSO

Conclusion

To define an operational semantics for TSO, we assumed:

- A set **Memories** of memories
- A set **WriteOp** of write operations
- A function *update-memory***WriteOp** $\times$ **Memories** $\to$ **Memories** that updates a memory according to a write operation
- A set **MemsInit** of initial memories
- Two functions *lock* : **Ids** $\times$ **Locks** $\times$ **Memories** $\to$ **Memories** and *unlock* : **Ids** $\times$ **Locks** $\times$ **Memories** $\to$ **Memories**.
- The two predicates *lockable* and *unlockable*.

9.2.1 Examples

9.2.1.a Maps A *memory* maps variables $\mathcal{V}ar$ to values in $\mathcal{V}$.

A write operation is a pair : **WriteOp** $= \mathcal{V}ar \times \mathcal{V}$. Such a pair $(x, v) \in$ **WriteOp** means that the value v is written into the variable x.

The memory is updated in the following way:

$$\mathit{update\text{-}memory}((x, v), m) = (m[x \mapsto v], b[i \mapsto w])$$

The set of initial memories is the set of all memories **Memories** (or, as seen in Section 8.1, it may be the singleton $\{\lambda x.0\}$, according to C-norm).

The value of the mutex variable μ is the identifier of the thread that owns it, or the special symbol **none**.

The mutexes are locked and unlocked instantaneously in the shared memory, without using write buffers:

$$\begin{aligned} \mathit{lock}(i, \mu, m) &= m[\mu \mapsto i] \\ \mathit{unlock}(i, \mu, m) &= (m[\mu \mapsto \textbf{none}], b) \end{aligned}$$

A thread i can only lock a mutex μ in a store (m, b) if μ is lockable:

$$\mathit{lockable}(i, \mu, (m, b))) \overset{\text{def}}{\Leftrightarrow} m(\mu) = \textbf{none} \wedge b(i) = \epsilon$$

This means that a mutex is lockable only if it is free : two distinct threads can not own the same mutex.

Similarly i can unlock μ if and only if $\mathit{unlockable}(i, \mu, (m, b))$ holds, i.e.:

$$\mathit{unlockable}(i, \mu, (m, b))) \overset{\text{def}}{\Leftrightarrow} m(\mu) = i \wedge b(i) = \epsilon$$

To unlock a mutex, a thread must own it, and must have an empty buffer: a write operation generated when the thread own the mutex can not update the memory when the thread does not own any more the mutex.

Before to spawn a new thread, a thread have to synchronize its view of the memory with the shared memory. Hence, at creation, a thread and its new child have the same view of the memory, e.g., in Figure 6.2a, the thread created in ℓ_4 view $\&y$ as value of p and not $\&x$. Formally, $\mathit{spawnable}(i, (m, b)) \overset{\text{def}}{\Leftrightarrow} b(i) = \epsilon$.

9.2.1.b Gen/Kill Similarly to Section 8.2, the set of values is la lattice and a memory is an element of this lattice: $\mathbf{Memories} = \mathcal{V}$.

A write operation is a pair : $\mathbf{WriteOp} = \mathcal{V} \times \mathcal{V}$. Such a pair $(\mathit{gen}, \mathit{keep}) \in \mathbf{WriteOp}$ means intuitively that the values of *gen* are generated, and the values that are not in *keep* are killed.

The memory is updated in the following way:

$$\mathit{update\text{-}memory}((\mathit{gen}, \mathit{keep}), m) = (m \sqcap \mathit{keep}) \sqcup \mathit{gen}.$$

9.3 PSO

The PSO (Partial Store Ordering) model is similar to the TSO model. In the PSO model, a store is a pair (m, b) where $m \in \mathbf{Memories}$ is a memory and $b : \mathbf{Ids} \times \mathcal{V}\mathit{ar} \to \mathbf{Buffers}$ is a map from threads identifiers and variables to buffers.

Compared to TSO model, the function *memory-action* have an extra argument.

$$\begin{array}{l}\mathit{memory\text{-}action}(i, x, (m, b)) = (\mathit{update\text{-}memory}(op, m), b[i \mapsto w]) \\ \text{where } op = \mathit{fst}(b(i, x)) \text{ and } w = \mathit{deq}(b(i, x)).\end{array}$$

The set *System* is defined by $\mathit{System} = \{\tau \mid \Vdash_{\text{PSO}} \tau\}$, where $\Vdash_{\text{PSO}}$ is defined by the rules of Figure 9.3.

$$\frac{P(j) \text{ is defined} \qquad i \neq j}{\Vdash_{\text{PSO}} (i, P, \sigma, g) \rightarrow (j, P, \sigma, g)} \text{schedule}$$

$$\frac{\sigma = (m, b) \wedge b(i) \neq \epsilon}{\Vdash_{\text{PSO}} (i, P, \sigma, g) \rightarrow (i, P, \mathit{memory\text{-}action}(i, x, \sigma), g)} \text{memory}$$

Figure 9.3: System Transitions for PSO

Part III

From Single-threaded to Multithreaded: Core Model

CHAPTER 10

Intermediate Semantics

10.1 Basic Concepts

To prepare the grounds for abstraction, we introduce an intermediate semantics, called G-collecting semantics, which associates a function on configurations with each statement. The aim of this semantics is to associate with each statement a transfer function that will be abstracted (see Section 13) as an abstract transfer function.

A *concrete configuration* is a tuple $\mathtt{Q} = \langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$:

1. $\mathtt{S}$ is the current state of the system during an execution,
2. $\mathtt{G}$, for *guarantee*, represents what the current thread and its descendants can do
3. and $\mathtt{A}$, for *assume*, represents what the other threads can do.

Formally, $\mathtt{S}$ is a set of states, and $\mathtt{G}$ and $\mathtt{A}$ are sets of transitions containing *System*. The set of concrete configurations is a complete lattice for the ordering $\langle \mathtt{S}_1, \mathtt{G}_1, \mathtt{A}_1 \rangle \leqslant \langle \mathtt{S}_2, \mathtt{G}_2, \mathtt{A}_2 \rangle \Leftrightarrow \mathtt{S}_1 \subseteq \mathtt{S}_2 \wedge \mathtt{G}_1 \subseteq \mathtt{G}_2 \wedge \mathtt{A}_1 \subseteq \mathtt{A}_2$. Let **C-Configurations** the set of concrete configurations.

During an execution, after having encountered a state $s_0 = (j_0, P_0, \sigma_0, g_0)$ we distinguish two kinds of descendants of j_0:

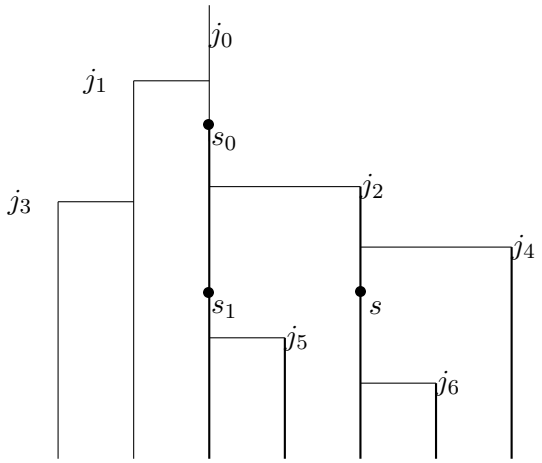


Figure 10.1: *after*

(i) those which already exist in state s_0 (except j_0 itself) and their descendants,

(ii) j_0 and its other descendants.

Each thread of kind (i) has been created by a statement executed by j_0. We call $after(s_0)$ the states from which a thread of kind (ii) can execute a transition. Formaly *after* is defined by:

Definition 10.1. We define the set $after(s)$ of states *after* s:

$$\begin{aligned} after(i, P, \sigma, g) &\stackrel{\text{def}}{=} \{(j, P', \sigma', g \cdot g') \in \mathbf{States} | j \in desc_{g'}(i)\} \\ &= \{(j, P', \sigma', g \cdot g') \in \mathbf{States} | i \preceq_{g'} j\} \end{aligned}$$

We also define the relation $\prec_{after}$ by:

$$s \prec_{after} s' \stackrel{\text{def}}{\Leftrightarrow} s' \in after(s).$$

In Fig. 10.1, the thick lines describe all the states encountered while executing the program that fall into $after(s_0)$. In this figure, $s_1, s \in after(s_0)$.

Lemma 10.1. *The relation $\prec_{after}$ is a pre-ordering on* **States**.

Proof. Let s_0, s_1, and s_2 such that $s_0 \prec_{after} s_1$ and $s_1 \prec_{after} s_2$.

Let $(i_0, P_0, \sigma_0, g_0) = s_0$ and $(i_1, P_1, \sigma_1, g_1) = s_1$. By Claim 7.1, we can define $g'_1 = g_0^{-1} \cdot g_1$. Because $i_0 \preceq_{g'_1} i_0$, $i_0 \in desc_{g'_1}(i_0)$.

Given that $s_0 \prec_{after} s_1$, we state that $s_1 \in after(s_0)$. Let $s_2 = (i_2, P_2, \sigma_2, g_2) \in after(s_1)$. Therefore, there exists g'_2 such that $g_2 = g_1 \cdot g'_2 = g_0 \cdot g'_1 \cdot g'_2$ and $i_2 \in desc_{g'_2}(i_1)$. Because $s_1 \in after(s_0)$, by definition, $i_1 \in desc_{g'_2}(i_0)$. Therefore $i_1 \in desc_{g'_2}(i_1) \cap desc_{g'_1 \cdot g'_2}(i_0)$. According to Lemma 7.4, $desc_{g'_2}(i_1) \subseteq desc_{g'_1 \cdot g'_2}(i_0)$. Hence $i_2 \in desc_{g'_1 \cdot g'_2}(i_0)$ and therefore $s_2 \in after(s_0)$. □

The following lemma is a corollary:

Lemma 10.2. *If $s_1 \in after(s_0)$ then $after(s_1) \subseteq after(s_0)$*

All states are after all initial states.

Lemma 10.3. *For all $P \in \mathbb{P}$ and $\sigma \in$ **Stores**, and $s \in$ **States**:*

$$(\boldsymbol{main}, P, \sigma, \epsilon) \lessdot_{after} s.$$

As a consequence we have the following lemma:

Lemma 10.4. *If Init is the set of initial states of a program and $s \in Init$, then $after(s) =$ **States**.*

If an execution of a program go from a state s_0 to a state s_1 with the same current thread, therefore s_1 is after s_0:

Lemma 10.5. *Let $(s_0, s_1) \in$ **Transitions***.*
If $thread(s_0) = thread(s_1)$ then $s_1 \in after(s_0)$.

Proof. Let $(i_0, P_0, \sigma_0, g_0) = s_0$ and $(i_1, P_1, \sigma_1, g_1) = s_1$. By Claim 7.1, we can define $g'_1 = g_0^{-1} \cdot g_1$, i.e., g'_1 is such that $g_1 = g_0 \cdot g'_1$. Because $i_0 \preceq_{g'_1} i_0$, $i_0 \in desc_{g'_1}(i_0)$. Therefore, if $thread(s) = thread(s')$, i.e., $i_1 = i_0$, then $s_1 \in after(s_0)$ (By definition of *after*). □

When a schedule transition is executed, the current thread changes. The future descendants of the past current thread and the new current thread are not the sames. This is formalized by the following lemma:

Lemma 10.6. *If $(s_1, s_2) \in Schedule$ then $after(s_1) \cap after(s_2) = \emptyset$.*

Proof. Let $(i_1, P_1, \sigma_1, g_1) = s_1$ and $i_2 = thread(s_2)$. Therefore $(i_2, P_1, \sigma_1, g_1) = s_2$. Let $s = (i, P, \sigma, g) \in after(s_1) \cap after(s_2)$.

By definition of *after*, there exists g' such that $g = g_1 \cdot g'$, $i \in desc_{g'}(i_1)$ and $i \in desc_{g'}(i_2)$. Furthermore i_1 and i_2 are in $Dom(P_1)$. Therefore i_1 and i_2 are either created in g_1, or are $\boldsymbol{main}$. Hence, i_1 and i_2 cannot be created in g'. Therefore, $i_2 \notin desc_{g'}(i_1)$ and therefore $desc_{g'}(i_2) \subseteq desc_{\epsilon \cdot g'}(i_1)$. Using Lemma 7.4 we conclude that $desc_{g'}(i_1) \cap desc_{g'}(i_2) = \emptyset$. This is a contradiction with $i \in desc_{g'}(i_1)$ and $i \in desc_{g'}(i_2)$. □

During the execution of a set of transitions T that do not create thread, the set of descendants does not increase:

Lemma 10.7. *Let T a conservative[1] set of transitions.*
Let $s = (i, P, \sigma, g)$ and $s = (i', P', \sigma', g \cdot g')$ be two states.
If $(s, s') \in (\mathtt{A}_{|\overline{after(s_0)}} \cup T)^\star$ then $desc_{g'}(i) = \{i\}$.

[1] This concept is defined in Section 7.3.

Proof. Let $s_0, \ldots, s_n$ a sequence of states such that $s_0 = s$, for all $k \in \{0, \ldots, n-1\}$, $(s_k, s_{k+1}) \in \mathtt{A}_{|\overline{after(s_0)}} \cup T$, and $s_n = s'$.

For all k, let $(i_k, P_k, \sigma_k, g_k) = s_k$. According to Claim 7.1, we can define, for all $k \geqslant 1$, $g'_k = g_{k-1}^{-1} \cdot g_k$. Therefore $g_n = g_0 \cdot g'_1 \cdot \ldots \cdot g'_n$ and $g' = g'_1 \cdot \ldots \cdot g'_n$.

Assume by contradiction that there exists a thread j such that $i \leftarrow_{g'} j$. Therefore, there exists a label ℓ and an integer $k \leqslant 1$ such that $(i, \ell, j) \in g'_k$. By definition of transitions, $i_k = i = i_0$. Therefore, according to Lemma 10.5, $s_{k+1} \in after(s_0)$. Given that $(s_{k-1}, s_k) \in \mathtt{A}_{|\overline{after(s_0)}} \cup T$, we conclude that $(s_{k-1}, s_k) \in T$. Because T is conservative, $g_{k-1} = g_k$ and then $g'_k = g_{k-1}^{-1} \cdot g_k = \epsilon$. Therefore $(i_0, \ell, j) \notin g'_k$. □

These lemmas has a consequence on *after*:

Lemma 10.8. *Let T a conservative set of transitions.*
If $(s_0, s_1) \in (\mathtt{A}_{|\overline{after(s_0)}} \cup T)^\star$ and $s_1 \in after(s_0)$ then $thread(s_1) = thread(s_0)$.

Proof. Let $(i_0, P_0, \sigma_0, g_0) = s_0$ and $(i_1, P_1, \sigma, g_0 \cdot g_1) = s_1$. By Lemma 10.7 $desc_{g_1}(i_0) = \{i_0\}$ and by definition of *after*, $i_1 \in desc_{g_1}(i_0)$. □

Lemma 10.9. *Let T_1 a conservative set of transitions.*

Let s_0, s_1, s three states such that:

- $(s_0, s_1) \in T_1^\star$,
- $thread(s_0) = thread(s_1)$,
- *and* $(s_1, s) \in \mathbf{Transitions}^\star$.

If $s \in after(s_0)$ then $s \in after(s_1)$.

Proof. According to Claim 7.1, the following definitions are correct:
$(i_0, P_0, \sigma_0, g_0) \stackrel{\text{def}}{=} s_0$, $(i_1, P_1, \sigma, g_0 \cdot g_1) \stackrel{\text{def}}{=} s_1$ and $(i, P, \sigma, g_0 \cdot g_1 \cdot g) \stackrel{\text{def}}{=} s$.

By Lemma 10.7 $desc_{g_1}(i_0) = \{i_0\}$ and by definition of *after*, $i_1 \in desc_{g_1}\{i_0\}$. According to Lemma 7.2, $desc_{g_1 \cdot g}(i_0) = desc_g(desc_{g_1}(i_0)) = desc_g(i_0)$.

Because $s \in after(s_0)$, $i desc_{g_1 \cdot g}(\{i_0\})$, therefore $i \in desc_g(i_0)$. Hence $s \in after(s_1)$. □

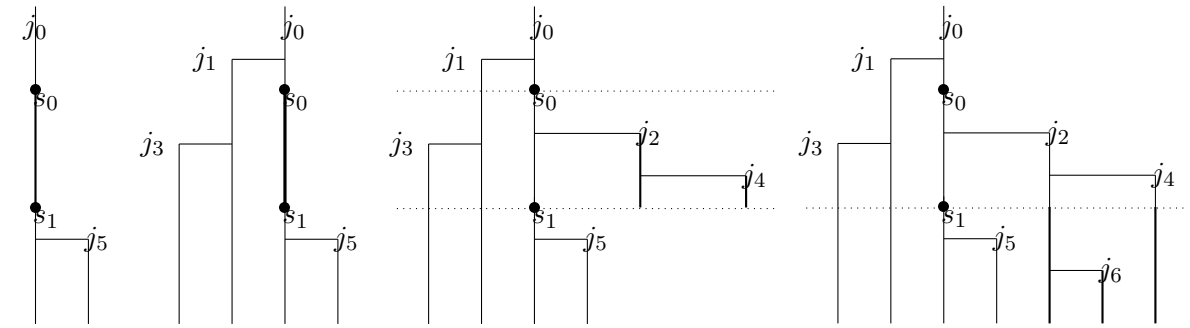
10.2 Definition of the G-collecting Semantics

The definition of the G-collecting semantics $[\![^\ell stmt, \ell']\!]$ of a statement $^\ell stmt, \ell'$ requires some intermediate relations and sets. The formal definition is given by the following definition:

Definition 10.2.

$$[\![^\ell stmt, \ell']\!]\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle \stackrel{\text{def}}{=} \langle \mathtt{S}', \mathtt{G} \cup \mathtt{Self} \cup \mathtt{Par} \cup \mathtt{Sub}, \mathtt{A} \cup \mathtt{Par} \cup \mathtt{Sub}\rangle$$

$$\{\!|^\ell stmt, \ell'|\!\}\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle \stackrel{\text{def}}{=} [\mathtt{Reach}, \mathtt{Ext}, \mathtt{Self}, \mathtt{Par}, \mathtt{Sub}]$$



(a) `Reach` (b) `Reach` (c) `Par` (d) `Sub`

Figure 10.2: G-collecting Semantics

where:

$$\begin{aligned}
\texttt{Reach} &= \left\{ (s_0, s_1) \;\middle|\; \begin{array}{l} (s_0, s_1) \in \left[(\texttt{G}_{|after(s_0)} \cap \mathcal{T}r^{\ell}_{stmt,\ell'}) \cup \texttt{A}_{|\overline{after(s_0)}} \right]^\star \\ \wedge\, thread(s_0) = thread(s_1) \wedge label(s_0) = \ell \end{array} \right\} \\
\texttt{S}' &= \{ s_1 | s_1 \in \texttt{Reach}\langle \texttt{S} \rangle \wedge label(s_1) = \ell' \} \\
\texttt{Self} &= \{ (s, s') \in \mathcal{T}r^{\ell}_{stmt,\ell'} | s \in \texttt{Reach}\langle \texttt{S} \rangle \} \\
\texttt{Par} &= \{ (s, s') \in \mathcal{T}r^{\ell}_{stmt,\ell'} | \exists s_0 \in \texttt{S} : (s_0, s) \in \texttt{Reach}; Schedule \wedge s \in after(s_0) \} \\
\texttt{Ext}(s_0, s_1) &= \left[(\texttt{G}_{|after(s_0)} \cap \mathcal{T}r^{\ell}_{stmt,\ell'}) \cup \texttt{A}_{|\overline{after(s_0)}} \cup \texttt{G}_{|after(s_1)} \right]^\star \\
\texttt{Sub} &= \left\{ (s, s') \in \mathcal{T}r^{\ell}_{stmt,\ell'} \;\middle|\; \begin{array}{l} \exists s_0, s_1 \in \texttt{S} \times \texttt{S}' : (s_0, s_1) \in \texttt{Reach} \wedge \\ (s_1, s) \in \texttt{Ext}(s_0, s_1) \wedge s \in after(s_0) \setminus after(s_1) \end{array} \right\}
\end{aligned}$$

Let us read together, on some special cases shown in Fig. 10.2. This will explain the rather intimidating of Definition 10.2 step by step, introducing the necessary complications as they come along.

The statement is executed between states $s_0 = (j_0, P, \sigma, g)$ and $s_1 = (j_0, P', \sigma', g \cdot g')$.

Figure 10.2(a) describes the single-thread case: there is no thread interaction during the execution of ${}^{\ell}stmt, \ell'$. The thread j_5 is spawned after the execution of the statement. E.g., in Fig. 6.2a, ${}^{\ell_1}p := \&x; {}^{\ell_2}p := \&y, \ell_3$.

In this simple case, a state s is reachable from s_0 if and only if there exists a path from s_0 to s using only transitions done by the unique thread (these transitions should be in the guarantee `G`) and that are generated by the statement. `S'` represents the final states reachable from `S`. Finally, in this case:

$$\begin{aligned}
\texttt{Reach} &= \{ (s_0, s_1) \in \left[\texttt{G} \cap \mathcal{T}r^{\ell}_{stmt,\ell'} \right]^\star | label(s_0) = \ell \} \\
\texttt{S}' &= \{ s_1 \mid s_1 \in \texttt{Reach}(\texttt{S}) \wedge label(s_1) = \ell' \} \\
\texttt{Self} &= \{ (s, s') \in \mathcal{T}r^{\ell}_{stmt,\ell'} \mid s \in \texttt{Reach}(\texttt{S}) \} \\
[\![{}^{\ell}stmt, \ell']\!] \langle \texttt{S}, \texttt{G}, System \rangle &= \langle \texttt{S}', \texttt{G} \cup \texttt{Self}, System \rangle \\
\texttt{Par} &= \texttt{Sub} = \emptyset
\end{aligned}$$

Figure 10.2(b) is more complex: j_0 interferes with threads j_1 and j_3. These interferences are assumed to be in $\mathtt{A}$. Some states can be reached only with such interference transitions. E.g, consider the statement ${}^{\ell_{18}}y := 1; {}^{\ell_{19}}z := y, \ell_\infty$ in Fig. 6.2d: at the end of this statement, the value of z may be 3, because the statement ${}^{\ell_{17}}y := 3, \ell_\infty$ may be executed when the thread ***main*** is at label ℓ_{19}. Therefore, to avoid missing some reachable states, transitions of $\mathtt{A}$ are taken into account in the definition of $\mathtt{Reach}$. In Fig. 10.2(b), the statement ${}^{\ell}stmt, \ell'$ is executed by descendants of j_0 of kind (ii) (i.e., $after(s_0)$), and the interferences come from j_1 and j_3 which are descendants of kind (i) (i.e., in $\overline{after(s_0)}$). Finally, we find the complete formula of Definition 10.2:

$$\mathtt{Reach} = \left\{ (s_0, s_1) \;\middle|\; \begin{array}{l} (s_0, s_1) \in \left[(\mathtt{G}_{|after(s_0)} \cap \mathcal{T}r_{\ell\, stmt, \ell'}) \cup \mathtt{A}_{|\overline{after(s_0)}} \right]^\star \\ \wedge\, thread(s_0) = thread(s_1) \wedge label(s_0) = \ell \end{array} \right\}.$$

In Fig. 10.2(c), when j_0 executes the statement ${}^{\ell}stmt, \ell'$ it creates subthreads (j_2 and j_4) which execute transitions in parallel of the statement. The guarantee $\mathtt{G}$ is not supposed to contain only transitions executed by the current thread but also these transitions. These transitions, represented by thick lines in Fig. 10.2(c), are collected into the set $\mathtt{Par}$. Consider such a transition, it is executed in parallel of the statement, i.e., from a state of $System \circ \mathtt{Reach}(\{s_0\})$. Furthermore, this transition came from the statement, and not from an earlier thread, hence from $after(s_0)$.

$$\mathtt{Par} = \{(s, s') \in \mathcal{T}r_{\ell\, stmt, \ell'} \mid \exists s_0 \in \mathtt{S} : (s_0, s) \in System \circ \mathtt{Reach} \wedge s \in after(s_0)\}.$$

The threads created by j_0 when it executes the statement ${}^{\ell}stmt, \ell'$ may survive when this statement returns in s_1, as shown in Fig. 10.2(d). Such a thread i (here, i is j_4 or j_5 or j_6) can execute transitions that are not in $\mathtt{Par}$. $\mathtt{Sub}$ collects these transitions. The creation of i results of a ***create*** statement executed between s_0 and s_1. Hence, such a transition (s, s') is executed from a state in $after(s_0) \setminus after(s_1)$. The path from s_1 to s is comprised of transitions in $(\mathtt{G}_{|after(s_0)} \cap \mathcal{T}r_{\ell\, stmt, \ell'}) \cup \mathtt{A}_{|\overline{after(s_0)}}$ (similarly to $\mathtt{Reach}$) and of transitions of j_0 or j_5 under the dotted line, i.e., transitions in $\mathtt{G}_{|after(s_1)}$.

Figure 10.3 gives the beginning of a program execution. This execution begins as Figure 9.1. Let ${}^{\ell_1}stmt, \ell_\infty$ be the statement of Figure 7.4.

We consider:

- $[\mathtt{Reach}_1, \mathtt{Ext}_1, \mathtt{Self}_1, \mathtt{Par}_1, \mathtt{Sub}_1] = \left\{\!\!\left| {}^{\ell_1}stmt, \ell_\infty \right|\!\!\right\}\langle\{s_1\}, System, System\rangle$,
- $\langle \mathtt{S}_1, \mathtt{G}_1, \mathtt{A}_1 \rangle = [\![{}^{\ell_1}stmt, \ell_\infty]\!]\langle\{s_1\}, System, System\rangle$,

Applying the definitions, we state that:

- $\mathtt{Reach}_1 = \{(s, s) \mid label(s) = \ell_1\}$,
- $\mathtt{S}_1 = \emptyset$,
- $\mathtt{Self}_1 = \{(s_1, s_2)\} \cup System$,

Name	Threads	Control point	Buffers	Memory	Genealogy
s_1	$\underline{\boldsymbol{main}}$	ℓ_1	$\emptyset$	$x = 0$	ϵ
s_2	$\underline{\boldsymbol{main}}$	ℓ_2	$\emptyset$	$x = 0$	ϵ
s_3	$\underline{\boldsymbol{main}}$	ℓ_1	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
	i	ℓ_3	$\emptyset$		
s_4	$\boldsymbol{main}$	ℓ_1	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
	$\underline{i}$	ℓ_3	$\emptyset$		
s_5	$\boldsymbol{main}$	ℓ_1	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
	$\underline{i}$	ℓ_∞	$(x, 1)$		
s_6	$\underline{\boldsymbol{main}}$	ℓ_1	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
	i	ℓ_∞	$(x, 1)$		
s_7	$\underline{\boldsymbol{main}}$	ℓ_2	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
	i	ℓ_∞	$(x, 1)$		
s_8	$\underline{\boldsymbol{main}}$	ℓ_1	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
	i	ℓ_∞	$(x, 1)$		
	j	ℓ_3	$\emptyset$		
s_9	$\boldsymbol{main}$	ℓ_1	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
	i	ℓ_∞	$(x, 1)$		
	$\underline{j}$	ℓ_3	$\emptyset$		
s_8	$\underline{\boldsymbol{main}}$	ℓ_1	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
	i	ℓ_∞	$(x, 1)$		
	j	ℓ_3	$\emptyset$		
s_{10}	$\underline{\boldsymbol{main}}$	ℓ_2	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
	i	ℓ_∞	$(x, 1)$		
	j	ℓ_3	$\emptyset$		
s_{11}	$\boldsymbol{main}$	ℓ_2	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
	i	ℓ_∞	$(x, 1)$		
	$\underline{j}$	ℓ_3	$\emptyset$		
s_{12}	$\boldsymbol{main}$	ℓ_2	$\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
	i	ℓ_∞	$(x, 1)$		
	$\underline{j}$	ℓ_∞	$(x, 1)$		

Figure 10.3: Example of Execution

- $\mathtt{Par} = \mathit{System}$,
- $\mathtt{Sub} = \emptyset$,
- $\mathtt{G_1} = \{(s_1, s_2)\} \cup \mathit{System}$,
- and $\mathtt{A_1} = \mathit{System}$

Since the $\mathtt{G}$-component of $\langle\{s_1\}, \mathit{System}, \mathit{System}\rangle$ is System, we collect only a few number of transitions in $\mathtt{Self}_1$, $\mathtt{Par}_1$ and $\mathtt{Sub}_1$. Notice that we collect the transition (s_1, s_2) in $\mathtt{Self}_1$.

Now, let us consider:

- $[\mathtt{Reach}_2, \mathtt{Ext}_2, \mathtt{Self}_2, \mathtt{Par}_2, \mathtt{Sub}_2] = \left\{\!\!\left|{}^{\ell_1}\mathit{stmt}, \ell_\infty\right|\!\!\right\}\langle\{s_1\}, \mathtt{G_1}, \mathit{System}\rangle$,
- $\langle \mathtt{S}_2, \mathtt{G}_2, \mathtt{A}_2\rangle = [\![{}^{\ell_1}\mathit{stmt}, \ell_\infty]\!]\langle\{s_1\}, \mathtt{G_2}, \mathit{System}\rangle$,

Notice that:

- $\mathtt{Reach}_2 = \{(s_1, s_2)\} \cup \mathtt{Reach}_1$,
- $\mathtt{S}_2 = \emptyset$,
- $\mathtt{Self}_2 = \{(s_1, s_2), (s_2, s_3)\} \cup \mathit{System}$,
- and $\mathtt{G_2} = \{(s_1, s_2)\} \cup \mathit{System}$.

Figure 10.4 gives an alternative execution. This execution of the program of Figure 7.4 begins with the same states s_1, s_2 and s_3. Nevertheless, when in s_3, instead of going to s_4, in Figure 10.4, the system goes to s'_4, s'_5 and s'_6. After s'_6 the system goes back to the first execution of Figure 9.1. It go to state s_7.

Now we consider:

- $[\mathtt{Reach}_3, \mathtt{Ext}_3, \mathtt{Self}_3, \mathtt{Par}_3, \mathtt{Sub}_3] = \left\{\!\!\left|{}^{\ell_2}\mathit{create}({}^{\ell_3}x := x{+}1), \ell_1\right|\!\!\right\}\langle\{s_2\}, \mathbf{Transitions}, \mathit{System}\rangle$.
- $\langle \mathtt{S}_3, \mathtt{G}_3, \mathtt{A}_3\rangle = [\![{}^{\ell_2}\mathit{create}({}^{\ell_3}x := x + 1), \ell_1]\!]\langle\{s_2\}, \mathbf{Transitions}, \mathit{System}\rangle$

Notice that:

- $(s_2, s_3) \in \mathtt{Reach}_3$
- $s_3 \in \mathtt{S}_3$
- $(s_3, s'_4) \in \mathtt{Ext}_3(s_2, s_3)$

As a consequence, $(s_4, s_5) \in \mathtt{Par}$, but $(s'_5, s'_6) \notin \mathtt{Par}$. Actually, $(s'_5, s'_6) \in \mathtt{Sub}$.

Name	Threads	Control point	Buffers	Memory	Genealogy
s_1	$\underline{\boldsymbol{main}}$	ℓ_1	$\emptyset$	$x = 0$	ϵ
s_2	$\underline{\boldsymbol{main}}$	ℓ_2	$\emptyset$	$x = 0$	ϵ
s_3	$\underline{\boldsymbol{main}}$ i	ℓ_1 ℓ_3	$\emptyset$ $\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s'_4	$\underline{\boldsymbol{main}}$ i	ℓ_2 ℓ_3	$\emptyset$ $\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s'_5	$\boldsymbol{main}$ $\underline{i}$	ℓ_2 ℓ_3	$\emptyset$ $\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s'_6	$\boldsymbol{main}$ $\underline{i}$	ℓ_2 ℓ_∞	$\emptyset$ $(x, 1)$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s_7	$\underline{\boldsymbol{main}}$ i	ℓ_2 ℓ_∞	$\emptyset$ $(x, 1)$	$x = 0$	$(\boldsymbol{main}, \ell_2, i)$
s_8	$\underline{\boldsymbol{main}}$ i j	ℓ_1 ℓ_∞ ℓ_3	$\emptyset$ $(x, 1)$ $\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$
s_9	$\boldsymbol{main}$ i $\underline{j}$	ℓ_1 ℓ_∞ ℓ_3	$\emptyset$ $(x, 1)$ $\emptyset$	$x = 0$	$(\boldsymbol{main}, \ell_2, i) \cdot (\boldsymbol{main}, \ell_2, j)$

Figure 10.4: Alternative Execution

$$
\begin{array}{rcl}
\texttt{interfere}_{\texttt{A}}(\texttt{S}) & \stackrel{\text{def}}{=} & \left\{ s' \,\middle|\, \exists s \in \texttt{S} : \begin{array}{l} (s,s') \in (\texttt{A}_{|\overline{after(s)}} \cup System)^\star \\ \wedge\, thread(s) = thread(s') \end{array} \right\} \\
\texttt{post}(\ell) & \stackrel{\text{def}}{=} & \left\{ s' \,\middle|\, \begin{array}{l} \exists s = (i,P,\sigma,g \cdot (i,\ell,j)) \in \textbf{States} : \\ s' \in after(s) \end{array} \right\} \\
\texttt{schedule-child}(\texttt{S}) & \stackrel{\text{def}}{=} & \left\{ (j,P,\sigma,g') \,\middle|\, \exists i,g : \begin{array}{l} (i,P,\sigma,g') \in \texttt{S} \\ \wedge\, g' = g \cdot (i,\ell,j) \end{array} \right\} \\
\texttt{init-child}_\ell(\langle \texttt{S},\texttt{G},\texttt{A} \rangle) & \stackrel{\text{def}}{=} & \langle \texttt{interfere}_{\texttt{A} \cup (\texttt{G}_{|\texttt{post}(\ell)})} \circ \texttt{schedule-child}(\texttt{S}), \\
 & & \quad System, \texttt{A} \cup (\texttt{G}_{|\texttt{post}(\ell)}) \rangle \\
\texttt{combine}_{\langle \texttt{S},\texttt{G},\texttt{A} \rangle}(\texttt{G}') & \stackrel{\text{def}}{=} & \langle \texttt{interfere}_{\texttt{A} \cup \texttt{G}'}(\texttt{S}), \texttt{G} \cup \texttt{G}', \texttt{A} \cup \texttt{G}' \rangle \\
\texttt{execute-thread}_{f,\texttt{S},\texttt{A}}(\texttt{G}) & \stackrel{\text{def}}{=} & \texttt{G}' \text{ with } \langle \texttt{S}', \texttt{G}', \texttt{A}' \rangle = f \langle \texttt{S},\texttt{G},\texttt{A} \rangle \\
\texttt{guarantee}_f \langle \texttt{S},\texttt{G},\texttt{A} \rangle & \stackrel{\text{def}}{=} & \texttt{execute-thread}^{\uparrow\omega}_{f,\texttt{S},\texttt{A}}(\texttt{G})
\end{array}
$$

Figure 10.5: Basic semantic functions

10.3 Properties of the G-collecting Semantics

To prepare for our static analysis we provide a compositional analysis of the G-collecting semantics in Theorem 12.1 below. To this end, we introduce a set of helper functions, see Fig. 10.5.

The function $\texttt{interfere}_{\texttt{A}}(\texttt{S})$ returns states that are reachable from $\texttt{S}$ by applying interferences in $\texttt{A}$. Notice that these interferences do not change the label of the current thread:

Lemma 10.10. *Let $s = (i,P,\sigma,g)$ and $s' = (i',P',\sigma',g')$. If $(s,s') \in (\texttt{A}_{|\overline{after(s)}} \cup System)^\star$ then $P(i) = P'(i)$, i.e., $label(s) = P'(thread(s))$.*

If furthermore $thread(s) = thread(s')$ then $label(s) = label(s')$.

Proof. There exists a sequence of states $s_0, \ldots, s_n$ such that $s_0 = s$ and $s_n = s'$ and for all $k \in \{0,\ldots,n-1\}$, $(s_k, s_{k+1}) \in \texttt{A}_{|\overline{after(s)}} \cup System$.

Let $(i_k, P_k, \sigma_k, g_k) = s_k$. Let us prove by induction that $P_k(i) = P(i)$. If $(s_k, s_{k+1}) \in System$ and $P_k(i) = P(i)$ then $P_{k+1}(i) = P(i)$. If $(s_k, s_{k+1}) \in \texttt{A}_{|\overline{after(s)}}$ and $P_k(i) = P(i)$ then $s_k \notin after(s_k)$ and then $i_k \neq i$ and then $P_{k+1}(i) = P_k(i) = P(i)$. □

The function $\texttt{post}(\ell)$ computes the set of states that may be reached after having created a thread at label ℓ; `schedule-child` applies a schedule transition to the last child of the current thread. The function $\texttt{init-child}_\ell$ computes a configuration for the last child created at ℓ, taking into account interferences with its parent using $\texttt{post}(\ell)$; notice that we need here the genealogies to define $\texttt{post}(\ell)$ and then to have Theorem 12.1. The following Lemma shows the link between `interfere` and `post`(): the function `interfere` does not allow to enter into a new set $\texttt{post}(\ell)$.

Lemma 10.11. *If $s' \in \texttt{interfere}_{\texttt{A}}(\{s_0\}) \cap \texttt{post}(\ell)$ then $s_0 \in \texttt{post}(\ell)$*

Proof. Let $(i_0, P_0, \sigma_0, g_0) = s_0$ and $(i_0, P', \sigma', g_0 \cdot g') = s'$.

$s' \in \texttt{post}(\ell)$. Therefore, there exists $s = (i, P, \sigma, g \cdot (i, \ell, j))$ such that $s' \in \mathit{after}(s)$.

Hence, both $g \cdot (i, \ell, j)$ and g_0 are prefixes of $g_0 \cdot g'$. According to Lemma 2.7, two cases may occur:

- First case: $g \cdot (i, \ell, j) \leqslant_{\text{prefix}} g_0$. Therefore, there exists a genealogy g_1 such that: $g \cdot (i, \ell, j) \cdot g_1 = g_0$.

  Because $s' \in \mathit{after}(s)$, $i_0 \in \mathit{desc}_{g_1 \cdot g'}(i)$. According to Lemma 7.2 $i_0 \in \mathit{desc}_{g'}(\mathit{desc}(g_1))$ By definition of *desc*, there exists $j_0 \in \mathit{desc}(g_1)$ such that $i_0 \in \mathit{desc}_{g'}(j_0)$. Then, according to Lemma 7.4 $\mathit{desc}_{g'}(j_0) \subseteq \mathit{desc}_{g_1 \cdot g'}$. Hence $j_0 \preceq_{g_1 \cdot g'} i_0$ and $i_0 \preceq_{g_1 \cdot g'} j_0$. Then $i_0 = j_0$. Hence, $s_0 \in \mathit{after}(s)$ and therefore $s_0 \in \texttt{post}(\ell)$.

- Second case: $g_0 \leqslant_{\text{prefix}} g \cdot (i, \ell, j)$.

  Let $s'_0 = (i_0, P, \sigma, g \cdot (i, \ell, j))$. According to Lemma 7.1, $(s'_0, s') \in \textbf{Transitions}^\star$. Hence, according to Lemma 10.5, $s' \in \mathit{after}(s_0)$.

  Nevertheless, $(s, s_0) \in \mathit{Schedule}$, and, according to Lemma 10.6 $\mathit{after}(s) \cap \mathit{after}(s_0) = \emptyset$. But $s' \in \mathit{after}(s) \cap \mathit{after}(s_0)$. This is a contradiction.

