Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Logica di Hoare applicata su un linguaggio procedurale: implementazione Matita

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Fabio altadonna

Marzo 2024
Anno Accademico 2022-2024

Indice

Abstract

Nell'ambito dello sviluppo del software, garantire la correttezza dei programmi scritti é cruciale per fornire affidabilitá e sicurezza. La logica di Hoare[2], introdotta da Tony Hoare nel 1969, fornisce un approccio formale per la specifica e la verifica della correttezza dei programmi. Tale approccio si basa sull'idea di specificare le precondizioni e le postcondizioni per ciascun frammento di programma, definendo la nozione di *tripla di Hoare*, la quale consente di dimostrare logicamente che un determinato programma soddisfa determinate proprietá. Lo scopo del progetto presentato nel documento é quello di utilizzare il software Matita[1], per definire la sintassi e la semantica di un linguaggio imperativo con memoria statica (stack), memoria dinamica (heap) e parallelismo. Dopo aver creato il linguaggio, implementare, dimostrandone la validitá attraverso la funzionalitá di Proof Assistant di Matita, le regole di inferenza sulle triple di Hoare, per consentire in ultimo di effettuare dimostrazioni rigorose su programmi scritti nello stesso linguaggio. Il presente documento riepiloga in primis i concetti fondamentali riguardanti la teoria che sta alla base del sistema di inferenze di Hoare e di come quest'ultimo dipenda dal linguaggio di programmazione su cui si adopera. In seguito viene invece relazionato il processo di lavoro svolto su Matita per la relativa implementazione pratica.

Capitolo 1

Stato dell'arte e nozioni preliminari

1.1 Introduzione alla logica di Hoare

1.1.1 Predicati Hoare

Prima di introdurre il concetto di Tripla di Hoare, occorre definire la nozione di *stato* di un programma. Per far ció, si consideri un generico linguaggio di programmazione che consente iterazione, selezione e definizione/aggiornamento variabili: [3]

$\langle C \rangle$ ::= 'none'
| 'while' $\langle E \rangle$ 'do' $\langle C \rangle$ 'endwhile'
| 'if' $\langle E \rangle$ 'then' $\langle C \rangle$ 'else' $\langle C \rangle$ endif
| $\langle I \rangle$ ':=' $\langle E \rangle$
| $\langle C \rangle$; $\langle C \rangle$

[1] Definiamo lo stato di questo linguaggio di programmazione come una mappa tra variabili a valori. A titolo di esempio, il seguente programma in esecuzione: `x := 10 ; x := 20` modificherá lo stato 2 volte. Dopo il primo

[1]La sintassi delle espressioni $\langle E \rangle$ e degli identificatori $\langle I \rangle$ non é rilevante.

assegnamento, dallo stato vuoto, che non assegna nessun valore ad alcuna variabile, si passerá allo stato: $\{x \mapsto 10\}$ e in seguito, dopo il secondo, a: $\{x \mapsto 20\}$. Diventa quindi possibile scrivere predicati che hanno come variabile libera uno stato.

Predicato Hoare. Un predicato P si definisce *predicato Hoare* sse ha come variabile libera uno stato.

Per esempio si puó considerare il predicato Hoare $P(S) := S[x] > 10$ da interpretare come la variabile 'x' letta nello stato 'S' é > 10. Siccome in un linguaggio di programmazione imperativo come quello esemplificato sopra lo stato é mutevole per definizione, risulta evidente come lo stesso predicato possa essere valido in un determinato punto dell'esecuzione del programma e non valido in un altro. L'esempio riportato rende evidente questa proprietá: P vale solamente alla fine del secondo assegnamento, mentre non vale in tutto il resto del codice. Come notazione si scrive $S \Vdash P$ per indicare che lo stato S rende valido (soddisfa) il predicato Hoare P. Avendo definito questa particolare classe di predicati, si é molto vicini alla definizione di tripla di Hoare: intuitivamente, la precondizione e la postcondizione sono proprio predicati Hoare mentre il frammento di programma menzionato nell'introduzione avrá a che vedere con i comandi. Prima di poter attribuire una definizione univoca al concetto di tripla di Hoare occorre, tuttavia, trattare cosa s'intende per computazione e come si possa attribuire un significato inequivocabile alla sintassi di un linguaggio di programmazione.

1.1.2 Semantica dei linguaggi di programmazione

Nella sezione precedente si é adottato a titolo esemplificativo un semplice linguaggio di programmazione, enunciandone la sintassi. Per poter discutere formalmente e dettagliatamente di un linguaggio, tuttavia, é necessario avere consapevolezza, nello specifico, di come i comandi ad esso appartenenti possano mutare lo stato. La semantica di un linguaggio di programmazione svolge proprio tale compito, attribuendo un significato

univoco ai suoi comandi. Esistono diversi tipi di semantica [5], il presente documento tratterá sempre di semantica operazionale e, in particolare, di semantica operazionale strutturata (S.O.S.)[4]. La semantica operazionale attribuisce un significato ai comandi mediante una serie di regole d'inferenza. Tali regole descrivono, dato un comando e uno stato iniziale in cui esso viene eseguito, la transizione verso il nuovo stato eventualmente mutato dopo l'esecuzione del comando. La semantica operazionale, oltre a poter essere strutturata (quando lo schema delle transizioni dei comandi segue lo schema della relativa sintassi) o meno, si divide in 2 ulteriori categorie:

1. a grandi passi (big steps O.S.)
2. a piccoli passi (small steps O.S.).

La seconda adotta un sistema di regole d'inferenza nel quale ogni transizione apporta il minimo numero di modifiche allo stato in cui il comando analizzato viene eseguito. La prima, al contrario, adotta un approccio piú astratto, nel quale le transizioni apportano diverse modifiche allo stato iniziale. Data una semantica del primo tipo é quindi facile derivarne una del secondo, ma non viceversa. Inoltre, la semantica a grandi passi é piú semplice da trattare, visto che studia il comportamento dei comandi meno nello specifico. Di contro, essa é meno scalabile, fornisce meno controllo sulle transizioni ed é inapplicabile nel caso di programmazione concorrente, nella quale é fondamentale che ogni transizione sia indipendente e isolata. Come esempio significativo si riporta la semantica di un comando `while`$\langle E \rangle$*do*$\langle C \rangle$`endwhile`:

WHILE TRUE

$$\frac{\text{E} \hookrightarrow_b \text{true}}{\langle \texttt{while e do c endwhile}, S\rangle \hookrightarrow \langle \texttt{c; while do c endwhile}, S\rangle}$$

WHILE FALSE

$$\frac{\text{E} \hookrightarrow_b \text{false}}{\langle \texttt{while e do c endwhile}, S\rangle \hookrightarrow \langle \texttt{none}, S\rangle}$$

Le transizioni vengono descritte come usuali regole d'inferenza. Per comprendere il significato delle regole sopra occorre dare le seguenti definizioni (per il momento informali)[2]:

Transizione. Dati 2 stati S1, S2 e 2 comandi `c1`, `c2` $\langle S1, c1\rangle \hookrightarrow \langle S2, c2\rangle$ significa che eseguendo c1 nello stato S1, il programma passa allo stato S2, avendo come nuovo comando da eseguire c2.

Computazione. Dati 2 stati S1, S2 e 2 comandi `c1`, `c2`, se esiste una sequenza di transizioni $\langle S1, c1\rangle \hookrightarrow ... \hookrightarrow \langle S2, c2\rangle$ allora si indica $\langle S1, c1\rangle \hookrightarrow_\star \langle S2, c2\rangle$ dove $\hookrightarrow_\star$ é un predicato dello stesso tipo di $\hookrightarrow$ e si legge "computa in". Piú in generale, $\hookrightarrow_\star$ Viene detta *chiusura transitiva*

Computazione Finale. Dati 2 stati S1, S2 un comando `c1`. Si indica con $\langle S1, c1\rangle \hookrightarrow *_{\text{final}} S2$ la computazione in uno stato S2 finale, logicamente equivalente a $\exists c2.\langle S1, c1\rangle \hookrightarrow_\star \langle S2, c2\rangle \wedge$ S2 é stato finale. [6]

Nota su definizione formali. La trattazione formale delle definizioni enunciate é presente nella sezione relativa all'implementazione Matita. Per il momento é sufficiente avere un'idea informale per poter definire e comprendere i concetti teorici successivi

Nelle regole illustrate sopra, si assuma che $\hookrightarrow_b$ é sia il predicato che associa un'espressione al booleano in cui essa si valuta. Notare che, in questo caso, la transizione NON modifica in alcun modo lo stato, ma modifica soltanto il comando da eseguire. In caso `c` sia una dichiarazione di variabile e la guardia si valuti in `true` allora il comando `while` si valuterá in una composizione e sará la regola d'inferenza che descrive il comando `c1; c2` a occuparsi di aggiornare opportunamente lo stato. La nozione di "computazione" segue quindi direttamente dalle regole di transizione esplicitate nella semantica. A questo punto é possibile fornire una definizione di tripla Hoare.

[2]La trattazione formale di questi concetti é presente nella sezione che descrive l'implementazione Matita degli stessi.

1.1.3 Triple di Hoare

Tripla di Hoare. Dati 2 predicati Hoare P e Q, rispettivamente chiamati precondizione e postcondizione, un comando c, uno stato in cui esso viene eseguito S1 e uno stato finale S2, $\{P\}c\{Q\}$ viene definita come tripla di Hoare valida sse $\langle S1, c1\rangle \hookrightarrow *_{\text{final}} S2 \wedge S1 \Vdash P \implies S2 \Vdash Q$.

Piú semplicemente, $\{P\}c\{Q\}$ é valida se nello stato attuale vale P ed eseguendo "completamente" il comando c si arriva in uno stato in cui vale Q. Notare che in caso c sia una composizione `c1; c2`, consumarlo interamente vuol dire consumare (ricorsivamente) entrambe le componenti della composizione stessa. Si consideri ancora una volta il programma: `x := 10 ; x := 20`. A partire da esso sono valide, per esempio, le triple:

$$\{\texttt{TRUE}\}\ \texttt{x := 10}\ \{\texttt{S[x] > 0}\}$$

$$\{\texttt{S[x]>5}\}\ \texttt{x := 20}\ \{\texttt{S[x] > 10}\}$$

$$\{\texttt{TRUE}\}\ \texttt{x := 10; x := 20}\ \{\texttt{S[x] > 10}\}$$

Per non appesantire la scrittura delle triple, nel documento sará commesso un piccolo abuso di notazione tralasciando lo stato S e tenendo nelle precondizioni e nelle postcondizioni soltanto le variabili, assumendo sempre che esse siano valutate nello stato. La seconda tripla verrá indicata quindi semplicemente come $\{\texttt{x > 5}\}\ \texttt{x := 20}\ \{\texttt{x > 10}\}$.

1.2 Regole d'inferenza sulle triple

Per facilitare le dimostrazioni sulla correttezza dei programmi. i.e. sulle pre e post condizioni delle triple, viene definito, per ogni linguaggio, un sistema di regole d'inferenza che lega le triple di Hoare. É molto importante sottolineare come tali regole siano sensibili alla semantica attribuita ai comandi del linguaggio. Una regola valida per un primo linguaggio potrebbe non essere tale per un secondo. Di seguito si riportano, con una breve descrizione, le regole utilizzabili nel semplice linguaggio descritto

precedentemente, assumendo di attribuire ai relativi comandi una semantica standard.

NONE

$$\frac{}{\{P\}\ \texttt{none}\ \{P\}}$$

Dato che il comando `none` non modifica lo stato, lo stesso predicato Hoare é valido prima e dopo il comando.

ASSEGNAMENTO

$$\frac{}{\{P[e/x]\}\ \texttt{x := e}\ \{P\}}$$

La notazione $P[e/x]$ indica che il predicato viene valutato usando il valore dell'espressione `e` al posto di x.

IF

$$\frac{\{P \wedge e \hookrightarrow_b \text{true}\}\texttt{c1}\{\text{Q}\} \qquad \{P \wedge e \hookrightarrow_b \text{false}\}\texttt{c2}\{\text{Q}\}}{\{P\}\ \texttt{if e then c1 else c2 endif}\ \{Q\}}$$

Se sia il comando nel ramo `then`, sia quello nel ramo `else` verificano la stessa postcondizione, allora l'intero if la verifica

WHILE

$$\frac{\{P \wedge e \hookrightarrow_b \text{true}\}\texttt{c}\{\text{P}\}}{\{P\}\ \texttt{while e do c endwhile}\ \{P \wedge e \hookrightarrow_b false\ \}}$$

Se la validitá del predicato P viene mantenuta dopo l'esecuzione del corpo del while, allora sará valido prima e dopo aver eseguito l'intero ciclo. In questo contesto il predicato P viene detto "invariante".

COMPOSIZIONE

$$\frac{\{\text{P}\}\texttt{c1}\{\text{Q}\} \qquad \{\text{Q}\}\texttt{c2}\{\text{R}\}}{\{P\}\ \texttt{c1; c2}\ \{R\}}$$

Se eseguendo in sequenza i comandi `c1` e `c2` si verifica la postcondizione R, allora R sará postcondizione valida nella conclusione
Esiste, inoltre, una regola d'inferenza aggiuntiva, questa indipendente dalla semantica del linguaggio di programmazione:

CONSEQUENCE

$$\frac{Ps \implies Pw \qquad Qs \implies Qw \qquad \{\text{Pw}\}\texttt{c}\{\text{Qs}\}}{\{\text{Ps}\}\texttt{c}\{\text{Qw}\}}$$

Con questa regola d'inferenza si introduce un non-automatismo per le prove di correttezza, in quanto occorre trovare i predicati le cui implicazioni logiche siano coerenti con quelle delle premesse.

1.2.1 Accenni sulla logica di Hoare totale

Fino ad adesso si é analizzata la correttezza di un programma solamente sulla base dello stato, ovvero della mappa tra variabili e valori. É possibile descrivere in maniera piú stringente la correttezza di un programma, esaminando anche se esso termina o meno, attraverso la logica di Hoare totale. Nella definizione di tripla di Hoare totale, il predicato $\hookrightarrow_{*_{\text{final}}}$ (indicante che un comando computa in uno stato finale) viene spostato nella conclusione, con conseguente necessitá di essere dimostrato. L'unica regola impattata da tale cambiamento risulta essere quella del *while*, unico comando che potrebbe portare a una computazione infinita. Indicando con [P] c [Q] un tripla di Hoare con correttezza totale, si sostituisce dunque la regola del while con la seguente:

WHILE_TOTALE

$$\frac{[P \wedge e \hookrightarrow_b \text{true} \wedge t = n]\ \texttt{c}\ [P \wedge t < n] \qquad P \wedge e \hookrightarrow_b \text{true} \implies t \geq 0}{\texttt{[}P\texttt{] while e do c endwhile [}P \wedge e \hookrightarrow_b false\ \texttt{]}}$$

In tale regola "t" viene chiamata "variante" e garantisce che il numero di iterazioni sia finito, in quanto decresce a ogni computazione del corpo del while ed é positiva fintanto che la guardia viene valutata in true.