□

The function `execute-thread` computes a part of the guarantee (an under-approximation), given the semantics of a command represented as a function f from configuration to configuration. And `guarantee` iterates `execute-thread` to compute the whole guarantee, as shown by the following proposition:

Proposition 10.1 (Soundness of `guarantee`). *Let $\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$ a concrete configuration, ${}^{\ell}\mathit{stmt}, \ell'$ a statement and $\mathtt{G}_\infty = \texttt{guarantee}_{[\![{}^{\ell}\mathit{stmt}, \ell']\!]} \langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$. Let $s_0 \in \mathtt{S}$ and $s \in \mathit{after}(s_0)$ such that $(s, s') \in \mathcal{T}\!r_{{}^{\ell}\mathit{stmt}, \ell'}$.*

If $(s_0, s) \in \left[(\mathcal{T}\!r_{{}^{\ell}\mathit{stmt}, \ell'})_{|\mathit{after}(s_0)} \cup \mathtt{A}_{|\overline{\mathit{after}(s_0)}} \right]^\star$ then $(s, s') \in \mathtt{G}_\infty$

Proof. Let $\langle \mathtt{S}_k, \mathtt{G}_k, \mathtt{A}_k \rangle = \texttt{execute-thread}^k_{[\![{}^{\ell}\mathit{stmt}, \ell']\!], \mathtt{S}, \mathtt{A}}(\mathtt{G})$
and $[\mathtt{Reach}_k, \mathtt{Ext}_k, \mathtt{Self}_k, \mathtt{Par}_k, \mathtt{Sub}_k] = \left\{ \left| {}^{\ell}\mathit{stmt}, \ell' \right| \right\} \langle \mathtt{S}, \mathtt{G}_k, \mathtt{A} \rangle$
and $T = \mathcal{T}\!r_{{}^{\ell}\mathit{stmt}, \ell'}$

Let $s_0, \ldots, s_{n+1}$ a path such that $s_n = s$, $s_{n+1} = s'$ and for all k, $(s_k, s_{k+1}) \in \left[T_{|\mathit{after}(s_0)} \cup \mathtt{A}_{|\overline{\mathit{after}(s_0)}} \right]^\star$. Let m an arbitrary integer. Then, let k_0 the smallest k (if it exists) such that $(s_k, s_{k+1}) \in T_{|\mathit{after}(s_0)} \setminus \mathtt{G}_m$. Then, by definition, $(s_{k_0}, s_{k_0+1}) \in \mathtt{Self}_m \cup \mathtt{Par}_m \subseteq \mathtt{G}_{m+1} \subseteq \mathtt{G}_\infty$. □

This proposition shows how the G-collecting semantics is used to overapproximate the operational semantics.

During the execution of a statement ${}^{\ell}\mathit{stmt}, \ell'$, some interference transitions may be fired at any time. Nevertheless, the labels of the thread(s) executing the statement are still in a label of the statement:

Lemma 10.12. *If* $(s_0, s) \in (\mathcal{T}r_{\ell_{stmt,\ell'}} \cup \mathtt{A}_{|\overline{after(s_0)}})^*$, $label(s_0) \in Labs(^\ell stmt, \ell')$ *and* $s \in after(s_0)$ *then* $label(s) \in Labs(^\ell stmt, \ell')$.

Futhermore, if $label(s) = \ell'$ *or* $label(s) = \ell$ *then* $thread(s_0) = thread(s)$.

Proof. There exists a path $s_1, \ldots, s_n$ such that $s_n = s$ and for all $k \in \{0, \ldots, n-1\}$, $(s_k, s_{k-1}) \in \mathcal{T}r_{\ell_{stmt,\ell'}} \cup \mathtt{A}_{|\overline{after(s_0)}}$. Let $(i_0, P_0, \sigma_0, g_0) = s_0$ and for $k \geqslant 1$, let $(i_k, P_k, \sigma_k, g_0 \cdot g_k) = s_k$.

Let us prove by induction on k that $P_k(i) \in Labs(^\ell stmt, \ell')$ and for all $j \in desc_{g_k}(\{i_0\}) \setminus \{i_0\}$, $P_k(j) \in Labs_{child}(^\ell stmt, \ell')$.

Let us assume that k satisfies the induction property, and let us show that $k+1$ satifies the induction property.

In the case $(s_k, s_{k+1}) \in \mathtt{A}_{|\overline{after(s_0)}}$, $i_k \notin desc_{g_k}(\{i_0\})$ and then for all $j = desc_{g_k}(\{i_0\}) = desc_{g_{k+1}}(\{i_0\})$, $P_k(j) = P_{k+1}(j)$.

In the case $(s_k, s_{k+1}) \in \mathcal{T}r_{\ell_{stmt,\ell'}}$ and $i_k = i_0$, by Lemma 7.7, $P_{k+1}(i_k) \in Labs(^\ell stmt, \ell')$. Furthermore, if $j \in desc_{g_k}(\{i_0\})$ then $P_k(j) = P_{k+1}(j)$. If $j \in desc_{g_{k+1}}(\{i_0\}) \setminus desc_{g_k}(\{i_0\})$, then $j \in Dom(P_{k+1}) \setminus Dom(P_k)$ and by Lemma 7.9, $P_{k+1}(j) \in Labs_{child}(^\ell stmt, \ell')$.

In the case$(s_k, s_{k+1}) \in \mathcal{T}r_{\ell_{stmt,\ell'}}$ and $i_k = i_0$, we conclude similarly by Lemma 7.10. If $s \in after(s_0)$, then $i_n \in desc_{g_n}(\{i_0\})$ and therefore $label(s) \in Labs(^\ell stmt, \ell')$.

If $label(s) = \ell'$ or $label(s) = \ell$, then, because by Lemma 7.10, ℓ and ℓ' are not in $Labs_{child}(^\ell stmt, \ell')$, we have $thread(s_0) = thread(s)$. □

The following lemma summarizes the consequences on `Reach` of Lemmas 10.2, 10.5 and 10.12:

Lemma 10.13. *Let* $[\mathtt{Reach}, \mathtt{Ext}, \mathtt{Self}, \mathtt{Par}, \mathtt{Sub}] = \left\{\left|^\ell stmt, \ell'\right|\right\}\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle$.

If $(s_0, s) \in \mathtt{Reach}$ *therefore* $s \in after(s_0)$, $after(s) \subseteq after(s_0)$ *and* $label(s) \in Labs(^\ell stmt, \ell')$.

Proof. $(s_0, s) \in \mathtt{Reach}$. Therefore, by definition, $thread(s_0) = thread(s)$. Hence, according to Lemma 10.5, $s \in after(s_0)$. According to Lemma 10.2, $after(s) \subseteq after(s_0)$.

Given that $(s_0, s) \in \mathtt{Reach}$, we state that $(s_0, s) \in \left[(\mathtt{G}_{|after(s_0)} \cap \mathcal{T}r_{\ell_{stmt,\ell'}}) \cup \mathtt{A}_{|\overline{after(s_0)}}\right]^\star$. Hence, according to Lemma 10.12, $label(s) \in Labs(^\ell stmt, \ell')$. □

After a statement returns, some subthreads created during the execution of the statement may continue to be executed. We introduce a concept of coherence:

Definition 10.3. A set T of transitions is *coherent* with $^\ell stmt, \ell'$ and the states s_0 and s_1 if and only if:

$$\forall (s, s') \in T, s \in after(s_0) \cap \overline{after(s_1)} \wedge label(s) \in Labs(^\ell stmt, \ell') \Rightarrow (s, s') \in \mathcal{T}r^\ell stmt, \ell'.$$

Recall Figure 10.2d. A set of transition coherent with $^\ell stmt, \ell'$ and the states s_0 and s_1 of 10.2d may contain two kinds of transitions:

- Transitions (s, s') done by j_5, j_6 and j_4. These transitions are in $\mathcal{T}r_{\ell_{stmt,\ell'}}$ (in bold in Figure 10.2d), or are transitions of another statement (i.e., $label(s) \notin Labs(^\ell stmt, \ell')$).

- Transition done by other threads.

The Following Lemma ensures us that any transition executed by a thread created during the execution of ${}^{\ell}stmt, \ell'$ (i.e., between s_0 and s_1) is a transition generated by the statement ${}^{\ell}stmt, \ell'$.

Lemma 10.14. *Let T a set of transitions coherent with stmt, s_0 and s_1. For all $s = (i, P, \sigma, g) \in$ **States**, if $(s_1, s) \in T^\star$, therefore $\forall j \in desc_g(i_0) \setminus desc_{g_1^{-1}.g}(i_0)$, $P(j) \in Labs_{child}({}^{\ell}stmt, \ell')$.*

Proof. Let $i_0 = thread(s_0)$.

Let us prove by induction on $n \in \mathbb{N}$ that for all n, for all $s = (i \in, P \in, \sigma \in, g \in)$**States**, if $(s_1, s) \in T^n$, therefore $\forall j \in desc_g(i_0) \setminus desc_{g_1^{-1}.g}(i_0)$, $P(i) \in Labs_{child}({}^{\ell}stmt, \ell')$.

Let s such that $(s_1, s) \in T^{n+1}$. Therefore, there exists s_2 such that $(s_1, s_2) \in T^n$ and $(s_2, s) \in T$.

Let $s_2 = (i_2, P_2, \sigma_2, g_2)$. Let $j \in desc_g(i_0) \setminus desc_{g_1^{-1}.g}(i_0)$.

There are several cases:

- First case: $j \in desc_{g_2}(i_0)$. Therefore, by induction hypothesis, $P_2(j) \in Labs_{child}({}^{\ell}stmt, \ell')$. There is two cases.
  - First case $j = i_2$. Because $j \in desc_g(i_0)$, $s_2 \in after(s_0)$. Because $j \in desc_{g_1^{-1}.g}(i_0)$, $s_2 \notin after(s_0)$. Hence, because T is coherent, $(s_2, s) \in \mathcal{T}r^{\ell}stmt, \ell'$.

    Therefore, by Lemma 7.9, $j \in Labs_{child}({}^{\ell}stmt, \ell')$.
  - Second case $j \neq i_2$, hence, according to the definition of transitions, $P_2(j) = P(j)$. Therefore $P(j) \in Labs_{child}({}^{\ell}stmt, \ell')$.
- Second case: $j \in\notin desc_{g_2}(i_0)$. Therefore, by definition of transitions, $(i_2, P_2(i_2), j) \in g_2^{-1} \cdot h$. Therefore, according to Lemma 7.4, $desc_{g_2^{-1}.g}(i_2) \subseteq desc_g(i_0)$ and therefore $i_2 \in desc_g(i_0)$. Furthermore, because $j \in desc_{g_2^{-1}.g}(i_2)$ and $j \notin desc_{g_1^{-1}.g}(i_0)$, therefore, according to Lemma 7.4, $desc_{g_2^{-1}.g}(i_2) \cap desc_{g_1^{-1}.g}(i_0) = \emptyset$ and therefore $i_2 \notin desc_{g_1^{-1}.g}(i_0)$.

  Hence, by induction hypothesis, $P_2(i_2) \in Labs_{child}({}^{\ell}, stmt, \ell')$. And therefore, by Lemma 7.9, $P(j) \in Labs_{child}({}^{\ell}, stmt, \ell')$.

□

The following proposition is fundamental to prove the main properties of `Ext`(,) and `Sub`.

Proposition 10.2. *Let ${}^{\ell}stmt, \ell'$ a statement,*
$[\texttt{Reach}, \texttt{Ext}, \texttt{Self}, \texttt{Par}, \texttt{Sub}] = \left\{\left|{}^{\ell}stmt, \ell'\right.\right\}\langle \texttt{S}, \texttt{G}, \texttt{A}\rangle$. *Let $(s_0, s_1) \in$ `Reach` and T a set of transitions coherent with stmt, s_0 and s_1.*

Let s' and s'' such that $(s_1, s') \in T^\star$ and $(s', s'') \in T$. Therefore, if $s' \in after(s_0)$ then either $s' \in after(s_1)$ or $(s', s'') \in \mathcal{T}r_{\ell_{stmt}, \ell'}$.

Proof. Let $(i_0, P_0, \sigma_0, g_0) = s_0$ and $(i', P', \sigma', g') = s'$.
Either $s' \in \mathit{after}(s_1)$ or $s' \notin \mathit{after}(s_1)$.

- First case: $s' \in \mathit{after}(s_1)$, we have nothing to prove.
- Second case: $s' \notin \mathit{after}(s_1)$. Therefore $i' \in \mathit{desc}_{g_0 \cdot g'}(i_0) \setminus \mathit{desc}_{g'}(i_0)$, and by Lemma 10.14 $\mathit{label}(s') \in \mathit{Labs}_{\mathit{child}}({}^{\ell}\mathit{stmt}, \ell')$. Hence, by definition of T, $(s', s'') \in \mathcal{T}r_{\ell\,\mathit{stmt},\ell'}$.

□

The G-collecting semantics is a sound overapproximation on the operational semantics. In other words:

Theorem 10.1 (Soundness)**.** *Consider a program ${}^{\ell}\mathit{cmd}, \ell_\infty$ and its set of initial states Init. Let:*

$$\langle \mathtt{S}', \mathtt{G}', \mathtt{A}' \rangle \stackrel{def}{=} [\![{}^{\ell}\mathit{cmd}, \ell_\infty]\!] \langle \mathit{Init}, \mathtt{G}_\infty, \mathit{System} \rangle$$
$$\text{with } \mathtt{G}_\infty = \mathtt{guarantee}_{[\![{}^{\ell}\mathit{cmd}, \ell_\infty]\!]} \langle \mathit{Init}, \mathit{System}, \mathit{System} \rangle$$

Then:

$$\begin{aligned}
\mathtt{S}' &= \{(\boldsymbol{main}, P, \sigma, g) \in \mathcal{T}r^{\star}_{\ell\,\mathit{cmd},\ell_\infty} \langle \mathit{Init} \rangle \mid P(\boldsymbol{main}) = \ell_\infty\} \\
\mathtt{G}' &= \mathtt{G}_\infty = \{(s, s') \in \mathcal{T}r_{\ell\,\mathit{cmd},\ell_\infty} \mid s \in \mathcal{T}r^{\star}_{\ell\,\mathit{cmd},\ell_\infty} \langle \mathit{Init} \rangle\} \cup \mathit{System} \\
\mathtt{A}' &= \{(s, s') \in \mathcal{T}r_{\ell\,\mathit{cmd},\ell_\infty} \mid s \in \mathcal{T}r^{\star}_{\ell\,\mathit{cmd},\ell_\infty} \langle \mathit{Init} \rangle \wedge \mathit{thread}(s) \neq \boldsymbol{main}\} \\
&\quad \cup \mathit{System}
\end{aligned}$$

Proof. Let $[\mathtt{Reach}, \mathtt{Ext}, \mathtt{Self}, \mathtt{Par}, \mathtt{Sub}] = \left\{\!\!\left| {}^{\ell}\mathit{cmd}, \ell_\infty \right|\!\!\right\} \langle \mathit{Init}, \mathtt{G}_\infty, \mathit{System} \rangle$.

We only have to prove that $\mathtt{Reach} = \{s \in \mathcal{T}r^{\star}_{\ell\,\mathit{cmd},\ell_\infty} \langle \mathit{Init} \rangle \mid \mathit{thread}(s) = \boldsymbol{main}\}$. We conclude using Definition 10.2.

Let $s_1 \in \{s \in \mathcal{T}r^{\star}_{\ell\,\mathit{cmd},\ell_\infty} \langle \mathit{Init} \rangle \mid \mathit{thread}(s) = \boldsymbol{main}\}$.

There exists $s_0 \in \mathit{Init}$ such that $(s_0, s) \in \mathcal{T}r^{\star}_{\ell\,\mathit{cmd},\ell_\infty}$ By proposition 10.1, $(s_0, s) \in \mathtt{G}_\infty \cap \mathcal{T}r^{\star}_{\ell\,\mathit{cmd},\ell_\infty}$

By Lemma 10.4, $(s_0, s) \in (\mathtt{G}_{\infty\,|\mathit{after}(s_0)} \cap \mathcal{T}r^{\star}_{\ell\,\mathit{cmd},\ell_\infty}) \cup \mathit{System}_{\overline{|\mathit{after}(s_0)}}$. Hence (s_0, s). □

CHAPTER 11

Overapproximation of the Intermediate Semantics

To ease abstraction, we overapproximate the intermediate semantics by a denotational semantics. In this Chapter, each section give a way to overapproximate one statement. These overapproximations will be used to define a denotational semantics in Chapter 12.

The Proposition 11.1 of Section 11.1 allows to overapproximate basic statements. The proposition 11.2 of Section 11.2 allows to overapproximate the composition of two statements. Notice that the composition of statements can be overapproximated by the composition of their semantics. This is not trivial, since when a statement ${}^{\ell_1}cmd_1;{}^{\ell_2}cmd_2,\ell_3$ is executed, the command ${}^{\ell_1}cmd_1$ may spawn some threads that will interfere with the execution of the command ${}^{\ell_2}cmd_2$.

The proposition 11.3 of Section 11.3 allows to overapproximate *if* statements. The proof of this proposition is similar to the proof of Proposition 11.2, since a if statement look like a composition between a guard and a command.

The proposition 11.4 of Section 11.4 allows to overapproximate the composition of two statements. Notice that a while loop may create an infinite number of threads. And a thread created in the k^{th} iteration may interfere with the $k+1^{\text{th}}$ iteration of the loop, but not with the $k-1^{\text{th}}$ iteration.

Finally, the Proposition 11.5 of Section 11.5 allows to overapproximate thread creation.

11.1 Basic Statements

In this section, we exhibit an overapproximation of the semantics of basic statements. This overapproximation is given by Proposition 11.1

An execution path of a basic statement can be decomposed in interferences, then one transition of the basic statement, and then, some other interferences. The following lemma shows this. This lemma will allow us to prove Proposition 11.1.

Lemma 11.1. *Let* ${}^{\ell_1}basic, \ell_2$ *be a basic statement,*
and $[\texttt{Reach}, \texttt{Ext}, \texttt{Self}, \texttt{Par}, \texttt{Sub}] = \left\{\!\!\left|{}^{\ell_1}basic, \ell_2\right|\!\!\right\}\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle$. *Let* $(s_0, s) \in \texttt{Reach}$ *then:*

- *either* $s \in \texttt{interfere}_{\mathtt{A}}(\{s_0\})$ *and* $label(s) = \ell_1$,
- *or* $s \in \texttt{interfere}_{\mathtt{A}}(\mathcal{T}r^{-}_{\ell_1 basic,\ell_2}\langle \texttt{interfere}_{\mathtt{A}}(\{s_0\})\rangle)$
  and $label(s) = \ell_2$

Proof. By definition of Reach (See Definition 10.2), there exists $s_1, \ldots, s_n$ such that $s_n = s$ and for all $k \leqslant 0$, $(s_k, s_{k+1}) \in (\mathtt{G}_{|after(s_0)} \cap \mathcal{T}r_{\ell_1 basic,\ell_2}) \cup \mathtt{A}_{\overline{|after(s_0)}}$.

Either there exists k such that $(s_k, s_{k+1}) \in \mathtt{G}_{|after(s_0)} \cap \mathcal{T}r_{\ell_1 basic,\ell_2})$ or there do not exist such a k.

- First case, there do not exist such a k. Therefore $(s_0, s) \in (\mathtt{A}_{\overline{|after(s_0)}} \cup System)^\star$. By definition of Reach (See Definition 10.2), $thread(s_0) = thread(s)$. Therefore $s \in \texttt{interfere}_{\mathtt{A}}(\{s_0\})$. By Lemma 10.10, $label(s_0) = label(s)$, hence, $label(s) = \ell_1$.
- Second case, there exists such a k. Let k_0 be the smallest such k. Hence, by definition, $s_{k_0} \in after(s_0)$ and $(s_0, s_k) \in (\mathtt{A}_{\overline{|after(s_0)}} \cup System)^\star$. Hence, by Lemma 10.8, $thread(s_0) = thread(s_{k_0})$. Therefore $s_{k_0} \in \texttt{interfere}_{\mathtt{A}}(\mathtt{S})$.

  Furthermore, by Lemma 7.6, $label(s_{k_0+1}) = \ell_2$.

  Either there exists $k \geqslant k_0 + 1$ such that $(s_k, s_{k+1}) \in \mathtt{G}_{|after(s_0)} \cap \mathcal{T}r_{\ell_1 basic,\ell_2})$ or there does not exist such a k.

  - First case, there does not exist such a k. Therefore $(s_{k_0+1}, s) \in (\mathtt{A}_{\overline{|after(s_0)}} \cup System)^\star$. By definition of Reach (See Definition 10.2), $thread(s_1) = thread(s)$. Therefore $s \in \texttt{interfere}_{\mathtt{A}}(\{s_{k_0+1}\})$. By Lemma 10.10, $label(s_0) = label(s)$, hence, $label(s) = \ell_2$.
  - Second case, there exists such a k. Let k_1 be the smallest such k. By definition of k_1, $(s_{k_0+1}, s_{k_1}) \in (\mathtt{A}_{\overline{|after(s_0)}} \cup System)^\star$. By Lemma 10.10, $label(s_{k_0} + 1) = label(s_{k_1})$, therefore $label(s_{k_0} + 1) = \ell_2$. According to Lemma 7.6, this is a contradiction.

□

Given a basic statement ${}^{\ell_1}basic, \ell_2$ and $[\texttt{Reach}, \texttt{Ext}, \texttt{Self}, \texttt{Par}, \texttt{Sub}] = \left\{\!\!\left|{}^{\ell_1}basic, \ell_2\right|\!\!\right\}\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle$, we claim that:

Claim 11.2. $\texttt{Par} = \emptyset$.

Claim 11.3. $\texttt{Sub} = \emptyset$.

Claim 11.4. $\texttt{Self} \subseteq \{(s, s') \in \mathcal{T}r_{\ell_1 basic,\ell_2} \mid s \in \texttt{interfere}_\texttt{A}(\texttt{S})\} \cup \mathit{System}$.

Claim 11.5. $\texttt{S}' \subseteq \texttt{interfere}_\texttt{A}\big(\bar{\mathcal{T}r}_{\ell_1 basic,\ell_2}\langle\texttt{interfere}_\texttt{A}(\texttt{S})\rangle\big)$.

Claims 11.2 and 11.3 say that when a basic statement is executed, only one thread is executed. Notice that *spawn* creates a subthread, but does not execute it. The Claim 11.4 characterizes the transitions done by the current thread. The Claim 11.5 gives an overapproximation of $\texttt{S}'$, the set of states reached at the end of the execution of a basic statement.

We prove these claims in the following way.

proof of Claim 11.2. Let $(s, s') \in \texttt{Par}$. According to Definition 10.2, there exists $s_0 \in \texttt{S}$ such that $(s_0, s) \in \texttt{Reach}; \mathit{Schedule}$ and $s \in \mathit{after}(s_0)$. Therefore there exists s_1 such that $(s_0, s_1) \in \texttt{Reach}$ and $(s_1, s) \in \mathit{Schedule}$.

By Definition 10.2, $\mathit{thread}(s_0) = \mathit{thread}(s_1)$.

Given that $\mathcal{T}r_{\ell_1 basic,\ell_2}$ is conservative by Lemma 7.5, according to Lemma 10.8 $\mathit{thread}(s_0) = \mathit{thread}(s)$.

By definition of *Schedule*, $\mathit{thread}(s_1) \neq \mathit{thread}(s)$.

There is a contradiction, therefore $\texttt{Par} = \emptyset$. □

proof of Claim 11.3. Let $(s, s') \in \texttt{Sub}$. By definition, there exists $s_0 \in \texttt{S}$ and s_1 such that $(s_0, s_1) \in \texttt{Reach}$, $(s_1, s) \in \texttt{Ext}(s_0, s_1)$ and $s \in \mathit{after}(s_0) \setminus \mathit{after}(s_1)$.

In particular $(s_1, s) \in \big[(\texttt{A} \cup \texttt{G})_{|\overline{after(s_1)}} \cup \mathcal{T}r_{\ell_1 basic,\ell_2}\big]^\star$. Hence, by Lemma 10.8, $\mathit{thread}(s_1) = \mathit{thread}(s)$.

By definition of Reach, $\mathit{thread}(s_1) = \mathit{thread}(s_0)$ and then according to Lemma 10.5, $s \in \mathit{after}(s_0)$.

This is contradictory with Definition 10.2 that implies $s \in \mathit{after}(s_0) \setminus \mathit{after}(s_1)$. Hence $\texttt{Sub} = \emptyset$. □

proof of Claim 11.4. Let $(s, s') \in \texttt{Self} \setminus \mathit{System}$. Then $(s, s') \in \bar{\mathcal{T}r}_{\ell_1 basic,\ell_2}$ and $s \in \texttt{Reach}\langle\texttt{S}\rangle$. Then, there exists $s_0 \in \texttt{S}$ such that $(s_0, s) \in \texttt{Reach}$. Because $(s, s') \in \bar{\mathcal{T}r}_{\ell_1 basic,\ell_2}$, by Lemma 7.6, $\mathit{label}(s) = \ell_1 \neq \ell_2$. By Lemma 11.1, $s \in \texttt{interfere}_\texttt{A}(\{s_0\}) \subseteq \texttt{interfere}_\texttt{A}(\texttt{S})$. □

proof of Claim 11.5. Let $s \in \texttt{S}'$. Therefore, $\mathit{label}(s) = \ell_2$ and there exists $s_0 \in \texttt{S}$ such that $(s_0, s) \in \texttt{Reach}$.

Because $\mathit{label}(s) = \ell_2 \neq \ell_1$, according to Lemma 11.1:
$s \in \texttt{interfere}_\texttt{A}(\bar{\mathcal{T}r}_{\ell_1 basic,\ell_2}\langle\texttt{interfere}_\texttt{A}(\{s_0\})\rangle) \subseteq \texttt{interfere}_\texttt{A}(\bar{\mathcal{T}r}_{\ell_1 basic,\ell_2}\langle\texttt{interfere}_\texttt{A}(\texttt{S})\rangle)$
□

The following statement gives an overapproximation of the semantics of basic statements.

Proposition 11.1 (Basic statements). *Let* $^{\ell_1}basic, \ell_2$ *be a basic statement, then:*

$$[\![^{\ell_1}basic, \ell_2]\!]\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle \leqslant \langle \mathtt{S}'', \mathtt{G} \cup \mathtt{G}_{new}, \mathtt{A}\rangle$$

where $\mathtt{S}'' = \mathtt{interfere}_{\mathtt{A}}\big(\mathcal{T}r^{-}_{\ell_1 basic,\ell_2}\langle \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\rangle\big)$
and $\mathtt{G}_{new} = \{(s, s') \in \mathcal{T}r_{\ell_1 basic,\ell_2} \mid s \in \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\}$

Proof. This proposition is a straightforward consequence of Claims 11.2, 11.3, 11.4 and 11.5. □

11.2 Composition

Lemma 11.6. $\mathcal{T}r_{\ell_1 cmd_1;^{\ell_2} cmd_2,\ell_3} = \mathcal{T}r_{\ell_1 cmd_1,\ell_2} \cup \mathcal{T}r_{\ell_2 cmd_2,\ell_3}$

In this section, we consider an initial configuration : $\mathtt{Q}_0 = \langle \mathtt{S}_0, \mathtt{G}_0, \mathtt{A}_0\rangle$ and a sequence $^{\ell_1} cmd_1;^{\ell_2} cmd_2, \ell_3$. We write $\mathcal{T}r_1 = \mathcal{T}r_{\ell_1 cmd_1,\ell_2}$ and $\mathcal{T}r_2 = \mathcal{T}r_{\ell_2 cmd_2,\ell_3}$ and $\mathcal{T}r = \mathcal{T}r_{\ell_1 cmd_1;^{\ell_2} cmd_2,\ell_3}$

Define:
$\mathtt{Q}' = \langle \mathtt{S}', \mathtt{G}', \mathtt{A}'\rangle = [\![^{\ell_1} cmd_1;^{\ell_2} cmd_2, \ell_3]\!](\mathtt{Q}_0)$
$\mathtt{K} = [\mathtt{Reach}, \mathtt{Ext}, \mathtt{Self}, \mathtt{Par}, \mathtt{Sub}] = \big\{\!\!\big|^{\ell_1} cmd_1;^{\ell_2} cmd_2, \ell_3\big|\!\!\big\}(\mathtt{Q}_0)$
$\mathtt{Q}_1 = \langle \mathtt{S}_1, \mathtt{G}_1, \mathtt{A}_1\rangle = [\![^{\ell_1} cmd_1, \ell_2]\!](\mathtt{Q}_0)$
$\mathtt{K}_1 = [\mathtt{Reach}_1, \mathtt{Ext}_1, \mathtt{Self}_1, \mathtt{Par}_1, \mathtt{Sub}_1] = \big\{\!\!\big|^{\ell_1} cmd_1, \ell_2\big|\!\!\big\}(\mathtt{Q}_0)$
$\mathtt{Q}_2 = \langle \mathtt{S}_2, \mathtt{G}_2, \mathtt{A}_2\rangle = [\![^{\ell_2} cmd_2, \ell_3]\!](\mathtt{Q}_1)$
$\mathtt{K}_2 = [\mathtt{Reach}_2, \mathtt{Ext}_2, \mathtt{Self}_2, \mathtt{Par}_2, \mathtt{Sub}_2] = \big\{\!\!\big|^{\ell_2} cmd_2, \ell_3\big|\!\!\big\}(\mathtt{Q}_1)$

Lemma 11.7. *If* $(s, s') \in \mathcal{T}r$ *and* $label(s) \in Labs(^{\ell_1} cmd_1, \ell_2) \setminus \{\ell_2\}$ *then* $(s, s') \in \mathcal{T}r_1$.
If $(s, s') \in \mathcal{T}r$ *and* $label(s) \in Labs(^{\ell_2} cmd_2, \ell_3)$ *then* $(s, s') \in \mathcal{T}r_2$.

Proof. Let us consider the case $label(s) \in Labs(^{\ell_1} cmd_1, \ell_2) \setminus \{\ell_2\}$. Hence because labels of the command $^{\ell_1} cmd_1;^{\ell_2} cmd_2, \ell_3$ are pairwise distinct, $label(s) \notin Labs(^{\ell_2} cmd_3, \ell_3)$. By Lemma 7.8, $(s, s') \notin \mathcal{T}r_2$. Hence, by Lemma 11.6, $(s, s') \in \mathcal{T}r_1$

The case $label(s) \in Labs(^{\ell_2} cmd_2, \ell_3)$ is similar :
Because labels of the command $^{\ell_1} cmd_1;^{\ell_2} cmd_2, \ell_3$ are pairwise distinct, $label(s) \notin Labs(^{\ell_1} cmd_2, \ell_2)$. By Lemma 7.8, $(s, s') \notin \mathcal{T}r_1$. Hence, by Lemma 11.6, $(s, s') \in \mathcal{T}r_2$. □

Lemma 11.8. *Using the above notations, for every* $(s_0, s) \in \mathtt{Reach}$ *such that* $s_0 \in \mathtt{S}_0$,

(a) either $(s_0, s) \in \mathtt{Reach}_1$ *and* $label(s) \neq \ell_2$

(b) or there exists $s_1 \in \mathtt{S}_1$ *such that* $(s_0, s_1) \in \mathtt{Reach}_1$, $(s_1, s) \in \mathtt{Reach}_2$ *and* $(s_1, s) \in \mathtt{Ext}_2(s_0, s_1)$.

Proof. Let $(s_0, s) \in \mathtt{Reach}$. Hence there exists $s_1, \ldots, s_n$ such $s = s_n$ and that for all $k \leqslant 0$, $(s_k, s_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{0\,\overline{|after(s_0)}}$ and $thread(s_0) = thread(s_n)$.

Either there exists k such that $(s_k, s_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_2) \setminus System$, or there does not exists a such k.

- In the case where no such k exists, $(s_0, s) \in \mathtt{Reach}_1$.

  Either $\mathit{label}(s) \neq \ell_2$, or $\mathit{label}(s) \neq \ell_2$.

  - If $\mathit{label}(s) \neq \ell_2$, then we are in the case (a) of the lemma
  - If $\mathit{label}(s) \neq \ell_2$, therefore, $(s, s) \in \mathtt{Reach}_2$ and therefore $s \in \mathtt{S}_1$. Notice that $(s, s) \in \mathtt{Ext}_1(s_0, s)$. We are in the case (b) of the lemma.

- If such a k exists, let k_0 the smallest such k. By definition of k_0, and by Lemma 11.6 $(s_0, s_{k_0}) \in [\mathcal{T}r_1 \cup \mathtt{A}_{0|\overline{after(s_0)}}]^\star$ By definition of $\mathtt{Reach}$, $\mathit{label}(s_0) = \ell_1 \in \mathit{Labs}({}^{\ell_1}cmd_1, \ell_2)$. Because $(s_{k_0}, s_{k_0+1}) \in \mathtt{G}_{0|after(s_0)}$, then $s_{k_0} \in \mathit{after}(s_0)$. so, according to Lemma 10.12, $\mathit{label}(s_{k_0}) \in \mathit{Labs}({}^{\ell_1}cmd_1, \ell_2)$. Given that $(s_{k_0}, s_{k_0+1}) \in \mathcal{T}r_2 \setminus \mathit{System}$, according to Lemma 7.8, $\mathit{label}(s_{k_0}) \in \mathit{Labs}({}^{\ell_2}cmd_2, \ell_3)$. Hence $\mathit{label}(s_{k_0}) \in \mathit{Labs}({}^{\ell_2}cmd_2, \ell_3) \cap \mathit{Labs}({}^{\ell_1}cmd_1, \ell_2)$. Because the labels of ${}^{\ell_1}cmd_1; {}^{\ell_2}cmd_2, \ell_3$ are pairwise distinct, $\mathit{label}(s_{k_0}) = \ell_2$. Using Lemma 10.12, we conclude that $\mathit{thread}(s_0) = \mathit{thread}(s_1)$. Hence $(s, s_{k_0}) \in \mathtt{Reach}_1$.

  Let us show that $(s_{k_0}, s_n) \in \mathtt{Reach}_2$. Since $(s_0, s_n) \in \mathtt{Reach}$ and $\mathit{thread}(s_{k_0}) = \mathit{thread}(s_0)$ and $\mathit{label}(s_{k_0}) = \ell_2$, we just have to show that $(s_{k_0}, s_n) \in [(\mathtt{G}_{1|after(s_1)} \cap \mathcal{T}r) \cup \mathtt{A}_{1|\overline{after(s_1)}}]^\star$.

  For all $k \geqslant 0$, $(s_k, s_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{0|\overline{after(s_0)}}$. According to Lemma 11.6:
  For all $k \geqslant k_0$, $(s_k, s_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_1) \cup (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_2) \cup \mathtt{A}_{0|\overline{after(s_0)}}$.

  We want to prove, for all $k \geqslant k_0$, two things:

  (i) If $(s_k, s_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_2) \setminus \mathit{System}$ then $s_k \in \mathit{after}(s_1)$

  (ii) If $(s_k, s_{k+1}) \in \mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_1$ then $(s_k, s_{k+1}) \in \mathtt{Sub}_1 \subseteq \mathtt{A}_1$

  Either there exists a k that does not satisfy (i) and (ii), or there do not exist such a k.

  - First case, there does not exists such a k. By definition $s_{k_0} \in \mathit{after}(s_0)$, hence, according to Lemma 10.2, $\mathit{after}(s_{k_0}) \subseteq \mathit{after}(s_0)$ and then $\mathtt{A}_{0|\overline{after(s_0)}} \subseteq \mathtt{A}_{0|\overline{after(s_{k_0})}}$. Therefore $(s_{k_0}, s_n) \in \mathtt{Reach}_2$ and $(s_{k_0}, s_n) \in \mathtt{Ext}_1(s_0, s_{k_0})$.
    Since $\mathit{label}(s_{k_0}) = \ell_2$, $s_{k_0} \in \mathtt{S}_1$.
  - Let us consider the case where there exists such a k. Let k_1 the smallest such k.
    For all $k \in \{k_0, \ldots, k_1 - 1\}$, given that for all $k \geqslant k_0$, $(s_k, s_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_1) \cup (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_2) \cup \mathtt{A}_{0|\overline{after(s_0)}}$ there are three cases:
    - $(s_k, s_{k+1}) \in \mathtt{A}_{|\overline{after(s_0)}}$, and then $s_k \in \overline{\mathit{after}(s_0)}$.
    - $(s_k, s_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_1)$ and then $(s_k, s_{k+1}) \in \mathcal{T}r_1$.
    - $(s_k, s_{k+1}) \in \mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_2$ and then, by (i), $s_k \in \mathit{after}(s_1)$.

Hence $(s_{k_0}, s_{k_1}) \in \mathtt{Ext}_1(s_0, s_{k_0})$ and then if $(s_{k_1}, s_{k_1+1}) \in \mathcal{Tr}_1$ then $(s_{k_1}, s_{k_1+1}) \in \mathtt{Sub}_1$. Hence k_1 satisfies (ii).

We apply the proposition 10.2 with the statement ${}^{\ell_1}stmt, \ell_2$ and $T = \{(s_k, s_{k+1}) \mid k_0 \leqslant k < k_1\}$. Therefore, if $(s_{k_1}, s_{k_1+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{Tr}_2) \setminus System$, then $(s_{k_1}, s_{k_1+1}) \notin \mathcal{Tr}_1$ and $s_{k_1} \in after(s_0)$. Hence k_1 satisfies (i).

This is a contradiction with the definition of k_1, then k_1 cannot exist and this case is not possible.

□

Lemma 11.9. *Using the above notations, for every* $(s_0, s) \in \mathtt{Reach}$ *such that* $s_0 \in \mathtt{S}_0$ *and* $s' \in \mathtt{S}'$*, there exists* $s_1 \in \mathtt{S}_1$ *such that* $(s_0, s_1) \in \mathtt{Reach}_1$*,* $(s_1, s) \in \mathtt{Reach}_2$ *and* $(s_1, s) \in \mathtt{Ext}_1(s_0, s_1)$*.*

Proof. If $(s_0, s) \in \mathtt{Reach}_1$, then, according to Lemma 10.13, $label(s) \in Labs({}^{\ell_1}cmd_1, \ell_2)$. In this case $label(s) \neq \ell_3$. This is not possible because $s \in \mathtt{S}'$.

Therefore, according to Lemma 11.8 there exists $s_1 \in \mathtt{S}_1$ such that $(s_0, s_1) \in \mathtt{Reach}_1$, $(s_1, s) \in \mathtt{Reach}_2$ and $(s_1, s) \in \mathtt{Ext}_1(s_0, s_1)$ □

Lemma 11.10. *Using the notations of this section, let* $s_0 \in \mathtt{S}_0, s_1 \in \mathtt{S}_1, s_2 \in \mathtt{S}_2, s \in \mathbf{States}$ *such that* $(s_0, s_1) \in \mathtt{Reach}_1$*,* $(s_1, s_2) \in \mathtt{Reach}_2 \cap \mathtt{Ext}_1(s_0, s_1)$ *and* $(s_2, s) \in \mathtt{Ext}(s_0, s_2)$*. Therefore* $(s_1, s) \in \mathtt{Ext}_1(s_0, s_1)$*.*

Proof. Recall that:

$\mathtt{Ext}(s_0, s_2) = \left[(\mathtt{G}_{0|after(s_0)} \cap \mathcal{Tr}) \cup \mathtt{A}_{0|\overline{after(s_0)}} \cup \mathtt{G}_{0|after(s_2)}\right]^\star$

$\mathtt{Ext}_1(s_0, s_1) = \left[(\mathtt{G}_{0|after(s_0)} \cap \mathcal{Tr}_1) \cup \mathtt{A}_{0|\overline{after(s_0)}} \cup \mathtt{G}_{0|after(s_1)}\right]^\star$

We do a proof similar to the proof of Lemma 11.8.

Let $s'_0, \ldots, s'_n$ a sequence of states such that $s_2 = s'_0$ and $s = s'_n$ and for all k, $(s'_k, s'_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{Tr}) \cup \mathtt{A}_{0|\overline{after(s_0)}} \cup \mathtt{G}_{0|after(s_2)}$.

Given Lemma 11.6, we just have to prove that for all k, if $(s_k, s_{k+1}) \in \mathtt{G}_{0|after(s_0)} \cap \mathcal{Tr}_2$ then $(s_k, s_{k+1}) \in \mathtt{G}_{0|after(s_1)}$. Hence, we just have to prove that for all k, if $(s_k, s_{k+1}) \in \mathtt{G}_{0|after(s_0)} \cap \mathcal{Tr}_2$ then $s_k \in after(s_1)$.

Either there exists a k that $(s_k, s_{k+1}) \in \mathtt{G}_{0|after(s_0)} \cap \mathcal{Tr}_2$ and $s_k \notin after(s_1)$ or there do not exist such a k.

- First case, there does not exists a such k.
- Second case there exists such a k. Let k_1 the smallest such k.

  By definition of k_1, $(s_2, s'_{k_1}) \in \mathtt{Ext}_1(s_0, s_1,)$. Given that $(s_1, s_2) \in \mathtt{Ext}_1(s_0, s_1,)$, hence $(s_1, s'_{k_1}) \in \mathtt{Ext}_1(s_0, s_1,)$.

  We apply the proposition 10.2 with the statement ${}^{\ell_1}stmt, \ell_2$ and $T = \{(s'_k, s'_{k+1}) \mid 0 \leqslant k < k_1\}$. Therefore, if $(s'_{k_1}, s'_{k_1+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{Tr}_2) \setminus System$, then $(s'_{k_1}, s'_{k_1+1}) \notin \mathcal{Tr}_1$ and $s'_{k_1} \in after(s_1)$.

  This is a contradiction with the definition of k_1, then k_1 cannot exist and this case is not possible.

□

Lemma 11.11. *Using the notations of this section, let* $s_0 \in \mathtt{S}_0, s_1 \in \mathtt{S}_1, s_2 \in \mathtt{S}_2, s$ *such that* $(s_0, s_1) \in \mathtt{Reach}_1$, $(s_1, s_2) \in \mathtt{Reach}_2 \cap \mathtt{Ext}_1(s_0, s_1)$ *and* $(s_2, s) \in \mathtt{Ext}(s_0, s_2)$. *Therefore* $(s_2, s) \in \mathtt{Ext}_2(s_1, s_2)$.

Proof. Recall that

- $\mathtt{Ext}(s_0, s_2) = \left[(\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{0|\overline{after(s_0)}} \cup \mathtt{G}_{0|after(s_2)}\right]^\star$
- $\mathtt{Ext}_2(s_1, s_2) = \left[(\mathtt{G}_{1|after(s_1)} \cap \mathcal{T}r_2) \cup \mathtt{A}_{1|\overline{after(s_1)}} \cup \mathtt{G}_{1|after(s_2)}\right]^\star$

Let $s'_0, \ldots, s'_n$ a sequence of states such that $s_2 = s'_0$ and $s = s'_n$ and for all k, $(s'_k, s'_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{0|\overline{after(s_0)}} \cup \mathtt{G}_{0|after(s_2)}$. Let us prove that for all k, $(s'_k, s'_{k+1}) \in (\mathtt{G}_{1|after(s_1)} \cap \mathcal{T}r_2) \cup \mathtt{A}_{1|\overline{after(s_1)}} \cup \mathtt{G}_{1|after(s_2)}$.
Let $k_0 \in \{0, \ldots, n\}$.

- First case $(s'_{k_0}, s'_{k_0+1}) \in \mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r_1$. According to Lemma 11.10, $(s_1, s) \in \mathtt{Ext}_1(s_0, s_1)$. Either $s'_{k_0} \in after(s_1)$ or $s'_{k_0} \notin after(s_1)$.
  - First case $s'_{k_0} \in after(s_1)$. We apply Proposition 10.2, with the statement ${}^{\ell_2}cmd_2, \ell_3$ then either $(s'_{k_0}, s'_{k_0+1}) \in \mathcal{T}r_2$ or $s'_{k_0} \in after(s_2) \cup \overline{after(s_1)}$.
    - First case, $(s'_{k_0}, s'_{k_0+1}) \in \mathcal{T}r_2$, therefore $(s'_{k_0}, s'_{k_0+1}) \in \mathcal{T}r_2 \cap \mathcal{T}r_1 = System$ (by Lemma 7.7)
    - Second case, $s'_{k_0} \in after(s_2)$, therefore $(s'_{k_0}, s'_{k_0+1}) \in \mathtt{G}_{1|after(s_2)}$.
  - Second case: $s'_{k_0} \notin after(s_1)$, therefore $s'_{k_0} \in after(s_0) \setminus after(s_1)$. According to Lemma 11.10, $(s_1, s'_{k_0}) \in \mathtt{Ext}_1(s_0, s_1)$ and therefore $(s'_{k_0}, s'_{k_0+1}) \in \mathtt{Sub}_1 \subseteq \mathtt{A}_1$, therefore $(s'_{k_0}, s'_{k_0+1}) \in \mathtt{A}_{1|\overline{after(s_1)}}$.
- Second case $(s'_{k_0}, s'_{k_0+1}) \in \mathtt{A}_{0|\overline{after(s_0)}}$. Hence $s_{k_0} \in afters_0$. By Lemma 10.2, $after(s_0) \subseteq after(s_1)$. Furthermore, by Definition 10.2 $\mathtt{A}_0 \subseteq \mathtt{A}_1$. Hence $(s'_{k_0}, s'_{k_0+1}) \in \mathtt{A}_{1|\overline{after(s_1)}}$
- Third case: $(s'_{k_0}, s'_{k_0+1}) \in \mathtt{G}_{0|after(s_2)}$. According to Definition 10.2 $\mathtt{G}_0 \subseteq \mathtt{G}_1$. Hence $(s'_{k_0}, s'_{k_0+1}) \in \mathtt{G}_{1|after(s_2)}$.

□

To prove the Proposition 11.2, we have to prove that $\mathtt{Q}_2 \geqslant \mathtt{Q}'$. We claim that

(a) $\mathtt{S}' \subseteq \mathtt{S}_2$,

(b) $\mathtt{Self}' \subseteq \mathtt{Self}_1 \cup \mathtt{Self}_2$,

(c) $\mathtt{Par}' \subseteq \mathtt{Par}_1 \cup \mathtt{Par}_2 \cup \mathtt{Sub}_1$,

(d) $\mathtt{Sub}' \subseteq \mathtt{Sub}_1 \cup \mathtt{Sub}_2$.

Using these claims and the definition of the semantics $[\![\cdot]\!]$, we conclude that $\mathtt{Q}_2 \geqslant \mathtt{Q}'$.

Now, we prove these claims:

Claim 11.12. *Using the notations of this section,* $\mathtt{S}' \subseteq \mathtt{S}_2$.

Proof. Let $s \in \mathtt{S}'$, so there exists $s_0 \in \mathtt{S}$ such that $(s_0, s) \in \mathtt{Reach}'$ and $\mathit{label}(s) = \ell_3$. According to Lemma 11.9 there exists $s_1 \in \mathtt{S}_1$ such that $(s_1, s) \in \mathtt{Reach}_2$. Therefore $s \in \mathtt{S}_2$. □

Claim 11.13. *Using the notations of this section,* $\mathtt{Self}' \subseteq \mathtt{Self}_1 \cup \mathtt{Self}_2$.

Proof. Let $(s, s') \in \mathtt{Self}'$. So $(s, s') \in \mathcal{Tr}$, and there exists $s_0 \in \mathtt{S}$ such that $(s_0, s) \in \mathtt{Reach}'$.

According to Lemma 11.8 either $(s_0, s) \in \mathtt{Reach}_1$ and $\mathit{label}(s) \neq \ell_2$, or there exists $s_1 \in \mathtt{S}_1$ such that $(s_0, s_1) \in \mathtt{Reach}_1$ and $(s_1, s) \in \mathtt{Reach}_2$.

- In the first case, according to Lemma 10.13, $\mathit{label}(s) \in \mathit{Labs}({}^{\ell_1}\mathit{cmd}_1, \ell_2)$. Since $\mathit{label}(s) \neq \ell_2$ and by Lemma 11.7, $(s, s') \in \mathcal{Tr}_1$. Hence, by definition, $(s, s') \in \mathtt{Self}_1$
- In the second case, by Lemma 10.12, $\mathit{label}(s') \in \mathit{Labs}({}^{\ell_2}\mathit{cmd}_2, \ell_3)$. Since $(s, s') \in \mathcal{Tr}$, by Lemma 11.7 $(s, s') \in \mathcal{Tr}_2$. Given that $s \in \mathtt{Reach}\langle \mathtt{S}_1 \rangle$ and $(s, s') \in \mathcal{Tr}_2$, we conclude that $(s, s') \in \mathtt{Self}_2$.

□

Claim 11.14. *Using the notations of this section* $\mathtt{Par}' \subseteq \mathtt{Par}_1 \cup \mathtt{Par}_2 \cup \mathtt{Sub}_1$.

Proof. Let $(s, s') \in \mathtt{Par}'$. Therefore, $(s, s') \in \mathcal{Tr}$ and there exists $s_0 \in \mathtt{S}_0$ and s_2 such that $(s_0, s_2) \in \mathtt{Reach}'$, $(s_2, s) \in \mathit{Schedule}$ and $s \in \mathit{after}(s_0)$. According to Lemma 11.8 there are two cases:

- First case: $(s_0, s_2) \in \mathtt{Reach}_1$ and $\mathit{label}(s_2) \neq \ell_2$. Then, using the fact that $\mathit{System} \subseteq \mathcal{Tr}_1$, $(s_0, s) \in (\mathcal{Tr}_1 \cup \mathtt{A}_{0|\overline{\mathit{after}(s_0)}})^\star$. Because $s \in \mathit{after}(s_0)$, by Lemma 10.12, $\mathit{label}(s) \in \mathit{Labs}({}^{\ell_1}\mathit{cmd}_1, \ell_2) \setminus \{\ell_2\}$. Hence, according to Lemma 11.7, $(s, s') \in \mathcal{Tr}_1$. We conclude that $(s, s') \in \mathtt{Par}_1$.
- Second case: There exists $s_1 \in \mathtt{S}_1$ such that $(s_0, s_1) \in \mathtt{Reach}_1$, $(s_1, s_2) \in \mathtt{Reach}_2$ and $(s_1, s_2) \in \mathtt{Ext}_1(s_0, s_1)$. Hence $(s_1, s) \in \mathtt{Ext}_1(s_0, s_1); \mathit{Schedule} = \mathtt{Ext}_1(s_0, s_1)$.

  Either $s \in \mathit{after}(s_1)$ or $s \notin \mathit{after}(s_1)$.

  - If $s \in \mathit{after}(s_1)$, then, because $(s_1, s) \in \mathtt{Reach}_2; \mathit{Schedule}$, by Lemma 10.12, $\mathit{label}(s) \in \mathit{Labs}({}^{\ell_2}\mathit{cmd}_2, \ell_3)$. So, in this case, by Lemma 11.7, $(s, s') \in \mathcal{Tr}_2$ and then $(s, s') \in \mathtt{Par}_2$.
  - We consider the case $s \notin \mathit{after}(s_1)$. Given that $(s_0, s_1) \in \mathtt{Reach}$, $(s_1, s) \in \mathtt{Ext}_1(s_1, s_2)$, so by Proposition 10.2, $(s, s') \in \mathcal{Tr}_1$. Hence, $(s, s') \in \mathtt{Sub}_1$.

□

Claim 11.15. *Using the notations of this section* $\mathtt{Sub}' \subseteq \mathtt{Sub}_1 \cup \mathtt{Sub}_2$.

Proof. Let $(s, s') \in \mathtt{Sub}'$. Then, there exists s_0 and s_2 such that $(s_0, s_2) \in \mathtt{Reach}'$ and $(s_2, s) \in \mathtt{Ext}(s_0, s_2)$. According to Lemma 11.9, there exists $s_1 \in \mathtt{S}_1$ such that $(s_0, s_1) \in \mathtt{Reach}_1$ and $(s_1, s_2) \in \mathtt{Reach}_2$ and $(s_1, s_2) \in \mathtt{Ext}_1(s_0, s_1)$.

By Lemma 11.10 and Lemma 11.11, $(s_1, s) \in \mathtt{Ext}_1(s_0, s_1)$ and $(s_2, s) \in \mathtt{Ext}_2(s_1, s_2)$. Either $s \notin \mathit{after}(s_1)$ or $s \in \mathit{after}(s_1)$.

- First case: $s \notin \mathit{after}(s_1)$. Because $s \in \mathit{after}(s_0)$, then $s \in \mathit{after}(s_0) \setminus \mathit{after}(s_1)$. Furthermore, given that $(s_0, s_1) \in \mathtt{Reach}_1$ and $(s_1, s) \in \mathtt{Reach}_2$, by Proposition 10.2, $(s, s') \in \mathcal{T}r_1$. We conclude that $(s, s') \in \mathtt{Sub}_1$.
- Second case: $s \in \mathit{after}(s_1)$. Because $s \in \mathit{after}(s_0) \setminus \mathit{after}(s_2)$, $s \in \mathit{after}(s_1) \setminus \mathit{after}(s_2)$. By Lemma 10.12, $\mathit{label}(s) \in \mathit{Labs}({}^{\ell_2}\mathit{cmd}_2, \ell_2)$. Hence, by Lemma 11.7, $(s, s') \in \mathcal{T}r_2$ and therefore, $(s, s') \in \mathtt{Sub}_2$.

□

Proposition 11.2. *For each concrete configuration* $\mathtt{Q}$*:*
$[\![{}^{\ell_1}\mathit{cmd}_1; {}^{\ell_2}\mathit{cmd}_2, \ell_3]\!](\mathtt{Q}) \leqslant [\![{}^{\ell_2}\mathit{cmd}_2, \ell_3]\!] \circ [\![{}^{\ell_1}\mathit{cmd}_1, \ell_2]\!](\mathtt{Q})$.

Proof. This is a consequence of Definition 10.2 and of Claims 11.12, 11.13, 11.14 and 11.15. □

11.3 *if* statements

In this section, we consider a command ${}^{\ell_1}\mathit{if}(\mathit{cond})\mathit{then}\{{}^{\ell_2}\mathit{cmd}_1\}\mathit{else}\{{}^{\ell_3}\mathit{cmd}_2\}, \ell_4$ and an initial configuration $\mathtt{Q}_0 = \langle \mathtt{S}_0, \mathtt{G}_0, \mathtt{A}_0 \rangle$

Let $\langle \mathtt{S}', \mathtt{G}', \mathtt{A}' \rangle = [\![{}^{\ell_1}\mathit{if}(\mathit{cond})\mathit{then}\{{}^{\ell_2}\mathit{cmd}\}\mathit{else}\{{}^{\ell_3}\mathit{cmd}\}, \ell_4]\!]\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$.
Let $[\mathtt{Reach}', \mathtt{Ext}', \mathtt{Self}', \mathtt{Par}', \mathtt{Sub}'] = \left\{\!\!\left|{}^{\ell_1}\mathit{if}(\mathit{cond})\mathit{then}\{{}^{\ell_2}\mathit{cmd}\}\mathit{else}\{{}^{\ell_3}\mathit{cmd}\}, \ell_4\right|\!\!\right\}\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$.
Let $\langle \mathtt{S}_+, \mathtt{G}_+, \mathtt{A}_+ \rangle = [\![{}^{\ell_1}\mathit{guard}\,\mathit{cond}, \ell_2]\!]\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$.
Let $[\mathtt{Reach}_+, \mathtt{Ext}_+, \mathtt{Self}_+, \mathtt{Par}_+, \mathtt{Sub}_+] = \left\{\!\!\left|{}^{\ell_1}\mathit{guard}\,\mathit{cond}, \ell_2\right|\!\!\right\}\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$.
Let $\langle \mathtt{S}_1, \mathtt{G}_1, \mathtt{A}_1 \rangle = [\![{}^{\ell_2}\mathit{cmd}_1, \ell_4]\!]\langle \mathtt{S}_+, \mathtt{G}_+, \mathtt{A}_+ \rangle$.
Let $[\mathtt{Reach}_1, \mathtt{Ext}_1, \mathtt{Self}_1, \mathtt{Par}_1, \mathtt{Sub}_1] = \left\{\!\!\left|{}^{\ell_2}\mathit{cmd}_1, \ell_4\right|\!\!\right\}\langle \mathtt{S}_+, \mathtt{G}_+, \mathtt{A}_+ \rangle$.
Let $\langle \mathtt{S}_\neg, \mathtt{G}_\neg, \mathtt{A}_\neg \rangle = [\![{}^{\ell_1}\mathit{guard}\neg\mathit{cond}, \ell_3]\!]\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$.
Let $[\mathtt{Reach}_\neg, \mathtt{Ext}_\neg, \mathtt{Self}_\neg, \mathtt{Par}_\neg, \mathtt{Sub}_\neg] = \left\{\!\!\left|{}^{\ell_1}\mathit{guard}\neg\mathit{cond}, \ell_3\right|\!\!\right\}\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$.
Let $\langle \mathtt{S}_2, \mathtt{G}_2, \mathtt{A}_2 \rangle = [\![{}^{\ell_3}\mathit{cmd}_2, \ell_4]\!]\langle \mathtt{S}_\neg, \mathtt{G}_\neg, \mathtt{A}_\neg \rangle$.
Let $[\mathtt{Reach}_2, \mathtt{Ext}_2, \mathtt{Self}_2, \mathtt{Par}_2, \mathtt{Sub}_2] = \left\{\!\!\left|{}^{\ell_3}\mathit{cmd}_2, \ell_4\right|\!\!\right\}\langle \mathtt{S}_\neg, \mathtt{G}_\neg, \mathtt{A}_\neg \rangle$.
Let $\mathcal{T}r = \mathcal{T}r_{{}^{\ell_1}\mathit{if}(\mathit{cond})\mathit{then}\{{}^{\ell_2}\mathit{cmd}_1\}\mathit{else}\{{}^{\ell_3}\mathit{cmd}_2\}, \ell_4}$.
Let $\mathcal{T}r_+ = \mathcal{T}r_{{}^{\ell_1}\mathit{if}(\mathit{cond})\mathit{then}\{{}^{\ell_2}\mathit{cmd}\}\mathit{else}\{{}^{\ell_3}\mathit{cmd}\}, \ell_4}$.
Let $\mathcal{T}r_\neg = \mathcal{T}r_{{}^{\ell_1}\mathit{if}(\mathit{cond})\mathit{then}\{{}^{\ell_2}\mathit{cmd}_1\}\mathit{else}\{{}^{\ell_3}\mathit{cmd}_2\}, \ell_4}$.
Let $\mathcal{T}r_1 = \mathcal{T}r_{{}^{\ell_2}\mathit{cmd}_1, \ell_4}$.
Let $\mathcal{T}r_2 = \mathcal{T}r_{{}^{\ell_3}\mathit{cmd}_2, \ell_4}$.

Lemma 11.16. $\mathcal{T}r_{\ell_1\, \mathit{if}(cond)\mathit{then}\{^{\ell_2}\, cmd\}\mathit{else}\{^{\ell_3}\, cmd\},\ell_4} = \mathcal{T}r_{\ell_1\, \mathit{guard}\, cond,\ell_2} \cup \mathcal{T}r_{\ell_2\, cmd_1,\ell_4} \cup \mathcal{T}r_{\ell_1\, \mathit{guard}\, \neg\, cond,\ell_3} \cup \mathcal{T}r_{\ell_3\, cmd_1,\ell_4}$.

Lemma 11.17. *If* $(s_0, s) \in \mathtt{Reach}$ *and* $s_0 \in \mathtt{S}_0$*, then, one of the three following properties hold:*

1. $s \in \mathtt{interfere}_{\mathtt{A}_0}(\{s_0\})$,
2. *or there exists* $s_1 \in \mathtt{S}_+$ *such that* $(s_1, s) \in \mathtt{Reach}_1 \cap \mathtt{Ext}_+(s_0, s_1)$
3. *or there exists* $s_1 \in \mathtt{S}_\neg$ *such that* $(s_1, s) \in \mathtt{Reach}_2 \cap \mathtt{Ext}_\neg(s_0, s_1)$

Proof. There exists a sequence of states $s'0, \ldots, s'_n$ such that $s'_0 = s_0$ and $s'_n = s$ for all k, $(s'_k, s'_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{0|\overline{after(s_0)}}$.

Either there exists a k such that $(s'_k, s'_{k+1}) \in (\mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r) \setminus \mathit{System}$ either such a k does not exists.

- First case: there does not exists such a k. Hence $s \in \mathtt{interfere}_{\mathtt{A}_0}(\{s_0\})$.
- Second case: there is such a k: let k_0 the smallest such k.

  Because $(s'_{k_0}, s'_{k_0+1}) \in \mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r$, $s'_{k_0} \in after(s_0)$. By Lemma 10.8, $thread(s_0) = thread(s'_{k_0})$. By Lemma 10.10, $label(s_0) = label(s'_0) = \ell_1$. Therefore, due to Lemmas 7.7 and 11.16, $(s'_{k_0}, s'_{k_0+1}) \in \mathcal{T}r_{\ell_1\, \mathit{guard}\, cond,\ell_2} \cup \mathcal{T}r_{\ell_1\, \mathit{guard}\, \neg\, cond,\ell_3}$.

  Either $(s'_{k_0}, s'_{k_0+1}) \in \mathcal{T}r_{\ell_1\, \mathit{guard}\, cond,\ell_2}$ or $(s'_{k_0}, s'_{k_0+1}) \in \mathcal{T}r_{\ell_1\, \mathit{guard}\, \neg\, cond,\ell_3}$.

  - In the first case, $(s'_{k_0}, s'_{k_0+1}) \in \mathcal{T}r_{\ell_1\, \mathit{guard}(cond),\ell_2}$. By Lemma 7.6, $thread(s'_{k_0}) = thread(s'_{k_0+1})$ and $label(s'_{k_0+1}) = \ell_2$. Therefore, $(s_0, s'_{k_0+1}) \in \mathtt{Reach}_+$ and $s'_{k_0+1} \in \mathtt{S}_+$.

    Let us prove, that $\forall k \in \{k_0 + 1, \ldots, n\}$, $(s'_k, s'_{k+1}) \in (\mathtt{G}_{0|after(s_{k_0+1})} \cap \mathcal{T}r_{\ell_2\, cmd,\ell_4}) \cup \mathtt{A}_{0|\overline{after(s_0)}}$.

    Assume by contradiction that there exists a k such that $(s'_k, s'_{k+1}) \notin (\mathtt{G}_{0|after(s_{k_0+1})} \cap \mathcal{T}r_{\ell_2\, cmd,\ell_4}) \cup \mathtt{A}_{0|\overline{after(s_0)}}$. Let k_1 the smallest such k. Therefore $(s'_{k_1}, s'_{k_1+1}) \in \mathtt{G}_{0|after(s_0)} \cap \mathcal{T}r$. Hence $s'_{k_1} \in after(s_0)$. By minimality of k_1, we can apply Proposition 10.2. Hence, either $s'_{k_1} \in after(s'_{k_0+1})$ or $(s'_{k_1}, s'_{k_1+1}) \in \mathcal{T}r_+$.

    - First case $s_{k_1} \in after(s_{k_0})$. Therefore $(s_{k_1}, s_{k_1+1}) \in \mathtt{G}_{0|after(s_{k_0})} \cap \mathcal{T}r$. This is in contradiction with the definition of k_1. This case is not possible.
    - Second case $(s'_{k_1}, s'_{k_1+1}) \in \mathcal{T}r_+$ and $(s_{k_1} \in after(s_0) \setminus after(s_1)$. Hence (s'_{k_1}, s'_{k_1+1}). By minimality of k_1, $(s_{k_0+1}, s_{k_1}) \in \mathtt{Ext}_+(s_0, s_{k_0+1})$. Hence $(s_{k_0+1}, s_{k_1}) \in \mathtt{Sub}_+$. Nevertheless, according to Claim 11.3, $\mathtt{Sub}_+ = \emptyset$. There is a contradiction, this case is not possible.

    Given that $\mathtt{G}_1 \supset \mathtt{G}_0$ and $\mathtt{A}_1 \supset \mathtt{A}_0$ and, given that $s'_{k_0} \in after(s_0)$, by Lemma 10.2 $after(s'_{k_0+1}) \subseteq after(s_0)$, we conclude that $(s_{k_0+1}, s) \in \mathtt{Reach}_1 \cap \mathtt{Ext}_+(s_0, s_{k_0+1})$.
  - In the second case, $(s'_{k_0}, s'_{k_0+1}) \in \mathcal{T}r_{\ell_1\, \mathit{guard}(\neg\, cond),\ell_3}$. The proof is similar to the first case. We are in the case 3. of the lemma.

□

Claim 11.18. $\mathtt{S}' \subseteq \mathtt{S}_1 \cup \mathtt{S}_2$

Proof. Let $s \in \mathtt{S}'$. Therefore there exists $s_0 \in \mathtt{S}_0$ such that $(s_0, s) \in \mathtt{Reach}$ and $\mathit{label}(s) = \ell_4 \neq \ell_1$. Hence, due to Lemma 10.10, $s \notin \mathtt{interfere}_{\mathtt{A}_0}\{s_0\}$.

According to Lemma 11.17, there exists s_1 such that either (1) $s_1 \in \mathtt{S}_+$ and $(s_1, s) \in \mathtt{Reach}_1 \cap \mathtt{Ext}_+(s_0, s_1)$, (2) or, $s_1 \in \mathtt{S}_\neg$ and $(s_1, s) \in \mathtt{Reach}_2 \cap \mathtt{Ext}_\neg(s_0, s_1)$.

In the first case, by definition, $s \in \mathtt{S}_1$ and in the second case $s \in \mathtt{S}_2$ □

Claim 11.19. $\mathtt{Self} \subseteq \mathtt{Self}_+ \cup \mathtt{Self}_1 \cup \mathtt{Self}_\neg \cup \mathtt{Self}_2$.

Proof. Let $(s, s') \in \mathtt{Self}$. Then, there exists $s_0 \in \mathtt{S}_0$ such that $(s_0, s) \in \mathtt{Reach}$.

According to Lemma 11.17, there is three cases:

- First case $s \in \mathtt{interfere}_{\mathtt{A}_0}(\{s_0\})$. By Lemma 10.10, $\mathit{label}(s) = \ell_1$. Hence, by Lemmas 7.7 and 11.16, $(s, s') \in \mathcal{T}r_{\ell_1\, \mathsf{guard}\, cond, \ell_2} \cup \mathcal{T}r_{\ell_1\, \mathsf{guard}\neg\, cond, \ell_3}$. Hence, $(s, s') \in \mathtt{Self}_+ \cup \mathtt{Self}_\neg$.
- Second case there exists s_1 such that $s_1 \in \mathtt{S}_+$ and $(s_1, s) \in \mathtt{Reach}_1 \cap \mathtt{Ext}_+(s_0, s_1)$. Hence, by Lemmas 7.7 and 11.16, $(s_1, s_k) \in \mathcal{T}r_{\ell_2\, cmd_1, \ell_4}$ and therefore $(s, s') \in \mathtt{Self}_1$.
- Third case there exists s_1 such that $s_1 \in \mathtt{S}_\neg$ and $(s_1, s) \in \mathtt{Reach}_2 \cap \mathtt{Ext}_\neg(s_0, s_1)$. Hence, by Lemmas 7.7 and 11.16, $(s_1, s_k) \in \mathcal{T}r_{\ell_3\, cmd_2, \ell_4}$ and therefore $(s, s') \in \mathtt{Self}_2$.

□

Claim 11.20. $\mathtt{Par} \subseteq \mathtt{Par}_1 \cup \mathtt{Par}_2$.

Proof. Let $(s, s') \in \mathtt{Par}$. Therefore, there exists $s_0 \in \mathtt{S}_0$ and s_2 such that $(s_0, s_2) \in \mathtt{Reach}$ and $(s_2, s) \in \mathit{System}$ and $s \in \mathit{after}(s_0)$. Notice that $\mathit{thread}(s_0) = \mathit{thread}(s_2) \neq \mathit{thread}(s)$.

According to Lemma 11.17, there is three cases:

- First case, $s_2 \in \mathtt{interfere}_{\mathtt{A}_0}(\{s_0\})$. Hence, due to Lema 10.8, $\mathit{thread}(s) = \mathit{thread}(s_0)$. This is contradictory.
- Second case: there exists s_1 such that $s_1 \in \mathtt{S}_+$ and $(s_1, s) \in \mathtt{Reach}_1 \cap \mathtt{Ext}_+(s_0, s_1)$. By Lemma 10.12, $\mathit{label}(s) \in \mathit{Labs}({}^{\ell_2}cmd_1, \ell_4)$ and therefore, by Lemmas 11.16 and 7.7, $(s, s') \in \mathcal{T}r_{\ell_2\, cmd_1, \ell_4}$. Hence, $(s, s') \in \mathtt{Par}_1$.
- Third case: there exists s_1 such that $s_1 \in \mathtt{S}_-$ and $(s_1, s) \in \mathtt{Reach}_1 \cap \mathtt{Ext}_-(s_0, s_1)$. By Lemma 10.12, $\mathit{label}(s) \in \mathit{Labs}({}^{\ell_3}cmd_2, \ell_4)$ and therefore, by Lemmas 11.16 and 7.7, $(s, s') \in \mathcal{T}r_{\ell_3\, cmd_2, \ell_4}$. Hence, $(s, s') \in \mathtt{Par}_2$.

□

Claim 11.21. $\mathtt{Sub} \subseteq \mathtt{Sub}_1 \cup \mathtt{Sub}_2$.

Proof. Let $(s, s') \in \mathtt{Sub}$. Therefore, there exists $s_0 \in \mathtt{S}_0$ and $s_2 \in \mathtt{S}'$ such that $(s_0, s_2) \in \mathtt{Reach}$ and $(s_2, s) \in \mathtt{Ext}(s_0, s_2)$ and $s \in \mathit{after}(s_0) \setminus \mathit{after}(s_2)$. Notice that $\mathit{thread}(s_0) = \mathit{thread}(s_2) \neq \mathit{thread}(s)$.

According to Lemma 11.17 there are three cases:

- First case: $s_2 \in \mathtt{interfere}(\{s_0\})$. Hence, due to Lemma 10.10, $\mathit{label}(s_2) = \ell_1$. This is contradictory with $s_2 \in \mathtt{S}'$.
- Second case: there exists s_1 such that either $s_1 \in \mathtt{S}_+$ and $(s_1, s) \in \mathtt{Reach}_1 \cap \mathtt{Ext}_+(s_0, s_1)$. By Lemma 10.9, $s \in \mathit{after}(s_1)$. By Lemma 10.12, $\mathit{label}(s) \in \mathit{Labs}({}^{\ell_2}\mathit{cmd}_1, \ell_4)$. Because $s \notin \mathit{after}(s_2)$, by Proposition 10.2, $(s, s') \in \mathcal{T}r_{{}^{\ell_1}\mathit{cmd}_1, \ell_2}$. Hence, $(s, s') \in \mathtt{Sub}_1$.
- Third case: there exists s_1 such that either $s_1 \in \mathtt{S}_-$ and $(s_1, s) \in \mathtt{Reach}_2 \cap \mathtt{Ext}_-(s_0, s_1)$. This case is very similar to the second case:

  By Lemma 10.9, $s \in \mathit{after}(s_1)$. By Lemma 10.12, $\mathit{label}(s) \in \mathit{Labs}({}^{\ell_3}\mathit{cmd}_2, \ell_4)$. Because $s \notin \mathit{after}(s_2)$, by Proposition 10.2, $(s, s') \in \mathcal{T}r_{{}^{\ell_1}\mathit{cmd}_1, \ell_2}$. Hence, $(s, s') \in \mathtt{Sub}_1$

□

Proposition 11.3. *For all concrete configuration* $\mathtt{Q}$*:*

$$[\![{}^{\ell_1}\mathit{if}((\mathit{cond})\mathit{then}\{{}^{\ell_2}\mathit{cmd}_1\}\mathit{else}\{{}^{\ell_4}\mathit{cmd}_2\}, \ell_3]\!](\mathtt{Q}) \leqslant \begin{array}{l}[\![{}^{\ell_2}\mathit{cmd}_1, \ell_3]\!] \circ [\![{}^{\ell_1}\mathit{guard}(\mathit{cond}), \ell_2]\!](\mathtt{Q}) \\ \sqcup [\![{}^{\ell_4}\mathit{cmd}_2, \ell_3]\!] \circ [\![{}^{\ell_1}\mathit{guard}(\neg\mathit{cond}), \ell_4]\!](\mathtt{Q})\end{array}$$

11.4 While loops

In this section, we consider a command ${}^{\ell_1}\mathit{while}(\mathit{cond})\{{}^{\ell_2}\mathit{cmd}\}, \ell_3$ and an initial configuration $\mathtt{Q}_0 = \langle \mathtt{S}_0, \mathtt{G}_0, \mathtt{A}_0 \rangle$.
Let $\mathtt{Q}' = \langle \mathtt{S}', \mathtt{G}', \mathtt{A}' \rangle = [\![{}^{\ell_1}\mathit{while}(\mathit{cond})\{{}^{\ell_2}\mathit{cmd}\}, \ell_3]\!]\mathtt{Q}_0$.
Let $\mathtt{Q}_\omega = \langle \mathtt{S}_\omega, \mathtt{G}_\omega, \mathtt{A}_\omega \rangle = \mathtt{loop}^{\uparrow\omega}(\mathtt{Q}_0)$.
Let $\mathtt{Q}'' = \langle \mathtt{S}'', \mathtt{G}'', \mathtt{A}'' \rangle = [\![{}^{\ell_1}\mathit{while}(\mathit{cond})\{{}^{\ell_2}\mathit{cmd}\}, \ell_3]\!]\mathtt{Q}_\omega$.
Let $\mathtt{K} = [\mathtt{Reach}, \mathtt{Ext}, \mathtt{Self}, \mathtt{Par}, \mathtt{Sub}] = \left\{\!\!\left|{}^{\ell_1}\mathit{while}(\mathit{cond})\{{}^{\ell_2}\mathit{cmd}\}, \ell_3\right|\!\!\right\}\mathtt{Q}_\omega$.
Let $\mathtt{Q}_+ = \langle \mathtt{S}_+, \mathtt{G}_+, \mathtt{A}_+ \rangle = [\![{}^{\ell_1}\mathit{guard}(\mathit{cond}), \ell_2]\!](\mathtt{Q}_\omega)$.
Let $\mathtt{K}_+ = [\mathtt{Reach}_+, \mathtt{Ext}_+, \mathtt{Self}_+, \mathtt{Par}_+, \mathtt{Sub}_+] = \left\{\!\!\left|{}^{\ell_1}\mathit{guard}(\mathit{cond}), \ell_2\right|\!\!\right\}(\mathtt{Q}_\omega)$.
Let $\mathtt{K}_{\mathit{cmd}} = [\mathtt{Reach}_{\mathit{cmd}}, \mathtt{Ext}_{\mathit{cmd}}, \mathtt{Self}_{\mathit{cmd}}, \mathtt{Par}_{\mathit{cmd}}, \mathtt{Sub}_{\mathit{cmd}}] = \left\{\!\!\left|{}^{\ell_2}\mathit{cmd}, \ell_1\right|\!\!\right\}(\mathtt{Q}_+)$.
Let $\mathtt{Q}_\neg = \langle \mathtt{S}_\neg, \mathtt{G}_\neg, \mathtt{A}_\neg \rangle = [\![{}^{\ell_1}\mathit{guard}(\neg\mathit{cond}), \ell_3]\!]\mathtt{Q}_\omega$.
Let $\mathtt{K}_\neg = [\mathtt{Reach}_\neg, \mathtt{Ext}_\neg, \mathtt{Self}_\neg, \mathtt{Par}_\neg, \mathtt{Sub}_\neg] = \left\{\!\!\left|{}^{\ell_1}\mathit{guard}(\neg\mathit{cond}), \ell_3\right|\!\!\right\}\mathtt{Q}_\omega$.
Let $\mathcal{T}r = \mathcal{T}r_{{}^{\ell_1}\mathit{while}(\mathit{cond})\{{}^{\ell_2}\mathit{cmd}\}, \ell_3}$.

Lemma 11.22.

$$\mathcal{T}r_{{}^{\ell_1}\mathit{while}(\mathit{cond})\{{}^{\ell_2}\mathit{cmd}\}, \ell_3} = \mathcal{T}r_{{}^{\ell_1}\mathit{guard}(\neg\mathit{cond}), \ell_3} \cup \mathcal{T}r_{{}^{\ell_1}\mathit{guard}(\mathit{cond}), \ell_2} \cup \mathcal{T}r_{{}^{\ell_2}\mathit{cmd}, \ell_1}$$

Notice that, by definition, $\mathtt{Q}_0 \leqslant \mathtt{Q}_\omega$

Lemma 11.23. *We use the above notations. Let $s_0, s_1, \dots, s_n, \dots, s_m$ a sequence of states such that for all $k \in \{0, \dots, m-1\}$, $(s_k, s_{k+1}) \in (\mathtt{G}_{\omega|after(s_0)} \cap \mathcal{Tr}) \cup \mathtt{A}_{\omega|\overline{after(s_0)}}$.*

If $(s_0, s_m) \in \mathtt{Reach}_\omega$, $(s_0, s_n) \in \mathtt{Reach}_\omega$ and $s_n \in \mathtt{S}_\omega$ then for all $k \geqslant n$, $(s_k, s_{k+1}) \in (\mathtt{G}_{\omega|after(s_n)} \cap \mathcal{Tr}) \cup \mathtt{A}_{\omega|\overline{after(s_n)}}$.

Proof. For all k, $(s_k, s_{k+1}) \in (\mathtt{G}_{\omega|after(s_n)} \cap \mathcal{Tr}) \cup (\mathtt{G}_{\omega|after(s_0) \setminus after(s_n)} \cap \mathcal{Tr}) \cup \mathtt{A}_{\omega|\overline{after(s_0)}}$.

Let $k_0 \geqslant n$ such that $(s_{k_0}, s_{k_0+1}) \in (\mathtt{G}_{\omega|after(s_0) \setminus after(s_n)} \cap \mathcal{Tr})$. Notice that $(s_n, s_{k_0}) \in \mathtt{Ext}_\omega(s_0, s_n)$ and $s_{k_0} \in after(s_0) \setminus after(s_n)$. Hence, $(s_{k_0}, s_{k_0+1}) \in \mathtt{Sub}_\omega \subseteq \mathtt{A}_\omega$. Therefore $(s_{k_0}, s_{k_0+1}) \in \mathtt{A}_{\omega|\overline{after(s_1)}}$.

In addition to this, according to Lemma 10.13, $after(s_n) \subseteq after(s_0)$, so, for all $k \geqslant n$, $(s_k, s_{k+1}) \in (\mathtt{G}_{\omega|after(s_n)} \cap \mathcal{Tr}) \cup \mathtt{A}_{\omega|\overline{after(s_n)}}$. □

Lemma 11.24. *Using the notations of this section, given $s_0 \in \mathtt{S}_\omega$ and $s \in$ **States**, if $(s_0, s) \in \mathtt{Reach}$, therefore there exists $s'_0 \in \mathtt{S}_\omega$ such that $(s_0, s'_0) \in \mathtt{Reach}$ and*

1. *either $(s'_0, s) \in \mathtt{Reach}_\neg$,*
2. *or there exists $s'_1 \in \mathtt{S}_+$ such that $(s'_0, s'_1) \in \mathtt{Reach}_+$ and $(s'_1, s) \in \mathtt{Reach}_{cmd}$ and $label(s) \neq \ell_1$.*

Proof. There exists a sequence $s_0, \dots, s_n$ such that for all $k \in \{0, \dots, n-1\}$, $(s_k, s_{k+1}) \in (\mathtt{G}_{\omega|after(s_0)} \cap \mathcal{Tr}) \cup \mathtt{A}_{\omega|\overline{after(s_0)}}$ and $s_n = s$.