1.3 Separation Logic

1.3.1 introduzione alla Sep. Logic

Sono state esplicitate le regole d'inferenza per un linguaggio con memoria statica. Il prossimo passo é analizzare la casistica in cui il linguaggio considerato supporti allocazione/deallocazione di memoria dinamica (heap). Si considerino quindi i comandi aggiuntivi

$\langle C \rangle$ ::= ...
| 'alloc' $\langle E \rangle$'
| 'mem_write' $\langle A \rangle \langle E \rangle \langle E \rangle$
| 'mem_read' $\langle A \rangle \langle E \rangle \langle E \rangle$
| 'dealloc' $\langle A \rangle$

dove $\langle A \rangle$ indica un indirizzo e dove l'allocazione alloca una cella di memoria con grandezza pari alla valutazione dell'espressione passata, la scrittura scrive nella cella di memoria puntata dall'indirizzo, al dato indice, la valutazione dell'espressione. Appare evidente come la nozione di stato introdotta all'inizio del capito sia da rimodulare per accogliere queste nuove introduzioni. Ecco allora che ad esso, oltre alla mappa variabile-valore presente dal vecchio linguaggio, viene aggiunta una ulteriore mappa indirizzo-dati_heap. A questo punto sarebbe teoricamente possibile usare i connettivi della logica predicativa per aggiungere ai predicati Hoare controlli sulla memoria dinamica. Tuttavia, l'utilizzo di predicati simili risulta poco scalabile e tedioso. Vengono allora introdotti dei predicati aggiuntivi facenti parte di ció che viene chiamata "Separation Logic" [7].

- `emp`: indica che nello stato l'heap non ha allocazioni
- $a \longmapsto v$: indica che all'indirizzo x é allocato il dato v
- $P * Q$**, separating conjunction**: indica che esistono 2 memorie disgiunte, la cui unione é la memoria attuale, tali che lo stato considerando la prima memoria soddisfa P e lo stato considerando la seconda memoria soddisfa Q

- $P \mathrel{-\!\!\circledast} Q$, **separating implication**: indica che se esiste una memoria disgiunta dall'attuale che soddisfa P, allora posso unire tale memoria con l'attuale per soddisfare Q.

1.3.2 Regole d'inferenza per la Sep. Logic

Con i nuovi predicati, é ora possibile integrare nuove regole d'inferenza sulle triple di Hoare per trattare la correttezza di programmi che usano i nuovi comandi modificanti la memoria dinamica. Le regole descritte nella presente sezione ricoprono un ruolo esclusivamente esemplificativo, considerando una semantica intuitiva del linguaggio descritto.

DEALLOC

$$\frac{}{\{a \longmapsto v * P \ \}\ \texttt{dealloc a}\ \{P\}}$$

ALLOC

$$\frac{}{\{\texttt{emp}\}\ \texttt{alloc x n}\ \{\exists a.a \longmapsto v \wedge len(v) = nP\}}$$

READ

$$\frac{}{\{\texttt{emp}\}\ \texttt{alloc x n}\ \{\exists a.a \longmapsto v \wedge len(v) = nP\}}$$

LOOKUP

$$\frac{}{\{p \longmapsto v \wedge d = v[i] \wedge P[d/x]\}\ \texttt{mem\_read x i p}\ \{P\}}$$

WRITE

$$\frac{}{\{p \longmapsto c * P\}\ \texttt{mem\_write p i v}\ \{p \longmapsto c' * P \wedge c'[i] = v \wedge \forall i' \neq i.c'[i'] = c[i']\ \}}$$

Notare come tutte le regole, ma in particolare la lookup e la write, siano estremamente dipendenti dalla semantica adottata nel linguaggio.
Nell'esempio presente si sta assumendo che una cella di memoria si

comporti similmente a un vettore (nella write infatti si esplicita nella postcondizione che l'indirizzo p debba puntare allo stesso vettore di partenza meno che all'indice). A ogni modo, se non é possibile applicare direttamente una tra le regole esplicitate é possibile usare il *framing*.

1.3.3 Regola di framing

Il principio fondamentale della Separation Logic é la *Framing Rule*:

REGOLA DI FRAMING

$$\frac{\{P\}\ \texttt{c}\ \{Q\}}{\{P * R\}\ \texttt{c}\ \{Q * R\}}$$

Tale inferenza vale solo se c non modifica alcuna variabile "osservata" nel predicato R e consente di ragionare localmente sulla memoria modificata dai programmi. La correttezza della regola dipende dalla semantica utilizzata nel linguaggio di programmazione esaminato. Consideriamo di attribuire 2 semantiche differenti al comando `dealloc a`, nel caso in cui l'indirizzo `a` non punti a una cella allocata: [8]

1) il comando solleva errore di memoria;

2) il comando agisce come skip.

Nel caso 2 é risulterebbe valida la tripla $\{emp\}$ `dealloc a` $\{emp\}$ e applicando la Frame Rule: $\{emp * p \mapsto [1]\}$ `dealloc p` $\{emp * p \mapsto [1]\}$ da cui l'assurdo $\{p \mapsto [1]\}$ `dealloc p` $\{p \mapsto [1]\}$. Le condizioni necessarie e sufficienti per rendere corretta la regola sono:

Safety Monotonicity: se un comando `c` termina eseguendo a partire da uno stato con memoria m1, allora esso termina anche da uno stato con memoria m2, dove m2 include m1.

Frame Property: se l'esecuzione di un comando `c` a partire da un certo stato con memoria m termina in uno stato con memoria m2 e se lo stesso comando termina per un qualche stato finale partendo da uno

stato con stesso store e memoria m' inclusa in m, allora esiste una computazione dallo stato con memoria m' con `c` a uno stato con memoria m2', inclusa in m2. Inoltre, considerando la differenza tra m e m', vale che questa é inclusa in m2.

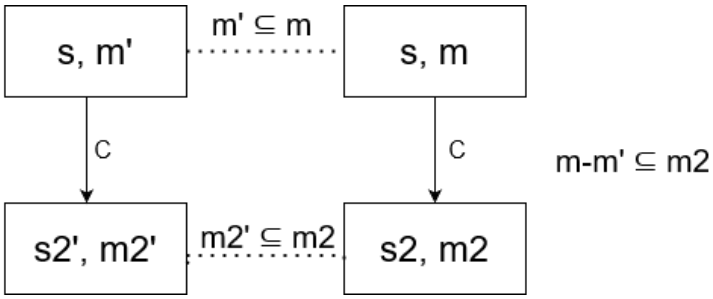

Figura 1.1: Frame property

1.4 Cenni sulla concorrenza

Si introduca ora il comando di composizione concorrente `c1||c2`, la cui semantica é quella di eseguire i 2 comandi `c1` e `c2` in maniera concorrente (i.e. con interleaving). Risulta allora corretta la regola di inferenza:

$$\text{COMP. CONCORRENTE}\quad \frac{\{P\}\ \texttt{c1}\ \{Q\} \qquad \{R\}\ \texttt{c2}\ \{S\}}{\{P * R\}\ \texttt{c1||c2}\ \{Q * S\}}$$

L'inferenza vale quando c1 e c2 non modificano variabili libere di P e R. La regola puó inoltre essere generalizzata per una composizione di n comandi [9]:

$$\text{COMP. CONCORRENTE}\quad \frac{\{P_1\}\ \texttt{c}_1\ \{Q_1\} \qquad \ldots \qquad \{P_n\}\ \texttt{c}_n\ \{Q_n\}}{\{P_1 * \ldots * P_n\}\ \texttt{c}_1||\ldots||\texttt{c}_n\ \{Q_1 * \ldots * Q_n\}}$$

Oltre a questo, esistono diverse tecniche per specificare altre regole di inferenza. Le regole relative alla concorrenza, piú che tutte le altre variano

enormemente in base al linguaggio scelto, e alle primitive di sincronizzazione possedute. L'idea generale é utilizzare il concetto di *permessi* all'interno dei diversi processi/thread concorrenti.

Capitolo 2

Implementazione linguaggi

2.1 Introduzione e modalitá di lavoro

Il lavoro svolto, oltre alla comprensione esaustiva dei concetti teorici esplicitati nel capitolo precedente, ha richiesto uno studio anche del software Matita e del suo paradigma di programmazione e dimostrazione. Prima di iniziare la vera e propria stesura del codice Matita ho quindi letto il tutorial fornito [11] e diviso il lavoro in:

1. scrittura linguaggio "*while*" (solo iterazione e selezione)
2. aggiunta memoria heap dinamica
3. aggiunta stack e gestione funzioni
4. aggiunti thread e computazioni parallele
5. dimostrazioni sulla logica di Hoare (capitolo successivo)

Ci sono state diverse modifiche operative nel corso della progettazione; il codice Matita relativo al primo linguaggio non é uguale a quello del secondo meno lo heap; e anche l'implementazione della logica di Hoare ha apportato collateralmente modifiche retroattive a quanto giá scritto. Nel presente documento verranno trattati i linguaggi comprendenti le modifiche

apportate nel file finale, per evitare incoerenze tra le varie sezioni. Dopo la stesura di ogni linguaggio, inoltre, sono stati scritti dei piccoli programmi di prova nel linguaggio definito, integrando operativamente anche una fase di testing. Il file finale, che non include i test effettuati sui linguaggi, comprende piú di 2000 righe di codice Matita, tra definizioni e dimostrazioni. L'implementazione della logica di Hoare é arrivata alla Separation Logic, implementandone i connettivi ma non dimostrando tutte le regole d'inferenza necessarie a rendere completo il sistema di deduzione.

2.2 Caratteritiche generali

Le caratteristiche generali, comuni alle varie espansioni del linguaggio prodotto sono:

variabili identificate tramite numeri naturali

i tipi del linguaggio compredono naturali, booleani e puntatori

il linguaggio non possiede uno stato di errore

il linguaggio non verifica il tipaggio delle espressioni

il linguaggio é privo di ottimizzazioni [1]

non é stata definita una notazione user-space completa

2.3 Linguaggio while

2.3.1 Sintassi

La sintassi del linguaggio é stata definita in Matita attraverso 2 tipi induttivi **Exp** e **C**, rispettivamente per espressioni e comandi. La sintassi

[1] Le strutture dati "underlying" per gestire i diversi tipi di stato sono liste e le ricerche lineari

dei comandi é la stessa di quella vista alla prima sezione, che si traduce in Matita nel codice:

```
inductive C: Type[0] ≝
| none: C
| def: nat → Exp → C
| comp: C → C → C
| if_then_else: Exp → C → C → C
| while_do: Exp → C → C
.
```

Mentre per la sintassi delle espressioni si ha il tipo induttivo:

```
inductive Exp: Type[0] ≝
| nat_exp: ℕ → Exp
| bool_exp: bool → Exp
| sum_exp: Exp → Exp→ Exp
| value_of: ℕ → Exp
| eq_exp: Exp → Exp → Exp
| le_exp: Exp → Exp → Exp
| and_exp: Exp → Exp → Exp
| not_exp: Exp → Exp
.
```

La definizione dell'elenco dei tipi linguaggio é:

```
inductive Value: Type[0] ≝
| nat_value: ℕ → Value
| bool_value: bool → Value
.
```

E quella riguardante gli identificatori delle variabili:

```
inductive Var_Id: Type[0] ≝
| var: ℕ → Var_Id
.
```

2.3.2 Definizione stato e semantica

Nel linguaggio lo stato viene definito come la coppia $\langle M, C\rangle$ (dove M é la mappa variabile-valore descritta nel primo capitolo). In Matita ho definito un tipo di record:

```
record State: Type[0] ≝ {
  store: list (ℕ × Value) ;
  cmd: C
}.
```

Descrizione regole di inferenza

É noto che una regola d'inferenza é un modo alternativo di scrivere un'implicazione:

$$\frac{A}{B} \equiv (A \implies B)$$

Pertanto in Matita é possibile scrivere le regole di inferenze come predicati induttivi, dove un predicato induttivo é' un particolare tipo induttivo, che ha come ultimo valore una `Prop`[2]. Definiamo allora il predicato induttivo `Exp_Eval`, che dati un espressione, un valore e uno store in cui effettuare la valutazione, é verificato quando l'espressione si valuta proprio nel valore.

```
inductive Exp_Eval: Exp → Value → list (Var_Id × Value) → Prop ≝
| natural: ∀n, s. Exp_Eval (nat_exp n) (nat_value n) s
| boolean: ∀b, s. Exp_Eval (bool_exp b) (bool_value b) s
| variable: ∀i, s, v. find_var_store = Some ? v →
    Exp_Eval (value_of i) v s
| sum: ∀a, b, n1, n2, s. Exp_Eval a (nat_value n1) →
                         Exp_Eval b (nat_value n2) →
      Exp_Eval (sum_exp a b) (nat_value (n1 + n2))
| eq_b: ∀a, b, b1, b2, s. Exp_Eval a (bool_value b1) s →
                          Exp_Eval b (bool_value b2) s →
      Exp_Eval (eq_exp a b) (bool_value (b_equals b1 b2)) s

| eq_n: ∀a, b, n1, n2, s. Exp_Eval a (nat_value n1) →
                          Exp_Eval b (nat_value n2) →
      Exp_Eval (eq_exp a b) (bool_value (eqb n1 n2)) s

| le: ∀a, b, n1, n2, s.  Exp_Eval a (nat_value n1) →
                          Exp_Eval b (nat_value n2) →
      Exp_Eval (le_exp a b) (bool_value (leb n1 n2)) s

| logical_and: ∀a, b, b1, b2, s.
                          Exp_Eval a (bool_value b1) →
                          Exp_Eval b (bool_value b2) →
      Exp_Eval (and_exp a b) (bool_value (andb b1 b2)) s

| logical_not: ∀a, s, b. Exp_Eval a (bool_value b) →
      Exp_Eval (not_exp a) (bool_value (notb b)) s
.
```

Per chiarezza, il caso sum é la regola d' inferenza:

$$\frac{\text{SUM}\quad \text{a} \hookrightarrow_s \text{n1} \qquad \text{b} \hookrightarrow_s \text{n2}}{\text{sum a b} \hookrightarrow_s \text{n1} + \text{n2}}$$

[2]il tipo Matita comprendente tutte le proposizioni

Dove `x` $\hookrightarrow_s$ `y` $\equiv$ `Exp_Eval x y s`.
Notare come all'interno del predicato induttivo si faccia riferimento a funzioni definite nello user-space di Matita, come la `find_var_store` [3] o lo stesso operatore + di somma.