Let k_0 the biggest k such that the following properties hold:

(1) $s_k \in \mathtt{S}_\omega$,

(2) for all $k' \in \{k, \dots, n-1\}$, $(s_{k'}, s_{k'+1}) \in (\mathtt{G}_{\omega|after(s_{k_0})} \cap \mathcal{Tr}) \cup \mathtt{A}_{\omega|\overline{after(s_{k_0})}}$,

Such a k exists because 0 satisfy properties (1) and (2). By definition of the sequence $s_0, \dots, s_n$, $(s_0, s'_{k_0}) \in \mathtt{Reach}$.

Either there exists $k \in \{k_0, \dots, n-1\}$ such that $(s_k, s_{k+1}) \in \mathtt{G}_{\omega|after(s_0)} \cap \mathcal{Tr}_{\ell_1\, \mathsf{while}(cond)\{^{\ell_2}\, cmd\}, \ell_3} \setminus System$ or such a k does not exist.

- First case, such a k does not exist. Therefore for all $k \in \{k_0, \dots, n-1\}$, $(s_k, s_{k+1}) \in System \cup \mathtt{A}_{\omega|\overline{after(s_0)}}$. Hence $(s_0, s) \in \mathtt{Reach}_+ \cap \mathtt{Reach}_\neg \subseteq \mathtt{Reach}_\neg$.
- Second case : such a k exists. Let k_1 the smallest such k.

  Therefore $(s_{k_1}, s_{k_1+1}) \in \mathtt{G}_{\omega|after(s_{k_0})}$, so, $s_{k_1} \in after(s_0)$. According to Lemma 10.8, $thread(s_{k_0}) = thread(s_{k_1})$. By Lemma 10.10, $label(s_{k_0}) = label(s_{k_1})$. But $label(s_{k_0}) = \ell_1$, therefore, by Lemma 7.8, $(s_{k_1}, s_{k_1+1}) \notin \mathcal{Tr}_{\ell_2\, cmd, \ell_1}$. Therefore, by Lemma 11.22, either $(s_{k_1}, s_{k_1+1}) \in \mathcal{Tr}_{\ell_1\, \mathsf{guard}(\neg cond), \ell_3}$ or $(s_{k_1}, s_{k_1+1}) \in \mathcal{Tr}_{\ell_1\, \mathsf{guard}(cond), \ell_2}$.

  - First case: $(s_{k_1}, s_{k_1+1}) \in \mathcal{Tr}_{\ell_1\, \mathsf{guard}(\neg cond), \ell_3}$. By Lemma 7.6, $label(s_{k_1+1}) = \ell_3$. Either there exists $k \geqslant k_0$ such that $(s_k, s_{k+1}) \notin \mathtt{A}_{\omega|\overline{after(s_0)}} \cup System$ or not.

- First case: such a k exists. Let k_2 be the smallest such k.
  By minimality of k_2, $(s_{k_1}, s_{k_2}) \in [\mathtt{A}_{\omega_{|\overline{after(s_0)}}} \cup System]^\star$. By definition of k_2, $(s_{k_2}, s_{k_2+1}) \in \mathtt{G}_{\omega_{|after(s_0)}} \cap \mathcal{T}r$. Therefore $s_{k_2} \in after(s_{k_0})$, then by Lemma 10.8, $thread(s_{k_2}) = thread(s_{k_1+1})$. By Lemma 10.10, $label(s_k) = label(s_{k_1+1}) = \ell_3$. So, by Lemma 7.8, $(s_{k_2}, s_{k+1}) \in System$. This is contradictory with the definition of k_2.
- Second case: for all $k > k_1$, $(s_k, s_{k+1}) \in \mathtt{A}_{\omega_{|\overline{after(s_{k_0})}}} \cup System$. Hence $(s_{k_0}, s) \in [\mathtt{A}_{\omega_{|\overline{after(s_{k_0})}}} \cup System]^\star$ and then $(s_{k_0}, s) \in \mathtt{Reach}_\neg$.

- Second case: $(s_{k_1}, s_{k_1+1}) \in \mathcal{T}r^{\ell_1}_{\mathit{guard}(cond),\ell_2}$.
  Therefore, by Lemma 7.6, $s_{k_1+1} \in \mathtt{S}_+$. Either there exists $k_3 > k_1$ such that $(s_{k_3}, s_{k_3+1}) \in \mathtt{G}_{|after(s_{k_0})} \cap (\mathcal{T}r^{\ell_1}_{\mathit{guard}(\neg cond),\ell_3} \cup \mathcal{T}r^{\ell_1}_{\mathit{guard}(cond),\ell_3})$ or there does not exist such a k_3.
  - First case: such a k_3 exists, therefore, by Lemma 7.6, $label(s_{k_3}) = \ell_1$. According to Lemma 10.12, $thread(s) = thread(s_{k_0})$. Hence, $(s_{k_0}, s_{k_3}) \in \mathtt{Reach}_\omega$.
    So, by Lemma 11.23, for all $k \in \{k_2, \ldots, n-1\}$, $(s_k, s_{k+1}) \in (\mathtt{G}_{\omega_{|after(s_{k_2})}} \cap \mathcal{T}r) \cup \mathtt{A}_{\omega_{|\overline{after(s_{k_2})}}}$. This is contradictory with the maximality of k_0. Therefore k_2 does not exist.
  - Second case: for all $k > k_1$, $(s_k, s_{k+1}) \in (\mathtt{G}_{\omega_{|after(s_{k_0})}} \cap \mathcal{T}r^{\ell_2}_{cmd,\ell_1}) \cup \mathtt{A}_{\omega_{|\overline{after(s_{k_0})}}}$.
    According to proposition 10.2, for all $k > k_1$, $(s_k, s_{k+1}) \in (\mathtt{G}_{\omega_{|after(s_{k_2})}} \cap \mathcal{T}r^{\ell_2}_{cmd,\ell_1}) \cup \mathtt{A}_{\omega_{|\overline{after(s_{k_0})}}}$. Therefore, $(s_{k_1}, s) \in \mathtt{Reach}_\omega$

Assume by contradiction that $label(s) = \ell_1$. Therefore, according to Lemma 10.12 $thread(s) = thread(s_{k_1})$ and then $s \in \mathtt{S}_\omega$. This is contradictory with the maximality of k_0.

We choose $s'_0 \stackrel{\text{def}}{=} s_{k_0}$, $s'_1 \stackrel{\text{def}}{=} s_{k_1}$

□

Lemma 11.25. *Using the notations of this section, if $s \in \mathtt{Reach}\langle \mathtt{S}_0 \rangle$, then, there exists $s_0 \in \mathtt{S}_\omega$ such that:*

1. *either $(s_0, s) \in \mathtt{Reach}_\neg$,*
2. *or there exists $s'_1 \in \mathtt{S}_+$ such that $(s_0, s'_1) \in \mathtt{Reach}_+$ and $(s'_1, s) \in \mathtt{Reach}_{cmd}$ and $label(s) \neq \ell_1$.*

Proof. Notice that $\mathtt{S}_0 \subseteq \mathtt{S}_\omega$. It is a straightforward consequence of Lemma 11.24. □

Claim 11.26. *Using the notation of this section $\mathtt{S}' \subseteq \mathtt{S}_\neg$.*

Proof. Let $s \in \mathtt{S}'$, therefore, $s \in \mathtt{Reach}\langle \mathtt{S}_0 \rangle$. Furthermore, $label(s) = \ell_3$. Hence, according to Lemma 10.13, for all $s_1 \in \mathbf{States}$, $(s_1, s) \notin \mathtt{Reach}_\omega$. Therefore, according to Lemma 11.25, there exists $s_0 \in \mathtt{S}_\omega$ such that $(s_0, s) \in \mathtt{Reach}_\neg$. Hence $s \in \mathtt{S}_\neg$. □

Claim 11.27. $\mathtt{Self} \subseteq \mathtt{Self}_\neg \cup \mathtt{Self}_+ \cup \mathtt{Self}_{cmd}$

Proof. Let $(s, s') \in \mathtt{Self}$. According to Lemma 11.22, $(s, s') \in \mathcal{T}r_{\ell_1\, \mathsf{guard}(\neg cond),\ell_3} \cup \mathcal{T}r_{\ell_1\, \mathsf{guard}(cond),\ell_2} \cup \mathcal{T}r_{\ell_2\, cmd,\ell_1}$.

- First case: $(s, s') \in \mathcal{T}r_{\ell_1\, \mathsf{guard}(\neg cond),\ell_3} \cup \mathcal{T}r_{\ell_1\, \mathsf{guard}(cond),\ell_2}$. Due to Lemma 7.6, $label(s) = \ell_1$ Hence, according to Lemma 11.25, either $(s_0, s) \in \mathtt{Reach}_\neg$ or $label(s) \neq \ell_1$. Therefore $(s_0, s) \in \mathtt{Reach}_\neg$.

  According to Lemma 11.1, either $label(s) = \ell_2 \neq \ell_1$ (contradiction) or $s \in \mathtt{interfere}_{\mathtt{A}_0}(\mathtt{S}_0) \subseteq \mathtt{Reach}_\neg\langle \mathtt{S}_\omega \rangle \cap \mathtt{Reach}_+\langle \mathtt{S}_\omega \rangle$. Therefore either $(s, s') \in \mathtt{Self}_\neg$ or $(s, s') \in \mathtt{Self}_+$.

- Second case: $(s, s') \in \mathcal{T}r_{\ell_2\, cmd,\ell_1}$. Therefore, according to Lemma 7.7, $label(s) \in Labs(^{\ell_2} cmd, \ell_1) \setminus \{\ell_1\}$. If $s'' \in \mathtt{Reach}_\neg\langle \mathtt{S}_\omega \rangle$, then, by Lemma 11.1, $label(s'') \in \{\ell_1, \ell_3\}$. Hence, $s \notin \mathtt{Reach}_\neg\langle \mathtt{S}_\omega \rangle$. So, by Lemma 11.25, there exists $s \in \mathtt{S}_0$ and $s_1 \in \mathtt{S}_+$ such that $(s_0, s_1) \in \mathtt{Reach}_+$ and $(s_1, s) \in \mathtt{Reach}_{cmd}$. According to Proposition 10.2, $(s, s') \in after(s_1)$ and therefore $(s, s') \in \mathtt{Self}_{cmd}$.

□

Claim 11.28. $\mathtt{Par} \subseteq \mathtt{Par}_{cmd}$

Proof. Let $(s, s') \in \mathtt{Par}$. There exists $s_0 \in \mathtt{S}_0$ and s_2 such that $(s_0, s_2) \in \mathtt{Reach}_\omega$ and $(s_2, s) \in Schedule$ and $s \in after(s_0)$. By Lemma 11.1, either $(s_0, s_2) \in \mathtt{Reach}_\neg$ or there exists $s_1 \in \mathtt{S}_+$ such that $(s_0, s_1) \in \mathtt{Reach}_+$ and $(s_1, s_2) \in \mathtt{Reach}_{cmd}$ and $label(s_2) \neq \ell_1$.

- In the first case, because $s \in after(s_0)$ and $\mathcal{T}r_\neg$ is conservative (See Lemma 7.5), by Lemma 10.8, $thread(s) = thread(s_0)$. But, by definition of *Schedule* and $\mathtt{Reach}_\neg$, $thread(s_2) \neq thread(s)$ and $thread(s_0) = thread(s_2)$. This is contradictory.

- In the second case, by Proposition 10.2, $s \in after(s_1)$. Because $thread(s) \neq thread(s_0) = thread(s_2)$, by Lemma 10.12, $label(s) \in Labs(^{\ell_2} cmd, \ell_1) \setminus \{\ell_2\}$. Therefore, by Lemmas 11.22 and 7.6, $(s, s') \in \mathcal{T}r_{\ell_2\, cmd,\ell_1}$. Hence $(s, s') \in \mathtt{Par}_{cmd}$

□

Claim 11.29. $\mathtt{Sub} \subseteq \mathtt{Sub}_\neg$

Proof. Let $(s, s') \in \mathtt{Sub}$. Therefore, there exists $s_0 \in \mathtt{S}_\omega$ and $s_1 \in \mathtt{S}'$ such that $(s_0, s_1) \in \mathtt{Reach}$ and $(s_1, s) \in \mathtt{Ext}(s_0, s_1)$.

Notice that $label(s_1) = \ell_3$, therefore, according to Lemma 10.13, $s_1 \notin \mathtt{Reach}_+; \mathtt{Reach}_{cmd}\langle \mathtt{S}_\omega \rangle$. Hence, by Lemma 11.24, there exists $s'_0 \in \mathtt{S}_\omega$ such that $(s_0, s'_0) \in \mathtt{Reach}$ and $(s'_0, s_1) \in \mathtt{Reach}_\neg$.

$(s_1, s) \in \mathtt{Ext}(s_0, s_1) \subseteq \left[(\mathtt{G}_{\omega|after(s_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{\omega|\overline{after(s_0)}} \cup \mathtt{G}_{\omega|after(s_1)}\right]^\star$.

Hence: $(s_1, s) \in \left[(\mathtt{G}_{\omega|after(s_0) \setminus after(s'_0)} \cap \mathcal{T}r) \cup (\mathtt{G}_{\omega|after(s'_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{\omega|\overline{after(s_0)}} \cup \mathtt{G}_{\omega|after(s_1)}\right]^\star$. According to Definition 10.2:

$(s_1, s) \in \left[(\mathtt{Sub}_{\omega|after(s_0)\setminus after(s'_0)}) \cup (\mathtt{G}_{\omega|after(s'_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{\omega|\overline{after(s_0)}} \cup \mathtt{G}_{\omega|after(s_1)}\right]^\star$
Therefore: $(s_1, s) \in \left[(\mathtt{Sub}_{\omega|\overline{after(s'_0)}}) \cup (\mathtt{G}_{\omega|after(s'_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{\omega|\overline{after(s_0)}} \cup \mathtt{G}_{\omega|after(s_1)}\right]^\star$.

Because, by Lemma 10.13, $after(s'_0) \subseteq after(s_0)$. Hence: $\mathtt{A}_{\omega|\overline{after(s_0)}} \subseteq \mathtt{A}_{\omega|\overline{after(s'_0)}}$.
Therefore: $(s_1, s) \in \left[(\mathtt{Sub}_{\omega|\overline{after(s'_0)}}) \cup (\mathtt{G}_{\omega|after(s'_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{\omega|\overline{after(s'_0)}} \cup \mathtt{G}_{\omega|after(s_1)}\right]^\star$.

Because $\mathtt{Sub}_\omega \subseteq \mathtt{A}_\omega$, $(s_1, s) \in \left[(\mathtt{G}_{\omega|after(s'_0)} \cap \mathcal{T}r) \cup \mathtt{A}_{\omega|\overline{after(s'_0)}} \cup \mathtt{G}_{\omega|after(s_1)}\right]^\star$.
This means that $(s_1, s) \in \mathtt{Ext}_\omega(s'_0, s_1)$.
Therefore:
$(s_1, s) \in \left[(\mathtt{G}_{\omega|after(s'_0)\setminus after(s_1)} \cap \mathcal{T}r) \cup (\mathtt{G}_{\omega|after(s_1)} \cap \mathcal{T}r) \cup \mathtt{A}_{\omega|\overline{after(s'_0)}} \cup \mathtt{G}_{\omega|after(s_1)}\right]^\star$
By Proposition 10.2, $(s_1, s) \in (\mathtt{G}_{\omega|after(s_0)} \cap \mathcal{T}r_{\ell_1\, \mathsf{guard}(\neg cond),\ell_2}) \cup (\mathtt{G}_{\omega|after(s_1)} \cap \mathcal{T}r \setminus \mathcal{T}r_{\ell_1\, \mathsf{guard}(\neg cond),\ell_2}) \cup \mathtt{A}_{\omega|\overline{after(s_0)}} \cup \mathtt{G}_{\omega|after(s_1)} = \mathtt{Ext}_\neg(s_1, s_2)$. □

Proposition 11.4. $[\![^{\ell_1}\mathsf{while}(cond)\{^{\ell_2} cmd\}, \ell_3]\!](\mathtt{Q}) \leqslant [\![^{\ell_1}\mathsf{guard}(\neg cond), \ell_3]\!] \circ \mathtt{loop}^{\uparrow\omega}(\mathtt{Q})$
with $\mathtt{loop}(\mathtt{Q}') = \left([\![^{\ell_2} cmd, \ell_1]\!] \circ [\![|^{\ell_1}\mathsf{guard}(cond), \ell_2|]\!](\mathtt{Q}')\right) \sqcup \mathtt{Q}'$

Proof. It is a consequence of Claims 11.26, 11.27, 11.28 and 11.29. □

11.5 Thread Creation

Let $\mathtt{Q}_0 = \langle \mathtt{S}_0, \mathtt{G}_0, \mathtt{A}_0 \rangle$ a configuration.
Let $\mathtt{Q}' = \langle \mathtt{S}', \mathtt{G}', \mathtt{A}' \rangle = [\![^{\ell_1}\mathsf{create}(^{\ell_2} cmd), \ell_3]\!](\mathtt{Q}_0)$
Let $\mathtt{K} = [\mathtt{Reach}, \mathtt{Ext}, \mathtt{Self}, \mathtt{Par}, \mathtt{Sub}] = \left\{\!\!\left|^{\ell_1}\mathsf{create}(^{\ell_2} cmd), \ell_3\right|\!\!\right\}(\mathtt{Q}_0)$
Let $\mathtt{Q}_1 = \langle \mathtt{S}_1, \mathtt{G}_1, \mathtt{A}_1 \rangle = [\![^{\ell_1}\mathsf{spawn}(\ell_2), \ell_3]\!](\mathtt{Q}_0)$
Let $\mathtt{K}_1 = [\mathtt{Reach}_1, \mathtt{Ext}_1, \mathtt{Self}_1, \mathtt{Par}_1, \mathtt{Sub}_1] = \left\{\!\!\left|^{\ell_1}\mathsf{spawn}(\ell_2), \ell_3\right|\!\!\right\}(\mathtt{Q}_0)$
Let $\mathtt{Q}_2 = \langle \mathtt{S}_2, \mathtt{G}_2, \mathtt{A}_2 \rangle = \mathtt{init\text{-}child}_{\ell_2}(\mathtt{Q}_1)$
Let $\mathtt{G}_\infty = \mathit{guarantee}_{\ell_2\, cmd, \ell_\infty}(\mathtt{Q}_2)$
Let $\mathtt{K}_3 = [\mathtt{Reach}_3, \mathtt{Ext}_3, \mathtt{Self}_3, \mathtt{Par}_3, \mathtt{Sub}_3] = \left\{\!\!\left|^{\ell_2} cmd, \ell_\infty\right|\!\!\right\}\langle \mathtt{S}_2, \mathtt{G}_\infty, \mathtt{A}_2 \rangle$
Let $\mathtt{Q}_3 = \langle \mathtt{S}_3, \mathtt{G}_3, \mathtt{A}_3 \rangle = \mathtt{combine}_{\mathtt{Q}_0}(\mathtt{G}_\infty)$
Let $\mathcal{T}r = \mathcal{T}r_{\ell_1\, \mathsf{create}(^{\ell_2} cmd), \ell_3}$

Lemma 11.30. $\mathcal{T}r_{\ell_1\, \mathsf{create}(^{\ell_2} cmd), \ell_3} = \mathcal{T}r_{\ell_1\, \mathsf{spawn}(\ell_2), \ell_3} \cup \mathcal{T}r_{\ell_2\, cmd, \ell_\infty}$

When a thread i_0 executes $^{\ell_1}\mathsf{create}(^{\ell_2} cmd), \ell_2$, it creates some thread i (See Figure 11.1). The main idea is that three kinds of transitions may interfere with i.

- Transitions that may be fired by some of i_0 that have been created before i (e.g., t_1), or by a descendant of such a thread (e.g., t_6). These transitions are represented in green on the figure, and collected in $\mathtt{A}_0$.

- Transitions fired by i_0 after having created i, or transitions fired by some descendant of i_0 created after i. These transitions (in blue on the figure) are collected in $\mathtt{G}_{|\mathtt{post}(\ell_2)}$.

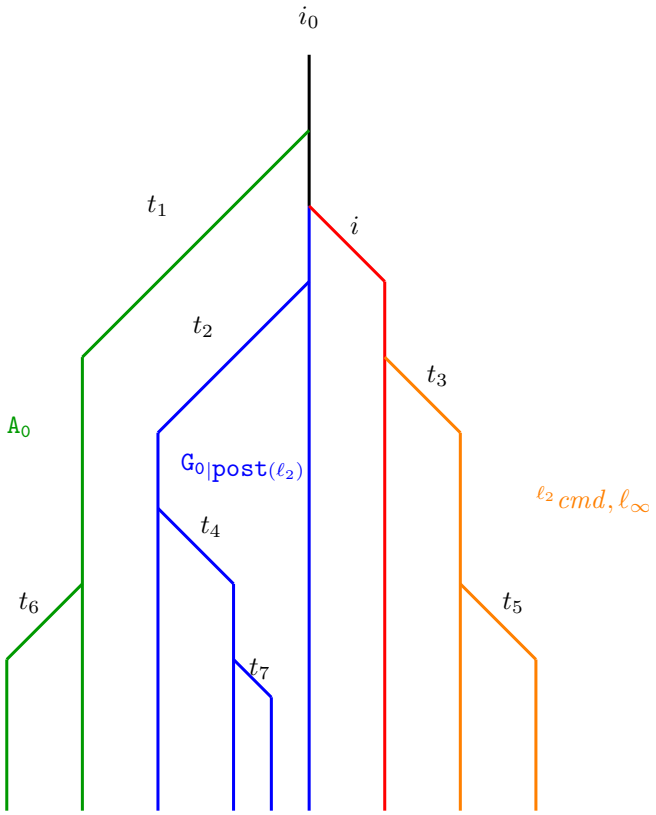


Figure 11.1: Thread Creation

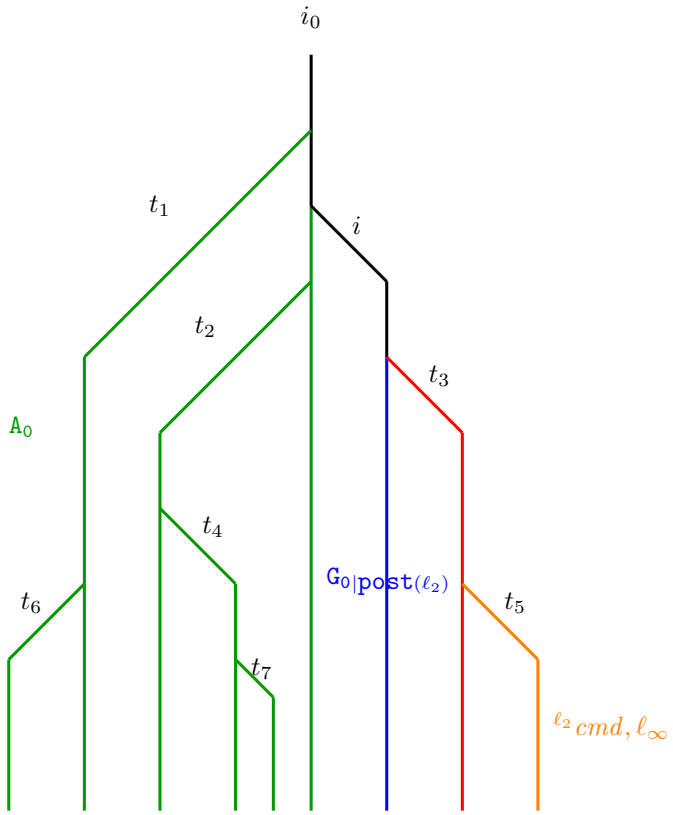


Figure 11.2: Thread Creation

- Transitions of descendants of i. These transitions are represented in orange. All these transitions are generated by the statement ${}^{\ell_2}cmd, \ell_\infty$.

Figure 11.2 is a second example. The thread i creates the thread t_3

Lemma 11.31. *Let s_0, s_1, s_2 and s be four states such that $(s_0, s_1) \in \texttt{Reach}_1$, $s_2 \in \texttt{schedule-child}\{s_1\}$, $label(s_1) = \ell_3$, $(s_2, s) \in \textbf{Transitions}^\star$ and $s \in after(s_0)$.*

Therefore, $s \in after(s_1) \cup after(s_2)$.

Proof. According to Lemma 11.1, there exists s'_0 and s'_1 such that, $s'_0 \in \texttt{interfere}_{\texttt{A}_0}\{s_0\}$, $(s'_0, s'_1) \in \mathcal{T}\!r^-_{\ell_1\, spawn(\ell_2),\ell_3}$, and $s_1 \in \texttt{interfere}_{\texttt{A}_0}\{s'_1\}$.

By Lemmas 10.8 and 7.7, $thread(s_0) = thread(s'_0) = thread(s'_1) = thread(s_1)$.

Let $i_0 = thread(s_0)$ and $i = thread(s)$.

Let g_0, g'_0, j, g_1 and g such that, respectively, the genealogy of s_0, s'_0, s''_0, s_1, s_2, s is g_0, $g_0 \cdot g'_0$, $g_0 \cdot g'_0 \cdot (i_0, \ell_2, j)$, $g_0 \cdot g'_0 \cdot (i_0, \ell_2, j) \cdot g_1$, $g_0 \cdot g'_0 \cdot (i_0, \ell_2, j) \cdot g_1$, $g_0 \cdot g'_0 \cdot (i_0, \ell_2, j) \cdot g_1 \cdot g$. Notice that s_1 and s_2 have the same genealogy.

Because $(s_0, s'_0) \in [\texttt{A}_0{}_{|\overline{after(s_0)}} \cup System]^*$, by Lemma 10.7, $desc_{g'_0}(i_0) = \{i_0\}$.

Because $(s''_1, s_1) \in [\texttt{A}_0{}_{|\overline{after(s_0)}} \cup System]^*$, by Lemma 10.7, $desc_{(i_0,\ell_2,j)\cdot g_1}(i_0) = desc_{(i_0,\ell_2,j)}\{i_0\} = \{i_0, j\}$.

By Lemma 7.2, $desc_{g'_0\cdot(i_0,\ell_2,j)\cdot g_1\cdot g}(i_0) = desc_g[desc_{(i_0,\ell_2,j)\cdot g_1}(desc_{g'_0}\{i_0\})] = desc_g\{i_0, j\}$ By Lemma 7.2, $desc_{g'_0\cdot(i_0,\ell_2,j)\cdot g_1\cdot g}(i_0) = desc_g(\{i_0\}) \cup desc_g(\{j\})$.

Because $s \in after(s_0)$, $i \in desc_{g'_0\cdot(i_0,\ell_2,j)\cdot g_2\cdot g}(i_0)$. Therefore either $i \in desc_g(i_0)$ or $i \in desc_g(j)$. If $i \in desc_g(i_0)$ then $s \in after(s_1)$. If $i \in desc_g(j)$ then $s \in after(s_2)$. □

Lemma 11.32. *If $(s_0, s) \in \texttt{Reach}$ then:*

- *either $s \in \texttt{interfere}_{\texttt{A}_0}(s_0)$ and $label(s) = \ell_1$*
- *or there exists s_1, s_2, s_3 such that $(s_0, s_1) \in \texttt{Reach}_1$, $(s_1, s_2) \in Schedule$, $(s_2, s_3) \in \texttt{Reach}_3 \cap \texttt{Ext}_1(s_0, s_1)$, $(s_3, s) \in Schedule$ and $s_2 \in \texttt{schedule-child}\{s_1\}$. Furthermore $label(s_1) = label(s) = \ell_3$ and $s \in \texttt{interfere}_{\texttt{G}_0{}_{|\texttt{post}(\ell_2)} \cup \texttt{A}_0}\{s_1\}$.*

Proof. $(s_0, s) \in \texttt{Reach}$, therefore, there exists a sequence s'_0, s'_n such that $s'_0 = s_0$, $s'_n = s$ and for every $k \in \{0, \ldots, n-1\}$, $(s_k, s_{k+1}) \in [(\texttt{G}_0{}_{|after(s_0)} \cap \mathcal{T}\!r) \cup \texttt{A}_0{}_{|\overline{after(s_0)}}]^\star$.

Two cases may occur:

- First case for every $k \in \{0, \ldots, n-1\}$, $(s'_k, s'_{k+1}) \in [\texttt{A}_0{}_{|\overline{after(s_0)}} \cup System$. Therefore $s \in \texttt{interfere}_{\texttt{A}_0}(s_0)$ and by Lemma 10.10, $label(s) = \ell_1$.
- Second case: there exists $k \in \{0, \ldots, n-1\}$ such that $(s'_k, s'_{k+1}) \notin [\texttt{A}_0{}_{|\overline{after(s_0)}} \cup System$. Let k_0 the smallest such k.

  $(s'_{k_0}, s'_{k_0+1}) \in (\texttt{G}_0{}_{|after(s_0)} \cap \mathcal{T}\!r)$. Therefore $s_{k_0} \in after(s_0)$. Due to Lemma 10.8 $thread(s'_0) = thread(s_0)$.

  Let $s_1 = s'_{k_0+1}$. According to Lemma 7.6, $thread(s_1) = thread(s'_{k_0}) = thread(s_0)$ and $label(s_1) = \ell_3$. Therefore $(s_0, s_1) \in \texttt{Reach}_1$.

By definition of `schedule-child`, $\texttt{schedule-child}\{s_1\}$ is a singleton. hence, let s_2 such that

$$\{s_2\} = \texttt{schedule-child}\{s_1\}.$$

Therefore, $(s_2, s_1) \in \mathit{Schedule}$. Let $(i, P, \sigma, g) = s$ and $s_3 = (\mathit{thread}(s_1), P, \sigma, g)$. Therefore, $(s, s_3) \in \mathit{Schedule}$ and

$$(s_3, s) \in \mathit{Schedule}.$$

Either there exists $k \in \{k_0, n-1\}$ such that $(s_k, s_{k+1}) \in (\texttt{G}_{0|\mathit{after}(s_0)} \cap \mathcal{T}r^{\diagdown})\mathit{System}$ and $s_k \notin \mathit{after}(s_2)$ or not.

- First case: such a k exists. Let k_1 the smallest such a k. According to Lemma 11.31, $s_{k_1} \in \mathit{after}(s_1)$. Therefore, by Lemma 10.8, $\mathit{thread}(s) = \mathit{thread}(s_1)$ and by Lemma 10.10, $\mathit{label}(s) = \mathit{label}(s_1) = \ell_3$. This is contradictory with Lemma 7.7 which implies $\mathit{label}(s) \neq \ell_3$.
- Second case: there does not exists such a k. Hence: $(s_2, s_3) \in \mathit{Schedule}$; $[(\texttt{G}_{0|\mathit{after}(s_2)} \cap \mathcal{T}r) \cup \texttt{A}_{0|\overline{\mathit{after}(s_0)}}]^\star$; $\mathit{Schedule} = [(\texttt{G}_{0|\mathit{after}(s_2)} \cap \mathcal{T}r) \cup \texttt{A}_{0|\overline{\mathit{after}(s_0)}}]^\star$. Hence:

  $$(s_2, s_3) \in \texttt{Ext}_1(s_0, s_1).$$

  Furthermore by Lemma 10.2, $\mathit{after}(s_2) \subseteq \mathit{after}(s_0)$. Hence $(s_2, s_3) \in [(\texttt{G}_{0|\mathit{after}(s_2)} \cap \mathcal{T}r) \cup \texttt{A}_{0|\overline{\mathit{after}(s_2)}}]^\star$. Therefore, by Proposition 10.1:

  $$(s_2, s_3) \in \texttt{Reach}_3.$$

  Given that, by definition of $\texttt{post}(\ell_2)$, $\mathit{after}(s_2) \subseteq \texttt{post}(\ell_2)$:

  $$s \in \texttt{interfere}_{\texttt{G}_{0|\texttt{post}(\ell_2)} \cup \texttt{A}_0}\{s_1\}.$$

□

Claim 11.33. $\texttt{S}' \subseteq \texttt{interfere}_{\texttt{G}_0 \cup \texttt{A}_0}(\texttt{S}_1)$.

Proof. Let $s \in \texttt{S}'$. Therefore there exists $s_0 \in \texttt{S}_0$ such that $(s_0, s) \in \texttt{Reach}$ and $\mathit{label}(s) = \ell_3 \neq \ell_1$. According to Lemma 11.32 there exists s_1 such that $(s_0, s_1) \in \texttt{Reach}_1$, $\mathit{label}(s_1) = \ell_3$ and $s \in \texttt{interfere}_{\texttt{G}_0 \cup \texttt{A}_0}\{s_1\}$. Therefore $s_1 \in \texttt{S}_1$ and $s \in \texttt{interfere}_{\texttt{G}_0 \cup \texttt{A}_0}(\texttt{S}_1)$. □

Claim 11.34. $\texttt{Self} \subseteq \texttt{Self}_1$.

Proof. Let $(s, s') \in \texttt{Self}$. According to Lemma 7.7, $\mathit{label}(s) \neq \ell_3$. There exists $s_0 \in \texttt{S}_0$ such that $(s_0, s) \in \texttt{Reach}$. Therefore, according to lemma 11.32, $s \in \texttt{interfere}_{\texttt{A}_0}\{s_0\}$. Therefore $(s_0, s) \in \texttt{Reach}_1$ and, by Lemma 10.10, $\mathit{label}(s) = \ell_1$. Due to Lemmas 7.8 and 11.30, $(s, s') \in \mathcal{T}r_{\ell_1\, \mathit{spawn}(\ell_2), \ell_3}$. Hence $(s, s') \in \texttt{Self}_1$. □

Claim 11.35. $\texttt{Par} \subseteq \texttt{Self}_3 \cup \texttt{Par}_3$.

Proof. Let $(s, s') \in \mathtt{Par}$. Therefore, there exists $s_0 \in \mathtt{S}_0$ such that $(s_0, s) \in \mathtt{Reach}$; *Schedule* and $s \in \mathit{after}(s_0)$. Notice that by definition of *Schedule*, $\mathit{thread}(s_0) \neq \mathit{thread}(s)$.

Assume by contradiction, that $s \in \mathit{Schedule}\langle \mathtt{interfere}_{\mathtt{A}_0}\{s_0\}\rangle$. Due to Lemma 10.8, $\mathit{thread}(s_0) = \mathit{thread}(s)$. This is contradictory.

Hence, by Lemma 11.32, there exists s_1, s_2, s_3 such that $(s_0, s_1) \in \mathtt{Reach}_1$, $(s_1, s_2) \in \mathit{Schedule}$, $(s_2, s_3) \in \mathtt{Reach}_3$, $(s_3, s) \in \mathit{Schedule}$, $s_2 \in \mathtt{schedule\text{-}child}\{s_1\}$, and $\mathit{label}(s_1) = \mathit{label}(s) = \ell_3$.

Hence, $s_1 \in \mathtt{S}_1$, $s_2 \in \mathtt{S}_2$.

According to Lemma 10.6 $\mathit{after}(s_1) \cap \mathit{after}(s_2) = \emptyset$. Given that $(s_2, s) \in \mathtt{Reach}; \mathit{Schedule}; \mathit{Schedule}$, $(s_2, s) \in (\mathtt{G}_0 \cup \mathtt{A}_0)^\star_{|\overline{\mathit{after}(s_1)}}$. Hence, du to Lemma 11.23, $s \in \mathit{after}(s_2)$.

If $\mathit{thread}(s) = \mathit{thread}(s_2)$, then $(s_2, s) \in \mathtt{Reach}_3$ and $(s, s') \in \mathtt{Self}_3$. If $\mathit{thread}(s) \neq \mathit{thread}(s_2)$, then $(s, s') \in \mathtt{Par}_3$. □

Claim 11.36. $\mathtt{Sub} \subseteq \mathtt{Self}_3 \cup \mathtt{Par}_3$.

Proof. Let $(s, s') \in \mathtt{Sub}$. There exists s_0, s_4 such that $(s_0, s_4) \in \mathtt{Reach}$ and $(s_4, s) \in \mathtt{Ext}(s_0, s_4)$ and $s_4 \in \mathtt{S}'$. By Lemma 11.32, there exists s_1, s_2, s_3 such that $(s_0, s_1) \in \mathtt{Reach}_1$, $s_2 \in \mathtt{schedule\text{-}child}_{\mathtt{A}}(\{s_1\})$, $(s_2, s_3) \in \mathtt{Reach}_3 \cap \mathtt{Ext}_1(s_0, s_1)$ and $(s_3, s_4) \in \mathit{Schedule}$.

Furthermore, $s \in \mathit{after}(s_0) \setminus \mathit{after}(s_4)$. Due to Lemma 11.31, either $s \in \mathit{after}(s_1) \setminus \mathit{after}(s_4)$ or $s \in \mathit{after}(s_2) \setminus \mathit{after}(s_4)$.

Assume by contradiction that $s \in \mathit{after}(s_1) \setminus \mathit{after}(s_4)$. Therefore $(s, s') \in \mathtt{Sub}_1$. But, by Claim 11.3, $\mathtt{Sub}_1 = \emptyset$. Therefore $s \in \mathit{after}(s_2) \setminus \mathit{after}(s_4)$.

Let $(i, P, \sigma, g) = s$ and $s_5 = (\mathit{thread}(s_2), P_5, \sigma_5, g_5)$.

Given that $(s_4, s) \in \mathtt{Ext}(s_0, s_4)$, $(s_4, s) \in \big[(\mathtt{G}_{0|\mathit{after}(s_0)} \cap \mathcal{Tr}) \cup \mathtt{A}_{2|\overline{\mathit{after}(s_0)}}\big]^*$ and by Lemma 11.31, $(s_4, s) \in \big[(\mathtt{G}_{0|\mathit{after}(s_1) \cup \mathit{after}(s_2)} \cap \mathcal{Tr}) \cup \mathtt{A}_{2|\overline{\mathit{after}(s_0)}}\big]^*$.

By definition of $\mathtt{post}$, $\mathit{after}(s_1) \subseteq \mathtt{post}(\ell_2)$. Furthermore by Lemma 10.6, $\mathit{after}(s_1) \cap \mathit{after}(s_2) = \emptyset$. Therefore $\mathit{after}(s_1) \subseteq \mathtt{post}(\ell_2) \setminus \mathit{after}(s_2)$. Hence, $(s_4, s) \in \big[(\mathtt{G}_{0|\mathit{after}(s_2)} \cap \mathcal{Tr}) \cup \mathtt{A}_{2|\overline{\mathit{after}(s_0)}} \cup \mathtt{G}_{0|\mathtt{post}(\ell_2) \setminus \mathit{after}(s_2)}\big]^*$. By Lemma 10.2, $\mathit{after}(s_2) \subseteq \mathit{after}(s)$, therefore $(s_4, s) \in \big[(\mathtt{G}_{0|\mathit{after}(s_2)} \cap \mathcal{Tr}) \cup (\mathtt{A}_2 \cup \mathtt{G}_{0|\mathtt{post}(\ell_2)})_{|\overline{\mathit{after}(s_0)}}\big]^*$. By Proposition 10.1, $(s_4, s) \in \big[(\mathtt{G}_{\infty|\mathit{after}(s_2)} \cap \mathcal{Tr}) \cup (\mathtt{A}_2 \cup \mathtt{G}_{0|\mathtt{post}(\ell_2)})_{|\overline{\mathit{after}(s_0)}}\big]^*$.

Let $(i, P, \sigma, g) = s$ and $s_5 = (\mathit{thread}(s_2), P, \sigma, g)$. Therefore, $(s_2, s_5) \in \mathtt{Reach}_3$.

If $i = \mathit{thread}(s_2)$, then $s_5 = s$ and $(s, s') \in \mathtt{Self}_3$. If $i \neq \mathit{thread}(s_2)$, then $(s_5, s) \in \mathit{Schedule}$ and $(s, s') \in \mathtt{Par}_3$. □

Proposition 11.5. $[\![^{\ell_1}\mathit{create}(^{\ell_2}\mathit{cmd}), \ell_3]\!](\mathtt{Q}) \leqslant \mathtt{combine}_{\mathtt{Q}'} \circ \mathtt{guarantee}_{[\![^{\ell_2}\mathit{cmd}, \ell_\infty]\!]} \circ \mathtt{init\text{-}child}_{\ell_2}(\mathtt{Q}')$ *with* $\mathtt{Q}' = [\![^{\ell_1}\mathit{spawn}(\ell_2), \ell_3]\!](\mathtt{Q})$

CHAPTER 12

Denotational Intermediate Semantics

12.1 Definition

We introduce the denotational intermediate semantics $[\![|\cdot|]\!]$:

Definition 12.1 (Basic statements)**.** Let ${}^{\ell_1}basic, \ell_2$ be a basic statement, then:
$[\![|^{\ell_1}basic, \ell_2|]\!]\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle \stackrel{\text{def}}{=} \langle \mathtt{S}'', \mathtt{G} \cup \mathtt{G}_{\text{new}}, \mathtt{A}\rangle$
where $\mathtt{S}'' = \mathtt{interfere}_{\mathtt{A}}\big(\mathcal{T}r^{-}_{\ell_1 basic,\ell_2}\langle \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\rangle\big)$
and $\mathtt{G}_{\text{new}} = \{(s, s') \in \mathcal{T}r_{\ell_1 basic,\ell_2} \mid s \in \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\}$.

Definition 12.2. For each concrete configuration $\mathtt{Q}$:

1. $[\![|^{\ell_1} cmd_1; {}^{\ell_2} cmd_2, \ell_3|]\!](\mathtt{Q}) \stackrel{\text{def}}{=} [\![|^{\ell_2} cmd_2, \ell_3|]\!] \circ [\![|^{\ell_1} cmd_1, \ell_2|]\!](\mathtt{Q})$

2. $[\![|^{\ell_1} \mathsf{if}((cond)\mathsf{then}\{^{\ell_2} cmd_1\}\mathsf{else}\{^{\ell_4} cmd_2\}, \ell_3|]\!](\mathtt{Q}) \stackrel{\text{def}}{=}$
$[\![|^{\ell_2} cmd_1, \ell_3|]\!] \circ [\![|^{\ell_1} \mathsf{guard}(cond), \ell_2|]\!](\mathtt{Q}) \sqcup [\![|^{\ell_4} cmd_2, \ell_3|]\!] \circ [\![|^{\ell_1} \mathsf{guard}(\neg cond), \ell_4|]\!](\mathtt{Q})$

3. $[\![|^{\ell_1} \mathsf{while}(cond)\{^{\ell_2} cmd\}, \ell_3|]\!](\mathtt{Q}) \stackrel{\text{def}}{=} [\![|^{\ell_1} \mathsf{guard}(\neg cond), \ell_3|]\!] \circ \mathtt{loop}^{\uparrow\omega}(\mathtt{Q})$
with $\mathtt{loop}(\mathtt{Q}') = \big([\![|^{\ell_2} cmd, \ell_1|]\!] \circ [\![|^{\ell_1} \mathsf{guard}(cond), \ell_2|]\!](\mathtt{Q}')\big) \sqcup \mathtt{Q}'$

4. $[\![|^{\ell_1} \mathsf{create}(^{\ell_2} cmd), \ell_3|]\!](\mathtt{Q}) \stackrel{\text{def}}{=} \mathtt{combine}_{\mathtt{Q}'} \circ \mathtt{guarantee}_{[\![|^{\ell_2} cmd, \ell_\infty|]\!]} \circ \mathtt{init\text{-}child}_{\ell_2}(\mathtt{Q}')$
with $\mathtt{Q}' = [\![|^{\ell_1} \mathsf{spawn}(\ell_2), \ell_3|]\!](\mathtt{Q})$

While points 1 and 3 are as expected, the semantics of ${}^{\ell_1}\mathsf{create}({}^{\ell_2}cmd), \ell_3$ (point 4) computes interferences which will arise from executing the child and its descendants with `guarantee` and then combines this result with the configuration of the current thread.

The next theorem shows how the G-collecting semantics is over-approximated by our intermediate denotational semantics, and is the key point in defining the abstract semantics.

Theorem 12.1 (Soundness)**.** *For each concrete configuration* $\mathtt{Q}$ *and each statement* ${}^{\ell}stmt, \ell'$*:* $[\![{}^{\ell}stmt, \ell']\!](\mathtt{Q}) \leqslant [\![|{}^{\ell}stmt, \ell'|]\!](\mathtt{Q})$.

Proof. This theorem is a consequence of Propositions 10.1, 11.1, 11.2, 11.3, 11.4 and 11.5. □

From the point of view of Galois connections, consider the lattice of concrete configurations **C-Configurations**, and the Galois connection $\alpha_{\mathrm{id}}, \gamma_{\mathrm{id}}$ from **C-Configurations** to **C-Configurations** defined by $\alpha_{\mathrm{id}} = \gamma_{\mathrm{id}} = \lambda\mathtt{Q}.\mathtt{Q}$. For all statements ${}^{\ell}stmt, \ell'$ The semantics $[\![|{}^{\ell}stmt, \ell'|]\!]$ is an abstraction[2] of $[\![{}^{\ell}stmt, \ell']\!]$ for this Galois connection.

The main advantages of the intermediate denotational semantics, comparing with the G-collecting semantics are:

- The intermediate denotational semantics is defined by induction on statements.
- There exist a pseudo-algorithm that computes the intermediate denotational semantics by induction on statements. This pseudo-algorithm applies the inductive definition. This is not a true algorithm since some fixpoint computations need an infinite time.

The abstract semantics (See Part IV) will overapproximate this semantics and be computable.

12.2 Connection Between the Denotational Intermediate Semantics and the Operational Semantics

12.2.1 Soundness

Recall that $\mathcal{T}r^{\star}_{\ell_{cmd},\ell_{\infty}}(Init)$ is the set of states that occur on paths starting from *Init*. $\mathtt{S}'$ represents all final states reachable by the whole program from an initial state. $\mathtt{G}'$ represents all transitions that may be done during any execution of the program and $\mathtt{A}'$ represents transitions of children of ***main***.

The following proposition states that the denotational semantics is an overapproximation of the operational semantics.

[2]The concept of *abstraction* is defined by Definition 3.3

Proposition 12.1 (Soundness). *Consider a program* ${}^{\ell}cmd, \ell_\infty$ *and its set of initial states Init. Let:*

$$\langle \mathtt{S}', \mathtt{G}', \mathtt{A}' \rangle \stackrel{def}{=} [\![|^{\ell} cmd, \ell_\infty |]\!] \langle Init, \mathtt{G}_\infty, System \rangle$$
$$with\ \mathtt{G}_\infty = \mathtt{guarantee}_{[\![|^{\ell} cmd, \ell_\infty |]\!]} \langle Init, System, System \rangle$$

Then:

$$\begin{array}{rcl} \mathtt{S}' & \supseteq & \{(\boldsymbol{main}, P, \sigma, g) \in \mathcal{T}r^{\star}_{\ell\, cmd, \ell_\infty} \langle Init \rangle \mid P(\boldsymbol{main}) = \ell_\infty\} \\ \mathtt{G}' & = & \mathtt{G}_\infty \supseteq \{(s, s') \in \mathcal{T}r_{\ell\, cmd, \ell_\infty} \mid s \in \mathcal{T}r^{\star}_{\ell\, cmd, \ell_\infty} \langle Init \rangle\} \cup System \\ \mathtt{A}' & \supseteq & \{(s, s') \in \mathcal{T}r_{\ell\, cmd, \ell_\infty} \mid s \in \mathcal{T}r^{\star}_{\ell\, cmd, \ell_\infty} \langle Init \rangle \wedge thread(s) \neq \boldsymbol{main}\} \\ & & \cup System \end{array}$$

Proof. This is a consequence of Theorem 10.1 and Theorem 12.1. □

12.2.2 Completeness

Since the intermediate denotational semantics is an overapproximation of the G-collecting semantics we may wonder if we lose precision. Obviously, the two semantics are not equal. Let us consider the program $x := 1$:
Let $\langle \mathtt{S}', \mathtt{G}', \mathtt{A}' \rangle = [\![x := 1]\!] \langle Init, System, System \rangle$ and $\langle \mathtt{S}'', \mathtt{G}'', \mathtt{A}'' \rangle = [\![|x := 1|]\!] \langle Init, System, System \rangle$. It is straightforward to check that $\mathtt{S}' = \emptyset$ and $\mathtt{S}'' \neq \emptyset$. Nevertheless, we will show that when we compute the guarantee of the whole semantics of a program, the two semantics coincide.

We introduce the concept of "*consistent*". A consistent configuration is a configuration without unreachable states or transitions. Formally:

Definition 12.3. A concrete configuration $\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$ is *consistent* with a statement ${}^{\ell}stmt, \ell'$ if and only if the three following properties hold:

(a) $\mathtt{S} \subseteq \mathcal{T}r^{\star}_{\ell\, stmt, \ell'} \langle Init \rangle$

(b) $\mathtt{G} \subseteq \{(s, s') \in \mathcal{T}r_{\ell\, stmt, \ell'} \mid s \in \mathcal{T}r^{\star}_{\ell\, stmt, \ell'} \langle Init \rangle\} \cup System$

(c) $\mathtt{A} \subseteq \{(s, s') \in \mathcal{T}r_{\ell\, stmt, \ell'} \mid s \in \mathcal{T}r^{\star}_{\ell\, cmd, \ell'} \langle Init \rangle\} \cup System$

Obviously, consistence is an invariant:

Lemma 12.1. *We consider a concrete configuration* $\mathtt{Q}$ *and two statements* ${}^{\ell}stmt, \ell'$ *and* ${}^{\ell_1}stmt_1, \ell_2$ *such that:* $\mathcal{T}r_{\ell_1\, stmt_1, \ell_2} \subseteq \mathcal{T}r_{\ell\, stmt, \ell'}$.

If $\mathtt{Q}$ *is consistent with* ${}^{\ell}stmt, \ell'$ *then* $[\![|^{\ell_1} stmt_1, \ell_2 |]\!](\mathtt{Q})$ *is also consistent with* ${}^{\ell}stmt, \ell'$.

Proof. We make a proof by induction on statements. The lemma is trivial for basic statements, and induction is straightforward[3]. □

We notice that the semantics constraints $\mathtt{S}'$:

Lemma 12.2. *Let* $\langle \mathtt{S}', \mathtt{G}', \mathtt{A}' \rangle = [\![{}^{\ell} stmt, \ell']\!] \langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle$. *Therefore for all* $(i', P', \sigma', g') \in \mathtt{S}'$, $P'(i') = \ell'$ *and there exists* $s \in \mathtt{S}$ *such that* $i = thread(s)$.

[3] We just need to consider each case.

Proof. As for previous lemma, the proof is done by induction on statements. □

When we compute inductively the semantics of a program, we will encounter two kinds of configurations:

(a) configurations that represent execution of the main thread.

(b) configurations that represent the execution of some other threads.

The configurations of kind (a) are called "principal" and the configurations of kind (b) are called "secondary". Formally:

Definition 12.4. A concrete configuration is *principal* if and only if the two following properties hold:

(a) $\forall s \in \mathtt{S}, thread(s) = \boldsymbol{main}$.

(b) $\forall (s, s') \in \mathtt{A}, thread(s) \neq \boldsymbol{main}$

Definition 12.5. A configuration is *secondary* if and only if the two following properties hold:

(a) $\forall s \in \mathtt{S}, thread(s) \neq \boldsymbol{main}$.

(b) $\forall (s, s') \in \mathtt{G}, thread(s) \neq \boldsymbol{main}$

Secondary configurations remain secondary:

Lemma 12.3. *If a configuration* $\mathtt{Q}$ *is secondary, therefore, for all statements* ${}^{\ell}stmt, \ell'$*, the configuration* $[\![|{}^{\ell}stmt, \ell'|]\!]\mathtt{Q}$ *is secondary.*

Proof. By induction on statements, using Lemma 12.2. □

The function $\mathtt{init\text{-}child}_{\ell}$ transforms a principal configuration into a secondary configuration: i.e, if we were executing the ***main*** thread, therefore, after $\mathtt{init\text{-}child}_{\ell}$ we execute some descendant(s) of the ***main*** thread. Formally:

Lemma 12.4. *If* $\mathtt{Q}$ *is a principal configuration therefore* $\mathtt{init\text{-}child}_{\ell}(\mathtt{Q})$ *is an secondary configuration.*

Proof. This is a consequence of the definition of $\mathtt{init\text{-}child}_{\ell}$. □

A principal configuration remains principal. Notice that, in the proof of the Lemma, we need to deal with Secondary configurations.

Lemma 12.5. *If a configuration* $\mathtt{Q}$ *is principal, therefore, for all statements* ${}^{\ell}stmt, \ell'$*, the configuration* $[\![|{}^{\ell}stmt, \ell'|]\!]\mathtt{Q}$ *is principal.*

Proof. The proof is done by induction on statements. All cases except *create* are straightforward.

Recall that $[\![|^{\ell_1}\textit{create}(^{\ell_2}cmd),\ell_3|]\!](\mathtt{Q}) = \mathtt{combine}_{\mathtt{Q}'}\circ\mathtt{guarantee}_{[\![|^{\ell_2}cmd,\ell_\infty|]\!]}\circ\mathtt{init\text{-}child}_{\ell_2}(\mathtt{Q}')$ with $\mathtt{Q}' = [\![|^{\ell_1}\textit{spawn}(\ell_2),\ell_3|]\!](\mathtt{Q})$

According to Lemma 12.4, $\mathtt{init\text{-}child}_{\ell_2}(\mathtt{Q}')$ is secondary, therefore, by Lemma 12.3, for all $(s,s') \in \mathtt{G}''$, $\mathit{thread}(s) \neq \boldsymbol{main}$ where $\langle \mathtt{S}'', \mathtt{G}'', \mathtt{A}''\rangle = \mathtt{guarantee}_{[\![|^{\ell_2}cmd,\ell_\infty|]\!]}\circ\mathtt{init\text{-}child}_{\ell_2}(\mathtt{Q}')$. Hence we conclude. □

Now, we can conclude that the denotational intermediate semantics is better than only sound, it is also complete:

Proposition 12.2 (Completeness)**.** *Consider a program* $^{\ell}cmd,\ell_\infty$ *and its set of initial states Init. Let:*

$$\langle \mathtt{S}', \mathtt{G}', \mathtt{A}'\rangle \stackrel{def}{=} [\![|^{\ell}cmd,\ell_\infty|]\!]\langle \mathit{Init}, \mathtt{G}_\infty, \mathit{System}\rangle$$

$$\textit{with } \mathtt{G}_\infty = \mathtt{guarantee}_{[\![|^{\ell}cmd,\ell_\infty|]\!]}\langle \mathit{Init}, \mathit{System}, \mathit{System}\rangle$$

Then:

$$\begin{array}{rcl}
\mathtt{S}' & \subseteq & \{(\boldsymbol{main}, P, \sigma, g) \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle \mid P(\boldsymbol{main}) = \ell_\infty\} \\
\mathtt{G}' & = & \mathtt{G}_\infty \subseteq \{(s,s') \in \mathcal{T}r_{\ell_{cmd,\ell_\infty}} \mid s \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle\} \cup \mathit{System} \\
\mathtt{A}' & \subseteq & \{(s,s') \in \mathcal{T}r_{\ell_{cmd,\ell_\infty}} \mid s \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle \wedge \mathit{thread}(s) \neq \boldsymbol{main}\} \\
 & & \cup \mathit{System}
\end{array}$$

Proof. Lemma 12.1 proves that $\mathtt{G}' \subseteq \{(s,s') \in \mathcal{T}r_{\ell_{cmd,\ell_\infty}} \mid s \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle\} \cup \mathit{System}$ and $\mathtt{A}' \subseteq \{(s,s') \in \mathcal{T}r_{\ell_{cmd,\ell_\infty}} \mid s \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle\} \cup \mathit{System}$.

Lemma 12.5 permits to conclude that $\mathtt{A}' \subseteq \{(s,s') \in \mathcal{T}r_{\ell_{cmd,\ell_\infty}} \mid s \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle \wedge \mathit{thread}(s) \neq \boldsymbol{main}\} \cup \mathit{System}$.

Lemma 12.1 proves that $\mathtt{S}' \subseteq \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle$. And, with Lemma 12.2 we conclude that: $\mathtt{S}' \subseteq \{(\boldsymbol{main}, P, \sigma, g) \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle \mid P(\boldsymbol{main}) = \ell_\infty\}$. □

12.2.3 Conclusion

The following theorem summarizes the two previous propositions:

Theorem 12.2 (Connection with the operational semantics)**.** *Consider a program* $^{\ell}cmd,\ell_\infty$ *and its set of initial states Init. Let:*

$$\langle \mathtt{S}', \mathtt{G}', \mathtt{A}'\rangle \stackrel{def}{=} [\![|^{\ell}cmd,\ell_\infty|]\!]\langle \mathit{Init}, \mathtt{G}_\infty, \mathit{System}\rangle$$

$$\textit{with } \mathtt{G}_\infty = \mathtt{guarantee}_{[\![|^{\ell}cmd,\ell_\infty|]\!]}\langle \mathit{Init}, \mathit{System}, \mathit{System}\rangle$$

Then:

$$\begin{array}{rcl}
\mathtt{S}' & = & \{(\boldsymbol{main}, P, \sigma, g) \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle \mid P(\boldsymbol{main}) = \ell_\infty\} \\
\mathtt{G}' & = & \mathtt{G}_\infty = \{(s,s') \in \mathcal{T}r_{\ell_{cmd,\ell_\infty}} \mid s \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle\} \cup \mathit{System} \\
\mathtt{A}' & = & \{(s,s') \in \mathcal{T}r_{\ell_{cmd,\ell_\infty}} \mid s \in \mathcal{T}r^{\star}_{\ell_{cmd,\ell_\infty}}\langle \mathit{Init}\rangle \wedge \mathit{thread}(s) \neq \boldsymbol{main}\} \\
 & & \cup \mathit{System}
\end{array}$$

Part IV

Abstract Semantics

CHAPTER 13

Generic Abstraction for Interleaving Semantics

13.1 Abstraction

We use the abstract interpretation methodology. Our concrete lattices are the powersets $\mathcal{P}(\textbf{States})$ and $\mathcal{P}(\textbf{Transitions})$ ordered by inclusion. Remember, our goal is to adapt any given single-thread analysis in a multithreaded setting. Accordingly, we are given an abstract complete lattice $\mathscr{D}$ of abstract states and an abstract complete lattice $\mathscr{R}$ of abstract transitions. These concrete and abstract lattices are linked by two Galois connections, respectively $\alpha_{\mathscr{D}}, \gamma_{\mathscr{D}}$ and $\alpha_{\mathscr{R}}, \gamma_{\mathscr{R}}$: We consider four Galois connections described in Fig. 13.2 and we assume that the two first Galois connections are given, and build the other two one from them. We assume that abstractions of states and transitions depend only on stores and current threads and that all the transitions that leave the store and the current thread unchanged are in $\gamma_{\mathscr{R}}(\bot)$. This assumption allows us to abstract *guard* and *spawn* as the least abstract transition $\bot$.

We also assume we are given the abstract operators of Figure 13.1, which are correct abstractions[1] of the corresponding concrete functions. The labels ℓ and ℓ' are implicitely

[1] According to definition 3.3.

Concrete function	Abstract function
$\lambda \mathtt{S}.\mathcal{T}\!r^{-}_{\ell_{\mathit{action},\ell'}}\langle \mathtt{S} \rangle$	$\mathit{elem}_{\mathit{action}} : \mathscr{D} \to \mathscr{D}$
$\lambda \mathtt{S}.(\mathcal{T}\!r^{-}_{\ell_{\mathit{action},\ell'}})_{\mid \mathtt{S}}$	$\mathit{elem\text{-}inter}_{\mathit{action}} : \quad \mathscr{D} \to \mathscr{R}$
$\lambda \mathtt{S}.\mathcal{T}\!r^{-}_{\ell_{\mathit{guard}(\mathit{cond}),\ell'}}\langle \mathtt{S} \rangle$	$\mathit{enforce}_{\mathit{cond}} : \mathscr{D} \to \mathscr{D}$
$\lambda \mathtt{A}, \mathtt{S}.\mathtt{interfere}_{\mathtt{A}}(\mathtt{S})$	$\mathit{inter} : \mathscr{R} \times \mathscr{D} \to \mathscr{D}$
$\mathtt{schedule\text{-}child}_{\ell}$	$\mathit{schedule\text{-}child} : \mathscr{D} \to \mathscr{D}$
$\lambda \mathtt{S}.\alpha_{\mathrm{E}}[(\mathcal{T}\!r^{-}_{\ell_{\mathit{action},\ell'}})_{\mid \mathtt{S}}]$	$\mathit{error}_{\mathit{action}} : \mathscr{D} \to \mathcal{P}(\mathbf{Errors})$
$\lambda \mathtt{S}.\alpha_{\mathrm{E}}[(\mathcal{T}\!r^{-}_{\ell_{\mathit{guard}(\mathit{cond}),\ell'}})_{\mid \mathtt{S}}]$	$\mathit{error}_{\mathit{cond}} : \mathscr{D} \to \mathcal{P}(\mathbf{Errors})$

Figure 13.1: Given Abstractions

universally quantified, e.g., $\mathit{elem}_{\mathit{action}}$ is an abstraction of $\lambda \mathtt{S}.\mathcal{T}\!r^{-}_{\ell_{\mathit{action},\ell'}}\langle \mathtt{S} \rangle$ for all labels ℓ and ℓ'.

The function $\mathit{elem}_{\ell_{\mathit{action},\ell'}}$ abstracts the fact to fire exactly one transition specific to the statement ${}^{\ell}\mathit{action}, \ell'$. Notice that, $\mathit{elem}_{\ell_{\mathit{action},\ell'}}$ is the abstraction of the canonical[2] function $f_{\mathcal{T}\!r^{-}_{\ell_{\mathit{action},\ell'}}}$ associated to $\mathcal{T}\!r^{-}_{\ell_{\mathit{action},\ell'}}$. The function $\mathit{elem\text{-}inter}_{\ell_{\mathit{action},\ell'}}(\mathcal{C})$ abstract the fact to collect all transitions generated by the statement ${}^{\ell}\mathit{action}, \ell'$ that may be fired from a state of $\gamma_{\mathscr{D}}(\mathcal{C})$. Functions $\mathit{enforce}_{\ell_{\mathit{cond},\ell'}}$ and $\mathit{enforce\text{-}inter}_{\ell_{\mathit{cond},\ell'}}$ do the same thing for *guard* statements.

To handle errors, we assume a function $\mathtt{error} : \mathbf{Transitions} \to \mathcal{P}(\mathbf{Errors})$ from the set of transitions **Transitions** to some set of errors **Errors**. The set of errors represents possibles run-time errors, e.g.:

$$\mathbf{Errors} = \{\mathbf{array\text{-}overflow}, \mathbf{division\text{-}by\text{-}zero}, \mathbf{NULL\text{-}pointer\text{-}dereference}\}.$$

The function $\mathtt{error}(\tau)$ associates to each transition τ, the set of errors that transition may make. This gives us a Galois connection $\alpha_{\mathrm{E}}, \gamma_{\mathrm{E}}$ from the lattice $\mathcal{P}(\mathbf{Transitions})$ of set of transitions to the lattice of set of errors $\mathcal{P}(\mathbf{Errors})$:

$$\begin{array}{rcl} \alpha_{\mathrm{E}}(\mathtt{G}) & \stackrel{\text{def}}{=} & \bigcup_{\tau \in \mathtt{G}} \mathtt{error}(\tau) \\ \gamma_{\mathrm{E}}(\mathcal{E}) & \stackrel{\text{def}}{=} & \{\tau \mid \mathtt{error}(\tau) \subseteq \mathcal{E}\}. \end{array}$$

The function $\mathit{error}_{\mathit{action}} : \mathscr{D} \to \mathcal{P}(\mathbf{Errors})$ abstracts the possible error transitions that may be fired when applying a action *action* from a set of states $\mathtt{S}$. For instance, recall the Euclides algorithm given in Figure 3.6. At line 6, they may be a division by zero, $\mathit{error}_{\mathtt{r:=a\%b}}$ returns **division-by-zero** is the value of `b` may be zero. We assume that transitions that are not generated by a statement of the form ${}^{\ell}\mathit{action}, \ell'$ or ${}^{\ell}\mathit{guard}(\mathit{cond}), \ell'$ cannot generates errors, e.g., *spawn* statements does not make errors.

Since the number of thread during an execution may be infinite (E.g., see program of Figure 7.4) we need to abstract threads. A thread will be abstracted by the label where

[2]Recall that, in Section 2.2 we associate to each binary relation R a canonical function f_R defined by: $f_R(S) \stackrel{\text{def}}{=} R\langle S \rangle$.

Name	Concrete Elements	Abstract Elements	Definitions
$\gamma_{\mathscr{D}}, \alpha_{\mathscr{D}}$	$\mathtt{S} \in \mathcal{P}(\mathbf{States})$	$\mathcal{C} \in \mathscr{D}$	
$\gamma_{\mathscr{R}}, \alpha_{\mathscr{R}}$	$\mathtt{A} \in \mathcal{P}(\mathbf{Transitions})$	$I \in \mathscr{R}$	
$\gamma_{\mathrm{L}}, \alpha_{\mathrm{L}}$	$\mathtt{S} \in \mathcal{P}(\mathbf{States})$	$\mathcal{L} \in \mathcal{P}(\mathbb{L})$	$\alpha_{\mathrm{L}}(\mathtt{S}) = \{\ell \in \mathbb{L} \mid \mathtt{S} \cap \mathtt{post}(\ell) \neq \emptyset\}$ $\gamma_{\mathrm{L}}(\mathbb{L}) = \mathbf{States}$ $\gamma_{\mathrm{L}}(\mathcal{L}) = \bigcap_{\ell \in \mathbb{L} \setminus \mathcal{L}} \overline{\mathtt{post}(\ell)}$
$\gamma_{\mathrm{K}}, \alpha_{\mathrm{K}}$	$\mathtt{G} \in \mathcal{P}(\mathbf{Transitions})$	$\mathcal{K} \in \mathscr{R}^{\mathbb{L}}$	$\alpha_{\mathrm{K}}(\mathtt{G}) = \lambda\ell.\alpha_{\mathscr{R}}(\mathtt{G}_{\mid \mathtt{post}(\ell)})$ $\gamma_{\mathrm{K}}(\mathcal{K}) = \{(s,s') \in \mathbf{Transitions} \mid \forall \ell \in \mathbb{L},$ $\in \mathtt{post}(\ell) \Rightarrow (s,s') \in \gamma_{\mathscr{R}}(\mathcal{K}(\ell))\}$

Figure 13.2: Galois Connections

it have been created. E.g., in Figure 7.3, the threads i and j will be abstracted by the same label ℓ_2. The set of abstract threads is then the set of labels $\mathbb{L} \subseteq \mathbf{Labels}$ in which a thread may be created. We define a Galois connection between $\mathcal{P}(\mathbf{States})$ and $\mathcal{P}(\mathbb{L})$: $\alpha_{\mathrm{L}}(\mathtt{S}) = \{\ell \in \mathbb{L} \mid \mathtt{S} \cap \mathtt{post}(\ell) \neq \emptyset\}$ and $\gamma_{\mathrm{L}}(\mathcal{L}) = \bigcap_{\ell \in \mathbb{L} \setminus \mathcal{L}} \overline{\mathtt{post}(\ell)}$ (by convention, this set is **States** when $\mathcal{L} = \mathbb{L}$). The set $\alpha_{\mathrm{L}}(\mathtt{S})$ represents the set of labels that may have been encountered before reaching this point of the program.

Note that we have two distinct ways of abstracting states (i, P, σ, g), either by using $\alpha_{\mathscr{D}}$, or by using α_{L} which only depends on the genealogy g and the current thread i. The latter is specific to the multithreaded case, and is used to infer information about possible interferences. Recall Section 4.7.2: In Figure 4.8, when the thread j_2 reaches the bullet, it can fire transitions. Such a transition (s, s') cannot interfere with j_6. The abstraction α_{L} detects this point, since $\ell_6 \notin \alpha_{\mathrm{L}}(s)$, where ℓ_6 is the label in which j_6 has been created.

We also need to abstract the $\mathtt{G}$ component of the G-collecting semantics, and most importantly $\mathtt{G}_{\mid \mathtt{post}(\ell)}$ as used in the definition of `init-child` (Fig. 10.5), itself required in the semantics of *create* (Def. 12.2, item 4). Notice that $\mathtt{post}()$ is called only on labels of $\mathbb{L}$ and never on labels of $\mathbf{Labels} \setminus \mathbb{L}$. The purpose of α_{K} is to abstract precisely $\mathtt{G}_{\mid \mathtt{post}(\ell)}$: for each $\mathtt{G} \in \mathcal{P}(\mathcal{T}r)$, $\mathcal{K} = \alpha_{\mathrm{K}}(\mathtt{G}) \in \mathscr{R}^{\mathbb{L}}$ maps each label ℓ to the abstract interference (in $\mathscr{R}$) on threads created at ℓ and their descendants. Additionally, we assume an extra element $\ell_\star \in \mathbf{Labels}$, never used in statements, and extend $\mathtt{post}()$ so that $\mathtt{post}(\ell_\star) = \mathbf{States}$. This trick allows us to represent an abstraction of $\mathtt{G}$ itself as $\mathcal{K}(\ell_\star)$.

Definition 13.1. *Abstract configurations* are tuples $\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E} \rangle \in \mathscr{D} \times \mathcal{P}(\mathbf{Labels}) \times \mathscr{R}^{\mathbf{Labels}} \times \mathscr{R} \times \mathcal{P}(\mathbf{Errors})$ and $\mathcal{C}$ is saturated with respect to interferences, i.e., $\mathit{inter}_I(\mathcal{C}) = \mathcal{C}$ and $\ell_\star \in \mathcal{L}$. The meaning of each component of an abstract configuration is given by the Galois connection $\alpha_{\mathrm{cfg}}, \gamma_{\mathrm{cfg}}$:

$$\alpha_{\mathrm{cfg}}\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle \stackrel{\text{def}}{=} \langle \mathit{inter}_{\alpha_{\mathscr{R}}(\mathtt{A})}(\alpha_{\mathscr{D}}(\mathtt{S})), \alpha_{\mathrm{L}}(\mathtt{S}), \alpha_{\mathrm{K}}(\mathtt{G}), \alpha_{\mathscr{R}}(\mathtt{A}), \alpha_{\mathrm{E}}(\mathtt{G}) \rangle$$
$$\gamma_{\mathrm{cfg}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E} \rangle \stackrel{\text{def}}{=} \langle \gamma_{\mathscr{D}}(\mathcal{C}) \cap \gamma_{\mathrm{L}}(\mathcal{L}), \gamma_{\mathrm{K}}(\mathcal{K}) \cap \gamma_{\mathrm{E}}(\mathcal{E}), \gamma_{\mathscr{R}}(I) \rangle$$

We call **A-Configurations** the set of abstract configurations.

In other words:

- $\mathcal{C}$ abstracts the possible current stores of $\mathtt{S}$
- $\mathcal{L}$ abstracts the threads encountered so far in the execution.
- $\mathcal{K}(\ell)$ abstracts possible interferences with a thread created in ℓ.
- I is an abstraction of interferences $\mathtt{A}$.
- $\mathcal{E}$ collects all errors that may occur during the execution of the program.

13.2 Semantics of Commands

The abstract semantics is then derived from the concrete. Fig. 13.3 gives the abstract counterpart of the functions of Fig. 10.5. To ensure termination we use a widening[3] operator $\triangledown$ (See P. Cousot and R. Cousot papers [CC92, CC91] and Section 3.3), i.e., we approximate fixpoints $f^{\uparrow\omega}$ by $f^{\uparrow\triangledown}$.

Definition. 13.2 gives the abstract semantics, derived from Definitions 12.1 and 12.2. Our final algorithm is to compute recursively $\mathit{guarantee}_{(\!|^{\ell}\, cmd, \ell_\infty |\!)}$ applied to the initial configuration.

Definition 13.2. For any abstract configuration $\mathcal{Q}$:

$$
\begin{array}{rcl}
(\!|^{\ell} \mathit{action}, \ell' |\!)\,\mathcal{Q} & \stackrel{\text{def}}{=} & \mathit{basic}_{\mathit{action}}(\mathcal{Q}) \\
(\!|^{\ell_1} cmd_1 ;^{\ell_2} cmd_2 |\!)\,\mathcal{Q} & \stackrel{\text{def}}{=} & (\!|^{\ell_2} cmd_2 |\!) \circ (\!|^{\ell_1} cmd_1 |\!)(\mathcal{Q}) \\
(\!|^{\ell_1} \mathit{while}(cond)\{^{\ell_2} cmd\} |\!)\,\mathcal{Q} & \stackrel{\text{def}}{=} & \mathit{guard}_{\neg cond} \circ \mathit{loop}^{\uparrow\triangledown}(\mathcal{Q}) \\
\text{with } \mathit{loop}(\mathcal{Q}') & \stackrel{\text{def}}{=} & \left((\!|^{\ell_2} cmd, \ell_1 |\!) \circ \mathit{guard}_{cond}\, \mathcal{Q}'\right) \sqcup \mathcal{Q}' \\
(\!|^{\ell_1} \mathit{create}(^{\ell_2} cmd) |\!)\,\mathcal{Q} & \stackrel{\text{def}}{=} & \mathit{combine}_{\mathcal{Q}'} \circ \mathit{guarantee}_{(\!|^{\ell_2} cmd |\!)} \circ \mathit{child\text{-}spawn}_{\ell_2}(\mathcal{Q}) \\
\text{with } \mathcal{Q}' & \stackrel{\text{def}}{=} & \mathit{spawn}_{\ell_2}(\mathcal{Q})
\end{array}
$$

The following lemma ensures us that $\mathit{elem\text{-}inter}_{\mathit{action}}$ gives us an abstraction of the set of transition $\mathtt{G}_{\text{new}}$ introduced in Proposition 11.1 and in Definition 12.1.

Lemma 13.1. *Let:*

- $\langle \mathtt{S}', \mathtt{G}', \mathtt{A}' \rangle = [\![|\mathit{action}|]\!](\gamma_{cfg}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E} \rangle)$
- $\langle \mathcal{C}', \mathcal{L}', \mathcal{K}', I', \mathcal{E}' \rangle = \mathit{basic}_{\mathit{action}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E} \rangle$

[3] We can also use a narrowing operator. We do not give the details on narrowing here, this is an orthogonal problem.

$$
\begin{array}{rcl}
\mathit{basic}_{\mathit{action}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle & \stackrel{\text{def}}{=} & \langle \mathit{inter}_I \circ \mathit{elem}_{\mathit{action}}(\mathcal{C}), \mathcal{L}, \mathcal{K} \sqcup \mathcal{K}_{\text{new}}, I, \mathcal{E} \cup \mathit{error}(\mathcal{C})\rangle \\
\text{with } \mathcal{K}_{\text{new}} & \stackrel{\text{def}}{=} & \lambda \ell. \begin{cases} \mathit{elem\text{-}inter}_{\mathit{action}}(\mathcal{C}) & \text{if } \ell \in \mathcal{L} \\ \bot & \text{if } \ell \notin \mathcal{L} \end{cases} \\
\mathit{guard}_{\mathit{cond}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle & \stackrel{\text{def}}{=} & \langle \mathit{inter}_I \circ \mathit{enforce}_{\mathit{cond}}(\mathcal{C}), \mathcal{L}, \mathcal{K}, I, \mathcal{E} \cup \mathit{error}_{\mathit{cond}}(\mathcal{C})\rangle \\
\mathit{spawn}_{\ell}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle & \stackrel{\text{def}}{=} & \langle \mathcal{C}, \mathcal{L} \cup \{\ell\}, \mathcal{K}, I, \mathcal{E}\rangle \\
\mathit{child\text{-}spawn}_{\ell}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle & \stackrel{\text{def}}{=} & \langle \mathit{inter}_{I \sqcup \mathcal{K}(\ell)}(\mathcal{C}), \mathcal{L}, \lambda \ell. \bot, I \sqcup \mathcal{K}(\ell), \emptyset\rangle \\
\mathit{combine}_{\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle}(\mathcal{K}', \mathcal{E}') & \stackrel{\text{def}}{=} & \langle \mathit{inter}_{I \sqcup \mathcal{K}'(\ell_\star)}(\mathcal{C}), \mathcal{L}, \mathcal{K} \sqcup \mathcal{K}', I \sqcup \mathcal{K}'(\ell_\star), \mathcal{E} \cup \mathcal{E}'\rangle \\
\mathit{execute\text{-}thread}_{f^\sharp, \mathcal{C}, \mathcal{L}, I}(\mathcal{K}, \mathcal{E}) & \stackrel{\text{def}}{=} & (\mathcal{K}', \mathcal{E}') \\
\text{with } \langle \mathcal{C}', \mathcal{L}', \mathcal{K}', I', \mathcal{E}'\rangle & \stackrel{\text{def}}{=} & f^\sharp(\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle) \\
\mathit{guarantee}_{f^\sharp}(\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle) & \stackrel{\text{def}}{=} & \mathit{execute\text{-}thread}^{\uparrow\triangledown}_{f^\sharp, \mathcal{C}, \mathcal{L}, I}(\mathcal{K}, \mathcal{E})
\end{array}
$$

Figure 13.3: Basic Abstract Semantic Functions

- $\mathtt{G}_{new} = \{(s, s') \in \mathcal{T}r_{\ell_1\, \mathit{action}, \ell_2} \mid s \in \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\}$.

Therefore:

$$
\begin{array}{rcl}
\alpha_{\mathscr{R}}(\mathtt{G}_{new}) & \leqslant & \mathit{elem\text{-}inter}_{\mathit{action}}(\mathcal{C}) \\
\alpha_{E}(\mathtt{G}_{new}) & \leqslant & \mathit{error}_{\mathit{action}}(\mathcal{C})
\end{array}
$$

Proof. Since abstract configurations are saturated with respect to interferences (See Definition 13.1), $\mathtt{interfere}_{\mathtt{A}}(\gamma_{\mathscr{D}}(\mathcal{C})) = \gamma_{\mathscr{D}}(\mathcal{C})$.
Therefore $\mathtt{G}_{\text{new}} \subseteq \{(s, s') \in \mathcal{T}r_{\ell_1 \mathit{basic}, \ell_2} \mid s \in \gamma_{\mathscr{D}}(\mathcal{C})\}$.
Hence $\alpha_{\mathscr{R}}(\mathtt{G}_{\text{new}}) \leqslant \mathit{elem\text{-}inter}_{\ell_{lv:=e, \ell'}}(\mathcal{C})$ and $\alpha_{\mathscr{R}}(\mathcal{E}) \leqslant \mathit{error}_{\mathit{action}}(\mathcal{C})$. □

Recall that, given a set of transitions $\mathtt{G}$ we need to abstract precisely the subset $\mathtt{G}_{|\mathtt{post}(\ell)}$. The following lemma shows the link between α_{L} and $\mathtt{G}_{|\mathtt{post}(\ell)}$.

Lemma 13.2. *Given a set of transitions* $\mathtt{G}$*:*

$$\mathtt{G}_{|\mathtt{post}(\ell)} = \emptyset \Leftrightarrow \ell \notin \alpha_L(\{s \mid \exists s' : (s, s') \in \mathtt{G}\})$$

Proof. Let $\mathtt{S} = \{s \mid \exists s' : (s, s') \in \mathtt{G}\}$. According to Definition 2.4, $\mathtt{G}_{|\mathtt{post}(\ell)} = \emptyset$ is equivalent to $\mathtt{post}(\ell) \cap \mathtt{S} = \emptyset$. This is equivalent to $\ell \notin \alpha_{\mathrm{L}}(\mathtt{S})$. □

The function *inter* abstracts `interfere` for the abstract lattice $\mathscr{R}$. As showned by the following lemma, the identity function $id : \mathcal{P}(\mathbb{L}) \to \mathcal{P}(\mathbb{L})$ is an abstraction of `interfere` for the abstract lattice $\mathcal{P}(\mathbb{L})$.