2.3.2.1 Tipo option

Matita definisce il tipo `option`:

```
inductive option (A:Type[0]) : Type[0] ≝
|None : option A
|Some : A → option A.
```

Intuitivamente, é un tipo che "potrebbe" contenere un dato. Il tipo `option` si presta molto bene, pertanto, a essere il tipo che le funzioni di ricerca (come `find_var_store`) ritornano. Conseguentemente la condizione `find_var_store s i = Some ?  v` nella semantica delle espressioni, assicura che il predicato NON possa essere valutato (e quindi che il programma scritto in user-space si blocchi) nel caso l'espressione contenga una variabile non definita nello store.

La semantica dei comandi, ovvero l'insieme di regole d'inferenza che descrivono una **transizione** é descritta nel predicato induttivo Step:

```
inductive Step: State → State → Prop ≝
| left_none: ∀s, c.
      Step (mk_State s (none; c)) (mk_State s c)
| none: ∀s. Step (mk_State s none) (mk_State s none)
| def: ∀s, i, e, v, s2.
                          Exp_Eval e v s →
                          s2 = mk_State( ow_store s ⟨i, v⟩) none →
      Step (mk_State s (def i e)) s2

| comp: ∀s1, s2, c1, c2, c1'.
                           c1 ≠ none →
                           Step (mk_State s1 c1)(mk_State s2 c1') →

      Step (mk_State s1 (c1; c2)) (mk_State s2 (c1'; c2))

| if_then_else_true: ∀s, be, c_true, c_false.
                           Exp_Eval be (bool_value true) s →
      Step ( mk_State s (if_then_else be c_true c_false)) (mk_State s c_true)

| if_then_else_false: ∀s, be, c_true, c_false.
```

[3] funzione ricorsiva sullo store che trova il valore a cui la variabile fa riferimento

```
                              Exp_Eval be (bool_value false) s →
      Step ( mk_State s (if_then_else be c_true c_false)) (mk_State s c_false)

| while_true: ∀s, be, c_true.

                              Exp_Eval be (bool_value true) s →
      Step (mk_State s (while_do be c_true)) (mk_State s (c_true; (while_do be c_true)))

| while_false: ∀s, be, c_true.
                              Exp_Eval be (bool_value false) s →
      Step (mk_State s (while_do be c_true)) (mk_State s h none)
.
```

Prima di spiegare la semantica, occorre specificare le notazioni Matita specificate per la composizione `comp` di 2 comandi, per il comando `if_then` e per il comando `while_do`:

```
notation "c1; c2" right associative with precedence 50 for @{comp $c1 $c2}.
notation "'c_if' e 'c_then' c1 'c_else' c2 'endif'" non associative with precedence 19 for @{if_then_else $e $c1 $c2}
notation "'c_while' e 'c_do' c 'endwhile'" non associative with precedence 19 for @{while_do $e $c}.
```

In particolare, la composizione `c1; c2` viene definita associativa a destra. Ció significa che una composizione `c1; c2; c3; c4` viene trattata nel linguaggio, come `c1; (c2; c3; c4)` necessitando quindi una derivazione di tipo *leftmost*. Considerando inoltre, il comando `def` (assegnamento) ha transizione verso `none`, é necessario specificare la regola aggiuntiva `left_none`, oltre a quella della composizione. Per rendere la semantica deterministica, infine, occorre aggiungere la precondizione `c1` $\neq$ `none` nella composizione, evitando che, sia quest'ultima regola, sia la `left_none` possano essere applicate in caso di comando con struttura `none; c2`. Il resto dei comandi gode di una semantica standard, con il `while` che si esaurisce in `none` quando la sua guardia é falsa e in una composione corpo-ciclo quando viene verificata. L'assegnamento, attraverso la funzione definita tramite ricorsione in Matita `ow_store` muta lo store aggiungendo una nuova *entry* (ovvero coppia variabile-valore) se la variabile da aggiornare non é definita, oppure modificandone il valore giá associato. Oltre al predicato `Step` che descrive una singola transizione ($\hookrightarrow$) é stato definito `Computation`:

```
inductive Computation: State → State → Prop ≝
| zero: ∀s. Computation s s
| more: ∀s1, s2, s3.
                        Step s1 s2 → Computation s2 s3 →
      Computation s1 s3
.
```

Seguendo la terminologia descritta nel capitolo introduttivo, `Computation` rappresenta la chiusura transitiva $\hookrightarrow_\star$. Come immediata conseguenza delle definizioni, infatti, si ha che `Computation` gode della proprietá transitiva:

```
theorem comp_transitivity: ∀s1, s2, s3. Computation s1 s2 → Computation s2 s3 →
Computation s1 s3.
```

La dimostrazione della proprietá é per induzione sulla prima ipotesi. Esplicitata la semantica del linguaggio, appare immediato come una computazione sia finale se conduce in uno stato in cui il comando é `none`, quindi non é necessaria un'ulteriore definizione per $\hookrightarrow *_{\text{final}}$.

2.3.3 Note su testing

A titolo dimostrativo, solamente per quanto riguarda il linguaggio descritto nella presente sezione, illustreró lo schema di testing seguito durante tutto il progetto . Per prima cosa ho definito uno store di test[4]:

```
definition test_store ≝ ⟨0, nat_value 20⟩::⟨1, nat_value 30⟩::⟨2, nat_value 10⟩
                        ::⟨3, bool_value true⟩::⟨4, bool_value false⟩::⟨5, bool_value true⟩
                        ::⟨6, nat_value 20⟩::[].
```

2.3.3.1 Espressioni

Per ogni tipo di espressione consentito nel linguaggio, ne ho definita una di prova. In ultimo, ho dimostrato la validitá del predicato `Exp_Eval` applicato sullo store definito e sull'espressione di prova, con il valore atteso in cui essa deve essere valutata. Per esempio, si riporta il testing della valutazione di un'espressione `eq_b` (uguaglianza tra booleani):

```
definition test_bool_eq_variable ≝ eq_exp (value_of (var 3))(value_of (var 5)).
example testing_bool_eq_variable: Exp_Eval test_bool_eq_variable (bool_value true) test_store.
normalize @eq_b[||/2/|/2/]
// qed.
```

[4] nei linguaggi successivi lo store sará il record contenente heap/stack/thread a seconda del caso

2.3.3.2 Comandi

Per i comandi ho seguito un approccio simile, definendo un comando da eseguire e verificando l'esistenza di una computazione $\hookrightarrow_\star$ fino allo stato atteso.

```
definition test_if_true ≝ mk_State []
          (c_if (bool_exp true) c_then
                def (var 0) (nat_exp 1)
          c_else
                def (var 0) (nat_exp 2)
          endif).
example testing_if_true: Computation test_if_true (mk_State [][⟨var 0, nat_value 1⟩]) .
  @more normalize [@(mk_State [] (def (var 0) (nat_exp 1)))]/2/
  /6/
 qed.
```

2.3.4 Principali funzioni introdotte

```
let rec find_var_store l i on l : option Value ≝
match l with
[ nil ⟹ None ?
| cons p tl ⟹ if (eq_var_id (fst ? ? p ) i) then Some ? (snd ? ? p) else find_var_store tl i
].
let rec ow_store (l: list (Var_Id × Value)) u on l ≝ match l with
[ nil ⟹ []
| cons h t ⟹ if (eq_var_id (fst ? ? h ) (fst ? ? u ))
             then (u :: t)
             else  (h :: ow_store t u )
].
```

2.3.5 Principali dimostrazioni

```
theorem comp_transitivity: ∀s1, s2, s3. Computation s1 s2 → Computation s2 s3 →
                                        Computation s1 s3.
  #s1 #s2 #s3 #c1 elim c1 −c1 [ #si //] #s10 #s20 #s30 #step #comp #ih
  #comp2 @more
 [@s20 | @step −step] @(ih comp2)
qed.
```

2.4 linguaggio Heap

2.4.1 Implementazione celle di memoria

Il nuovo linguaggio consente l'allocazione e la deallocazione di dati in memoria Heap. Il concetto di "cella di memoria" viene implementato attraverso il tipo `Vector` di Matita. Nello specifico, viene usato un *sigma-tipo*, `DPair` in Matita, ovvero una coppia $\langle x, y \rangle$ dove $y = f(x)$. Usare il tipo `DPair` in Matita é necessario data la definizione di `Vector`:

```
record Vector (A:Type[0]) (n: ℕ): Type[0] ≝
{ vec :> list A;
len: length ? vec = n
}.
```

Che non consentirebbe di annotare il tipo `Vector Value` senza conoscere la lunghezza dello stesso. Si usa allora il concetto di *currying*[12], definendo come funzione $f(x)$ del sigma-tipo `Vector Value`. A questo punto l'annotazione di tipo per le celle di memoria sará `DPair ℕ (Vector Value)`, il quale consente di specificare un vettore di valori di lunghezza generica. A ogni modo, quando si definisce un vettore rimane obbligatorio fornire una prova della coerenza della sua lunghezza.

2.4.2 Sintassi

Espandendo il linguaggio precedente viene aggiunto il tipo "puntatore" ai valori consentiti:

```
...
| ptr_value: Address → Value
...
```

Dove `Address`, esattamente come `Var_Id`, incapsula i naturali:

```
inductive Address: Type[0] ≝
| addr:  ℕ → Address
.
```

Le espressioni comprendono ora anche

```
...
| deref_exp: Exp → Exp → Exp
...
```

E i comandi

```
...
| malloc: Var_Id → Exp → C
| free: Exp  → C
| deref_write: Exp  → Exp → Exp → C
...
```

2.4.3 Semantica

La memoria dinamica nel linguaggio é definita con il record

```
definition Heap_Data ≝ DPair ℕ (Vector Value).
record Heap: Type[0] ≝ {
  mem: list(Address × Heap_Data);
  first_free: Address
}.
```

Dove `Heap_Data` é l'annotazione di tipo per le celle di memoria discussa precedentemente. `first_free` indica il primo indirizzo di memoria libero. Ogniqualvolta una cella di memoria viene allocata, `first_free` viene incrementato. La "compattazione" della memoria non viene gestita nel linguaggio, perció se un indirizzo precedente a `first_free` viene liberato, esso non potrá essere piú riallocato.
Il tipo `State` include ora anche l'heap:

```
record State: Type[0] ≝ {
  store: list (ℕ × Value);
  heap: Heap;
  cmd: C
}.
```

Il predicato `Exp_Eval`, per consentire la corretta valutazione delle espressioni `deref_read`, dipende, nel nuovo linguaggio, anche dallo Heap, quindi il suo tipo diventa:

```
Exp → Value → list (ℕ × Value) → list(Address × Heap_Data) → Prop
```

L'espressione `deref_read` viene valutata secondo la regola:

```
...
| deref_read: ∀s, m, addr_exp, a, index_exp, i, found, res.

    Exp_Eval addr_exp (ptr_value a) s m →
    Exp_Eval index_exp (nat_value i) s m →
    Some ? found = mem_search m a →
    res = (nth i ? (dpi2 ? ? found) (nat_value 0))  →

  Exp_Eval (deref_exp addr_exp index_exp) res s m
...
```

Notare come, similmente al caso discusso nel precedente linguaggio, il predicato non possa essere valutato quando l'indirizzo usato nell'espressione non punti ad alcuna cella di memoria. La funzione `nth i`, assume in input un indice e un `Vector`[5] e ritorna il valore dello stesso all'indice. La funzione, inoltre, prende un ulteriore parametro: un valore di default in caso l'indice sia maggiore della lunghezza del vettore. In virtú di tale valore di default, il predicato sará eseguito nel caso l'indice non sia valido nella cella di memoria trovata.
Per quanto concerne la semantica dei comandi, vengono aggiunte le regole:

```
...
| malloc: ∀s, h1, h2, e, i, n.
                Exp_Eval e (nat_value n) s  (mem h1) →
                h2 = allocate h1 n →
      Step ( mk_State s h1 (malloc i e) ) (mk_State s h2 (def i (ptr_exp (last_allocated h2))))
| free: ∀s, h1, h2, e, a.
                Exp_Eval e (ptr_value a) s (mem h1) →
                Some ? h2 = deallocate h1 a →
      Step (mk_State s h1 (free e)) (mk_State s h2 none)

| write_mem: ∀s, h1, h2, addr_exp, index_exp, value_exp, a, i, v .

                Exp_Eval addr_exp (ptr_value a) s  (mem h1) →
                Exp_Eval value_exp v s (mem h1) →
                Exp_Eval index_exp (nat_value i) s (mem h1) →
                h2 = heap_write h1 a i v  →
      Step (mk_State s h1 (deref_write addr_exp index_exp value_exp)) (mk_State s h2 none)
...
```

Basate sulle funzioni: `allocate`, `last_allocated`, `deallocate`, `heap_write`. Tra queste, l'unica che potrebbe portare a un fallimento é la deallocate. La deallocate, come é possibile notare nella parte dedicata alle funzioni, ritorna un Heap solamente se la funzione free viene effettuata con successo, ovvero quando l'indirizzo da deallocare é effettivamente presente in memoria. Ció servirá a impedire il caso assurdo menzionato alla sezione 1.3.3.