Lemma 13.3. $\alpha_L(\mathtt{S}) = \alpha_L(\mathtt{interfere}_{\mathtt{A}}(\mathtt{S}))$.

Proof. This is a consequence of Lemma 10.11. □

The $\mathit{basic}_{\ell\,\mathit{action},\ell'}$ function updates $\mathcal{K}$ by adding the modification of the store to all labels encountered so far (those which are in $\mathcal{L}$). It does not change $\mathcal{L}$ because no thread is created. Notice that in the case of a non-relational store, we can simplify function basic using the fact that $\mathit{inter}_I \circ \mathit{elem}_{x:=e}(\mathcal{C}) = \mathcal{C}[x \mapsto \mathit{val}_{\mathcal{C}}(e) \sqcup I(x)]$. The following lemma proves that this function is a correct abstraction of the semantics of ${}^{\ell}\mathit{action}, \ell'$.

Proposition 13.1. *For all labels ℓ and ℓ', $\mathit{basic}_{\mathit{action}}$ is an abstraction of $[\![|{}^{\ell}\mathit{action}, \ell'|]\!]$.*

Proof. Let us consider that $\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle = (\gamma_{\mathrm{cfg}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle)$, and $\langle \mathtt{S}', \mathtt{G}', \mathtt{A}'\rangle = [\![|{}^{\ell}\mathit{action}, \ell'|]\!]\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle$ and $\langle \mathcal{C}', \mathcal{L}', \mathcal{K}', I', \mathcal{E}'\rangle = \mathit{basic}_{\mathit{action}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle$.

- Let us prove that $\alpha_{\mathscr{D}}(\mathtt{S}') \leqslant \mathcal{C}'$.

  By Definition 12.1, $\mathtt{S}' = \mathtt{interfere}_{\mathtt{A}}\big(\mathcal{T}\!\mathit{r}^{-}_{\ell\,\mathit{action},\ell'}\langle \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\rangle\big)$. Therefore, by definition, $\alpha(S') \leqslant \mathit{inter}_I \circ \mathit{elem}_{\mathit{action}} \circ \mathit{inter}_I(\mathcal{C})$. But abstract configurations are saturated with respect to interferences, therefore: $\alpha(S') \leqslant \mathit{inter}_I \circ \mathit{elem}_{\mathit{action}}(\mathcal{C})$. Hence $\alpha_{\mathscr{D}}(\mathtt{S}') \leqslant \mathcal{C}'$.

- Let us prove that $\alpha_{\mathrm{L}}(\mathtt{S}') \leqslant \mathcal{L}' = \mathcal{L}$. According to Lemma 13.3, $\alpha_{\mathrm{L}}(\mathtt{S}') = \alpha_{\mathrm{L}}(\mathtt{interfere}_{\mathtt{A}}(\mathtt{S}))$. Since transitions of $\mathcal{T}\!\mathit{r}^{-}_{\ell\,\mathit{action},\ell'}$ does not modify the current thread or the genealogy (See Lemma 7.6), $\alpha_{\mathrm{L}}(\mathcal{T}\!\mathit{r}^{-}_{\ell\,\mathit{action},\ell'}\langle \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\rangle = \alpha_{\mathrm{L}}(\mathtt{interfere}_{\mathtt{A}}(\mathtt{S})$. And we conclude using a second time Lemma 13.3

- Let us prove that $\alpha_{\mathrm{K}}(\mathtt{G}') \leqslant \mathcal{K}$.

  According to Definition 12.1, $\mathtt{G}' = \mathtt{G} \cup \mathtt{G}_{\mathrm{new}}$ with $\mathtt{G}_{\mathrm{new}} = \{(s, s') \in \mathcal{T}\!\mathit{r}_{\ell_1 \mathit{basic},\ell_2} \mid s \in \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\}$. By Lemma 13.1, $\alpha_{\mathscr{R}}(\mathtt{G}_{\mathrm{new}}) \leqslant \mathit{elem\text{-}inter}_{\ell\, lv:=e,\ell'}(\mathcal{C})$.

  $$\alpha_{\mathrm{K}}(\mathtt{G}_{\mathrm{new}}) = \lambda\ell.\alpha_{\mathscr{R}}(\mathtt{G}_{\mathrm{new}\,|\mathtt{post}(\ell)}) \quad \leqslant \lambda\ell. \begin{cases} \bot & \text{if } \mathtt{G}_{\mathrm{new}\,|\mathtt{post}(\ell)} = \emptyset \\ \alpha_{\mathscr{R}}(\mathtt{G}_{\mathrm{new}}) & \text{otherwise} \end{cases}$$

  Notice that, due to Lemma 13.2:

  $$\mathtt{G}_{\mathrm{new}\,|\mathtt{post}(\ell)} = \emptyset \Leftrightarrow \ell \notin \alpha_{\mathrm{L}}(\mathtt{interfere}_{\mathtt{A}}(\mathtt{S})).$$

  Due to Lemma 13.3:

  $$\mathtt{G}_{\mathrm{new}\,|\mathtt{post}(\ell)} = \emptyset \Leftrightarrow \ell \notin \alpha_{\mathrm{L}}(\mathtt{S}).$$

  Therefore

  $$\alpha_{\mathrm{K}}(\mathtt{G}_{\mathrm{new}}) \quad \leqslant \lambda\ell. \begin{cases} \bot & \text{if } \ell \notin \alpha_{\mathrm{L}}(\mathtt{S}). \\ \alpha_{\mathscr{R}}(\mathtt{G}_{\mathrm{new}}) & \text{otherwise} \end{cases}$$

- Let us prove that $\alpha_{\mathscr{R}}(\mathtt{A}') \leqslant I'$. This is obvious since $\mathtt{A} = \mathtt{A}'$ and $I = I'$ and $\alpha_{\mathscr{R}}(\mathtt{A}) \leqslant I$.

Let us prove that $\alpha_{\mathrm{E}}(\mathtt{G}') \leqslant \mathcal{E}'$. This is a consequence of Lemma 13.1. □

The guards are abstracted in the same way:

Proposition 13.2. *For all labels ℓ and ℓ', $\mathit{guard}_{\ell\, cond,\ell'}$ is an abstraction of $[\![|^{\ell}\mathsf{guard}(cond), \ell'|]\!]$.*

Proof. Same as Proposition 13.1, using the facts that, by hypothesis on the abstract lattice of transitions $\mathscr{R}$: $\alpha_{\mathscr{R}}(\mathcal{T}r_{\ell\, \mathsf{guard}(cond),\ell'}) = \bot$ □

Proposition 13.3. *spawn_{ℓ_2} is an abstraction of $[\![|^{\ell_1}\mathsf{spawn}(\ell_2), \ell_3|]\!]$*

Proof. Let $\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle = (\gamma_{\mathrm{cfg}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle)$, and $\langle \mathtt{S}', \mathtt{G}', \mathtt{A}'\rangle = [\![|^{\ell_1}\mathsf{spawn}(\ell_2), \ell_3|]\!]\langle \mathtt{S}, \mathtt{G}, \mathtt{A}\rangle$ and $\langle \mathcal{C}', \mathcal{L}', \mathcal{K}', I', \mathcal{E}'\rangle = \mathit{spawn}_{\ell_2}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle$.

- Let us prove that $\alpha_{\mathscr{D}}(\mathtt{S}') \leqslant \mathcal{C}' = \mathcal{C}$.

  By Definition 12.1, $\mathtt{S}' = \mathtt{interfere}_{\mathtt{A}}\big(\mathcal{T}\tilde{r}_{\ell_1\, \mathsf{spawn}(\ell_2),\ell_3}\langle \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\rangle\big)$. Furthermore, $\mathcal{T}\tilde{r}_{\ell_1\, \mathsf{spawn}(\ell_2),\ell_3}$ does not modify the current thread and does not modify the store.

  Therefore, by definition, $\alpha(S') \leqslant \mathit{inter}_I \circ \mathit{inter}_I(\mathcal{C}) = \mathcal{C}$.

- Let us prove that $\alpha_{\mathrm{L}}(\mathtt{S}') \leqslant \mathcal{L}' = \mathcal{L}$. According to Lemma 13.3, $\alpha_{\mathrm{L}}(\mathtt{S}') = \alpha_{\mathrm{L}}(\mathtt{interfere}_{\mathtt{A}}(\mathtt{S}))$. $\alpha_{\mathrm{L}}(\mathcal{T}\tilde{r}_{\ell\, \mathsf{create},\ell'}\langle \mathtt{interfere}_{\mathtt{A}}(\mathtt{S})\rangle = \alpha_{\mathrm{L}}(\mathtt{interfere}_{\mathtt{A}}(\mathtt{S}) \cup \{\ell_2\}$. And we conclude using a second time Lemma 13.3

- Let us prove that $\alpha_{\mathrm{K}}(\mathtt{G}') \leqslant \mathcal{K}'$. Same proof as for Proposition 13.1, using the fact that transitions generated by the statement $^{\ell_1}\mathsf{spawn}(\ell_2), \ell_3$ have no effect on the current thread nor the store.

- We prove that $\alpha_{\mathscr{R}}(\mathtt{A}') \leqslant I'$ and $\alpha_{\mathrm{E}}(\mathtt{G}') \leqslant \mathcal{E}'$ in the same way than for Proposition 13.1.

□

The functions of Fig. 13.3 abstract the corresponding functions of the G-collecting semantics (See Fig. 10.5).

Proposition 13.4. *The abstract functions $\mathit{child\text{-}spawn}_{\ell_2}$, $\mathit{combine}$ and $\mathit{guarantee}_{(\!|^{\ell}\, cmd,\ell'|\!)}$ are abstractions of the concrete functions $\mathtt{init\text{-}child}_{\ell_1} \circ [\![^{\ell_1}\mathsf{spawn}(\ell_2), \ell_3]\!]$, $\mathtt{combine}$ and $\mathtt{guarantee}_{[\![^{\ell}\, cmd,\ell_\infty]\!]}$ respectively.*

Proof. This is a consequence of Proposition 3.1. □

The abstract semantics is defined by induction on syntax, see Fig. 13.2:

Theorem 13.1 (Soundness)**.** *$(\!|cmd, \ell|\!)$ is an abstraction of $[\![cmd, \ell]\!]$.*

Proof. This is a consequence of Propositions 3.1, 13.1, 13.2, 13.3 and 13.4. □

CHAPTER 14

Abstract Domains for Sequential Consistency

We show some concrete and abstract stores that can be used in practice. For each domain, we give the abstract lattices $\mathscr{D}$ and $\mathscr{R}$, the Galois connections of Figure 13.2 and the abstract functions of Figure 13.1.

14.1 Maps

14.1.1 Main Abstraction

Concrete stores are described in Chapter 8.1. They are maps from the set of variables $\mathcal{V}ar$ to some set $\mathcal{V}$ of *concrete values*.

Abstract stores are maps from $\mathcal{V}ar$ to some complete lattice $\mathcal{V}^\sharp$ of *abstract values*, e.g., **NotZero** (see section 3.4), **Ranges** [CC04] (See Section 3.2), or string lengths [AGH06]. Abstract stores are ordered by the pointwise ordering (recall Definition 2.10). Assuming a Galois connection α_V, γ_V between $\mathcal{V}$ and $\mathcal{V}^\sharp$, we define a Galois connection $\alpha_{\text{map}}, \gamma_{\text{map}}$

between set of concrete stores and abstract stores:

$$\begin{aligned}\alpha_{\text{map}}(\{\sigma\}) &\stackrel{\text{def}}{=} \lambda x.\alpha_V(\sigma(x))\\ \gamma_{\text{map}}(\sigma^\sharp) &\stackrel{\text{def}}{=} \{\sigma \mid \forall x, \sigma(x) \in \gamma_V(\sigma^\sharp(x))\}.\end{aligned}$$

Both *abstract states* and *abstract transitions* are encoded as abstract stores, i.e., $\mathscr{D} = \mathscr{R} = (\mathcal{V}^\sharp)^{\mathcal{V}ar}$. As required, abstract states depends only of stores and current threads. Actually, they depend only of stores.

$$\begin{aligned}\alpha_{\mathscr{D}}(\{(i,P,\sigma,g)\}) &\stackrel{\text{def}}{=} \alpha_{\text{map}}(\sigma)\\ \gamma_{\mathscr{D}}(\sigma^\sharp) &\stackrel{\text{def}}{=} \{(i,P,\sigma,g) \mid \sigma \in \gamma_{\text{map}}(\sigma^\sharp)\}.\end{aligned}$$

The abstraction of transitions remembers only information on modified variable: it recalls the possible new values of a written variable.

$$\alpha_{\mathscr{R}}(\{((i,P,\sigma,g),(i',P',\sigma',g'))\}) \stackrel{\text{def}}{=} \lambda x. \begin{cases}\alpha_V(\sigma'(x)) & \text{if } \sigma'(x) \neq \sigma(x)\\ \bot & \text{otherwise}\end{cases}$$

$$\gamma_{\mathscr{R}}(\sigma^\sharp) \stackrel{\text{def}}{=} \{((i,P,\sigma,g),(i',P',\sigma',g')) \mid \forall x \in \mathcal{V}ar, \sigma'(x) = \sigma(x) \vee \sigma'(x) \in \gamma_{\text{map}}(\sigma^\sharp)\}.$$

We give for these domains the primitives of Fig. 13.1. Let $val_{\mathcal{C}}(e)$ and $addr_{\mathcal{C}}(lv)$ be the abstract value of the expression e and the set of variables that may be represented by lv, respectively, in the context $\mathcal{C}$; and let $true^\sharp$ and $false^\sharp$ be the abstractions of *true* and *false* respectively.

$$\begin{aligned}\mathit{elem}_{x:=e}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C}[x \mapsto val_{\mathcal{C}}(e)]\\ \mathit{elem}_{lv:=e}(\mathcal{C}) &\stackrel{\text{def}}{=} \bigsqcup_{x \in addr_{\mathcal{C}}(lv)} \mathit{elem}_{x:=e}(\mathcal{C})\\ \mathit{elem\text{-}inter}_{lv:=e}(\mathcal{C}) &\stackrel{\text{def}}{=} \lambda x. \begin{cases} val_{\mathcal{C}}(e) & \text{if } x \in addr_{\mathcal{C}}(lv)\\ bot & \text{otherwise}\end{cases}\\ \mathit{inter}_I(\mathcal{C}) &\stackrel{\text{def}}{=} I \sqcup \mathcal{C}\\ \mathit{enforce}_x(\mathcal{C}) &\stackrel{\text{def}}{=} \begin{cases}\mathcal{C}[x \mapsto \mathcal{C}(x) \sqcap true^\sharp] & \text{if } \mathcal{C}(x) \sqcap true^\sharp \neq \bot\\ \bot & \text{otherwise}\end{cases}\\ \mathit{enforce}_{\neg x}(\mathcal{C}) &\stackrel{\text{def}}{=} \begin{cases}\mathcal{C}[x \mapsto \mathcal{C}(x) \sqcap false^\sharp] & \text{if } \mathcal{C}(x) \sqcap false^\sharp \neq \bot\\ \bot & \text{otherwise}\end{cases}\\ \mathit{schedule\text{-}child}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathit{elem}_{\mathit{lock}(\mu)}(\mathcal{C}) \stackrel{\text{def}}{=} \mathit{elem}_{\mathit{unlock}(\mu)}(\mathcal{C}) \stackrel{\text{def}}{=} \mathcal{C}\\ \mathit{elem\text{-}inter}_{\mathit{lock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathit{elem\text{-}inter}_{\mathit{unlock}(\mu)}(\mathcal{C}) \stackrel{\text{def}}{=} \bot\end{aligned}$$

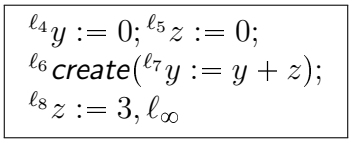


Figure 14.1: Example

The function *elem*() updates the abstract value of the modified variable.

Notice that this abstraction is a separate product (See Definition 3.4) of $card(\mathcal{V}ar)$ times the concrete lattice $\mathcal{V}ar$. The abstractions of guards take into account that, in a separate product, $(y, \bot)$, $(\bot, y)$ and $(\bot, \bot)$ have the same concretization. The abstraction of guard takes into account this point. If $x \sqcap false^\sharp$ is empty, this means that, in the concrete world, the system cannot pass the guard. Therefore, after the guard, the concrete value is $\emptyset$.

In this abstraction, we do not take into account the locks. But a simple product or even a reduced product with a domain for locks will allow us to take locks into account.

14.1.2 Errors

Errors will depend on what we want to detects. We give here a simple example, with

$$\textbf{Errors} = \{\textbf{division-by-zero}\}.$$

Given an assignment *assign* equal to $lv_0 := e_0$, we write $e \rightarrowtail\!\!\!\!\!\rightarrow \mathit{assign}$ to say that e is a subexpression of e_0 or of lv_0. The function *error* is defined by (all free variables are implicitly existentially quantified):

$$\mathit{error}_{\mathsf{a}}(\mathcal{C}) = \{\textbf{division-by-zero}\} \overset{\text{def}}{\Leftrightarrow} e_1 \% e_2 \rightarrowtail\!\!\!\!\!\rightarrow \mathsf{a} \vee e_1 / e_2 \rightarrowtail\!\!\!\!\!\rightarrow \mathsf{a} \wedge \mathit{val}_{\mathcal{C}}(e_2) = [l, u] \wedge l \leqslant 0 \leqslant u$$

14.1.3 Example

Consider the program of Fig. 14.1 and the abstract store **Ranges** of ranges [CC04]. We apply our algorithm to this example, giving a run-through (See Figure 14.2).

Our algorithm computes *execute-thread* (line 1 to 7). The fixpoint is not reached, so we compute *execute-thread* a second time (line 8 to 14). Then, the fixpoint is reached, as a third application of *execute-thread* will confirm (not shown).

We do not give $\mathcal{E}$ in this example since $\mathcal{E} = \emptyset$ for all lines.

14.2 Cartesian Abstraction

In Cartesian abstraction [MPR06b, MPR06a] (See 4.5.2), stores are maps. Nevertheless, the set of variable $\mathcal{V}ar$ is divided into two parts: shared variables $\mathcal{V}ar_{\text{shared}}$ and private

Line		$\mathcal{C}$	$\mathcal{L}$	$\mathcal{K}$	$\mathcal{I}$
1	Initial configuration	$y = ?$ $z = ?$	$\{\ell_\star\}$	$\bot$	$\bot$
2	$(\!\|^{\ell_4} y := 0, \ell_5 \|\!)$	$y = 0$ $z = ?$	$\{\ell_\star\}$	$\ell_\star \mapsto y = 0$	$\bot$
3	$(\!\|^{\ell_5} z := 0, \ell_6 \|\!)$	$y = 0$ $z = 0$	$\{\ell_\star\}$	$\ell_\star \mapsto y = 0, z = 0$	$\bot$
4	$\mathit{child\text{-}spawn}_{\ell_7}$	$y = 0$ $z = 0$	$\{\ell_\star\}$	$\bot$	$\bot$
5	$(\!\|^{\ell_7} y := y + z, \ell_\infty \|\!)$	$y = 0$ $z = 0$	$\{\ell_\star\}$	$\ell_\star \mapsto y = 0$	$\bot$
6	$\mathit{combine}_{\mathit{spawn}_{\ell_7}(\cdot)}$	$y = 0$ $z = 0$	$\{\ell_\star, \ell_7\}$	$\ell_\star \mapsto y = 0, z = 0$	$y = 0$
7	$(\!\|^{\ell_8} z := 3, \ell_\infty \|\!)$	$y = 0$ $z = 3$	$\{\ell_\star, \ell_7\}$	$\ell_\star \mapsto \begin{cases} y = 0 \\ z = [0,3] \end{cases}$ $\ell_7 \mapsto z = 3$	$y = 0$
8	Initial configuration	$y = ?$ $z = ?$	$\{\ell_\star\}$	$\ell_\star \mapsto \begin{cases} y = 0 \\ z = [0,3] \end{cases}$ $\ell_7 \mapsto z = 3$	$\bot$
9	$(\!\|^{\ell_4} y := 0, \ell_5 \|\!)$	$y = 0$ $z = ?$	$\{\ell_\star\}$	$\ell_\star \mapsto \begin{cases} y = 0 \\ z = [0,3] \end{cases}$ $\ell_7 \mapsto z = 3$	$\bot$
10	$(\!\|^{\ell_5} z := 0, \ell_6 \|\!)$	$y = 0$ $z = 0$	$\{\ell_\star\}$	$\ell_\star \mapsto \begin{cases} y = 0 \\ z = [0,3] \end{cases}$ $\ell_7 \mapsto z = 3$	$\bot$
11	$\mathit{child\text{-}spawn}_{\ell_7}$	$y = 0$ $z = [0,3]$	$\{\ell_\star\}$	$\bot$	$z = 3$
12	$(\!\|^{\ell_7} y := y + z, \ell_\infty \|\!)$	$y = [0,3]$ $z = [0,3]$	$\{\ell_\star\}$	$\ell_\star \mapsto y = [0,3]$	$z = 3$
13	$\mathit{combine}_{\mathit{spawn}_{\ell_7}(\cdot)}$	$y = [0,3]$ $z = 0$	$\{\ell_\star, \ell_7\}$	$\ell_\star \mapsto \begin{cases} y = [0,3] \\ z = [0,3] \end{cases}$ $\ell_7 \mapsto z = 3$	$y = [0,3]$
14	$(\!\|^{\ell_8} z := 3, \ell_\infty \|\!)$	$y = [0,3]$ $z = 3$	$\{\ell_\star, \ell_7\}$	$\ell_\star \mapsto \begin{cases} y = [0,3] \\ z = [0,3] \end{cases}$ $\ell_7 \mapsto z = 3$	$y = [0,3]$

Figure 14.2: Abstract Example

variables $\mathcal{V}ar_{\text{private}}$. each thread has its own copy of the private variables as in OpenMP model [Boa08].

The set *GlobalStore* is the set of maps from $\mathcal{V}ar_{\text{shared}}$ to the set of concrete values $\mathcal{V}$ and *Localstore* is the set of maps from $\mathcal{V}ar_{\text{shared}}$ to the set of concrete values. Finally, the set of stores is defined by:

$$\textbf{Stores} = \mathit{GlobalStore} \times \mathit{Localstore}^{\textbf{Ids}}.$$

In this concrete model, a thread may only reads the shared variable and its own private variables.

The abstract lattices $\mathscr{D}$ and $\mathscr{R}$ are defined as following:

$$\begin{aligned} \mathscr{D} &\stackrel{\text{def}}{=} \mathcal{P}(\mathit{GlobalStore} \times \mathit{Localstore}) \\ \mathscr{R} &\stackrel{\text{def}}{=} \mathcal{P}(\mathit{GlobalStore} \times \mathit{GlobalStore}) \end{aligned}$$

To define Galois connections, we just need to define the abstraction function on singletons (as shown in 3.2). On states, $\alpha_{\mathscr{D}}$ forget all information on the private variables of other threads. It keeps information only on global variables and on private variables of the current thread:

$$\alpha_{\mathscr{D}}(\{(i, P, (\text{globs}, ls), g)\}) \stackrel{\text{def}}{=} (\text{globs}, ls(i))$$

The abstract transitions keep information on how the global variables are modified. The link between local and global variables is lost:

$$\alpha_{\mathscr{R}}(\{\big((i, P, (\text{globs}, ls), g), (i', P', (\text{globs}', ls'), g')\big)\}) \stackrel{\text{def}}{=} (\text{globs}, \text{globs}').$$

$$\begin{aligned} \mathit{elem}_{\mathit{action}}(\mathcal{C}) &\stackrel{\text{def}}{=} \alpha_{\mathscr{D}}\big(\mathcal{T}\!\mathit{r}_{\ell_{\mathit{action},\ell'}}\langle\gamma_{\mathscr{D}}(\mathcal{C})\rangle\big) \\ \mathit{elem\text{-}inter}_{\mathit{action}}(\mathcal{C}) &\stackrel{\text{def}}{=} \alpha_{\mathscr{R}}((\mathcal{T}\!\mathit{r}_{\ell_{\mathit{action},\ell'}})_{|\gamma_{\mathscr{D}}(\mathcal{C})}) \\ \mathit{schedule\text{-}child}(\mathcal{C}) &\stackrel{\text{def}}{=} \{(glob, l) \mid \exists glob_2, l_2 : (glob, l_2) \in \mathcal{C} \wedge (glob_2, l) \in \textbf{StoresInit}\} \end{aligned}$$

14.3 Gen/Kill Analyses

In such analyses [SS00] the set of stores is a complete lattice, e.g., sets of initialized variables, sets of edges of a point-to graph (See Section 8.2 and Section 4.6). As for maps, the abstract states and abstract transitions are the same: $\mathscr{D} = \mathscr{R} = \textbf{Stores}\mathcal{V}$.

Each gen/kill analysis gives, for each basic action, two elements of the lattice **Stores**:

- $\mathtt{gen}(\mathit{action}, \sigma)$
- and $\mathtt{keep}(\mathit{action}, \sigma)$.

These sets may take the current store σ into account (e.g. Rugina and Rinard's "strong flag" [RR99, RR03]); *gen*. The Galois connections are defined by:

$$\begin{array}{rcl}
\alpha_{\mathscr{D}}(\{(i, P, \sigma, g)\}) & \stackrel{\text{def}}{=} & \{\sigma\} \\
\alpha(\{((i_1, P_1, \sigma_1, g_1), (i_2, P_2, \sigma_2, g_2))\}) & \stackrel{\text{def}}{=} & \bigcap_{\sigma : \sigma_2 \leqslant \sigma_1 \sqcup \sigma} \sigma \\
\gamma_{\mathscr{R}}(\sigma^\sharp) & \stackrel{\text{def}}{=} & \left\{ \big((i_1, P_1, \sigma_1, g_1), (i_2, P_2, \sigma_2, g_2)\big) \mid \sigma_2 \leqslant \sigma_1 \sqcup \sigma^\sharp \right\}
\end{array}$$

$$\begin{array}{rcl}
\mathit{elem}_{\mathit{action}}(\mathcal{C}) & \stackrel{\text{def}}{=} & (\mathcal{C} \sqcap \mathtt{keep}(\mathit{action}, \sigma)) \cup \mathtt{gen}(\mathit{action}, \sigma) \\
\mathit{elem\text{-}inter}_{\mathit{action}}(\mathcal{C}) & \stackrel{\text{def}}{=} & \mathtt{gen}(\mathit{action}, \sigma) \\
\mathit{inter}_I(\mathcal{C}) & \stackrel{\text{def}}{=} & I \sqcup \mathcal{C} \\
\mathit{enforce}_x(\mathcal{C}) & \stackrel{\text{def}}{=} & \mathcal{C}
\end{array}$$

Notice that, as it is standard in Gen/Kill analyses [SS00, LMO07], these domains model *if* statements by non-deterministic choices.

CHAPTER 15

Abstraction for Weak Memory Models

We define abstractions for weak memory models. These abstractions are similar to those given for interleaving semantics. This chapter gives only the difference between abstraction for weak memory models and abstractions of Chapter 13. Notice that our model is designed to handle both strong and weak memory models, this is why this chaper is brief: only few modifications are needed to handle TSO and PSO models.

We also assume an abstract lattice of states $\mathscr{D}$ and an abstract lattice of transitions $\mathscr{R}$.

In weak memory models, we have write operations. these operations may be protected by a lock:

Definition 15.1. A lock protects a write operation *op* if it is held when *op* is in the buffer. $\mathbf{ProtWrite} = \mathcal{P}(\mathbf{WriteOp} \times \mathcal{P}(\mathbf{Locks}))$ is the set of write operations *protected* by locks.

Given the abstract lattices $\mathscr{D}$ for states and $\mathscr{R}$ for transitions we consider six Galois connections described in Fig. 15.1. Notice that this is the same Galois connections as for interleaving semantics (Recall Figure 13.2) plus two new Galois connections $\alpha_{\text{op}}, \gamma_{\text{op}}$ and $\alpha_{\text{buf}}, \gamma_{\text{buf}}$. We assume that the three first Galois connections are given. We require $\alpha_{\mathscr{R}}$ and

Name	Concrete Elements	Abstract Elements	Definitions
$\gamma_{\mathscr{D}}, \alpha_{\mathscr{D}}$	$\mathtt{S} \in \mathcal{P}(\textbf{States})$	$\mathcal{C} \in \mathscr{D}$	
$\gamma_{\mathscr{R}}, \alpha_{\mathscr{R}}$	$\mathtt{A} \in \mathcal{P}(\textbf{Transitions})$	$I \in \mathscr{R}$	
$\gamma_{\text{op}}, \alpha_{\text{op}}$	**ProtWrite**	$I \in \mathscr{R}$	
γ_{buf}, α_{buf}	$\mathcal{P}(\textbf{States})$	$I \in \mathscr{R}$	$\alpha_{\text{buf}}(\{s\}) = \bigsqcup_{j \neq i} \bigsqcup_{op \in b(j)} \alpha_{\text{op}}\{(op, \mathtt{mutex}(j,s))\}$ where $s = (i, P, (m, b), g)$
$\gamma_{\text{L}}, \alpha_{\text{L}}$	$\mathtt{S} \in \mathcal{P}(\textbf{States})$	$\mathcal{L} \in \mathcal{P}(\mathbb{L})$	$\alpha_{\text{L}}(\mathtt{S}) = \{\ell \in \mathbb{L} \mid \mathtt{S} \cap \mathtt{post}(\ell) \neq \emptyset\}$ $\gamma_{\text{L}}(\mathbb{L}) = \textbf{States}$ $\gamma_{\text{L}}(\mathcal{L}) = \bigcap_{\ell \in \mathbb{L} \setminus \mathcal{L}} \overline{\mathtt{post}(\ell)}$
$\gamma_{\text{K}}, \alpha_{\text{K}}$	$\mathtt{G} \in \mathcal{P}(\textbf{Transitions})$	$\mathcal{K} \in \mathscr{R}^{\mathbb{L}}$	$\alpha_{\text{K}}(\mathtt{G}) = \lambda \ell . \alpha_{\mathscr{R}}(\mathtt{G}_{\mid \mathtt{post}(\ell)})$ $\gamma_{\text{K}}(\mathcal{K}) = \{(s, s') \in \textbf{Transitions} \mid \forall \ell \in \mathbb{L},$ $\in \mathtt{post}(\ell) \Rightarrow (s, s') \in \gamma_{\mathscr{R}}(\mathcal{K}(\ell))\}$

Figure 15.1: Galois Connections for Weak memory Models

α_{op} (from which α_{buf} is defined) to be *compatible* in the sense that:

$$\alpha_{\text{buf}}(\mathtt{interfere}_{\mathtt{A}}(\mathtt{S})) \leqslant \alpha_{\mathscr{R}}(\mathtt{A}) \sqcup \alpha_{\text{buf}}(\mathtt{S}).$$

This requirement states that applying interferences in $\mathtt{A}$ to $\mathtt{S}$ in the abstract can be computed by combining the effect of interferences ($\alpha_{\mathscr{R}}$) with effects that are pending from buffers in current states ($\alpha_{\text{buf}}(\mathtt{S})$). The requirement will be satisfied in all examples.

For buffer abstraction α_{buf} we introduce the set of locks owned by a thread j in a state $s = (i, P, (m, b), g)$:

$$\mathtt{mutex}(j, s) \stackrel{\text{def}}{=} \{\mu \in \textbf{Locks} \mid m(\mu) = j\}.$$

Intuitively $\alpha_{\text{buf}}(\{s\})$ represents how the thread buffers other than the current thread can modify the memory meaning both locks and pending writes.

The set of abstract configurations is the same as for the interleaving semantics, defined in definition 13.1. Nevertheless, the Galois connection is not the same:

Definition 15.2. *Abstract configurations* are tuples $\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E} \rangle \in \mathscr{D} \times \mathcal{P}(\textbf{Labels}) \times \mathscr{R}^{\textbf{Labels}} \times \mathscr{R} \times \mathcal{P}(\textbf{Errors})$ and $\mathcal{C}$ is saturated with respect to interferences, i.e., $\mathit{inter}_I(\mathcal{C}) = \mathcal{C}$ and $\ell_\star \in \mathcal{L}$. The Galois connection between concrete[4] and abstract configurations is:

$$\alpha_{\text{cfg}}\langle \mathtt{S}, \mathtt{G}, \mathtt{A} \rangle \stackrel{\text{def}}{=} \langle \mathit{inter}_{\alpha_{\mathscr{R}}(\mathtt{A})}(\alpha_{\mathscr{D}}(\mathtt{S})), \alpha_{\text{L}}(\mathtt{S}), \alpha_{\text{K}}(\mathtt{G}), \alpha_{\mathscr{R}}(\mathtt{A}) \sqcup \alpha_{\text{buf}}(\mathtt{S}), \alpha_{\text{E}}(\mathtt{G}) \rangle$$
$$\gamma_{\text{cfg}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E} \rangle \stackrel{\text{def}}{=} \langle \gamma_{\mathscr{D}}(\mathcal{C}) \cap \gamma_{\text{L}}(\mathcal{L}) \cap \gamma_{\text{buf}}(I), \gamma_{\text{K}}(\mathcal{K}) \cap \gamma_{\text{E}}(\mathcal{E}), \gamma_{\mathscr{R}}(I) \rangle$$

For weak memory models, we need the primitives of Fig. 15.2. Notice that these primitives are the same as those given in Figure 13.1 in Section 14.1, except for *error*. Since

[4]Recall definition 12.3.

Concrete function	Abstract function
$\lambda \mathtt{S}.\mathcal{T}\bar{r}_{\ell\,\mathit{action},\ell'}\langle \mathtt{S}\rangle$	$\mathit{elem}_{\mathit{action}} : \mathscr{D} \to \mathscr{D}$
$\lambda \mathtt{S}.(\mathcal{T}\bar{r}_{\ell\,\mathit{action},\ell'})_{\mid \mathtt{S}}$	$\mathit{elem\text{-}inter}_{\mathit{action}} : \quad \mathscr{D} \to \mathscr{R}$
$\lambda \mathtt{S}.\mathcal{T}\bar{r}_{\ell\,\mathit{guard}(\mathit{cond}),\ell'}\langle \mathtt{S}\rangle$	$\mathit{enforce}_{\mathit{cond}} : \mathscr{D} \to \mathscr{D}$
$\lambda \mathtt{A}, \mathtt{S}.\mathtt{interfere}_{\mathtt{A}}(\mathtt{S})$	$\mathit{inter} : \mathscr{R} \times \mathscr{D} \to \mathscr{D}$
$\mathtt{schedule\text{-}child}_{\ell}$	$\mathit{schedule\text{-}child} : \mathscr{D} \to \mathscr{D}$
$\lambda \mathtt{S}.\alpha_{\mathrm{E}}[(\mathcal{T}\bar{r}_{\ell\,\mathit{action},\ell'})_{\mid \mathtt{S}}]$	$\mathit{error}_{\mathit{action}} : \mathscr{D} \times \mathscr{R} \to \mathcal{P}(\mathbf{Errors})$
$\lambda \mathtt{S}.\alpha_{\mathrm{E}}[(\mathcal{T}\bar{r}_{\ell\,\mathit{action},\ell'})_{\mid \mathtt{S}}]$	$\mathit{error}_{\mathit{cond}} : \mathscr{D} \times \mathscr{R} \to \mathcal{P}(\mathbf{Errors})$

Figure 15.2: Given Abstractions For Weak Memory Models

$$
\begin{array}{rcl}
\mathit{basic}_{\mathit{action}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle & \stackrel{\text{def}}{=} & \langle \mathit{inter}_I \circ \mathit{elem}_{\mathit{action}}(\mathcal{C}), \mathcal{L}, \mathcal{K} \sqcup \mathcal{K}_{\text{new}}, I, \mathcal{E} \cup \mathit{error}(\mathcal{C}, I)\rangle \\
\text{with } \mathcal{K}_{\text{new}} & \stackrel{\text{def}}{=} & \lambda \ell. \begin{cases} \mathit{elem\text{-}inter}_{\mathit{action}}(\mathcal{C}) & \text{if } \ell \in \mathcal{L} \\ \bot & \text{if } \ell \notin \mathcal{L} \end{cases} \\
\mathit{guard}_{\mathit{cond}}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle & \stackrel{\text{def}}{=} & \langle \mathit{inter}_I \circ \mathit{enforce}_{\mathit{cond}}(\mathcal{C}), \mathcal{L}, \mathcal{K}, I, \mathcal{E} \cup \mathit{error}_{\mathit{cond}}(\mathcal{C}, I)\rangle
\end{array}
$$

Figure 15.3: Basic Abstract Semantic Functions for Weak memory Models

error abstracts states, in the sequential consistent model, this function takes as argument an abstract state in $\mathscr{D}$. But, in the weak memory model abstractions, states are abstracted with $\alpha_{\mathscr{D}}$ and with α_{buf}, hence, *error* can take an extra argument: a set of states abstracted with α_{buf}. Furthermore, $\alpha_{\mathscr{D}}$ may forget (E.g., Maps in Section 16.1) all information about buffers, and keep only information on the view of the memory by the current thread. With its new argument, the function *error* may detect an error that depends on buffers and not only on the view (E.g. Section 16.2.2).

To define the abstract semantics, we use functions of Figure 13.3. But, in the case of a weak memory model, we must do some modifications: we give two arguments to the function *error*. The Figure 15.3 gives the difference between the sequential consistency case and the case of weak memory models. The abstract semantics is then defined as in sequential consistency case, see Definition 13.2.

CHAPTER 16

Abstract Domains for Weak Memory Models

As for sequential consistency, we give some examples of domains for weak memory models. Here we give some domains for TSO (recall Section 9.2), but this domain can be adapted for the PSO model (see Section 9.3).

16.1 Maps

The concrete stores are those described in Section 9.2.1.a. They are a pair (m, b) where m is a map from variables to values and b a function that maps threads to their write buffers.

Abstract memories are maps from $\mathcal{V}ar$ to some complete lattice $\mathcal{V}^\sharp$ of *abstract values*. As in Section 14.1, we assume a Galois connection α_V, γ_V between $\mathcal{V}$ and $\mathcal{V}^\sharp$, we reuse the Galois connection $\alpha_{\text{map}}, \gamma_{\text{map}}$ given in Section 14.1. between set of concrete memories and abstract memories.

Both *abstract states* and *abstract transitions* are encoded as abstract memories, i.e., $\mathscr{D} = \mathscr{R} = (\mathcal{V}^\sharp)^{\mathcal{V}ar}$. Abstract states only keep information about the view of the current

thread, ignoring the shared memory:

$$\begin{aligned} \alpha_{\mathscr{D}}(\{(i,P,\sigma,g)\}) &\stackrel{\text{def}}{=} \alpha_{\text{map}}(\mathit{view}(i,\sigma)) \\ \gamma_{\mathscr{D}}(\sigma^\sharp) &\stackrel{\text{def}}{=} \{(i,P,\sigma,g) \mid \mathit{view}(i,\sigma) \in \gamma_{\text{map}}(\sigma^\sharp)\}. \end{aligned}$$

Notice that we do not keep information on the shared memory, but only on the view of each thread. The Galois connection α_{buf} will recover information from buffers, which cannot be deduced from the view only.

To define $\alpha_{\mathscr{R}}$, introduce the function $\mathit{new\text{-}write} : \mathcal{T}r \to \mathcal{P}(\mathbf{WriteOp} \times \mathcal{P}(\mathbf{Locks}))$, that, given a transition τ returns the set of new protected writes in the buffer of the current thread (a singleton or the empty set):

$$\begin{aligned} \mathit{new\text{-}write}(\tau) &\stackrel{\text{def}}{=} \begin{cases} \{\big(op, \texttt{mutex}(i, s_1)\big)\} & \text{if } b'(i) = \mathit{enq}(op, b(i)) \\ \emptyset & \text{otherwise} \end{cases} \\ \text{where } \tau &= (s_1, s_2) = \big((i,P,(m,b),g),(i',P',(m',b'),g')\big). \end{aligned}$$

To abstract transitions and write buffers, we abstract the written variables and their new values:

$$\gamma_{\text{op}}(I) = \{((x,v),M) | v \in \gamma_V(I(x)) \wedge M \subseteq \mathbf{Locks}\}$$

$$\alpha_{\mathscr{R}}(\{\tau\}) = \alpha_{\text{op}}(\mathit{new\text{-}write}(\tau)).$$

Notice that $\alpha_{\text{op}}, \gamma_{\text{op}}$ do not keep information on locks. To handle locks, we need another domain.

16.2 Protected Variables

This abstraction infers which lock protects which variable. An error is raised when two different threads access the same variable (one is a write) but with disjoints sets of locks.

16.2.1 Lattice of Abstract States

We define an abstract complete lattice for locks: $\mathbf{SetLocks}^\downarrow = \mathcal{P}(\mathbf{Locks})$. An abstract state represents locks that *must* be held at some point, we order the lattice $\mathscr{D} = \mathbf{SetLocks}^\downarrow$ by the reverse inclusion ordering, i.e., $\mathcal{C}_1 \leqslant \mathcal{C}_2 \Leftrightarrow \mathcal{C}_1 \supseteq \mathcal{C}_2$. Hence $\mathcal{C}_1 \sqcup \mathcal{C}_2 = \mathcal{C}_1 \cap \mathcal{C}_2$. The Galois connection with concrete states is formally defined by:

$$\alpha_{\mathscr{D}}(\mathtt{S}) = \bigsqcup_{s \in \mathtt{S}} \texttt{mutex}(\mathit{thread}(s), s).$$

Functions of Fig. 13.1 are given here:

$$
\begin{aligned}
\mathit{elem}_{lv:=e}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \\
\mathit{enforce}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \\
\mathit{elem}_{\mathit{lock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \cup \{\mu\} \\
\mathit{schedule\text{-}child}(\mathcal{C}) &\stackrel{\text{def}}{=} \emptyset \\
\mathit{elem}_{\mathit{unlock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \setminus \{\mu\}
\end{aligned}
$$

Notice that $\mathit{elem}_{lv:=e}$ and *enforce* do nothing. This domain does not handle writes or guards.

16.2.2 Lattice of Abstract Transitions

We define the domain $\mathbf{ProtVars} = \mathscr{R}$ of *protected variables* for transitions. In this domain, each variable in $\mathcal{V}\mathit{ar}$ is abstracted by the set of locks that are held when that variable is accessed, i.e., the locks that protect the variable. Formally: $\mathscr{R} = \mathbf{SetLocks}^{\mathcal{V}\mathit{ar}}$. The Galois connection is defined by:

$$
\begin{aligned}
\alpha_{\mathscr{R}}(\{\tau\}) &\stackrel{\text{def}}{=} \alpha_{\text{op}}(\mathit{new\text{-}write}(\tau)) \\
\gamma_{\text{op}}(I) &\stackrel{\text{def}}{=} \{((x,v), I(x)) \mid v \in \mathcal{V}\}.
\end{aligned}
$$

We define the following functions:

$$
\begin{aligned}
\mathit{inter}_I(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \\
\mathit{elem\text{-}inter}_{x:=e}(\mathcal{C}) &\stackrel{\text{def}}{=} \lambda y. \begin{cases} \mathcal{C} & \text{if } y = x \\ \bot & \text{if } y \neq x \end{cases} \\
\mathit{elem\text{-}inter}_{\mathit{lock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \bot \\
\mathit{elem\text{-}inter}_{\mathit{unlock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \bot
\end{aligned}
$$

A datarace occurs when, at the same time, a thread write a variable and another thread wants to read or write the same variable. In our model, a datarace occurs when, in some store (m, b), a thread attempts to access (in read or write) to a variable x even though x is the write buffer of another thread. Hence, to check dataraces, we just have to check during an assignment if a variable accessed by the assignment is written in I.

Let $\mathtt{access}_{\mathtt{S}}(lv := e)$ the set of variables accessed by the assignment $lv := e$ in a context $\mathtt{S}$. E.g., the statement $x := y$ accesses to the variables x and y in any context. The statement $*x := y$ accesses to x, to y and to a third variable; this third variable depends of the value of x. If the value of x is $\&z$, then this statement accesses to x, y and z. Let $\mathit{access}_{\mathcal{C}}(lv := e)$ an abstraction of $\mathtt{access}_{\mathtt{S}}(lv := e)$. In a similar way we define $\mathit{access}_{\mathcal{C}}(\mathit{cond})$.

If neither lv nor e use pointers, then $\mathit{access}(lv := e)$ is the set of all variables that appear in lv or in e. If there is a pointer dereference, this domain does not know which variable

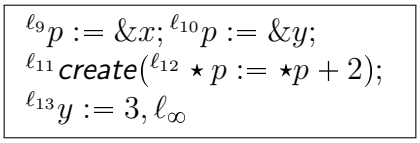


Figure 16.1: Data-race on y

is accessed, and therefore, a sound approximation will be that the statement $lv := e$ may access to any variable. This is not precise. It is standard that we combine two domains by computing their reduced product [Cou05, CC79, CFR+97, CMB+95, GT06], getting a more precise domain than both domains separately. Hence, we can compute the reduced product of this domain with a domain that handle pointers, e.g., maps domains of Section 16.1.

Finally, the errors are defined by:

$$\mathit{error}_{lv:=e}(\mathcal{C}, I) = \begin{cases} \emptyset & \text{if } \mathcal{C} \cap \bigcap_{y \in \mathit{access}_{\mathcal{C}}(lv:=e)} I(y) \neq \emptyset \\ \{\textbf{data-race}\} & \text{if } \mathcal{C} \cap \bigcap_{y \in \mathit{access}_{\mathcal{C}}(lv:=e)} I(y) = \emptyset \end{cases}$$

For guards, we have a similar definition:

$$\mathit{error}_{cond}(\mathcal{C}, I) = \begin{cases} \emptyset & \text{if } \mathcal{C} \cap \bigcap_{y \in \mathit{access}_{\mathcal{C}}(cond)} I(y) \neq \emptyset \\ \{\textbf{data-race}\} & \text{if } \mathcal{C} \cap \bigcap_{y \in \mathit{access}_{\mathcal{C}}(cond)} I(y) = \emptyset \end{cases}$$

When the current thread attempts to access a variable x, it holds the set $\mathcal{C}$ of mutexes. When another thread writes in x, holding the set $I(x)$ of mutexes. A data race occurs when the two sets are disjoints.

16.2.3 Reduced Product

The main drawback of this domain is that we need to overapproximate access. If a pointer is used in an assignment (e.g., in the assignment $^{\ell_2} \star p := 2$ in Fig. 6.2a), then *access* cannot be precise. This domain knows nothing about pointers, the set of variables accessed by $^{\ell_2} \star p := 2$ in overapproximated by the set of all variables: $\mathit{access}_{\ell_2 \star p:=2}(\mathcal{C}) = \mathcal{V}ar$.

To enhance precision, we may use the reduced product described in Section 3.4 with a domain of maps. For instance, the domains of maps where values are addresses of functions.

Hence, in Fig. 6.2a the reduced product detects that, when $^{\ell_2} \star p := 2$ is executed, p points to y and not to x, hence, there is no datarace.

Figure 16.2 gives an example of the analysis. The columns $\mathcal{C}$, $\mathcal{L}$ and $\mathcal{K}$ give the information of the domain of maps (values are ranges or addresses of variables). The last column gives the errors detected by the domain of protected variables.

On the program 16.1, the analysis will detect that there is a data-race on y.

		$\mathcal{C}$	$\mathcal{L}$	$\mathcal{K}$	I	Data-race
1	Initial Configuration	$y = 0$ $p = \text{NULL}$	$\{\ell_\star\}$	$\bot$	$\bot$	No
2	$(\!\|^{\ell_9} p := \&x, \ell_{10} \|\!)$	$y = 0$ $p = \&x$	$\{\ell_\star\}$	$\ell_\star \mapsto p = \&x$	$\bot$	No
3	$(\!\|^{\ell_{10}} p := \&y, \ell_{11} \|\!)$	$y = 0$ $p = \&y$	$\{\ell_\star\}$	$\ell_\star \mapsto p = \{\&x, \&y\}$	$\bot$	No
4	$\mathit{child\text{-}spawn}_{\ell_{12}}$	$y = 0$ $p = \&y$	$\{\ell_\star\}$	$\bot$	$\bot$	No
5	$(\!\|^{\ell_{12}} \star p :=$ $\star p + 2, \ell_\infty \|\!)$	$y = 2$ $p = \&y$	$\{\ell_\star\}$	$\ell_\star \mapsto y = 2$	$\bot$	No
6	$\mathit{combine}_{\mathit{spawn}_{\ell_{12}}(\cdot)}$	$y = [0, 2]$ $p = \&y$	$\{\ell_\star, \ell_{12}\}$	$\ell_\star \mapsto y = 2, p = \{\&x, \&y\}$	$y = 2$	No
7	$(\!\|^{\ell_{13}} y := 3, \ell_\infty \|\!)$	$y = [0, 3]$ $p = \&y$	$\{\ell_\star, \ell_{12}\}$	$\ell_\star \mapsto \begin{cases} y = [2, 3] \\ p = \{\&x, \&y\} \end{cases}$ $\ell_{11} \mapsto y = 3$	$y = 2$	Yes
8	Initial Configuration	$y = 0$ $p = \text{NULL}$	$\{\ell_\star\}$	$\ell_\star \mapsto \begin{cases} y = [2, 3] \\ p = \{\&x, \&y\} \end{cases}$ $\ell_{11} \mapsto y = 3$	$\bot$	No
9	$(\!\|^{\ell_9} p := \&x;$ $^{\ell_{10}} p := \&y, \ell_{11} \|\!)$	$y = 0$ $p = \&y$	$\{\ell_\star\}$	$\ell_\star \mapsto \begin{cases} y = [2, 3] \\ p = \{\&x, \&y\} \end{cases}$ $\ell_{11} \mapsto y = 3$	$\bot$	No
10	$\mathit{child\text{-}spawn}_{\ell_{11}}$	$y = 0$ $p = \&y$	$\{\ell_\star\}$	$\bot$	$y = 3$	No
11	$(\!\|^{\ell_{12}} \star p :=$ $\star p + 2, \ell_\infty \|\!)$	$y = [2, 3]$ $p = \&y$	$\{\ell_\star\}$	$\ell_\star \mapsto y = 2$	$\bot$	Yes

Figure 16.2: Example of Data-Race Detection

16.3 Set of Locks and Acquisition Histories

16.3.1 Lattice of Abstract States

We define an abstract complete lattice for locks: $\mathbf{SetLocks}^{\uparrow} = \mathcal{P}(\mathbf{Locks})$ as in Section 16.2.1. Nevertheless, we use this lattice to represent locks that *may* be held at some point, we order the lattice $\mathscr{D} = \mathbf{SetLocks}^{\uparrow}$ by the inclusion ordering (and not[5] by the reverse inclusion ordering), i.e., $\mathcal{C}_1 \leqslant \mathcal{C}_2 \Leftrightarrow \mathcal{C}_1 \subseteq \mathcal{C}_2$. Hence $\mathcal{C}_1 \sqcup \mathcal{C}_2 = \mathcal{C}_1 \cup \mathcal{C}_2$.

The functions on abstract states are the same as section 16.2.1:

$$\begin{aligned}
\mathit{elem}_{lv:=e}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \\
\mathit{enforce}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \\
\mathit{elem}_{\mathit{lock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \cup \{\mu\} \\
\mathit{schedule\text{-}child}(\mathcal{C}) &\stackrel{\text{def}}{=} \emptyset \\
\mathit{elem}_{\mathit{unlock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \setminus \{\mu\}
\end{aligned}$$

16.3.2 Lattice of Abstract Transitions

We consider the complete lattice $\mathbb{H} = \mathcal{P}(\mathbf{Locks})^{\mathbf{Locks}}$ of acquisition histories [KIG05, LMO08] ordered by the pointwise ordering. An acquisition history maps a mutex μ to the set of mutexes that *may* be acquired after μ is acquired.

Let us consider the operator ρ defined by: $\rho(h) = \lambda\mu_0.h(\mu_0) \cup \bigcup_{\mu \in h(\mu_0)} h(\mu)$. We consider the domain $\mathbb{H}_\rho = \{h \in \mathbb{H} \mid \rho(h) = h\}$. $\mathbb{H}_\rho$ is a complete lattice for the pointwise ordering. We use this lattice as the lattice of abstract transitions: $\mathscr{R} = \mathbb{H}_\rho$.

$$\begin{aligned}
\alpha_{\mathscr{R}}(\{(s_1, s_2)\}) &\stackrel{\text{def}}{=} \lambda\mu. \begin{cases} M_1 \setminus M_2 & \text{if } \mu \in M_1 \\ \emptyset & \text{otherwise} \end{cases} \\
\text{where } M_1 &= \mathtt{mutex}(\mathit{thread}(s_1), s_1) \\
\text{and } M_2 &= \mathtt{mutex}(\mathit{thread}(s_2), s_2)
\end{aligned}$$

$$\begin{aligned}
\mathit{elem\text{-}inter}_{\mathit{lock}(\mu_0)}(\mathcal{C}) &\stackrel{\text{def}}{=} \lambda\mu. \begin{cases} \mu_0 & \text{if } \mu \in \mathcal{C} \\ \emptyset & \text{otherwise.} \end{cases} \\
\mathit{elem\text{-}inter}_{\mathit{unlock}(\mu_0)} &\stackrel{\text{def}}{=} \bot
\end{aligned}$$

A deadlock occurs when several threads $i_1, \ldots, i_n$ attempt to acquire a lock owned by the next thread: i_1 attempts to lock μ_1 owned by i_2, i_2 attempts to lock μ_2 owned by i_3,. . . ,

[5] The only difference between $\mathbf{SetLocks}^{\uparrow}$ and $\mathbf{SetLocks}^{\downarrow}$ is the ordering.

i_n attempts to lock μ_n owned by i_1. To detect deadlocks, at each action $\mathit{lock}(\mu)$ we check whether there exists a sequence of locks $\mu_2, \ldots, \mu_n$ such that for every k, $\mu_k \in h(\mu_{k+1})$ and $\mu_n \in \mathcal{C}$. This is easy since $\rho(h) = h$ for all $h \in \mathscr{R}$:

$$\mathit{error}_{\mathit{lock}\mu_0}(\mathcal{C}, I) \stackrel{\text{def}}{=} \begin{cases} \{\textbf{Deadlock}\} & \text{if } \exists \mu \in \mathcal{C} : \mu \in I(\mu_0) \\ \{\textbf{Auto-Deadlock}\} & \text{if } \mu_0 \in \mathcal{C} \\ \emptyset & \text{otherwise.} \end{cases}$$

The intuition is that domain checks if the mutexes are locked in the same order in all threads. If a thread locks a mutex μ_1 and after a mutex μ_2 and another thread locks μ_2 and after μ_1, therefore we detect a deadlock.

The error **Auto-Deadlock** occurs when a thread attempts to lock a mutex it owns.

16.3.3 Anti-Chains of Acquisition Histories

We introduce an abstract domain based on acquisition histories $\mathbb{H} = \mathcal{P}(\textbf{Locks})^{\textbf{Locks}}$. $\mathbb{H}$ is ordered by the pointwise ordering. We use the lattice of upper-closed sets[6] of acquisitions histories: $\mathcal{P}^{\uparrow}(\mathbb{H}) = \{X \in \mathbb{H} \mid \forall x \in X \forall y \in \mathbb{H}, x \leqslant y \Rightarrow y \in X\}$. Recall[6] that an element of $\mathcal{P}^{\uparrow}(\mathbb{H})$ may be represented by a finite antichain of acquisition histories.

This domain reuse P. Lammich and M. Müler-Olm's ideas [LMO08] to detect precisely data races. As in P. Lammich and M. Müler-Olm analysis [LMO08] (See Section 4.7.3), we assume a set $A \stackrel{\text{def}}{=} \{U, V\}$ and a function *critic* from assignments to $\mathcal{P}(A) \setminus A = \{\emptyset, \{U\}, \{V\}\}$.

The abstract lattice for states is $\mathscr{D} = \mathcal{P}^{\uparrow}(\mathbb{H})$. The abstract lattice for transitions $\mathscr{R} = \mathcal{P}^{\uparrow}(\mathbb{H} \times \{U, V\})$. Notice that these sets may be represented by anti-chains (See Section 2.3.3). We do not need to represent all elements of these sets.

We introduce a function $\mathit{h\text{-}acquire} : \mathbb{H} \times \textbf{Locks} \to \mathbb{H}$, that, given an acquisition history h, acquires a new mutex. We encode in acquisition histories the fact that a mutex is owned, i.e., $\mu \in h(\mu)$ means that the mutex μ is owned by the current thread and $h(\mu) = \emptyset$ means that the mutex μ is free or owned by another thread.

$$\mathit{h\text{-}acquire}(h, \mu_0) \stackrel{\text{def}}{=} \lambda \mu. \begin{cases} h(\mu) \cup \mu_0 & \text{if } \mu \in h(\mu) \\ \{\mu_0\} & \text{if } \mu = \mu_0 \\ h(\mu) & \text{otherwise.} \end{cases}$$

Recall that $\otimes$ is a predicate that tells us if two acquisition histories may be interleaved, See Section 4.7.3.

[6] See Section 2.3.3.

Hence we define the functions of Figure 15.2 for this domain:

$$
\begin{aligned}
\mathit{elem}_{lv:=e}(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \\
\mathit{elem\text{-}inter}_{lv:=e}(\mathcal{C}) &\stackrel{\text{def}}{=} \lambda x. \begin{cases} \mathcal{C} \times \{U\} & \text{if } \mathit{critic}(lv := e) = U \\ \mathcal{C} \times \{V\} & \text{if } \mathit{critic}(lv := e) = V \\ \bot & \text{if } \mathit{critic}(lv := e) = \emptyset \end{cases} \\
\mathit{inter}_I(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \\
\mathit{enforce}_x(\mathcal{C}) &\stackrel{\text{def}}{=} \mathcal{C} \\
\mathit{schedule\text{-}child}(\mathcal{C}) &\stackrel{\text{def}}{=} \bot \\
\mathit{elem}_{\mathit{lock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \{\mathit{h\text{-}acquire}(h, \mu) \mid h \in \mathcal{C} \wedge h(\mu) = \emptyset\} \\
\mathit{elem}_{\mathit{unlock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \{h[\mu \mapsto \emptyset] \mid h \in \mathcal{C} \wedge \mu \in h(\mu)\} \\
\mathit{elem\text{-}inter}_{\mathit{lock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \bot \\
\mathit{elem\text{-}inter}_{\mathit{unlock}(\mu)}(\mathcal{C}) &\stackrel{\text{def}}{=} \bot \\
\mathit{error}_{lv:=e}(\mathcal{C}, I) &\stackrel{\text{def}}{=} \begin{cases} \textbf{Data-race} & \text{if } \exists h_C \in \mathcal{C} \exists (h_I, V) \in I : h_C \otimes h_I \\ \textbf{Data-race} & \text{if } \exists h_C \in \mathcal{C} \exists (h_I, V) \in I : h_C \otimes h_I \\ \bot & \text{if } \mathit{critic}(lv := e) = \emptyset \end{cases}
\end{aligned}
$$

CHAPTER 17

Language Extensions

In this chapter we discuss some language extensions, e.g., the *par* constructor that is used in several other analyses [KSV96, RR99, RR03, SS00].

17.1 Conditions and Actions

With our semantics, it is easy to add new kinds of conditions or actions. For instance, Figure 17.1 gives some possible extensions.

The *skip* is trivial, and is abstracted by the identity function: $(\!|^{\ell_1}\textit{skip}, \ell_2|\!)(Q) = Q$.

Nondeterministic choices is easy to handle in our concrete model[7]:

$$\forall \sigma, bool(\sigma, undet()) = true \wedge bool(\sigma, \neg undet()) = true.$$

Undeterministic choices allow to model:

- random functions
- external devices that measure some physical quantity

[7] Recall Chapter 7.

cond	::=		condition
	|	⋮	⋮
	|	$undet()$	Non-deterministic choice
	|	⋮	⋮
action	::=		basic action
	|	⋮	⋮
	|	$x :=$ *pthread_lock*(μ)	Lock that may fail
	|	*skip*	do nothing
	|	*copy*(lv_1, lv_2)	Copy
	|	⋮	⋮

Figure 17.1: Syntax for "Par" Constructor

- data from an user or from an unknown other program
- complex guards

The semantics of guards with non-deterministic choices is:

$$(\!|^{\ell_1} \textit{guard}(undet()), \ell_2|\!) \stackrel{\text{def}}{=} (\!|^{\ell_1} \textit{guard}(undet()), \ell_2|\!) \stackrel{\text{def}}{=} (\!|^{\ell_1} \textit{skip}, \ell_2|\!).$$

The *copy* allows to model copy of memory regions.

Until now, we assume that a lock operation never fails. Nevertheless, in real multi-threaded libraries, as recalled by V. Vojdani and V. Vene [VV07], it is a common practice when using Pthread Library to test whether the lock operation succeeded or not. V. Vojdani and V. Vene give the following example:

```
1 status = pthread_mutext_lock(m);
2 if (status != 0)
3     err_abort(status, "Lock mutex");
```

The semantics of $x :=$ *pthread_lock*(μ) then has to consider two cases: the case where the mutex is locked, and the case where the mutex is not locked. If the lock operation succeeds, the value of x is 0, if the lock operation fails, the value of x is ERROR.

$$(\!|x := \textit{pthread\_lock}(\mu)|\!)(Q) = (\!|x = 0|\!) \circ (\!|\textit{lock}(\mu)|\!)(Q) \sqcup (\!|x = \text{ERROR}|\!)(Q).$$

17.2 Par Constructor

17.2.1 Concrete Semantics

The *par* constructor is a different kind of parallelism.

$stmt$	$::=$		statement
	$\vert$	$\vdots$	$\vdots$
	$\vert$	${}^{\ell}\mathsf{join}\{\ell_1, \ell_2, \ldots, \ell_n\}, \ell'$	join
	$\vert$	$\vdots$	$\vdots$
cmd	$::=$		command
	$\vert$	$\vdots$	$\vdots$
	$\vert$	${}^{\ell_0}\mathsf{par}^{\ell'_0}\{{}^{\ell_1} cmd_1 \mid {}^{\ell_2} cmd_2\}$	binary parallelism
	$\vert$	${}^{\ell_0}\mathsf{par}_n^{\ell'_0}\{{}^{\ell_1} cmd_1 \mid {}^{\ell_2} cmd_2 \mid \ldots \mid {}^{\ell_n} cmd_n\}$	n-ary parallelism
	$\vert$	${}^{\ell_0}\mathsf{parfor}\{{}^{\ell} cmd\}$	parallel loop
	$\vert$	$\vdots$	$\vdots$

Figure 17.2: Syntax for "Par" Constructor

Our language described in Chapter 6 (See Figure 6.1) does not handle the constructor *par*. It is why we extend our language to handle *create* and *par* at the same time. Figure 17.2 explains how to extend the grammar of Figure 6.1 to add two new constructors. The classical *par* statement[KSV96, RR99, RR03, SS00] and the *parfor* constructor described by R. Rugina and M. Rinard [RR99, RR03]. We also add an intermediate statement, usefull to define the semantics of *par*.

The command ${}^{\ell_0}\mathsf{par}^{\ell'_0}\{{}^{\ell_1} cmd_1 \mid {}^{\ell_2} cmd_2\}$ executes the statements ${}^{\ell_1} cmd_1, \ell_\infty$ and ${}^{\ell_2} cmd_2, \ell_\infty$ in parallel.This command is generalized by par_n that executes n statements in parallel. The statement ${}^{\ell_0}\mathsf{parfor}^{\ell'_0}\{{}^{\ell} cmd\}$ launches an arbitrary number of times the same statement ${}^{\ell} cmd, \ell_\infty$. Notice that these commands have two labels ℓ_0 and ℓ'_0. The label ℓ_0 is as for other commands: it is the label of the beginning of the command. The second label ℓ'_0 represents a label in which a thread will wait its descendants.

We have to define the concrete rules (like rules of Figure 7.2) to state the precise semantics of *par* statements. These rules are given in Figure 17.3 for the binary *par* operator.

The rule "par spawn" explains how the binary *par* statement spawns two threads at the same time. The label $\ell_?$ is an intermediate label needed for the definition.

At the end of a *par* statement, all threads created by this statement will join. Notice that a set is joinable only under some conditions. It is why we have introduced the predicate *joinable*. In a sequentially consistent model, a thread is joinable if and only if it has ended its execution:

$$joinable(j, P, \sigma) \overset{\text{def}}{\Leftrightarrow} P(j) = \ell_\infty.$$

In a weak memory model (TSO or PSO), we need an extra condition: the thread buffer is empty, i.e., all writes done by the thread j have been taken into account in the global memory. Formally, in the TSO model:

$$joinable(j, P, (m, b)) \overset{\text{def}}{\Leftrightarrow} P(j) = \ell_\infty \wedge b(j) = \epsilon.$$

$$\frac{{}^{\ell_0}\mathsf{spawn}(\ell_1), \ell_? \Vdash s_0 \to s_1 \qquad {}^{\ell_?}\mathsf{spawn}(\ell_2), \ell'_0 \Vdash s_1 \to s_2}{{}^{\ell_0}\mathsf{par}^{\ell'_0}\{{}^{\ell_1} cmd_1 \mid {}^{\ell_2} cmd_2\}, \ell' \Vdash s_0 \to s_2} \text{ par spawn}$$

$$\frac{\forall j, (i, \ell_1, j) \in g \vee (i, \ell_2, j) \in g \Rightarrow joinable(j, P, \sigma)}{{}^{\ell_0}\mathsf{join}\{\ell_1, \ell_2\}, \ell' \Vdash (i, P, \sigma, g) \to (i, P[i \mapsto \ell'], \sigma, g)} \text{ binary join}$$

$$\frac{{}^{\ell'_0}\mathsf{join}\{\ell_1, \ell_2\}, \ell' \Vdash \tau}{{}^{\ell_0}\mathsf{par}^{\ell'_0}\{{}^{\ell_1} cmd_1 \mid {}^{\ell_2} cmd_2\}, \ell' \Vdash \tau} \text{ par join}$$

$$\frac{{}^{\ell_1} cmd_1, \ell_\infty \Vdash \tau}{{}^{\ell_0}\mathsf{par}^{\ell'_0}\{{}^{\ell_1} cmd_1, \ell'_1 \mid {}^{\ell_2} cmd_2, \ell'_2\}, \ell' \Vdash \tau} \text{ par body 1}$$

$$\frac{{}^{\ell_2} cmd_2, \ell_\infty \Vdash \tau}{{}^{\ell_0}\mathsf{par}^{\ell'_0}\{{}^{\ell_1} cmd_1 \mid {}^{\ell_2} cmd_2\}, \ell' \Vdash \tau} \text{ par body 2}$$

Figure 17.3: Rules for the Binary "Par" Constructor

Rules "par body" means that a *par* statement generates all transitions generated by its substatements. These rules are similar to "while body" and "then body" of Figure 7.2.

The constructor par_n generalized the constructor *par*. Its rules are given in Figure 17.4.

The statement ${}^{\ell_0}\mathsf{parfor}\{{}^{\ell_{cmd}}\}, \ell_2$ allows a thread to spawn an arbitrary number of thread, obligates it to wait for their termination, and then the statement returns. The rules for this statement are given in Figure 17.5. A thread that executes this statement is at label ℓ_0. According to rule "parfor spawn", it may spawn a thread staying at the label ℓ_0. For simplicity, we model this par the statement ${}^{\ell_0}\mathsf{spawn}(\ell_1), \ell_0$ (Notice that the label ℓ_0 appears twice). Nondeterminiscally, the thread may decide to go to label ℓ_2, this is the rule "parfor join".

Notice that, with this extension, win our language, a program may use both *par* and *create* constructors.

17.2.2 Intermediate Denotational Semantics

As for other constructors, we give the intermediate denotational semantics for *par* statements. For *create* statements we use an intermediate function `schedule-child` (See Figure 10.5). This function makes a schedule transition to the last spawned child. Nevertheless, *par* (and par_n and *parfor*) spawns several threads. Hence, we need an intermediate function $\texttt{schedule-children}_L$:

$$\texttt{schedule-children}_L(\mathtt{S}) \stackrel{\text{def}}{=} \left\{ (j, P, \sigma, g) \,\middle|\, \exists i \in \mathbf{Ids} \exists \ell \in L : \begin{array}{l} (i, P, \sigma, g) \in \mathtt{S} \\ \wedge (i, \ell, j) \in g \end{array} \right\}.$$

Given a set of labels $L \subseteq \mathbb{L}$, the function $\texttt{schedule-children}_L$ fire a schedule transition to all threads created by the current thread in any label of L.

$$\frac{\begin{array}{c}{}^{\ell_0}\mathsf{spawn}(\ell_1), \ell_{?0} \Vdash s_0 \to s_1 \\ \vdots \\ {}^{\ell_{?k}}\mathsf{spawn}(\ell_{k+1}), \ell_? k+1 \Vdash s_k \to s_{k+1} \\ \vdots \\ {}^{\ell_{?n-1}}\mathsf{spawn}(\ell_n), \ell'_0 \Vdash s_{n-1} \to s_n\end{array}}{{}^{\ell_0}\mathsf{par}^{\ell'_0}\{{}^{\ell_1} cmd_1 \mid {}^{\ell_2} cmd_2\}, \ell' \Vdash s_0 \to s_n}\ \text{n-par spawn}$$

$$\frac{\forall j, \exists \ell \in \{\ell_1, \ldots, \ell_n\}(i, \ell, j) \in g \Rightarrow joinable(j, P, \sigma)}{{}^{\ell_0}\mathsf{join}\{\ell_1, \ldots, \ell_n\}, \ell' \Vdash (i, P, \sigma, g) \to (i, P[i \mapsto \ell'], \sigma, g)}\ \text{general join}$$

$$\frac{{}^{\ell'_0}\mathsf{join}\{\ell_1, \ldots, \ell_n\}, \ell' \Vdash \tau}{{}^{\ell_0}\mathsf{par}^{\ell'_0}\{{}^{\ell_1} cmd_1 \mid \ldots \mid {}^{\ell_n} cmd_n\}, \ell' \Vdash \tau}\ \text{par join}$$

$$\frac{\exists k : {}^{\ell_k} cmd_k, \ell_\infty \Vdash \tau}{{}^{\ell_0}\mathsf{par}^{\ell'_0}\{{}^{\ell_1} cmd_1 \mid \ldots \mid {}^{\ell_n} cmd_n\}, \ell' \Vdash \tau}\ \text{par body 1}$$

Figure 17.4: Rules for the n-ary "Par" Constructor

$$\frac{{}^{\ell_0}\mathsf{spawn}(\ell_1), \ell_0 \Vdash \tau}{{}^{\ell_0}\mathsf{parfor}\{{}^{\ell_1} cmd\}, \ell_2 \Vdash \tau}\ \text{parfor spawn}$$

$$\frac{{}^{\ell_0}\mathsf{join}\{\ell_1\}, \ell_2 \Vdash \tau}{{}^{\ell_0}\mathsf{par}\{{}^{\ell_1} cmd\}, \ell_2 \Vdash \tau}\ \text{parfor join}$$

$$\frac{{}^{\ell_1} cmd_1, \ell_\infty \Vdash \tau}{{}^{\ell_0}\mathsf{par}\{{}^{\ell_1} cmd_1, \ell'_1 \mid {}^{\ell_2} cmd_2, \ell'_2\}, \ell' \Vdash \tau}\ \text{parfor body}$$

Figure 17.5: Rules for the"Parfor" Constructor

In addition to this, the threads created by a statement *par* will join at their termination. Hence we need an extra function, that makes a schedule transition to the father thread at termination:

$$\mathtt{join}_L(\mathtt{S}) \stackrel{\text{def}}{=} \left\{ (i,P,\sigma,g) \,\middle|\, \forall i \in \mathbf{Ids} \forall \ell \in L : \left[\begin{array}{l} (j,P,\sigma,g) \in \mathtt{S} \\ \wedge (i,\ell,j) \in g \end{array} \right] \Rightarrow P(j) = \ell_\infty \right\}.$$

The intermediate denotational semantics of *par*-like statements is then defined by:

$$\begin{array}{rcl} [\![|^{\ell_0} \mathit{par}^{\ell'_0}\{^{\ell_1} \mathit{stmt}_1 \mid {}^{\ell_2} \mathit{stmt}_2\}|]\!](\mathtt{Q}) & \stackrel{\text{def}}{=} & \langle \mathtt{join}_{\{\ell_1,\ell_2\}}(\mathtt{S}_1 \cap \mathtt{S}_2), \mathtt{G} \cup \mathtt{G}_1 \cup \mathtt{G}_2, \mathtt{A} \rangle \\ \text{where } \langle \mathtt{S},\mathtt{G},\mathtt{A} \rangle & = & [\![|^{\ell_?} \mathit{spawn}(\ell_2), \ell'_0|]\!] \circ [\![|^{\ell_0} \mathit{spawn}(\ell_1), \ell_?|]\!](\mathtt{Q}) \\ \text{and } \mathtt{S}_0 & = & \mathtt{schedule\text{-}children}_{\{\ell_1,\ell_2\}}(\mathtt{S}) \\ \text{and } \langle \mathtt{S}_1,\mathtt{G}_1,\mathtt{A}_1 \rangle & = & [\![|^{\ell_1} \mathit{stmt}_1, \ell_\infty|]\!] \langle \mathtt{interfere}_{\mathtt{A}_2}(\mathtt{S}_1), \mathtt{G}_1, \mathtt{A} \cup \mathtt{G}_2 \rangle \\ \text{and } \langle \mathtt{S}_2,\mathtt{G}_2,\mathtt{A}_2 \rangle & = & [\![|^{\ell_2} \mathit{stmt}_2, \ell_\infty|]\!] \langle \mathtt{interfere}_{\mathtt{A}_1}(\mathtt{S}_2), \mathtt{G}_2, \mathtt{A} \cup \mathtt{G}_1 \rangle \end{array}$$

Notice that $\langle \mathtt{S}_1,\mathtt{G}_1,\mathtt{A}_1 \rangle$ and $\langle \mathtt{S}_2,\mathtt{G}_2,\mathtt{A}_2 \rangle$ are defined by a fixpoint. This fixpoint is the equivalent of `guarantee` for create statements.

This definition straightforwardly generalize to par_n. Furthermore, we define the semantics of *parfor*:

$$\begin{array}{rcl} [\![|^{\ell_0} \mathit{parfor}\{^{\ell} \mathit{cmd}\}|]\!](\mathtt{Q}) & \stackrel{\text{def}}{=} & \langle \mathtt{join}_{\{\ell\}}(\mathtt{S}_1 \cap \mathtt{S}_2), \mathtt{G} \cup \mathtt{G}_1 \cup \mathtt{G}_2, \mathtt{A} \rangle \\ \text{where } \langle \mathtt{S},\mathtt{G},\mathtt{A} \rangle & = & [\![|^{\ell_0} \mathit{spawn}(\ell_1), \ell_0|]\!]^{\uparrow\omega}(\mathtt{Q}) \\ \text{and } \mathtt{S}_0 & = & \mathtt{schedule\text{-}children}_{\{\ell_1\}}(\mathtt{S}) \\ \text{and } \langle \mathtt{S}',\mathtt{G}',\mathtt{A}' \rangle & = & [\![|^{\ell_1} \mathit{stmt}_1, \ell_\infty|]\!] \langle \mathtt{interfere}_{\mathtt{A}'}(\mathtt{S}'), \mathtt{G}', \mathtt{A} \cup \mathtt{G}' \rangle \end{array}$$

17.2.3 Abstract Semantics

$$\begin{array}{rcl} {}^{\ell_0}\mathit{par}^{\ell'0}\{^{\ell_1} \mathit{cmd}_1 |^{\ell_2} \mathit{cmd}_2\}(\mathcal{Q}) & \stackrel{\text{def}}{=} & \mathit{exe\text{-}children}^{\uparrow\triangledown}(\mathcal{Q}) \\ \text{where } \mathit{exe\text{-}children}(\mathcal{Q}) & \stackrel{\text{def}}{=} & \langle \mathcal{C}_1 \sqcap \mathcal{C}_2, \mathcal{L}_1 \sqcup \mathcal{L}_2, \mathcal{K}_1 \sqcup \mathcal{K}_2, I_1 \sqcup I_2, \mathcal{E}_1 \sqcup \mathcal{E}_2 \rangle \\ \langle \mathcal{C}_1, \mathcal{L}_1, \mathcal{K}_1, I_1, \mathcal{E}_1 \rangle & \stackrel{\text{def}}{=} & (\!|^{\ell_1} \mathit{cmd}_1, \ell_\infty|\!) \circ \mathit{child\text{-}spawn}_{\ell_1} \circ \mathit{spawn}_{\ell_2}(\mathcal{Q}) \\ \langle \mathcal{C}_2, \mathcal{L}_2, \mathcal{K}_2, I_2, \mathcal{E}_2 \rangle & \stackrel{\text{def}}{=} & (\!|^{\ell_2} \mathit{cmd}_2, \ell_\infty|\!) \circ \mathit{child\text{-}spawn}_{\ell_2} \circ \mathit{spawn}_{\ell_1}(\mathcal{Q}) \end{array}$$

Figure 17.6: Abstract Semantics of "Par" Statements

Hence, we introduce the abstract semantics for *par* statements. This abstract semantics is described by Figure 17.6. This definition may be straightforwardly generalized to par_n. The case of the statement *parfor* is even simpler:

$${}^{\ell_0}\mathit{parfor}\{^{\ell} \mathit{cmd}\}(\mathcal{Q}) \stackrel{\text{def}}{=} \left((\!|^{\ell_1} \mathit{cmd}_1, \ell_\infty|\!) \circ \mathit{child\text{-}spawn}_{\ell} \circ \mathit{spawn}_{\ell} \right)^{\uparrow\triangledown}(\mathcal{Q})$$

Notice that, with this semantic extension, we may analyze programs with both *create* and *par* statements.

cmd	$::=$		command
	$\mid$	$\vdots$	$\vdots$
	$\mid$	${}^{\ell}\mathsf{call}(f)$	Function Call
	$\mid$	$\vdots$	$\vdots$

Figure 17.7: Syntax for "Call" Constructor

17.3 Function Calls

Until now, we only dealt with intraprocedural analysis. hence, to analyze a function, we need to analyze the body of the function each time the function is called. We extend the syntax of our language given in Figure 6.1 with the new constructor *call* given in Figure 17.7. In Chapter 6 programs were statements of the form ${}^{\ell}cmd, \ell_\infty$. Now programs are a pair with a statement of the form ${}^{\ell}cmd, \ell_\infty$ and a list $(f_1, {}^{\ell_1}cmd_1, \ell'_1), \ldots, (f_n, {}^{\ell_n}cmd_n, \ell'_n)$ of declaration functions. The abstract semantics $(\!|\cdot|\!)$ can trivially be extended to handle function calls: $(\!|{}^{\ell}\mathsf{call}(f), \ell'|\!) = (\!|{}^{\ell_1}body, \ell'_1|\!)$; we call $(\!|\cdot|\!)_0$ this extension of $(\!|\cdot|\!)$. Nevertheless, computing the semantics of the body of a function each time this function is called may be costly.