[5] in realtá prende una lista, ma il record `Vector` ha una coercion a `list`

2.4.4 Principali funzioni introdotte

```
definition allocate ≝ λh.λn.
  mk_Heap
  (⟨first_free h, (mk_DPair ? ? n (new_vector ? (nat_value 0) n))⟩::(mem h))
  (addr (match first_free h with [addr a ⟹ a ] + 1))

definition last_allocated ≝ λh.
  match mem h with
  [ nil ⟹ addr 0
  | cons hd tl ⟹ fst ? ? hd
  ]
let rec free (T: Type[0]) (mem: list(Address×T)) a on mem ≝
  match mem with
  [ nil ⟹ ⟨[], false⟩
  | cons h t ⟹
      if (eq_addr (fst ? ? h) a) then ⟨t, true⟩
      else let next ≝ free T t a in
      ⟨h::(fst ? ? next), snd ? ? next⟩
.
definition deallocate ≝ λh.λa.
  let freed ≝ (free ? (mem h) a ) in
  if (snd ? ? freed) then Some ? (mk_Heap (fst ? ? freed) (first_free h))
  else None ?
let rec mem_search (mem: list(Address×Heap_Data)) (a: Address) on mem ≝
  match mem with
  [ nil ⟹ None ?
  | cons h t ⟹ if (eq_addr (fst ? ? h) a) then Some ? ( snd ? ? h)
               else mem_search t a
let rec  mem_write (mem: list(Address×Heap_Data)) (a: Address) v i on mem ≝
  [ nil ⟹ mem
  | cons h t ⟹ let vector ≝ (dpi2 ? ? (snd ? ? h)) in
               if (eq_addr (fst ? ? h) a)
               then
                ((a, mk_DPair ? ? ? (replace_at_vector ? ? vector i v))::t)
               else (h::mem_write t a v i)
.
definition heap_write ≝ λh.λa.λi.λv.
  mk_Heap (mem_write (mem h) a v i) (first_free h)
let rec replace_at_list (A: Type[0]) (l: list A) (i: ℕ) (v: A) on l ≝
  match l with
  [ nil ⟹ [ ]
  | cons h t ⟹ match i with
    [ O ⟹ v::t
    | S n ⟹ h::replace_at_list A t n v
    ]
  ]
.
definition replace_at_vector ≝ λA: Type[0].λn: ℕ.λvector: (Vector A n).λi.λv.
 mk_Vector A n (replace_at_list A (vec A n vector) i v ) ?
./2/qed.
```

2.4.5 Principali dimostrazioni

```
lemma new_list_len: ∀n, A, v. (|make_list A v n|) = n.
#n#A#v elim n [ normalize // | #N#I normalize >I // .
qed.
lemma replace_list_len: ∀A, v, l, i. |(replace_at_list A l i v)| = |(l)|.
#A #v #l
elim l -l normalize
// #h #t #ih
#i2 cases i2
normalize //
qed.
```

2.5 linguaggio con Stack

2.5.1 Descrizione introduttiva

Il linguaggio descritto nella presente sezione é, tra tutti, quello che ha richiesto piú lavoro e che ha subito il maggior numero di ristrutturazioni. Il fondamento del linguaggio é lo stack, una pila di record di attivazione, contenenti uno store locale per le funzioni e uno per le variabili, mentre l'heap rimane unico e condiviso tra tutti i record. La struttura a pila garantisce uno *scope statico*[13] nel linguaggio di programmazione, pertanto una ricerca nello stack coinvolge, nel caso pessimo, TUTTI i record di attivazione allocati, sia per quanto riguarda una chiamata di funzione sia per quanto riguarda una lettura o un assegnamento di variabile. Le funzioni NON sono valori del linguaggio, quindi non possono essere passate ad altre funzioni come argomento e non possono essere valori di ritorno da un'altra funzione. Tuttavia, in quanto é presente il concetto di store locale per le funzioni, queste possono essere definite e ridefinite annidandole all'interno di altre funzioni. Come limitazione, non possono coesistere all'interno della stessa procedura 2 comandi di ritorno differenti. Ció si é reso necessario per mantenere, nel linguaggio, l'invariante che solamente la regola di composizione `c1; c2` possa agire su uno dei 2 comandi (il primo) separatamente dall'altro (tolto il caso `none`). In una prima stesura del linguaggio avevo infatti permesso piú `return` da una sola funzione. Allora, era necessario salvare nel record di attivazione il comando da cui ripartire una volta effettuato il ritorno dalla procedura. Ma ció implicava gestire il comando di fun_call esclusivamente come composizione, in quanto, gestendolo singolarmente, (`f_call f args`) non era possibile conoscere il comando successivo ad esso, mentre sarebbe stato possibile nel caso (`f_call f args; c`). Come ulteriore limitazione, il valore ritornato da una funzione viene assegnato a una variabile fissa (RETVAR), di sola lettura, mentre in una prima stesura la variabile era scelta dall'utente al

momento della chiamata di funzione. [6] Il motivo di quest'ultima limitazione ha a che fare con la gestione dei predicati Hoare di cui si discuterá piú avanti nel documento.

2.5.2 Sintassi

Per gestire i nomi delle funzioni é stato, come negli altri linguaggi, introdotto un nuovo tipo induttivo che incapsula i naturali:

```
inductive Fun_Id: Type[0] ≝
| fun: ℕ → Fun_Id
.
```

I comandi introdotti nel linguaggio sono i seguenti:

```
...
| return_cmd: Exp → C
| def_f: Fun_Id → DPair ℕ (Vector Var_Id) → C → C (*<funzione><argomenti><corpo>*)
| fun_call: Fun_Id → list Exp → C (*<funzione>, <parametri>*)
...
```

(Come per le celle di memoria, anche per gli argomenti sono stati usati i tipi dipendenti)

2.6 Semantica

Come prima cosa viene definito il tipo `Function`:

```
record Function: Type[0] ≝ {
fun_id: Fun_Id;
args: DPair ℕ (Vector Var_Id);
body: C;
one_return: returns_count body = 1 ∧ return_last body ∧ ¬while_if_return body
}.
```

Oltre necessitare a di un id, di un vettore di argomenti e di un corpo, il tipo necessita di una dimostrazione della validitá di quest'ultimo, ovvero della univocitá del suo return.
Viene poi definito il tipo `Activation_Record`:

```
record Activation_Record: Type[0] ≝ {
var_store: list (Var_Id × Value);
fun_store: list Function
}.
```

[6] questo comportamento impatterá anche sulla concorrenza

E in ultimo `State` viene modificato in:

```
record State: Type[0] ≝ {
  stack: list Activation_Record;
  heap: Heap;
  cmd: C
}.
```

Il predicato `Exps_Eval` necessita ora di valutare le variabili non in un singolo store, ma in tutta la pila, pertanto il suo tipo diventa:

```
Exp → Value → list Activation_Record → list(Address × Heap_Data) → Prop
```

e la funzione con cui cercare le variabili: `stack_search`.

Per valutare i parametri attuali quando viene eseguito `fun_call` é necessario il predicato `Exps_Evals`, che utilizza `Exps_Eval` per effettuare tale valutazione, costruendo, inoltre, sulla base del nome dei parametri formali specificati nella funzione, lo store da utilizzare nel nuovo record della pila:

```
inductive Exps_Evals: list Exp → list Value → list Var_Id → list (Var_Id × Value)
    → list Activation_Record → list (Address × option Heap_Data) → Prop ≝
| recursive: ∀exps_h, exps_t, vals_h, vals_t, vars_h, vars_t, store_h, store_t, s, h.
                Exp_Eval exps_h vals_h s h →
                ⟨vars_h, vals_h⟩ = store_h →
                Exps_Evals exps_t vals_t vars_t store_t s h  →
Exps_Evals (exps_h :: exps_t) (vals_h :: vals_t) (vars_h :: vars_t) (store_h :: store_t) s h

| base: ∀s, h. Exps_Evals [] [] [] [] s h
.
```

La semantica dei nuovi comandi nella step viene descritta dalle regole:

```
...
| return_cmd: ∀s_head, s_tail, h, e, v, s2.
        Exp_Eval e v (s_head :: s_tail) (mem h) →
        s2 = update_var_stack s_tail ⟨RETVAR, v⟩ →
    Step (mk_State (s_head :: s_tail) h (return_cmd e)) (mk_State s2 h none)

| def_f: ∀s, h, f_id, args, c, s2.
        (returns_count c = 1) ∧ (return_last c ) ∧ ¬while_if_return c →
        s2 = (mk_Activation_Record
                    (current_var_store s)
                    ((mk_Function f_id args c ?)::(current_fun_store s))
                  )::(tail ? s) →
    Step (mk_State s h (def_f f_id args c))(mk_State s2 h none)

| fun_call: ∀f_id, f, f_state, e_params, v_params, created_var_store, s, h.
        Some ? f = find_fun_stack s f_id →
        Exps_Evals e_params v_params (vec ? ? (dpi2 ? ? (args f))) created_var_store s (mem h) →
        f_state = mk_State ((mk_Activation_Record created_var_store [ ]) :: s) h (body  f) →
    Step (mk_State s h (fun_call f_id e_params)) f_state
...
```

Anche in questo caso, una chiamata di funzione dove la funzione richiesta non é definita nello stack comporta la non validitá del predicato. Notare inoltre come la regola per la definizione di una nuova funzione abbia come ipotesi la fondatezza del suo comando/corpo, al fine di poter costruire una `Function` usando tale ipotesi. La parte significativa dell'implementazione della semantica é nelle funzioni descritte alla pagina successiva. In particolare, la sovrascrittura di una variabile nello stack necessita di riconoscere a quale livello della pila fermarsi nella ricerca, ovvero in quale record di attivazione, partendo da quello attuale, é definita la variabile da cercare e sovrascrivere. Si fa notare come l'implementazione scelta non sia quella computazionalmente piú vantaggiosa, visto che `update_var_stack` scorre la pila 2 volte, nel caso in cui la variabile é definita in qualche record.

2.6.1 Principali funzioni introdotte

```
let rec find_var_store l i on l : option Value ≝
match l with
[ nil ⟹ None ?
| cons p tl ⟹ if (eq_var_id (fst ? ? p ) i) then Some ? (snd ? ? p) else find_var_store tl i
].
let rec stack_search (stack: list Activation_Record) i on stack ≝ match stack with
[ nil ⟹ None ?
| cons ar t ⟹ match find_var_store (var_store ar) i with
              [ None ⟹ stack_search t i
              | Some found ⟹ Some ? found
              ]
]
.
let rec ow_store (l: list (Var_Id × Value)) u on l ≝ match l with
[ nil ⟹ ⟨[], false⟩
| cons h t ⟹ if (eq_var_id (fst ? ? h ) (fst ? ? u ))
            then ⟨(u :: t), true⟩
            else let next ≝ ow_store t u in ⟨h :: (fst ? ? next), snd ? ? next⟩
].
let rec ow_stack (stack: list Activation_Record) v on stack ≝ match stack with
[ nil ⟹ []
| cons ar t ⟹ let updated ≝ ow_store (var_store ar) v in
              if snd ? ? updated then
                (mk_Activation_Record (fst ? ? updated)(fun_store ar)) :: t
              else ar :: (ow_stack t v)
].
definition current_fun_store ≝ λ(stack: list Activation_Record ).
   fun_store(hd ? stack (mk_Activation_Record [ ] [ ] ))
.
definition current_var_store ≝ λ(stack: list Activation_Record ).
   var_store(hd ? stack (mk_Activation_Record [ ] [ ] ))
.
definition update_var_stack ≝ λs.λv. match (stack_search s (fst ? ? v) ) with
[ None ⟹ (mk_Activation_Record (v::(current_var_store s)) (current_fun_store s) )::(tail ? s)
| Some d ⟹ ow_stack s v

]
.
let rec find_fun_store l f_id on l ≝
  match l with
  [ nil ⟹ None ?
  | cons h t ⟹ if (eq_fun_id (fun_id h) f_id) then Some ? h else find_fun_store t f_id
  ]
.
let rec find_fun_stack stack f_id on stack ≝ match stack with
[ nil ⟹ None ?
| cons ar t ⟹ match find_fun_store (fun_store ar) f_id with
              [ None ⟹ find_fun_stack t f_id
              | Some found ⟹ Some ? found
              ]
]
.
let rec find_fun_stack_list stack f_ids on f_ids ≝ match f_ids with
[ nil ⟹ []
| cons f_id t ⟹ match (find_fun_stack stack f_id) with
  [ None ⟹ find_fun_stack_list stack t
  | Some f ⟹ f::(find_fun_stack_list stack t)
  ]
]
```

```
let rec returns_count c on c ≝ match c with
[ comp c1 c2 ⟹ returns_count c1 + returns_count c2
| return_cmd e ⟹ 1
| _ ⟹ 0
].

let rec while_if_return c on c ≝ match c with
[ comp c1 c2 ⟹ while_if_return c1 ∨ while_if_return c2
| while_do e c ⟹ returns_count c > 0
| if_then_else e c1 c2 ⟹ returns_count c1 + returns_count c2 > 0
| _ ⟹ False
].


let rec return_last c on c ≝ match c with
[ comp c1 c2 ⟹ ∃e. c2 = return_cmd e
| return_cmd e ⟹ True
| _ ⟹ False
].
.
```

2.7 Linguaggio concorrente

2.7.1 Descrizione introduttiva

Nel linguaggio concorrente scelto si ha una condivisione dell'heap tra i vari thread, che mantengono invece una copia dello stack locale. Un thread é eseguito associandolo a una funzione, con istanziazione dei parametri attuali esattamente come avviene per un comando `fun_call`. Diversi thread possono partire in contemporanea, venendo associati a diverse funzioni (e argomenti). Quando un thread viene lanciato, questo utilizza una copia dello stack del thread padre, aggiungendo, come ultimo record di attivazione, lo store instanziato con i parametri appropriati. Data la proprietá del linguaggio con stack di permettere la definizione di funzioni annidate, é possibile che un thread definisca nuove funzioni locali e crei a sua volta altri thread usanto tali funzioni. Nell'implementazione attuale, tutti i thread (compresi thread padri-figli) possono scambiare informazioni esclusivamente utilizzando l'heap. Come primitive di concorrenza, per gestire la sincronizzazione, il linguaggio adotta `lock` e `unlock`, comandi che verranno descritti nella parte relativa alla semantica. Le caratteristiche finali del linguaggio sono

1. atomicitá nella valutazione espressioni
2. atomicitá nell'esecuzione comandi della sintassi

2.7.2 Sintassi

Per gestire i lock si usa, ancora una volta, un tipo induttivo che incapsula i naturali:

```
inductive Lock_Id: Type[0] ≝
| lock: ℕ → Lock_Id
.
```

Gli unici comandi aggiunti sono:

```
...
| spawn: list Fun_Id → list (list Exp) → C
| lock_cmd: Lock_Id → C
```

```
| unlock_cmd: Lock_Id → C
...
```

2.7.3 Semantica

Come menzionato nell'introduzione alla sezione, un thread, localmente, possiede solamente uno stack, oltre che il comando:

```
record Thread: Type[0] ≝ {
  t_stack: list Activation_Record;
  t_cmd: C
}.
```

Il tipo `State` rimane identico a quello descritto nel linguaggio precedente. Viene tuttavia introdotto un nuovo stato "globale", comprendente tutti i thread in esecuzione e la memoria heap tra essi condivisa.

```
record Global_State: Type[0] ≝ {
g_heap: Heap;
threads: list Thread;
locks: list (Lock_Id × bool) (*coppie ⟨id, free⟩*)
}.
```

Il predicato `Step` NON gestisce in alcun modo le primitive di concorrenza. Cosí facendo lo `State` utilizzato nei linguaggi precedenti descrive regole d'inferenza valide localmente a un thread, permettendo di ragionare sullo stato globale del programma in maniera separata. La semantica attribuiti ai nuovi comandi, nel predicato `Step` descrive semplicemente una transizione a none:

```
...
| spawn: ∀s, h, spawn_funs_ids, spawn_args.
    Step (mk_State s h (spawn spawn_funs_ids spawn_args )) (mk_State s h none)

| lock_unlock: ∀s, h, lock_id, c.
       c = lock_cmd lock_id ∨ c = unlock_cmd lock_id →
    Step (mk_State s h c) (mk_State s h none)
...
```

Per valutare correttamente il tipo `list (list Exp)` é necessario "liftare" ancora una volta il predicato `Exp_Eval`:

```
inductive Exps_Evals_List: list (list Exp) → list (list Value) → list (list Var_Id) →
| recursive: ∀exps_h, exps_t, vals_h, vals_t, vars_h, vars_t, stores_h, stores_t, s, h.
                        Exps_Evals exps_h vals_h vars_h stores_h s h →
                        Exps_Evals_List exps_t vals_t vars_t stores_t s h  →
      Exps_Evals_List (exps_h :: exps_t) (vals_h :: vals_t) (vars_h :: vars_t) (stores_h :: stores_t) s h

| base: ∀s, h. Exps_Evals_List [ ] [ ] [ ] [ ] s h
.
```

Lo stato globale viene modificato secondo le regole d'inferenza descritte nel nuovo predicato induttivo `Tick_Nth`:

```
inductive Tick_Nth: Global_State → ℕ → Global_State → Prop ≝

| tick_step: ∀s1, n, s2, chosen, thread_state, other_threads.

⟨Some ? chosen, other_threads⟩ = find_nth_and_remove ? (threads s1) n [ ] →
spawn_unfold (t_cmd chosen) = None ? →
lock_unfold (t_cmd chosen) = None ? →
t_cmd chosen ≠ none →
Step (mk_State (t_stack chosen)(g_heap s1)(t_cmd chosen)) thread_state →
s2 = mk_Global_State (heap thread_state)
((mk_Thread (stack thread_state) (cmd thread_state)) :: other_threads)
(locks s1) →

        Tick_Nth s1 n s2

| tick_lock: ∀s1, n, s2, chosen, thread_state, other_threads, lock_id,
        maybe_lock, lock_list_no_lock, found_lock, new_cmd, new_lock_list .

⟨Some ? chosen, other_threads⟩ = find_nth_and_remove ? (threads s1) n [ ] →
Step (mk_State (t_stack chosen)(g_heap s1)(t_cmd chosen)) thread_state →
lock_unfold (t_cmd chosen) = Some ? lock_id →
⟨maybe_lock, lock_list_no_lock⟩ = find_lock_and_remove (locks s1) lock_id [ ] →
found_lock = match maybe_lock with [None ⟹ N ⟨lock_id, true⟩| Some p ⟹ N p] →
⟨new_lock_list, new_cmd⟩ = lock found_lock lock_list_no_lock (t_cmd chosen) (cmd thread_state) →
s2 = mk_Global_State (heap thread_state)
((mk_Thread (stack thread_state) new_cmd) :: other_threads)
new_lock_list →

        Tick_Nth s1 n s2

| tick_unlock: ∀s1, n, s2, chosen, thread_state, other_threads, lock_id,
        lock_list_no_lock, found_lock.

⟨Some ? chosen, other_threads⟩ = find_nth_and_remove ? (threads s1) n [ ] →
unlock_unfold (t_cmd chosen) = Some ? lock_id →
Step (mk_State (t_stack chosen)(g_heap s1)(t_cmd chosen)) thread_state →
⟨Some ? found_lock, lock_list_no_lock⟩ = find_lock_and_remove (locks s1) lock_id [ ] →
s2 = mk_Global_State (heap thread_state)
((mk_Thread (stack thread_state) (cmd thread_state)) :: other_threads)
((⟨fst ?? found_lock, true⟩::lock_list_no_lock) →

   Tick_Nth s1 n s2

| tick_spawn: ∀s1, n, s2, chosen, thread_state, other_threads,
        spawn_funs_ids, spawn_params_exps, spawn_funs, spawn_params_values, spawn_stores,
        threads2, spawned_threads.


        ⟨Some ? chosen, other_threads⟩ = find_nth_and_remove Thread (threads s1) n [ ] →
        Step (mk_State (t_stack chosen)(g_heap s1)(t_cmd chosen)) thread_state →

        spawn_unfold (t_cmd chosen) = Some ? ⟨spawn_funs_ids, spawn_params_exps⟩ →

        spawn_funs = find_fun_stack_list (t_stack chosen) spawn_funs_ids →
        Exps_Evals_List spawn_params_exps
                                        spawn_params_values
                                        (funs_to_args_lists spawn_funs)
                                        spawn_stores
                                        (t_stack chosen)
                                          (mem(g_heap s1)) →
```

```
        Spawning
                          (funs_to_cmds_list spawn_funs)
                           spawn_stores
                           spawned_threads
                           (t_stack chosen) →


        threads2 = spawned_threads @((mk_Thread (stack thread_state) (cmd thread_state)) :: other_threads) →
        s2 = mk_Global_State (heap thread_state) threads2 (locks s1) →
    Tick_Nth s1 n s2

| remove: ∀s1, n, s2, chosen, other_threads.
        ⟨Some ? chosen, other_threads⟩ = find_nth_and_remove ? (threads s1) n [ ] →
        t_cmd chosen = none →
        s2 = mk_Global_State (g_heap s1) other_threads (locks s1) →
    Tick_Nth s1 n s2
| stall: ∀s, n.
    (|threads s|) = 0 →
    Tick_Nth s n s
.
```

Nel predicato:

1. si estrae dalla lista un thread
2. si controlla il comando che esso deve eseguire
3. in caso non sia un comando relativo alla concorrenza e non sia un `none` lo si "esegue" con `Step` e si reinserisce il thread estratto nella lista dei thread globali(con stack e comando aggiornati) modificando l'heap nel successivo stato globale raggiunto dal thread
4. in caso sia un comando `lock i`
   (a) si controlla nella lista dei lock globali se `i` é libero
   (b) se lo é si sovrascrive il lock (impostandolo a false)
   (c) altrimenti si imposta il comando successivo del thread a `lock i`
   (d) si reinserisce il thread nella lista dei thread in esecuzione
5. in caso sia un comando `unlock i`
   (a) se esiste `i` nella lista di lock lo si imposta a true, altrimenti lo si crea ex-novo
   (b) si reinserisce il thread nella lista dei thread in esecuzione

6. in caso sia un comando `spawn`

   (a) si utilizza `Spawning` in combinazione a `Exps_Evals_Lists` per ottenere una lista con tutti i nuovi thread creati

   (b) si reinserisce il thread nella lista dei thread in esecuzione, concatenandola alla nuova lista ottenuta al punto precedente

7. in caso sia un comando `none` si avanza senza reinserire il thread in lista (esso non ha piú comandi da eseguire)

Il predicato `Spawning`, che si occupa di "creare" la lista con i nuovi thread, allocandone lo store con la valutazione dei parametri é:

```
inductive Spawning:  list C → list (list (Var_Id × Value)) → list Thread → list Activation_Record →
 Prop ≝
| recursive: ∀cmds_h, cmd_t, stack, spawned_h, spawned_t,
              new_var_stores_h, new_var_stores_t, new_ar.

                            Spawning cmd_t new_var_stores_t spawned_t stack →
                            new_ar = mk_Activation_Record new_var_stores_h [ ] →
                            spawned_h = mk_Thread (new_ar :: stack) cmds_h →

     Spawning (cmds_h::cmd_t)(new_var_stores_h::new_var_stores_t)(spawned_h::spawned_t) stack

| base: ∀s. Spawning [ ][ ][ ] s
.
```

Avendo definito un predicato che consente di eseguire uno specifico thread, é sufficiente definire:

```
inductive Random_Tick: Global_State → Global_State → Prop ≝
|tick:  ∀s1, s2, chosen_pos.
              chosen_pos < |(threads s1)|→
              Tick_Nth s1 chosen_pos s2 →

      Random_Tick s1 s2
.


inductive Parallel_Computation_Random: Global_State → Global_State → Prop ≝
|zero: ∀s. Parallel_Computation_Random s s

|more: ∀s1, s2, s3.
              Random_Tick s1 s2 →
              Parallel_Computation_Random s2 s3 →
      Parallel_Computation_Random s1 s3
.
```

per ottenere un linguaggio che simuli, attraverso l'interleaving dei thread, un linguaggio concorrente non deterministico. Il predicato `Random_Tick` puó infatti essere verificato a partire da stati globali diversi, visto che la sua

condizione `chosen_pos < |(threads s1)|` é soddisfatta da piú di un naturale.
[7]

[7]in questo contesto sono stati utilizzati predicati induttivi per ottenere risultati (per esempio la lista dei nuovi thread) costruendoli induttivamente, similmente a come si usano le funzioni ricorsive

Capitolo 3

Implementazione Logica di Hoare

3.0.1 Dimostrazioni in Matita

Nelle pagine precedenti sono state accennate alcune dimostrazioni svolte in Matita. Prima di descrivere come siano state definite le triple di Hoare, e come si sia dimostrata la correttezza delle regole d'inferenza relative, occorre descrivere piú dettagliatamente come vengano effettuate le dimostrazioni in Matita.
Matita implementa la logica intuizionista [14], quindi NON valgono *terzo escluso e riduzione all'assurdo*. Il principale mezzo con cui si conducono le dimostrazioni é dunque l'analisi per casi, sia sulle ipotesi sia sulle conclusioni, in combinazione con l'uso appropriato dell'induzione strutturale [15]. Matita, come altri *Proof Assistant* mette a disposizione *tattiche* per poter introdurre o applicare ipotesi, effettuare analisi per casi, applicare l'induzione e in generale, completare le dimostrazioni. La difficoltá nelle dimostrazioni sta nel trovare le proprietá (ovvero predicati in logica predicativa) "invarianti", che mantengono la loro validitá applicando analisi per casi e induzione. Inoltre Matita é in grado di terminare prove applicando ipotesi ed enunciati dimostrati precedentemente attraverso automazione, indicando fino a che profonditá arrivare nell'albero di

dimostrazione. Nel presente documento sono allegate le dimostrazioni principali e discusse quelle che hanno richiesto un maggior sforzo per essere ultimate.

3.0.2 Definizioni predicati e triple di Hoare

Come menzionato nella prima sezione del presente documento, la nozione di *Predicato Hoare* é riconducibile a quella di predicato con variabile libera di tipo stato. In prima battuta, allora, si potrebbe pensare di definire la stessa nozione su Matita con l'annotazione di tipo `State` $\rightarrow$ `Prop`. Tuttavia un tipo del genere consentirebbe ai predicati di "osservare" i comandi, che fanno parte del tipo record `State`. Allora una possibile intuizione potrebbe essere quella di creare un ulteriore tipo, dedicato solamente ai predicati Hoare, esclusivamente considerando stack e memoria heap. Nella prima implementazione del progetto veniva seguita tale strada. La problematica che, tuttavia, sopraggiunge con tale definizione riguarda l'**ordine delle variabili e delle celle memoria**. I predicati cosí descritti, osservano, collateralmente, data l'implementazione mediante lista delle strutture di memoria, informazioni irrilevanti e controproducenti, che rendono estremamente tediose alcune dimostrazioni. Si consideri per esempio il predicato $P * Q$. Esso afferma che esistono 2 partizioni di memoria le quali soddisfano separatamente P e Q. Pur definendo in Matita la nozione di "partizione", attraverso funzioni sulla lista dell'heap, nulla vieta ai predicati, comunque, di osservare l'ordine delle allocazioni.
L'implementazione finale risolve la problematica definendo un predicato Hoare come predicato che ha come variabili libere funzioni di ricerca:

```
record Hoare_Lookup: Type[0] ≝ {
l_vars: (Var_Id → option Value);
l_funs: (Fun_Id → option Function);
l_mem: (Address → option Heap_Data)
}.
definition Hoare_P_Local ≝ Hoare_Lookup → Prop.
notation > "L ⊩ P" non associative with precedence 40 for @{$P $L}.
```

Ancora una volta si utilizza il *currying* per ottenere il tipo compatibile dal record `State`. Usando `S: State` si puó costruire univocamente il tipo

`Hoare_Lookup` corrispondente usando le funzioni di ricerca definite nei linguaggi precedenti:

```
 mk_Hoare_Lookup
(stack_search (stack S))
(find_fun_stack (stack S))
(mem_search (mem (heap S))) .
```

Viceversa, si puó costruire un qualsiasi predicato Hoare con le proiezioni delle funzioni di ricerca[1]. Per esempio "la variabile x (var 1) é ≥ 10" é definibile come `P:=` $\lambda L.$ `(l_vars L)(var 1)` $\geq$ `10`
Tornerá utile "espandere" un predicato in funzione `Hoare_Lookup` $\rightarrow$ `Prop` per usare i connettivi della logica predicativa ($\Longrightarrow$, $\wedge$ ...) tra predicati Hoare differenti. Consideriamo di dover connettere con una congiunzione logica il predicato precedente a `Q:=` $\lambda L.$ `(l_vars L)(var 1)` $\leq$ `30`. `(P` $\wedge$ `Q)` viene soddisfatto sse lo stato verifica entrambe le condizioni. Per definire ció in Matita espandiamo la definizione di `Hoare_P_Local` e usiamo il connettivo all'interno della funzione, definendo:
`R:=` $\lambda L.\ L \Vdash P \wedge L \Vdash Q.$

Applicando queste definizioni, il predicato `Exp_Eval` (con i relativi sulle liste) cambia ancora una volta tipo in:

```
Exp → Value → (Var_Id → option Value) → (Address → option Heap_Data) → Prop
```

Come ultimo punto sui predicati occorre definire **l'unico** assioma necessario per proseguire il lavoro:

```
axiom predicates_equality: ∀P:Hoare_P_Local. ∀s1, s2, f1, f2, h1, h2.
             mk_Hoare_Lookup s1 f1 h1 ⊩ P →
             ∀i. s1 i = s2 i →
             ∀i. f1 i = f2 i →
             ∀i. h1 i = h2 i →
     mk_Hoare_Lookup s2 f2 h2 ⊩ P
.
```

[1] non é possibile ovviamente costruire stack e heap completi partendo dalle loro funzioni di ricerca

Esso applica il concetto di *function extensionality*[2] ai predicati: se un predicato soddisfa uno stato S1, allora ogni stato S2 le cui funzioni di ricerca sono "estensionalmente" uguali a quelle di S1, é anch'esso soddisfatto dal predicato.