We define another concrete semantics for programs, based on the intermediate denotational semantics. We consider the concrete lattice[8] **Mon**(**C-Configurations**) ordered by the pointwise ordering. We define the semantics $\overrightarrow{[\![\cdot]\!]}$ of a program by:

$$\overrightarrow{[\![{}^{\ell}stmt, \ell']\!]}(f) \stackrel{\text{def}}{=} [\![{}^{\ell}stmt, \ell']\!] \circ f.$$

We consider the abstract complete lattice **Mon**(**A-Configurations**). The Galois connection between the concrete and the abstract lattice is defined by:

$$\begin{array}{rcl} \alpha_{\text{fun}}(f^\sharp) & \stackrel{\text{def}}{=} & \alpha_{\text{cfg}} \circ f^\sharp \circ \gamma_{\text{cfg}} \\ \gamma_{\text{fun}}(f^\sharp) & \stackrel{\text{def}}{=} & \gamma_{\text{cfg}} \circ f^\sharp \circ \alpha_{\text{cfg}} \end{array}$$

Then, we define an abstract semantics $\overrightarrow{(\!|\cdot|\!)}$:

$$\overrightarrow{(\!|{}^{\ell}stmt, \ell'|\!)}(f) \stackrel{\text{def}}{=} (\!|{}^{\ell}stmt, \ell'|\!)_0 \circ f.$$

The following proposition tells us that the semantics of function calls may be simplified:

Proposition 17.1. *Given a function f, its code ${}^{\ell_1}body, \ell'_1$ and an abstract configuration $\mathcal{Q}$:*

$$(\!|{}^{\ell}\mathsf{call}(f), \ell'|\!)_0(\mathcal{Q}) = \left(\overrightarrow{(\!|{}^{\ell_1}body, \ell'_1|\!)}(id)\right)(\mathcal{Q})$$

where id is the identity function, i.e., $\forall x, id(x) = x$.

[8] Recall that $\mathbf{Mon}(X)$ is the set of monotone functions from X to X (See Definition 2.11).

In this definition, we only need to compute one time the semantics of ${}^{\ell_1}body, \ell_1'$. Indeed, $\overrightarrow{(\!|{}^{\ell_1}body, \ell_1'|\!)}$ will be applied to the same argument id. Hence we do not need to compute the semantics of a function each time it is called.

Nevertheless, $\overrightarrow{(\!|\cdot|\!)}$ may be hard to compute. Hence, we need an abstraction of $\overrightarrow{(\!|\cdot|\!)}$. We consider an abstract domain $\overrightarrow{\mathscr{D}}$ and a Galois connection $\alpha_\rightarrow, \gamma_\rightarrow$ from the concrete lattice **Mon**(**A-Configurations**) to $\overrightarrow{\mathscr{D}}$. We also assume an operator $\overrightarrow{\circ}$ that is an abstraction of composition[9] $\circ$ of functions and a function $\overrightarrow{apply} : \overrightarrow{\mathscr{D}} \times$ **A-Configurations** $\rightarrow$ **A-Configurations** that is an abstraction of the application of a function, i.e., $\overrightarrow{apply}$ is an abstraction of $\lambda(f, \mathcal{Q}).f(\mathcal{Q})$. Furthermore, we assume an element $\overrightarrow{id}$ that is an abstraction of the identity function $\lambda x.x$. At the end, we assume an abstract thread creation function $\overrightarrow{create}$, that, given a semantics $\overrightarrow{(\!|{}^{\ell_2}cmd, \ell_\infty|\!)}$ that overappoximates $\overrightarrow{(\!|{}^{\ell_2}cmd, \ell_\infty|\!)}$, returns an abstraction of $\overrightarrow{(\!|{}^{\ell_1}\mathsf{create}({}^{\ell_2}cmd), \ell_3|\!)}$.

This allows us to define inductively a new semantics $\overrightarrow{(\!|\cdot|\!)}$:

Definition 17.1. For any abstract function f of body ${}^{\ell_1}body, \ell_1'$:

$$
\begin{aligned}
\overrightarrow{(\!|{}^{\ell}\mathsf{call}(f), \ell'|\!)}(f) &\stackrel{\text{def}}{=} \left(\overrightarrow{(\!|{}^{\ell_1}body, \ell_1'|\!)}(\overrightarrow{id})\right) \overrightarrow{\circ} f \\
\overrightarrow{(\!|{}^{\ell}\mathsf{action}, \ell'|\!)}(f) &\stackrel{\text{def}}{=} \left(\alpha_\rightarrow(\mathit{elem}_{\mathsf{action}})\right) \overrightarrow{\circ} f \\
\overrightarrow{(\!|{}^{\ell_1}cmd_1; {}^{\ell_2}cmd_2|\!)}(f) &\stackrel{\text{def}}{=} \overrightarrow{(\!|{}^{\ell_2}cmd_2|\!)} \circ \overrightarrow{(\!|{}^{\ell_1}cmd_1|\!)}(f) \\
\overrightarrow{(\!|{}^{\ell_1}\mathsf{while}(cond)\{{}^{\ell_2}cmd\}|\!)}(f) &\stackrel{\text{def}}{=} \overrightarrow{guard}_{\neg cond} \circ \mathit{loop}^{\uparrow\triangledown}(f) \\
\text{with } \mathit{loop}(g) &\stackrel{\text{def}}{=} \left(\overrightarrow{(\!|{}^{\ell_2}cmd, \ell_1|\!)} \circ \overrightarrow{guard}_{cond}(g)\right) \sqcup g \\
\text{and } \overrightarrow{guard}_{cond}(g) &\stackrel{\text{def}}{=} \left(\alpha_\rightarrow(guard_{cond})\right) \overrightarrow{\circ} g \\
\overrightarrow{(\!|{}^{\ell_1}\mathsf{create}({}^{\ell_2}cmd), \ell_2|\!)}(f) &\stackrel{\text{def}}{=} \overrightarrow{create}_{\overrightarrow{(\!|{}^{\ell_2}cmd, \ell_\infty|\!)}}(f) \\
\text{with } g &\stackrel{\text{def}}{=} \alpha_\rightarrow(spawn_{\ell_2}) \overrightarrow{\circ} g
\end{aligned}
$$

This is a similar definition as Definition 13.2, excepted that we also handle function calls. Furthermore, this semantics is sound:

Proposition 17.2. $\overrightarrow{(\!|\cdot|\!)}$ *is an abstraction of* $\overrightarrow{(\!|\cdot|\!)}$.

Hence, this semantics allows us to compute an overapproximation of the abstract semantics. For any abstract configuration $\mathcal{Q}$ and any statement ${}^{\ell}cmd, \ell' : (\!|{}^{\ell}cmd, \ell'|\!)(\mathcal{Q}) \leqslant \overrightarrow{apply}(\overrightarrow{(\!|{}^{\ell}cmd, \ell'|\!)}(\overrightarrow{id}), \mathcal{Q})$

[9] Recall Section 2.1.3.

17.3.1 Examples of Abstract Domains

17.3.1.a Pure Gen/Kill Analyses We want to define an abstract semantics $\overrightarrow{(\!|\cdot|\!)}$ for the gen/kill analysis of Section 14.3. The gen/kill analysis gives us, for each basic action, two elements of the lattice $\mathcal{V}$:

- $\texttt{gen}(\mathit{action}, \sigma)$
- and $\texttt{keep}(\mathit{action}, \sigma)$.

In the case of *pure* Gen/Kill analysis (See Section 4.6), the functions `gen` and `keep` does not depends on the store σ. Hence, we omit this argument and write: $\texttt{gen}(\mathit{action})$ and $\texttt{keep}(\mathit{action})$.

As abstract domain, we use the lattice $\mathbb{F}$ define in Section 4.6. An element of $\mathbb{F}$ may be represented by two elements of the lattice $\mathcal{V}$. Furthermore, according to Claim 4.2, $\mathbb{F}$ is stable under composition.

Our domain $\overrightarrow{\mathscr{D}}$ is then abstract configurations using $\mathbb{F}$ as abstract states and $\mathscr{R} = \mathcal{V}$ as abstract transitions.

We define the Galois $\alpha_{\rightarrow}, \gamma_{\rightarrow}$ connection as follow:

$$\gamma_{\rightarrow}\langle f, \mathcal{L}_0, \mathcal{K}_0, I_0, \mathcal{E}_0\rangle \stackrel{\text{def}}{=} \lambda\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle.\langle f(\mathcal{C}), \mathcal{L} \cup \mathcal{L}_0, \mathcal{K} \sqcup \mathcal{K}_0, I \sqcup I_0, \mathcal{E} \sqcup \mathcal{E}_0\rangle$$

The abstract application is then:

$$\overrightarrow{\mathit{apply}}(\langle f, \mathcal{L}_0, \mathcal{K}_0, I_0, \mathcal{E}_0\rangle, \langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle) \stackrel{\text{def}}{=} \langle f(\mathcal{C}), \mathcal{L} \cup \mathcal{L}_0, \mathcal{K} \sqcup \mathcal{K}_0, I \sqcup I_0, \mathcal{E} \sqcup \mathcal{E}_0\rangle$$

The abstract composition is defined by:

$$\langle f_0, \mathcal{L}_0, \mathcal{K}_0, I_0, \mathcal{E}_0\rangle \overrightarrow{\circ} \langle f_1, \mathcal{L}_1, \mathcal{K}_1, I_1, \mathcal{E}_1\rangle = \langle f_1 \circ f_2, \mathcal{L}_0 \cup \mathcal{L}_1, \mathcal{K}_0 \sqcup \mathcal{K}_1, I_0 \sqcup I_1, \mathcal{E}_0 \sqcup \mathcal{E}_1\rangle$$

The abstract thread creation is defined by:

$$\overrightarrow{\mathit{create}}_{\overrightarrow{(\!|\ell_2\, cmd, \ell_\infty|\!)}}(f) \stackrel{\text{def}}{=} \alpha_{\rightarrow}\left(\mathit{combine}_g \circ \mathit{guarantee}_{\overrightarrow{(\!|\ell_2\, cmd, \ell_\infty|\!)}} \circ \mathit{child\text{-}spawn}_{\ell_2}\right) \overrightarrow{\circ} f$$

The abstract function $\langle f, \mathcal{L}_0, \mathcal{K}_0, I_0, \mathcal{E}_0\rangle$

The semantics $\overrightarrow{(\!|\cdot|\!)}$ is similar to semantics $(\!|\cdot|\!)$ given in 14.3. The main difference is the set of abstract states.

17.3.2 Acquisition Histories

We want to define a semantics $\overrightarrow{(\!|\cdot|\!)}$ for the semantics $(\!|\cdot|\!)$ described in Section 16.3.3.

The domain $\overrightarrow{\mathscr{D}}$ is the set of abstract configurations with:

- $\mathscr{D} = \mathcal{P}^{\uparrow}(\mathbb{H})$ as set of abstract states
- $\mathscr{R} = \mathscr{R}_1 \times \mathscr{R}_2$ as set of abstract transitions, where $\mathscr{R}_1 = \mathscr{R}_2 = \mathcal{P}^{\uparrow}(\mathbb{H} \times \{U, V\})$

- $\mathcal{P}^{\uparrow}(\mathbb{H})$ as set of errors.

The two differences between abstraction configuration of $\overrightarrow{\mathscr{D}}$ and abstract configurations used in Section 16.3.3 is the set of errors and set of abstract transitions. The set $\mathscr{R}_1$ represents the end of the execution of the current thread and $\mathscr{R}_2$ represents the whole execution of a

The main reason is that an error may be reachable from some configurations and not from other configurations. The lattice $\mathcal{P}^{\uparrow}(\mathbb{H}) \times \mathcal{P}^{\uparrow}(\mathbb{H})$ will allow us to check whether a given error is reachable.

We define an intermediate function $\mathit{concat} : \mathscr{D} \times \mathscr{R}_1 \times \mathscr{R}_2 \to \mathcal{P}^{\uparrow}(\mathbb{H} \times \{U, V\})$:

$$\mathit{concat}(\mathcal{C}, H_1, H_2) \stackrel{\text{def}}{=} \{(h_C \sqcup h, X) \mid h_C \in \mathcal{C} \wedge (h, X) \in H_1\} \cup H_2$$

The abstract application is defined by:

$$\overrightarrow{\mathit{apply}}\big(\langle \mathcal{C}_0, \mathcal{L}_0, \mathcal{K}_0, I_0, \mathcal{E}_0\rangle, \langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle\big) \stackrel{\text{def}}{=} \langle \mathcal{C}_1, \mathcal{L}_1, \mathcal{K}_1, I_1, \mathcal{E}_1\rangle$$

$$\begin{array}{rcl}
\text{where } \mathcal{C}_1 & \stackrel{\text{def}}{=} & \{h_0 \sqcup h \mid h_0 \in \mathcal{C}_0 \wedge h \in \mathcal{C}\} \\
\mathcal{L}_1 & \stackrel{\text{def}}{=} & \mathcal{L}_0 \cup \mathcal{L}_0 \\
\mathcal{K}_1 & \stackrel{\text{def}}{=} & \lambda \ell.\mathit{concat}(\mathcal{C}, \mathcal{K}_0(\ell)) \\
I_1 & \stackrel{\text{def}}{=} & I \cup H_2 \text{ whith } (H_1, H_2) = I_0 \\
\mathcal{E}_1 & \stackrel{\text{def}}{=} & \begin{cases} \{\textbf{Data-Race}\} & \text{if } \exists h_C \in \mathcal{C} \exists h_E \in \mathcal{E}_0 : h_C \otimes h_E \\ \emptyset & \text{if } \mathit{otherwise} \end{cases}
\end{array}$$

The Galois connection between $\mathbf{Mon}(\mathbf{A\text{-}Configurations})$ and $\overrightarrow{\mathscr{D}}$ is defined by: $\gamma_{\rightarrow}(f) = \lambda \mathcal{Q}.\overrightarrow{\mathit{apply}}(f, \mathcal{Q})$.

The abstract composition is defined by:

$$\langle \mathcal{C}_1, \mathcal{L}_1, \mathcal{K}_1, I_1, \mathcal{E}_1\rangle \overrightarrow{\circ} \langle \mathcal{C}_2, \mathcal{L}_2, \mathcal{K}_2, I_2, \mathcal{E}_2\rangle = \langle \mathcal{C}[, \mathcal{L}[, \mathcal{K}[, I[, \mathcal{E}[\rangle_3]$$

$$\begin{array}{rcl}
\text{where } \mathcal{C}_3 & \stackrel{\text{def}}{=} & \{h_1 \sqcup h_2 \mid h_1 \in \mathcal{C}_1 \wedge h_2 \in \mathcal{C}_2\} \\
\mathcal{L}_3 & \stackrel{\text{def}}{=} & \mathcal{L}_0 \cup \mathcal{L}_0 \\
\mathcal{K}_3 & \stackrel{\text{def}}{=} & \mathcal{K}_1 \sqcup \mathcal{K}_2 \\
I_3 & \stackrel{\text{def}}{=} & I_1 \sqcup I_2 \\
\mathcal{E}_3 & \stackrel{\text{def}}{=} & \mathcal{E}_1 \sqcup \mathcal{E}_2
\end{array}$$

cmd	$::=$		command
	$\mid$	$\vdots$	$\vdots$
	$\mid$	${}^{\ell}\mathsf{SyncCall}_{\mu}(f)$	Function Call
	$\mid$	$\vdots$	$\vdots$

Figure 17.8: Syntax for "SyncCall" Constructor

Synchronized Function Call and Reentrant Monitors To handle reentrant monitors, we need as in P. Lammich and M. Müller-Olm paper [LMO08] the possibility to call a function synchronized with a monitor. This synchronized called are used in practice in some languages like JAVA [GJSB05, Section 8.4.3.6 Synchronized Methods].

Hence, we add a new constructor to our language, see Figure 17.8. The new constructor $\mathsf{SyncCall}_{\mu}(f)$ may be modeled by $\mathsf{lock}(\mu); \mathsf{call}(f); \mathsf{unlock}(\mu)$. Nevertheless, in a program that never uses lock nor unlock we may have a more precise abstraction of $\mathsf{SyncCall}_{\mu}(f)$ than of $\mathsf{lock}(\mu); \mathsf{call}(f); \mathsf{unlock}(\mu)$.

In the domain describes in Section 17.3.2, we have a more precise abstraction:

$$
\begin{aligned}
\overrightarrow{(\!|\mathsf{SyncCall}_{\mu_0}(f)|\!)}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle &\stackrel{\text{def}}{=} \langle \mathcal{C}_1, \mathcal{L}_1, \mathcal{K}_1, I_1, \mathcal{E}_1\rangle \sqcup \langle \mathcal{C}_2, \mathcal{L}_2, \mathcal{K}_2, I_2, \mathcal{E}_2\rangle \\
\text{where } \langle \mathcal{C}_1, \mathcal{L}_1, \mathcal{K}_1, I_1, \mathcal{E}_1\rangle &\stackrel{\text{def}}{=} \overrightarrow{(\!|\mathsf{call}(f)|\!)} \circ \overrightarrow{(\!|\mathsf{lock}(\mu)|\!)}\langle \mathcal{C}_{\mu_0}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle \\
\mathcal{C}_{\mu_0} &\stackrel{\text{def}}{=} \{h \in \mathcal{C} \mid \mu_0 \in h(\mu_0)\} \\
\langle \mathcal{C}_2, \mathcal{L}_2, \mathcal{K}_2, I_2, \mathcal{E}_2\rangle &\stackrel{\text{def}}{=} \overrightarrow{(\!|\mathsf{lock}(\mu)|\!)} \circ \overrightarrow{(\!|\mathsf{call}(f)|\!)} \circ \overrightarrow{(\!|\mathsf{lock}(\mu)|\!)}\langle \mathcal{C}_{\neg\mu_0}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle \\
\mathcal{C}_{\neg\mu_0} &\stackrel{\text{def}}{=} \uparrow \{h \in \mathcal{C} \mid \mu_0 \notin h(\mu_0)\}
\end{aligned}
$$

The set $\mathcal{C}$ is split into two sets $\mathcal{C}_{\mu_0}$ and $\mathcal{C}_{\neg\mu_0}$. $\mathcal{C}_{\mu_0}$ is the set of acquisition histories in which the mutex μ_0 is held. $\mathcal{C}_{\neg\mu_0}$ is the upper closure (the symbole $\uparrow$ means upper-closure) of the set of acquisition histories in which μ_0 is not hold. We must take the upper closure is the case of $\mathcal{C}_{\neg\mu_0}$, because the set $\{h \in \mathcal{C} \mid \mu_0 \notin h(\mu_0)\}$ is not in the abstract domain $\mathcal{P}^{\uparrow}(\mathbb{H})$.

This analysis is exact on some kind of programs. We call L-M-O (Lammich-Müller-Olm) programs programs such that:

- All guard uses non-deterministic choices
- No lock or unlock are used
- The program uses reentrant monitors through the primitive $\mathsf{SyncCall}$.

This analysis is exact on a large class of L-M-O programs. Let us consider the class Cl_0 of programs p such that in all execution path of p, whenever a thread i that owns a mutex μ spawns another thread j, i will release μ further in the execution.

This analysis is exact on the programs of Cl_0, i.e, if this analysis detects a data-race on a program p in Cl_0, then, there is a data-race in the concrete model. The analysis of P. Lammich and M. M. Müller-Olm [LMO08] is exact on all L-M-O programs.

To lose precision of programs that are in L-M-O but not is Cl_0 is not a problem in practice, since a programmer will avoid to lock definitively a mutex before spawning a thread.

17.3.3 Partial Functions

Functions calls can be handled using the concept of R. Wilson and M. Lam's partial functions [WL95]. Each time a function is called, we compute an abstraction of the semantics of its body for some abstract values of the arguments.

If the function is called a second time, we check if we may reuse the previous analysis of the function body, or if we had to re-analyze the function.

17.4 Conclusion

Our language may easily be extended to handle new features. New kinds of basic statement may be added without changing all the analysis. E.g., we may add a return value to the *lock* function and test if the *lock* fails.

The statement *par* may also be added. With this statement, we generalize the R. Rugina and M. C. Rinard analysis [RR99, RR03] of pointers. Furthermore, it is possible to analyze programs that use both *par* and *create* statements.

Part V

A Complete Static Analyzer: MT-Penjili

CHAPTER 18

Implementation

18.1 Penjili: The EADS Tool

The EADS company develops a static analysis tool called *Penjili*. This tool is based on abstract interpretation techniques. The Static Analysis Team that develops Penjili is composed of three permanents and two Phd students (including me).

Penjili exists since 2006 March; it detects array-overflows, `NULL` pointer dereference, invalid pointer dereference, division by zero and integer overflows. The tool is sound, in the sense that there is no false negative[1]. It analyzes programs in full fledged C (dynamic memory allocation is handle with a simple abstraction).

When I began my Phd, this tool was only able to analyze single-threaded programs. This was a major restriction since most programs (even embedded programs) are multi-threaded. Now, as a consequence of my work, Penjili handles multithreaded programs as well.

[1] Assuming there is no bug in the analyzer.

	L.o.C.	Parint	MT-Penjili	
		time	time	false alarms
Message	65	0.05	0.20s	0
Embedded	27 100	-	0.34s	7
Test 12	342	-	3.7s	1
Test 15	414	3.8	-	-

Figure 18.1: Benchmarks

18.2 Practical Results

The abstract semantics given in Part IV is denotational, so we may compute it recursively. Our final algorithm is to compute recursively $\mathit{guarantee}_{\ell_{cmd},\ell_\infty}$ applied to the initial configuration $\langle \top, \{\ell_\star\}, \lambda\ell.\bot, \bot, \emptyset\rangle$.

A large part of my work was to implement the analysis described in Part IV. The aim was to detect the same errors (array-overflows, `NULL` pointer dereference, invalid pointer dereference, division by zero and integer overflows) as for single-threaded programs.

I have implemented three tools. First I implement two proof-of-concept tools, then, I was in charge to extend Penjili so that it can handle multithreaded programs. For intellectual property reasons, the code is not given. This code is owned by EADS. These three tools use as entry a program written is the Newspeak language [HOL07, HOL08]. A program written in C code can be transformed into Newspeak code using the open source tool C2Newspeak. C2Newspeak is developped by EADS.

First, the proof-of-concept tool Parint is an standalone tool of 822 lines of code. Parint analyzes only programs with integer variables. It overapproximate integer values by ranges (see Section 2.3.2.a for the definition of the domain **Ranges**). This tool stubs the `pthread_create` function of the Pthread Library [IT04, Bar10, But06].

Second, the Tool MT-Penjili is implemented using the code of Penjili as a basis. It add 331 lines of code to Penjili. The main objective of this second proof-of-concept tool was to prove that my analysis is not restricted to "toy" analyzers and may be integrated into an industrial tool. MT-Penjili handle Pthread library but is not very precise: it was only a proof-of-concept tool.

Third, since MT-Penjili was working, I was responsible to integrate into the Penjili Tool my analysis. Now Penjili is able to analyze multithreaded programs using Pthread or Arinc. Integrating my analysis into Penjili needs 16 399 line modifications in Penjili source code and Penjili Benchmark suite. In the current version of Penjili (May 2010), the number of lines specific to multithread (i.e., that are executed only to analyse a multithreaded program) is 792, according to SlocCount, or 1294, according to `wc -l`. This represents 2.6% of the whole source code of Penjili.

In Table 18.1 we show some results on benchmarks of different sizes. L.o.C. means "Lines of Code". "Message" is a C file, with 3 threads: one thread sends an integer message

	LOC according to `wc -l`	LOC according to `sloccount`
Mini	101 000	64 000

Figure 18.2: Code Size of the "Mini" Program

to another through a shared variable. "Embedded" is extracted from embedded C code with two threads. "Test 12" and "Test 15" are sets of 12 and 15 files respectively, each one focusing on a specific thread interaction.

To give an idea of the precision of the analysis, we indicate how many false alarms were raised. Our preliminary experiments show that our algorithm loses precision in two ways: 1. through the (single-thread) abstraction on stores 2. by abstraction on interferences. Indeed, even though our algorithm takes the order of transitions into account for the current thread, it considers that interference transitions may be executed in an arbitrary order and arbitrary many times. This does not cause any loss in "Message", since the thread which sends the message never puts an incorrect value in the shared variable. Despite the fact that "Embedded" is a large excerpt of an actual industrial code, the loss of precision is moderate: 7 false alarms are reported on a total of 27 100 lines. Furthermore, it is because of this arbitrary order, that our analysis handles weak memory models.

Given this results, this analysis has been implemented in the industrial tool Penjili. Penjili was a tool that may check single-threaded programs. Now it is able to check multithreaded programs.

This analysis have been launched on an embedded software called "Mini". This software is quite large (See Figure 18.2), nevertheless we called it "Mini" because it is small compared to the softwares we want to analyze. The Figure 18.2 gives two ways to count the number of Source Line Code:

- Using the Linux Tool `wc -l` that counts the total number of lines of all source files.
- Using the tool SLocCount [Whe].

The Tool SLocCount does not count comments, blank lines and then is more accurate. Nevertheless, we also give the number of lines given by `wc -l` as other authors do.

Figure 18.3 gives the results of our analysis. In nearly 5h20min, we found only 233 alarms. Figure 18.4 gives more details on these alarms. The first column indicates which kind of alarm, e.g., we raise 12 "array out of bounds" alarms. The second column gives the number of alarms, and the third column gives the accuracy of the analysis. The accuracy of the analysis is the percentage of dangerous operations that have been proved correct, e.g, the tool prove that 99.44 percents of pointers dereferences are correct.

Furthermore, the fixpoint needed to computes *guarantee* is reached in only 3 steps.

We investigate where we lose time. The time needed to compute the first step in the *guarantee* fixpoint is 30min32s. Notice this time is less that $\frac{1}{3}$ of the time of 3 the iterations. The explanation is that, during the two other iterations, we discover new possible execution paths. When we sequentially executes the code of the threads, we can use the single-threaded analysis of Penjili on it. This needs 29min25s. This means that, to computation

Number of Alarms	233
Analysis time	5h 17min 21s
Analysis space	135.5 Mb
Number of iterations of the *guarantee* loop	3

Figure 18.3: Experimental Results of the Penjili Tool

Run-time error	Number of alarms	Accuracy
Array Out of Bounds	12	98,77%
Integer Overflow	193	88.24%
Division by Zero	0	100%
Invalid Pointer Dereference	28	99.44%

Figure 18.4: Penjili Alarms

time is increased by only 4% with our multithread domain $\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E} \rangle$ compared to the single-threaded domain $\mathcal{C}$.

18.3 Complexity

In practice, the analysis works and need a reasonnable amount of time (only some hours on a standard laptop) on large programs. We justify theoretically this point by a study of complexity.

The complexity of our algorithm greatly depends on widening and narrowing operators. Given a program ${}^{\ell_0}prog, \ell_\infty$, the *slowness* of the widening and narrowing in an integer w such that: widening-narrowing stops in always at most w steps on each loop and whenever *guarantee* is computed (which also requires doing an abstract fixpoint computation). Let the *nesting depth* of a program be the nesting depth of *while* and of *create* which[2] have a subcommand *create*.

Proposition 18.1. *Let d be the nesting depth, n the number of commands of our program, and, w the slowless of our widening. The time complexity of our analysis is $O(nw^{d+1})$ assuming operations on abstract stores are done in constant time.*

This is comparable to the $O(nw^d)$ complexity of the corresponding single-thread analysis, and certainly much better that the combinatorial explosion of interleaving-based analyses. Furthermore, this is beter than polynomial in an exponential number of states [FQ03].

Proof. Let $c({}^\ell cmd, \ell')$, $n({}^\ell cmd, \ell')$ and $d({}^\ell cmd, \ell')$ and $w({}^\ell cmd, \ell')$ be the complexity of analyzing ${}^\ell cmd, \ell'$, the size of ${}^\ell cmd, \ell'$ and the nesting depth of ${}^\ell cmd, \ell'$, the slowless of the widening and narrowing on ${}^\ell cmd, \ell'$ respectively. Let a and k the complexity of assign and of reading $\mathcal{K}(\ell)$ respectively.

[2] In our Semantics, each *create* needs a fixpoint computation, except *create* with no subcommand *create*.

Proposition 18.1 is a straightforward consequence of the following lemma[3]:

Lemma 18.1. *The complexity of computing* $(\!|{}^{\ell} cmd, \ell'|\!)_{\mathcal{Q}}$ *is* $O(an(w+k)w^{d-1})$

This lemma is proven by induction.
$c(lv := e) = a$
$c({}^{\ell_1} cmd_1; {}^{\ell_2} cmd_2, \ell_3) = c({}^{\ell_1} cmd_1, \ell_2) + c({}^{\ell_2} cmd_2, \ell_3)$
$c({}^{\ell_1}\textsf{while}(cond)\{{}^{\ell_2} cmd\}, \ell_3) \leqslant w({}^{\ell_1}\textsf{while}(cond)\{{}^{\ell_2} cmd\}, \ell_3) \times c({}^{\ell_2} cmd, \ell_1)$

If ${}^{\ell_2} cmd$ does not contain any subcommand *create*, then the fixpoint computation terminates in one step: $c({}^{\ell_1}\textsf{create}({}^{\ell_2} cmd), \ell_3) = k + c({}^{\ell_2} cmd)$
Else: $c({}^{\ell_1}\textsf{create}({}^{\ell_2} cmd), \ell_3) = k + w({}^{\ell_1}\textsf{create}({}^{\ell_2} cmd), \ell_3)) \times c({}^{\ell_2} cmd)$ □ □

18.3.1 Complexity of Operations on K

Notice that we have assumed that operation on $\mathscr{R}^{\mathbf{Labels}}$ are done in constant time in Proposition 18.1. This abstract store may be represented in different ways. The main problem is the complexity of the *basic* function, which computes a union for each element in $\mathcal{L}$. The naive approach is to represent $\mathcal{K} \in \mathscr{R}^{\mathbf{Labels}}$ as a map from $\mathcal{P}(\mathbf{Labels})$ to $\mathscr{R}$. Assuming that operations on maps are done in constant time, this approach yields a $O(tnw^d)$ complexity where t is the number[4] of *create*s in the program. We may also represent $\mathcal{K} \in \mathscr{R}^{\mathbf{Labels}}$ as some map $\mathcal{K}_M$ from $\mathcal{P}(\mathbf{Labels})$ to $\mathscr{R}$ such that $\mathcal{K}(\ell) = \bigcup_{\mathcal{L} \ni \ell} \mathcal{K}_M(\mathcal{L})$ and the function *basic* is done in constant time : $\textit{basic}_{lv:=e}\langle \mathcal{C}, \mathcal{L}, \mathcal{K}, I, \mathcal{E}\rangle \stackrel{\text{def}}{=} \langle \textit{inter}_I \circ \textit{elem}_{lv:=e}(\mathcal{C}), \mathcal{L}, \mathcal{K}_M[\mathcal{L} \mapsto \mathcal{K}_M(\mathcal{L}) \sqcup \textit{elem-inter}_{lv:=e}(\mathcal{C})], I\rangle$. Nevertheless, to access to the value $\mathcal{K}(\ell)$ may need up to t operations, which increases the complexity of *child-spawn* and *combine*. The complexity is then $O(n(w+t)w^{d-1})$.

18.3.2 Complexity of Widening

The slowness of the widening and narrowing operators, w, depends on the abstraction. Nevertheless, a widening is supposed to be fast.

Consider the classical widening on **Ranges** : $[x, x'] \triangledown [y, y'] = [z, z']$ where $z = \begin{cases} x & \text{if } y \geqslant x \\ -\infty & \text{else} \end{cases}$ and $z' = \begin{cases} x' & \text{if } y' \leqslant x' \\ +\infty & \text{else} \end{cases}$.
This widening never widen more than two times on the same variable. Therefore this widening is linear in the worst case.

[3] The functions arguments are omitted in the name of simplicity.

[4] This is different to the number of threads since an arbitrary number of threads may be created at the same location.

Part VI

Conclusion

CHAPTER 19

Conclusion

19.1 Conclusion

We have described a generic static analysis technique for multithreaded programs parametrized by a single-thread analysis framework and based on a form of rely-guarantee reasoning. To our knowledge, this is the first such *modular* framework: all previous analysis frameworks concentrated on a particular abstract domain. Such modularity allows us to leverage any static analysis technique to the multithreaded case. We have illustrated this by applying it to a large variety of abstract domains.

Our theoretical analysis generalizes cartesian abstraction [MPR06b, MPR06a, FQ03], it generalizes the P. Lammich and M. Müller-Olm Acquisition Histories analysis (with the same precision, for nearly all programs, see Section 17.3.2). And it generalizes R. Rugina and M. C. Rinard [RR99, RR03] analysis.

Our theoretical analysis allows us to use domains designed for the single-threaded case (e.g, string domains [AGH06], **Ranges**,...).

Furthemore, I have implemented this theoretical framework in an industrial tool (Penjili) and analyzed with it a large embedded program.

We have shown that our framework only incurred a moderate (low-degree polynomial) amount of added complexity. In particular, we avoid the combinatorial explosion of all

interleaving based approaches.

Our analysis is always correct, and produces reasonably precise information on the programs we tested.

19.2 Perspectives

As seen in Chapter 17 our analysis has been designed to be easily extended. We hope that this analysis can be extended to handle more kinds of programs, new kind of parrallel constructors. Some interesting parallel constructors are atomic blocks, invokation of another thread and synchronisation primitives.

An atomic block is executed sequentially, as if it was only one instruction. A primitive $\mathsf{atomic}\{cmd\}$ can be overapproximated by cmd. Nevertheless, if this is sound, this is not precise. We may wonder if we can update the $\mathcal{K}$-componnent of the abstract configuration in a more precise way. Furthermore, some blocks of instructions are executed "as if" they where atomics [Lip75, FF04, FFL05], e.g., due to mutexes. I hope this analysis may be extended to detect such blocks and then to enhance precision.

The invocation of another thread is allowed in the C# language. The main idea is that a thread may "invoke" another thread to execute a function f, i.e., the function f will be executed by the invoked thread. The invokation of another thread on a function f may be overapproximated by $\mathsf{create}(f)$, but we may hope to improve precision analysing this primitive since the execution of the function f cannot interfere with the invoked thread. Maybe we may detect, in some cases, which thread i is invoked, and then update in a different way the $\mathcal{K}$-componnent of abstract configurations.

There exists a large variety of synchronisation primitives, e.g., the Posix norm uses condition variables. A thread may wait on a variable, and another thread may launch a signal on a condition variable awaking a thread that waits on this variable. As explained by H. Seidl and B. Steffen [SS00], these synchronisations may be ignored, since they only reduce possible behaviors. Nevertheless I hope these synchronisations may be taken into consideration to improve precision. May be we can use a set like *after* to distinguish transitions fired after some synchronization point and transitions done before some synchronization point. Hence, in the abstract, we may define some $\mathcal{K}(\ell)$ where ℓ is the label of a synchronisation primitive.

CHAPTER 20

Index

CHAPTER 21

List of Figures

CHAPTER 22

Bibliography

[ABBM10] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *POPL '10*, pages 7–18, New York, NY, USA, 2010. ACM.

[AGH06] Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. Static Analysis of String Manipulations in Critical Embedded C Programs. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 35–51, Seoul, Korea, August 2006. Springer Verlag.

[Bar10] Blaise Barney. Posix threads programming, 2010. `https://computing.llnl.gov/tutorials/pthreads/`.

[BMOT05] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. pages 473–487, 2005.

[Boa08] "OpenMP Architecture Review Board". *OpenMP Application Program Interface*. Mai 2008.

[But06] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 2006.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CC91] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/-narrowing approaches to abstract interpretation. JTASPEFL '91, Bordeaux. *BIGRE*, 74:107–110, October 1991.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[CC04] P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.

[CDNB08] Christopher L. Conway, Dennis Dams, Kedar S. Namjoshi, and Clark Barrett. Pointer analysis, conditional soundness, and proving the absence of errors. In *SAS '08: Proceedings of the 15th international symposium on Static Analysis*, pages 62–77, Berlin, Heidelberg, 2008. Springer-Verlag.

[CFR+97] Agostino Cortesi, Gilberto Filé, Francesco Ranzato, Roberto Giacobazzi, and Catuscia Palamidessi. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1):7–47, 1997.

[CMB+95] Michael Codish, Anne Mulkers, Maurice Bruynooghe, Maria García de la Banda, and Manuel Hermenegildo. Improving abstract interpretations by combining domains. *ACM Trans. Program. Lang. Syst.*, 17(1):28–44, 1995.

[Cou96] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996.

[Cou05] P. Cousot. Forward relational infinitary static analysis, 2005.

[fCS98] "Supercomputing Technologies Group MIT Laboratory for Computer Science". *Cilk 5.4.6 - Reference Manual.* 1998.

[FF04] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM Press.

[FFL05] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 47–58, New York, NY, USA, 2005. ACM Press.

[FQ03] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–224. Springer, 2003.

[GBC+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. Technical report, 2007.

[GHK+98] Gierz, Hofmann, Keimel, Lawson, Mislove, and Scott. *A Compendium of Continuous Lattices*. second edition, 1998.

[GHK+03] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D. Scott. *Continuous Lattices and Domains*. Cambridge University Press, 2003.

[GJSB05] James Gosling, Bill Joy, Guy L. Steele, and Gilad Brach. *The Java Language Specification, Third Edition*. May 2005.

[GT06] Sumit Gulwani and Ashish Tiwari. Combining abstract interpreters. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 376–386, New York, NY, USA, 2006. ACM.

[HOL07] Charles Hymans and Olivier Levillain. Newspeak: Big Brother is compiling your code. Technical report, EADS France, 2007. This tool may be downloaded on `http://www.penjili.org/newspeak.html`.

[HOL08] Charles Hymans and Olivier Levillain. Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C. Technical report, EADS IW/SE, 2008. This tool may be downloaded on `http://www.penjili.org/newspeak.html`.

[Hym06] Charles Hymans. Presentation at lsv seminar, at ENS cachan, 2006.

[ISO99] ISO/IEC. *Programming Languages C*. 1999.

[ISO06] ISO/IEC. *Programming languages — C#*. 2006.

[IT04] IEEE and The Open Group. The Open Group base specifications issue 6 - IEEE Std 1003.1, 2004. `http://www.opengroup.org/onlinepubs/009695399/toc.htm`.

[KIG05] Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. Reasoning about threads communicating via locks. In *In Computer Aided Verification*, pages 505–518. Springer, 2005.

[KSV96] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, 18(3):268–299, 1996.

[Lam79] Leslie Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. IEEE, 1979.

[Lip75] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LMO07] Peter Lammich and Markus Müller-Olm. Precise fixpoint-based analysis of programs with thread-creation and procedures. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2007.

[LMO08] Peter Lammich and Markus Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In *SAS'08*, pages 205–220. Springer, 2008.

[Mic10] Microsoft. .NET framework general reference – design guidelines for class library developers, 2010.

[MPR06a] Er Malkis, Andreas Podelski, and Andrey Rybalchenko. Thread-modular verification and cartesian abstraction. In *Thread Verification workshop, TV06*, pages 21–22. Springer, 2006.

[MPR06b] Er Malkis, Andreas Podelski, and Andrey Rybalchenko. Thread-modular verification is cartesian abstraction. In *Interpretation, 3rd International Colloquium on Theoretical Aspects of Computing*, pages 21–22. Springer, 2006.

[MPR07] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. Precise thread-modular verification. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2007.

[OSS09] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI '06: Proceedings*

of the 2006 ACM SIGPLAN conference on Programming language design and implementation, pages 320–331, New York, NY, USA, 2006. ACM Press.

[RR99] Radu Rugina and Martin C. Rinard. Pointer analysis for multithreaded programs. In *PLDI*, pages 77–90, 1999.

[RR03] Radu Rugina and Martin C. Rinard. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25(1):70–116, 2003.

[SS00] Helmut Seidl and Bernhard Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nordic J. of Computing*, 7(4):375–400, 2000.

[Vic07] Paul Vick. *The Microsoft Visual Basic Language Specification – Version 9.0.* 2007.

[VMo03] Varmo Vene and Markus Muller-olm. Global invariants for analyzing multithreaded applications. In *In Proc. of Estonian Academy of Sciences: Phys., Math*, pages 413–436, 2003.

[VV07] Vesal Vojdani and Varmo Vene. Goblint: Path-sensitive data race analysis. In *SPLST*, 2007.

[Whe] David A. Wheeler. Sloccount.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. pages 1–12, 1995.