A questo punto si puó fornire la definizione di *tripla di Hoare*:

```
definition is_valid_triple_local ≝ λP:Hoare_P_Local.λc.λQ:Hoare_P_Local. ∀s1, h1, s2, h2.
          Computation (mk_State s1 h1 c) (mk_State s2 h2 none) →
          mk_Hoare_Lookup (stack_search s1)(find_fun_stack s1)(mem_search(mem h1))⊩ P →
  mk_Hoare_Lookup (stack_search s2)(find_fun_stack s2)(mem_search(mem h2))⊩ Q .

notation "{ P } c { Q }" non associative with precedence 45 for @{is_valid_triple_local $P $c $Q}.
```

che risulta coerente con quanto descritto nel capitolo introduttivo, data la natura di stato finale di ⟨`s2, h2, none`⟩.

[2] 2 funzioni sono considerate uguali se, eseguendole sui medesimi argomenti restituiscono lo stesso risultato

3.1 Regole d'inferenza linguaggio while

Le regole per i comandi relativi al primo linguaggio sono le stesse di quelle trattate nel capitolo introduttivo:

CONSEQUENCE

$$\frac{Ps \implies Pw \qquad Qs \implies Qw \qquad \{\text{Pw}\}\texttt{c}\{\text{Qs}\}}{\{\text{Ps}\}\texttt{c}\{\text{Qw}\}}$$

NONE

$$\frac{}{\{P\}\ \texttt{none}\ \{P\}}$$

IF

$$\frac{\{P \wedge e \hookrightarrow_b \text{true}\}\texttt{c1}\{\text{Q}\} \qquad \{P \wedge e \hookrightarrow_b \text{false}\}\texttt{c2}\{\text{Q}\}}{\{P\}\ \texttt{if e then c1 else c2 endif}\ \{Q\}}$$

ASSEGNAMENTO

$$\frac{}{\{P[e/x]\}\ \texttt{x := e}\ \{P\}}$$

COMPOSIZIONE

$$\frac{\{\text{P}\}\texttt{c1}\{\text{Q}\} \qquad \{\text{Q}\}\texttt{c2}\{\text{R}\}}{\{P\}\ \texttt{c1; c2}\ \{R\}}$$

WHILE

$$\frac{\{P \wedge e \hookrightarrow_b \text{true}\}\texttt{c}\{\text{P}\}}{\{P\}\ \texttt{while e do c endwhile}\ \{P \wedge e \hookrightarrow_b false\ \}}$$

3.1.0.1 Consequence, none, if e assegnamento

Le prime 3 regole vengono enunciate in Matita come:

```
theorem consequence_rule: ∀Ps, Pw, Qs, Qw, c.
                          (∀L. L ⊩ Ps → L ⊩ Pw) →
                          (∀L. L ⊩ Qs → L ⊩ Qw)→
                           { Pw }  c  { Qs }  →
       { Ps }  c  { Qw }
.
theorem valid_none: ∀P. { P }  none  { P } .

theorem valid_if_then_else: ∀P, Q, b_exp, c_true, c_false.
    { (λl. Exp_Eval b_exp (bool_value true)(l_vars l)(l_mem l) ∧ (P l) ) } c_true  { Q }  →
    { (λl. Exp_Eval b_exp (bool_value false)(l_vars l)(l_mem l)∧ (P l)) } c_false { Q }  →
    { P }  (c_if b_exp c_then c_true c_else c_false endif)  { Q } .
```

Le dimostrazioni sono abbastanza lineari, per quella riguardante la `none` é tuttavia necessario dimostrare un lemma che tornerá spesso utile, diretta conseguenza della definizione del caso base in `Computation`:

```
lemma none_next_Computation: ∀s1, s2. cmd s1 = none → Computation s1 s2 → s1 = s2.
```

Il caso dell'assegnamento é interessante e meno immediato. Occorre in primis definire come si faccia a valutare un predicato sostituendo in valore della variabile con quello dell'espressione:

```
definition hoare_predicate_sub: Hoare_P_Local → Var_Id × Value → Hoare_P_Local ≝ λP.λv.λl.
  let wrap ≝ λval. match ((l_vars l) val) with
    [ None ⟹ if eq_var_id (fst ? ? v) val then Some ? (snd ? ? v) else None ?
    | Some d ⟹ if eq_var_id (fst ? ? v) val then Some ? (snd ? ? v) else Some ? d
    ] in
  mk_Hoare_Lookup wrap (l_funs l) (l_mem l) ⊩ P
.
```

La definizione di `hoare_predicate_sub` agisce incapsulando la funzione di ricerca in stack del predicato a cui applicare la sostituzione, ritornando il valore sostituito quando l'utente cerca la variabile da sostituire o ritornando il valore cercato nello stack attuale nell'altro caso. Nella dimostrazione, applicando l'assioma di `predicates_equality` si arriva a dover dimostrare il lemma

```
lemma predicate_sub_invariant: ∀s, i, v_fst, v_snd.
     match (stack_search s i) with
     [None ⟹ if eq_var_id v_fst i then Some ? v_snd else None ?
     |Some d ⟹ if eq_var_id v_fst i then Some ? v_snd else Some ? d
     ]
                                        =
      stack_search (update_var_stack s ⟨v_fst, v_snd⟩) i.
```

il quale asserisce che aggiornare una variabile nello stack e applicare la funzione descritta sopra sulla variabile aggiornata NON modifica il risultato

delle ricerche. Il lemma, per essere dimostrato, necessita a catena di ulteriori lemmi riguardanti le proprietá delle funzioni di update e ricerca nello stack. Sono state incluse tutte le dimostrazioni dei lemmi relativi: 3.4.3. Oltre al lemma menzionato, é necessario dimostrare:

```
lemma find_fun_invariant_update_var_stack: ∀s, v, f_id.
  find_fun_stack (update_var_stack s v) f_id
                                        =
  find_fun_stack s f_id.
```

Per mostrare che l'aggiornamento di una variabile é indipendente dallo store delle funzioni.

3.1.0.2 Composizione e while

Per poter dimostrare le regole successive é necessario un lemma fondamentale:

```
lemma to_none_composition: ∀s1, h1, s2, h2, c1, c2.
                Computation (mk_State s1 h1 (c1;c2))(mk_State s2 h2 none) →
  (
       ∃sb, hb. Computation (mk_State s1 h1 c1) (mk_State sb hb none) ∧
       Computation (mk_State sb hb c2)(mk_State s2 h2 none)
  ).
```

L'enunciato asserisce che se una composizione termina in uno stato finale, allora il primo comando termina in uno stato intermedio che conduce al finale. La dimostrazione é per induzione sull'unica ipotesi, e si fissa come invariante il fatto che comando del primo stato é sempre una composizione. (Dettagli: 3.4.1). Combinando questo lemma con la proprietá transitiva della computazione si "sbloccano" molte dimostrazioni, tra cui quelle della validitá della regola della composizione e dell'iterazione. La prima, il cui enunciato in Matita é

```
theorem valid_composition: ∀P, R, Q, c1, c2.
                          { P }  c1  { Q }  →
                          { Q }  c2  { R }  →

        { P }  c1;c2  { R } .
```

segue direttamente dal lemma. Per la seconda invece é necessario un ulteriore lemma:

```
lemma while_termination: ∀c, b_exp, s1, h1, s2, h2.
                 Computation (mk_State s1 h1  (c_while b_exp c_do c endwhile)) (mk_State s2 h2 none) →

  Exp_Eval b_exp (bool_value false) (stack_search s2) (mem_search (mem h2)).
```

che garantisce, se un comando `while` termina, il fatto che nello stato finale la guardia é falsa. Anche quest'ultimo viene dimostrato per induzione, dove l'invariante del passo induttivo é:
$(cmd\ S1 = (processed\_c; c\_whileb\_exp\ c\_do\ c\ endwhile)$
$\vee\ cmd\ S1 = (c\_while\ b\_exp\ c\_do\ c\ endwhile)$. Con questo lemma si puó successivamente dimostrare, finalmente:

```
theorem valid_while: ∀Inv, b_exp, c.
       { λl. Inv l ∧ Exp_Eval b_exp (bool_value true)(l_vars l)(l_mem l) } c { Inv } →
   { Inv }  c_while b_exp c_do c endwhile  { λl. Inv l ∧ Exp_Eval b_exp (bool_value false)(l_vars l)(l_mem l) }
.
```

Anche qui, naturalmente, si usa l'induzione. L'invariante in questo caso é complessa.

```
{ λl. Inv l ∧ Exp_Eval b_exp (bool_value true)(l_vars l)(l_mem l) } c { Inv } →
(

(
Inv (mk_Hoare_Lookup (stack_search (stack S1)) (find_fun_stack (stack S1))(mem_search (mem (heap S1))))
                          ∧
(
cmd S1 = (c_while b_exp c_do c endwhile)  ∨ cmd S1 = none
)
)
∨

( cmd S1 = (processed_c; (c_while b_exp c_do c endwhile))  ∧

∃sf, hf, s0, h0.

Computation (mk_State s0 h0 c) (mk_State (stack S1) (heap S1) processed_c) ∧
Computation(mk_State (stack S1) (heap S1) processed_c)(mk_State sf hf none) ∧
Inv (mk_Hoare_Lookup (stack_search s0) (find_fun_stack s0)(mem_search (mem h0)))  ∧
Exp_Eval b_exp (bool_value true) (stack_search s0) (mem_search (mem h0)) ∧

Inv(mk_Hoare_Lookup (stack_search sf) (find_fun_stack sf)(mem_search (mem hf)))



)
```

La parte importante da analizzare é quella in cui si considera il caso induttivo `c1; while b do c endwhile`. Il comando `c1` é sicuramente una qualche derivazione del corpo del ciclo e quindi la sua esecuzione totale mantiene l'invariante dell'intero `while`.

3.2 Regole inferenza su funzioni

Nel linguaggio valgono inoltre le seguenti regole:

RETURN

$$\frac{}{\{P\texttt{[RETVAR/}E\texttt{]} \wedge R\}\ \texttt{return E}\ \{P\}}$$

Dove "P[RETVAR/E]", come nell'assegnamento, indica P valutato sostituendo il valore della variabile RETVAR con la valutazione dell'espressione E. A differenza dell'assegnamento, tuttavia, é necessario specificare un'ipotesi aggiuntiva per indicare che P possa osservare esclusivamente la RETVAR. Questo é necessario in quanto P potrebbe osservare una variabile nel record di attivazione da cui si ritorna. Siccome, per come sono definiti i predicati, non é possibile "scartare" l'ultimo record dalla valutazione di P, occorre limitarlo a osservare RETVAR. Ció non provoca una perdita di generalitá grazie alla regola successiva.

F_CALL

$$\frac{\{P\}\texttt{body f}\{Q\}}{\{P[store + args] \wedge R\}\ \texttt{f\_call f args}\ \{Q \wedge R\}}$$

Dove "$P[store + args]$" indica P valutato aggiungendo il record di attivazione corrispondente alla chiamata di f. Come condizioni si chiede, inoltre, che R osservi solamente la RETVAR e lo store delle funzioni.

3.2.1 Chiamata e ritorno

La regola relativa al comando `fun_call` necessita del lemma:

```
lemma fun_call_stack_variation: ∀S1, S2.
        Computation S1 S2 →
        ∀f_id, args.
        cmd S1 = fun_call f_id args →
        cmd S2 = none →
(∀f_id. find_fun_stack (stack S1) f_id = find_fun_stack (stack S2) f_id )
∧
(∀i. (eq_var_id RETVAR i) = false → stack_search(stack S1)i = stack_search(stack S2) i)
.
```

Esso garantisce che l'unica cosa a cambiare nello stack dopo il ritorno di una funzione é la RETVAR. Il lemma non é stato dimostrato per motivi di tempo, tuttavia é sicuramente dimostrabile utilizzando le proprietá note del `return`. L'enunciato della regola d'inferenza sulla chiamata di funzione é.

```
theorem valid_fun_call: ∀P, Q, R, f, f_id, e_params.
         { P } (body f) { Q }  →
        (∀s, f, h, s', h'. mk_Hoare_Lookup s f h ⊩ R →
        (∀i. eq_var_id RETVAR i = false → s i = s' i ) →
         mk_Hoare_Lookup s' f h' ⊩ R
        ) →

{ λL. ∀vs, new_s.
Some ? f = (l_funs L) f_id ∧
Exps_Evals e_params vs (vec ? ? (dpi2 ? ? (args f))) new_s (l_vars L) (l_mem L) ∧
(P (mk_Hoare_Lookup (push_function_data (l_vars L) new_s) (l_funs L) (l_mem L)  ))
∧ L⊩R } (fun_call f_id e_params) { λL. L ⊩ Q ∧ L ⊩ R } .
```

Che usa una definizione molto simile a quanto vista nell'assegnamento per simulare il push del record di attivazione sulla pila:

```
definition push_function_data ≝ λsl.λs.
  λv_id. match (find_var_store s v_id) with [None ⟹ sl v_id | Some d ⟹ Some ? d]
.
```

La dimostrazione segue uno schema abbastanza standard, andando per casi sull'ipotesi Computation fornita dalla tripla e applicando infine la side condition assieme al lemma menzionato.
Per quanto riguarda invece il `return` si ha l'enunciato:

```
theorem valid_return: ∀P, exp, retval.
                (∀s, s', f, f', h. mk_Hoare_Lookup s f h ⊩ P →
                                   s RETVAR = s' RETVAR →
                  mk_Hoare_Lookup s' f' h ⊩ P) →
       { λL. Exp_Eval exp retval (l_vars L)(l_mem L) ∧ L⊩(hoare_predicate_sub P ⟨RETVAR, retval⟩) }
      return_cmd exp  { P } .
```

Anche qui la dimostrazione é lineare. Tuttavia, per i lemmi visti fino a ora risulta impossibile, in quanto, dopo aver analizzato i casi rimanendo con

quello giusto, si hanno 2 ipotesi `Exp_Eval`, valutate nella stessa memoria e steso stack, sull'espressione di ritorno. Serve dunque un lemma che garantisce la non ambiguitá della valutazione delle espressioni: 2 espressioni uguali si valutano sempre nello stesso valore. Si tenga a mente questo lemma, lo si dimostrerá nella sezione successiva.

3.3 Separation Logic

3.3.1 Definizione connettivi

Per poter definire i connettivi della Separation Logic si definiscono, innanzitutto, i predicati che descrivono l'unione e la disgiunzione tra 2 memorie.

```
definition disjoint_mems ≝ λ(l1: Address → option Heap_Data).
                           λ(l2: Address → option Heap_Data).
                                 ∀a. (l1 a ≠ None ? → l2 a = None ?) ∧
                                     (l2 a ≠ None ? → l1 a = None ?)
.
definition is_mems_union ≝ λ(l1: Address → option Heap_Data).
                           λ(l2: Address → option Heap_Data).
                           λ(l3: Address → option Heap_Data).
                           ∀a, d. l3 a = Some ? d → (l1 a = Some ? d ∨ l2 a = Some ? d).
```

Con le nuove definizioni é possibile creare tutti i connettivi della Sep.Logic:

```
definition Star: Hoare_P_Local → Hoare_P_Local → Hoare_P_Local ≝ λP.λQ.λl.
  (
    ∃m1, m2. (disjoint_mems (mem_search m1) (mem_search m2))
                                  ∧
             (is_mems_union (mem_search m1) (mem_search m2) (l_mem l))
                                  ∧
    (mk_Hoare_Lookup (l_vars l) (l_funs l) (mem_search m1) ⊩ P)
                                  ∧
    ( mk_Hoare_Lookup (l_vars l) (l_funs l) (mem_search m2) ⊩ Q)
  )
.

definition Magic_Wand: Hoare_P_Local → Hoare_P_Local → Hoare_P_Local≝ λP.λQ.λl.
  ∀m, union.
  is_mems_union (mem_search m) (l_mem l) union →
  disjoint_mems (mem_search m) (l_mem l) →
  (mk_Hoare_Lookup (l_vars l) (l_funs l) (mem_search m)) ⊩ P →
  (mk_Hoare_Lookup (l_vars l) (l_funs l) union) ⊩ Q
.

definition Empty: Hoare_P_Local ≝ λl. ∀a. (l_mem l) a = None ? .

definition Points_To: Address → Heap_Data → Hoare_P_Local ≝ λa.λd.λl.
  (l_mem l) a = Some ? d.
```

```
notation "P1 * P2" left associative with precedence 45 for @{Star $P1 $P2}.
notation "a ↦ [d]" non associative with precedence 45 for @{Points_To $a $d}.
notation "P1 -* P2" right associative with precedence 21 for @{Magic_Wand $P1 $P2}.
```

Dimostrando la commutativitá dei predicati `is_mems_union` e `disjoint_mems` si dimostra facilmente la commutativitá di `Star`.

```
lemma disjoint_mems_comm: ∀m1, m2. disjoint_mems m1 m2 → disjoint_mems m2 m1.
  #m1 #m2 #H whd in H; whd #a lapply(H a) -H * #H1 #H2 @conj /2/
qed.
lemma is_mems_union_comm: ∀m1, m2, tot. is_mems_union m1 m2 tot → is_mems_union m2 m1 tot.
  #m1 #m2 #tot #H whd #a #d #H1 whd in H; lapply (H a d) -H * // #H2 /2/
qed.
theorem star_comm: ∀P1, P2, L. L ⊩ (P1 * P2) → L ⊩ (P2 * P1) .
  #P1 #P2 #L #H whd cases H #m1 * #m2 *** #H1 #H2 #H3 #H4 whd -H
  %{m2} %{m1} @conj // @conj // @conj /2/
qed.
```

3.3.1.1 Monotonicitá Exp_Eval

Prima di mostrare le regole di inferenza relative all'heap, é interessante far notare la seguente proprietá nel linguaggio:

```
lemma framing_lemma_eval: ∀e, s, m_joined, v.
    Exp_Eval e v s m_joined →
    ∀m1, m2, v'.
    is_mems_union m1 m2 m_joined →
    disjoint_mems m1 m2 →
    Exp_Eval e v' s m1 →

      v' = v.
```

Quella enunciata é esattamente la *Safety Monotonicity* enunciata alla sezione 1.3.3 nel capitolo introduttivo, applicata peró sulla valutazione delle espressioni e non sulle computazioni. Come corollario, inoltre é facilmente dimostrabile, scegliendo come seconda memoria quella vuota:

```
corollary expression_eval_injective: ∀e, v1, v2, s, h.
             Exp_Eval e v1 s h → Exp_Eval e v2 s h → v1 = v2.
```

che consente di ultimare la dimostrazione della regola del `return`.

3.3.2 Regole d'inferenza su heap

FREE

$$\frac{}{\{p \longmapsto v * P\ \}\ \texttt{free p}\ \{P\}}$$

MALLOC

$$\frac{}{\{P \wedge x = k\}\ \texttt{malloc x n}\ \{\exists a.(a \longmapsto [mk\_Vector\ \ n] * P) \wedge x = a\}}$$

La verifica sul valore di x nella tripla é necessaria in quanto la malloc si valuta in una def. É inoltre necessaria l'ipotesi che P NON osservi x

WRITE

$$\frac{}{\{\exists d.\ p \longmapsto [d] * P\}\ \texttt{deref\_write p i v}\ \{p \longmapsto [updated\_d] * P\}}$$

REGOLA DI FRAMING

$$\frac{\{P\}\ \texttt{c}\ \{Q\}}{\{P * R\}\ \texttt{c}\ \{Q * R\}}$$

3.3.3 Considerazioni su inferenze Heap

Per motivi di tempo, solamente la regola riguardante la deallocazione é stata dimostrata nel linguaggio, introducendo temporaneamente un assioma (dimostrabile) e un lemma, immediato da dimostrare data la definizione della funzione `free` in Matita:

```
lemma search_deallocated: ∀m, a.
mem_search(fst ? ? (free ? m a)) a = None ?.
lemma free_success: ∀h1, h2, a. Some ? h2 = deallocate h1 a →
      fst ? ? (free ? (mem h1) a) = mem h2.
  * #m1 #f1 #h2 #a #H whd in H:(???%); lapply H -H
  @(b_elim ... (snd ? ? (free Heap_Data m1 a))) #b #H whd in H:(???%); destruct //
qed.
```

Search deallocated é il lemma non dimostrato ed enuncia la proprietá della free di far fallire la ricerca in memoria dell'indirizzo appena liberato. Ció

necessita tuttavia di un ulteriore lemma, che garantisce l'unicitá degli indirizzi nella lista che descrive la memoria. Il secondo enunciato é semplicemente un corollario alla definizione della funzione. La regola della free viene quindi enunciata e dimostrata:

```
theorem valid_free: ∀P, e, a, d.
       { λL. Exp_Eval e (ptr_value a)(l_vars L)(l_mem L) ∧ L ⊩ (P * (a ↦ [d]) ) }
      free e { P }
.
```

Per dimostrare la regola sulla malloc é necessario dimostrare:

```
axiom disjunct_allocation: ∀h, n, m1.
            m1 = mem_search[⟨
             last_allocated(allocate h n),
             mk_DPair ℕ (Vector Value) n (new_vector ? (nat_value 0) n)
            ⟩] →
      is_mems_union (mem_search (mem h)) m1
              (mem_search (mem (allocate h n)))
      ∧
      disjoint_mems (mem_search (mem (h))) m1

.
```

Per quanto riguarda la regola sulla write, essa é del tutto da analizzare, ma non dovrebbe aver bisogno di ristrutturazioni ulteriori del linguaggio. La framing Rule necessita di alcune considerazioni: innanzitutto occorre dimostrare le proprietá presentate nella sezione 1.3.3. Inoltre, é necessario definire un algoritmo che verifichi, a partire da un `Hoare_Lookup` le variabili modificate dal comando `c` della tripla della regola di Framing.

3.4 Dimostrazioni

3.4.1 Miscellanea

```
lemma none_next_Step: ∀s1, s2. cmd s1 = none → Step s1 s2 → s1 = s2.
  #s1 #s2 #none_cmd #step destruct inversion  step
    try #a try #b try #c try #d try #e try #f try #g try #h try #i try #l try #m
       try #n  try #o  try #p try #q try #r destruct  try destruct /2/
qed.
lemma none_next_Computation: ∀s1, s2. cmd s1 = none → Computation s1 s2 → s1 = s2.
  #s1 #s2 #none_cmd #comp destruct inversion comp
//#s10 #s20 #s30 #step #H1 #H2 #H3 #H4 <H3 in step; #step
  lapply (none_next_Step ... s20 none_cmd step) #H5 destruct
  /2/
qed.
lemma to_none_composition: ∀s1, h1, s2, h2, c1, c2.
                                Computation (mk_State s1 h1 (c1;c2))(mk_State s2 h2 none) →
                           (
                             ∃sb, hb. Computation (mk_State s1 h1 c1) (mk_State sb hb none)
```

```
                                                    ∧
                                    Computation (mk_State sb hb c2)(mk_State s2 h2 none)
                            ).

  cut (∀S1, S2, c2.

    Computation S1 S2 →
    cmd S2 = none →
    ∀c1. cmd S1 = (c1; c2) →
    (
      ∃sb, hb. Computation (mk_State (stack S1) (heap S1) c1) (mk_State sb hb none)
                                  ∧
               Computation (mk_State sb hb c2) S2
    )

  )[|#H #Hp_2 #H1 #H2 #H3 #H4 #H5 #H7 cases (H ... H7)[|||//||||] /3/]
  #S1 #S2 #c2  #H elim H -H
  [#H1 #H2 #H3 #H4 >H2 in H4; #H4 destruct ]
  *#s1 #h1 #cmd1 *#s2 #h2 #cmd2 *#s3 #h3 #cmd3 #step #compH #IH
  #c3H #c1 #c1H destruct inversion step
  try #H1 try #H2 try #H3 try #H4 try #H5 try #H6 try #H7 try #H8 try #H9 try #H10
  try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H17 try destruct;
  [%{H1}%{H2} /2/
  | lapply (IH ? H7 ?) /2/ -IH * #sb * #hb * #IH1 #IH2 %{sb} %{hb} /3/
  ]@(Or_ind ... H5) #HH destruct
qed.

lemma while_termination: ∀c, b_exp, s1, h1, s2, h2.
                  Computation (mk_State s1 h1  (c_while b_exp c_do c endwhile)) (mk_State s2 h2 none) →

Exp_Eval b_exp (bool_value false) (stack_search s2) (mem_search (mem h2)).
cut (
∀S1, S2.
Computation S1 S2 →
cmd S2 = none →
∀s0, h0, c, processed_c, b_exp.
   (cmd S1 = (processed_c; c_while b_exp c_do c endwhile)
                                                  ∨
        cmd S1 = (c_while b_exp c_do c endwhile)
   ) →
   Computation (mk_State s0 h0 c) (mk_State (stack S1)(heap S1) processed_c)  →

Exp_Eval b_exp (bool_value false) (stack_search (stack S2)) (mem_search ( mem (heap S2)))
)
[|#H #c #b_exp #s1 #h1 #s2 #h2 #HCOMP @(H ... HCOMP ...)[//||||||//|//] ]
#S1 #S2 #H_comp elim H_comp -H_comp
[#s #s2_c_h #s0 #h0 #c #p_c #b_exp #s1_c_h #comph  >s2_c_h in s1_c_h; #abs cases abs #HH destruct]
*#s1 #h1 #cmd1 *#s2 #h2 #cmd2 *#s3 #h3 #cmd3 #step #compH #IH
#cmd3_h #s0 #h0 #c #p_c #b_exp #cmd1_h cases cmd1_h -cmd1_h destruct inversion step
try #H1 try #H2 try #H3 try #H4 try #H5 try #H6 try #H7 try #H8 try #H9 try #H10
try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H17 try destruct;
[||@(Or_ind ... H5) #HH destruct | | |@(Or_ind ... H5) #HH destruct]
[lapply (IH ... b_exp ... H7) //
|-H10 lapply (IH ? s0 h0 c H7 b_exp ? ?) /2/ @(comp_transitivity ... (mk_State H1 H3 H5)) /2/
|lapply (IH ? ? ? H4 H4 H3 ? ?) /2/
| lapply (none_next_Computation ... compH) // #HH destruct //
qed.
lemma framing_lemma_eval: ∀e, s, m_joined, v.
                          Exp_Eval e v s m_joined →
                          ∀m1, m2, v'.
                          is_mems_union m1 m2 m_joined →
                          disjoint_mems m1 m2 →
                          Exp_Eval e v' s m1 →
```

```
    v' = v.
#e #s #m_joined #v #H elim H -H

 [ #H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 inversion H9 try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H
   try #H18 try #H19 try #H20 try #H21 try #H22 try #H23 try #H24 try #H25 try #H26
   try #H27 try #H29 try #H29 try #H30 try #H31 try #H32 try #H33 try #H34 try #H35 try #H36
   try #H37 try #H39 try #H40 try #H41 try destruct //
 | #H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 inversion H9 try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H
   try #H18 try #H19 try #H20 try #H21 try #H22 try #H23 try #H24 try #H25 try #H26
   try #H27 try #H29 try #H29 try #H30 try #H31 try #H32 try #H33 try #H34 try #H35 try #H36
   try #H37 try #H39 try #H40 try #H41 try destruct //
 | #H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 inversion H9 try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H
   try #H18 try #H19 try #H20 try #H21 try #H22 try #H23 try #H24 try #H25 try #H26
   try #H27 try #H29 try #H29 try #H30 try #H31 try #H32 try #H33 try #H34 try #H35 try #H36
   try #H37 try #H39 try #H40 try #H41 try destruct //
 |#H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 #H10 #H11 inversion H11 -H11 try #H11 try #H12 try #H13 try #H14 try #H15
   try #H18 try #H19 try #H20 try #H21 try #H22 try #H23 try #H24 try #H25 try #H26
   try #H27 try #H29 try #H29 try #H30 try #H31 try #H32 try #H33 try #H34 try #H35 try #H36
   try #H37 try #H39 try #H40 try #H41 try destruct >H15 in H5; #HH destruct //
 | #H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 #H10 #H11 #H12 #H13 #H14 #H15 #H16 #H17
   #H18 inversion H18
   #H110 try #H111 try #H112 try #H113 try #H114 try #H115 try #H116 try #H117
   try #H118 try #H119 try #H120 try #H121 try #H122 try #H133
   try #H135 try #H136 try #H137 try #H138 try #H139 try #H140 destruct
   lapply(H11 ... H16 H17 H117) #HH1 lapply(H12 ... H16 H17 H118) #HH2 destruct //
 | #H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 #H10 #H11 #H12 #H13 #H14 #H15 #H16 #H17
   #H18 inversion H18
   #H110 try #H111 try #H112 try #H113 try #H114 try #H115 try #H116 try #H117
   try #H118 try #H119 try #H120 try #H121 try #H122 try #H133
   try #H135 try #H136 try #H137 try #H138 try #H139 try #H140 destruct
   lapply(H11 ... H16 H17 H117) #HH1 lapply(H12 ... H16 H17 H118) #HH2 destruct //
  | #H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 #H10 #H11 #H12 #H13 #H14 #H15 #H16 #H17
   #H18 inversion H18
   #H110 try #H111 try #H112 try #H113 try #H114 try #H115 try #H116 try #H117
   try #H118 try #H119 try #H120 try #H121 try #H122 try #H133
   try #H135 try #H136 try #H137 try #H138 try #H139 try #H140 destruct
   lapply(H11 ... H16 H17 H117) #HH1 lapply(H12 ... H16 H17 H118) #HH2 destruct //
 | #H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 #H10 #H11 #H12 #H13 #H14 #H15 #H16 #H17
   #H18 inversion H18
   #H110 try #H111 try #H112 try #H113 try #H114 try #H115 try #H116 try #H117
   try #H118 try #H119 try #H120 try #H121 try #H122 try #H133
   try #H135 try #H136 try #H137 try #H138 try #H139 try #H140 destruct
   lapply(H11 ... H16 H17 H117) #HH1 lapply(H12 ... H16 H17 H118) #HH2 destruct //
 | #H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 #H10 #H11 #H12 #H13 #H14 #H15 #H16 #H17
   #H18 inversion H18
   #H110 try #H111 try #H112 try #H113 try #H114 try #H115 try #H116 try #H117
   try #H118 try #H119 try #H120 try #H121 try #H122 try #H133
   try #H135 try #H136 try #H137 try #H138 try #H139 try #H140 destruct
   lapply(H11 ... H16 H17 H117) #HH1 lapply(H12 ... H16 H17 H118) #HH2 destruct //
 | #H1 #H2 #H3 #H4 #H5 #H6 #H7 #H8 #H9 #H10 #H11 #H12 #H13 #H14 inversion H14
   #H110 try #H111 try #H112 try #H113 try #H114 try #H115 try #H116 try #H117
   try #H118 try #H119 try #H120 try #H121 try #H122 try #H133
   try #H135 try #H136 try #H137 try #H138 try #H139 try #H140 destruct
   lapply(H8 ... H12 H13 H115) #HH1 destruct //
 ]
#s0 #h #addrexp #a #indexexp #i #found #res #H1 #H2 #H3 #H4 #H5
 #H6 #m1 #m2 #v' #memUNION #memDISJOINT #H7 inversion H7 -H7 try #H11 try #H12 try #H13 try #H14 try #H15 try #
   try #H18 try #H19 try #H20 try #H21 try #H22 try #H23 try #H24 try #H25 try #H26
   try #H27 try #H29 try #H29 try #H30 try #H31 try #H32 try #H33 try #H34 try #H35 try #H36
   try #H37 try #H39 try #H40 try #H41 try destruct -H23 -H24
   lapply(H5 H12 m2 (ptr_value H14) ? ? ?) // -H5
   lapply(H6 H12 m2 (nat_value H16) ? ? ?) // #HH1 #HH2 destruct
   -H6 whd in memUNION; lapply(memUNION a found ?)// *
   #HH destruct [<H21 in HH; #HH destruct //] -memUNION
```

```
    whd in memDISJOINT; lapply(memDISJOINT a) -memDISJOINT * #disj1 #disj2
    lapply (disj2 ?) [@nmk #abs >abs in HH; #abs2 destruct] #abs >abs in H21;
    #abs2 destruct
qed.
corollary expression_eval_injective: ∀e, v1, v2, s, h.
             Exp_Eval e v1 s h → Exp_Eval e v2 s h → v1 = v2.
  #e #v1 #v2 #s #h #H1 #H2 @framing_lemma_eval
  [@e|@s|@h|//|@h|@(λa.None ?)|/2/|whd #a @conj #HH // @False_ind @(absurd ... HH)/2/]
  //
qed.
```

3.4.2 Inferenze

```
theorem consequence_rule: ∀Ps, Pw, Qs, Qw, c.
                          (∀L. L ⊩ Ps → L ⊩ Pw) →
                          (∀L. L ⊩ Qs → L ⊩ Qw)→
                           { Pw }  c  { Qs }  →
       { Ps }  c  { Qw }
.
  #Ps #Pw #Qs #Qw #c #Ps_Pw #Qs_Qw #H #s1 #h1 #s2 #h2 #compH #precond
  lapply(Ps_Pw  ... precond) -Ps_Pw #h4
  lapply (H s1 h1 s2 h2 ? ?) // #h5 /2/
qed.

theorem valid_none: ∀P. { P }  none  { P } .
  #P #s1 #h1 #s2 #h2 #comp #h inversion comp
  try #a try #b try #c try #d try #e try #f try #g try #h try #i try #l try #m
       try #n  try #o  try #p try #q try #r  try destruct
  /2/
  lapply (none_next_Computation ... comp) // #H destruct //
qed.

theorem valid_if_then_else: ∀P, Q, b_exp, c_true, c_false.
        { (λl. Exp_Eval b_exp (bool_value true)(l_vars l)(l_mem l) ∧ (P l) ) } c_true  { Q }  →
        { (λl. Exp_Eval b_exp (bool_value false)(l_vars l)(l_mem l)∧ (P l)) } c_false { Q }  →
   { P }  (c_if b_exp c_then c_true c_else c_false endif)  { Q } .
  #P #Q #b_exp #c_true #c_false #H_true #H_false whd in H_true; whd in H_false;
  #s1 #h1 #s2 #h2 #comp #precond inversion comp -comp [#s0 #h0 #H0 destruct]
  #s10 #s20 #s3 #step #H1_1 #H2_1 #H3_1 #H4_1 destruct inversion step
  try #H1 try #H2 try #H3 try #H4 try #H5 try #H6 try #H7 try #H8 try #H9 try #H10
  try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H17 try destruct
  [@(H_true ... H1_1 ) /2/ | @(H_false ... H1_1 ) /2/] @(Or_ind ... H5) #HH destruct
qed.

theorem valid_def: ∀P, P2, exp, var.
             (P = λl. ∀val. Exp_Eval exp val (l_vars l)(l_mem l) → (hoare_predicate_sub P2 ⟨var, val⟩) l) →
       { P } def var exp  { P2 } .
  #P #P2 #exp #var #HP whd #s1 #h1 #s2 #h2 #comp #precond destruct
  whd in precond;
  destruct inversion comp -comp
  [#s #H1 #H2 destruct]
  #s10 #s20 #s3 #step_10_20 #comp_20_3 #H01 #H02 #H03 destruct -H01
  inversion step_10_20 -step_10_20
  try #H1 try #H2 try #H3 try #H4 try #H5 try #H6 try #H7 try #H8 try #H9 try #H10
  try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H17 try destruct;
  [|@(Or_ind ... H5) #HH destruct] whd
  lapply (precond ... H8) -precond #precond
  lapply (none_next_Computation ... comp_20_3) // #H9
  lapply(states_destruct ... H9) -H9 ** #S1 #S2 #S3 destruct whd in precond;

  @(predicates_equality ... precond) /2/
```

```
qed.

theorem valid_composition: ∀P, R, Q, c1, c2.
                          { P }  c1  { Q }  →
                          { Q }  c2  { R }  →

        { P }  c1;c2  { R } .

  #P #R #Q #c1 #c2 #HC1 #HC2 #s1 #h1 #s2 #h2 #comp #precond
  whd in HC1; whd in HC2;
  lapply (to_none_composition ... comp) * #sb * #hb * #c1_to_none #c2_to_none
  lapply (HC1 s1 h1 sb hb c1_to_none precond) /3/
qed.
theorem valid_while: ∀Inv, b_exp, c.
               { λl. Inv l ∧ Exp_Eval b_exp (bool_value true)(l_vars l)(l_mem l) } c { Inv } →
   { Inv }  c_while b_exp c_do c endwhile  { λl. Inv l ∧ Exp_Eval b_exp (bool_value false)(l_vars l)(l_mem l) }
.
cut (
∀S1, S2.
Computation S1 S2 →
cmd S2 = none →
∀c, Inv, b_exp, processed_c.
{ λl. Inv l ∧ Exp_Eval b_exp (bool_value true)(l_vars l)(l_mem l) } c { Inv } →
(

(
Inv (mk_Hoare_Lookup (stack_search (stack S1)) (find_fun_stack (stack S1))(mem_search (mem (heap S1))))
                          ∧
(
cmd S1 = (c_while b_exp c_do c endwhile)  ∨ cmd S1 = none
)
)
∨

( cmd S1 = (processed_c; (c_while b_exp c_do c endwhile))  ∧

∃sf, hf, s0, h0.

Computation (mk_State s0 h0 c) (mk_State (stack S1) (heap S1) processed_c) ∧
Computation(mk_State (stack S1) (heap S1) processed_c)(mk_State sf hf none) ∧
Inv (mk_Hoare_Lookup (stack_search s0) (find_fun_stack s0)(mem_search (mem h0)))  ∧
Exp_Eval b_exp (bool_value true) (stack_search s0) (mem_search (mem h0)) ∧

Inv(mk_Hoare_Lookup (stack_search sf) (find_fun_stack sf)(mem_search (mem hf)))



)
  )
  →

  Inv (mk_Hoare_Lookup (stack_search (stack S2)) (find_fun_stack (stack S2))(mem_search (mem (heap S2))))
 )

 [|#CutH #Inv #b_exp #c #H #s1 #h1 #s2 #h2 #comp #precond whd @conj /2/
  lapply (CutH ... comp ? c Inv b_exp ? H ?)[/4/||//|//]//
 ]
#S1 #S2 #compH elim compH -compH
[ try #H1 try #H2 try #H3 try #H4 try #H5 try #H6 try #H7
**[#H9 * #H10 >H2 in H10; #H10 destruct //]#H8>H2 in H8; #H8 destruct]
*#s1 #h1 #cmd1 *#s2 #h2 #cmd2 *#s3 #h3 #cmd3 #stepH #compH #IH
#cmd3H #c #Inv #b_exp #processed_c #H **
[#precond * #H2|#cmd1H #fcompH] destruct inversion stepH -stepH
try #H1 try #H2 try #H3 try #H4 try #H5 try #H6 try #H7 try #H8 try #H9 try #H10
```

```
try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H17 try destruct;
[||@(Or_ind ... H5) #HH destruct||@(Or_ind ... H5) #HH destruct
|||@(Or_ind ... H5) #HH destruct]
[lapply(to_none_composition ... compH) * #sb *#hb* #HH1 #HH2
@IH-IH[///|@H4|@H3|@H4|//|@or_intror @conj// %{sb} %{hb} %{H1} %{H2}
 @conj[ @conj // @conj // @conj //] @(H ... H1 H2 sb hb) // whd /2/]
|lapply(IH ? H4 Inv H3 ? H ?) [/3/|||//|//]//
|lapply(IH ? c Inv b_exp ? H ?)[/3/|||//|//]//
| cases(fcompH) #w * #x * #y * #z  ****#HH1 #HH2 #HH3 #HH4 #HH5 -fcompH
lapply(none_next_Computation ... HH2) // #HH10
 destruct  lapply(IH ? c Inv b_exp ? H ?) [/4/|||//|//]//
]
-H10
lapply(IH ? c Inv b_exp H7 H ?) //%2 @conj //   -IH
cases(fcompH) #w * #x * #y * #z  ****#HH1 #HH2 #HH3 #HH4 #HH5 -fcompH
lapply (to_none_composition ... compH) *#sb *#hb *#HB1 #HB2
%{sb} %{hb} %{y} %{z} @conj [@conj// @conj// @conj // @comp_transitivity[@(mk_State H1 H3 H5)| // |/2/]]
@H[@y|@z|@comp_transitivity[@(mk_State H2 H4 H7)|@comp_transitivity[@(mk_State H1 H3 H5)| // |/2/]]
//| whd @conj //]
qed.
theorem valid_fun_call: ∀P, Q, R, f, f_id, e_params.
         { P } (body f) { Q }  →
        (∀s, f, h, s', h'. mk_Hoare_Lookup s f h ⊩ R →
        (∀i. eq_var_id RETVAR i = false → s i = s' i ) →
         mk_Hoare_Lookup s' f h' ⊩ R
        ) →

{ λL. ∀vs, new_s.
Some ? f = (l_funs L) f_id ∧
Exps_Evals e_params vs (vec ? ? (dpi2 ? ? (args f))) new_s (l_vars L) (l_mem L) ∧
(P (mk_Hoare_Lookup (push_function_data (l_vars L) new_s) (l_funs L) (l_mem L)  ))
∧ L⊩R } (fun_call f_id e_params) { λL. L ⊩ Q ∧ L ⊩ R } .
  #P #Q #R #f #f_id #e_params #H #Rvar whd #s1 #h1 #s2 #h2 #compH #precond
  inversion compH [#a #b #c destruct] try #H1 try #H2 try #H3 try #H4 try #H5
  try #H6 try #H7 try #H8 try #H9 try #H10
  try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H17 try destruct
  inversion H4 try #H7 try #H8 try #H9 try #H10
  try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H17
  try #H18 try #H19 try #H20 try #H21 try destruct [|@(Or_ind ... H11) #HH destruct]
  -H6 lapply(precond ... H11 H12) -precond *** #precond1 #precond2 #precond3 #precond4
  <H15 in precond1; #precond1 destruct whd in H; cut(Q
  (mk_Hoare_Lookup (stack_search s2) (find_fun_stack s2) (mem_search (mem h2))))[
  @H [@((mk_Activation_Record H12 [])::H13)|@H14|//] -H -H16
   destruct
   cut ((push_function_data (stack_search H13) H12)  =
        (stack_search ((mk_Activation_Record H12 [])::H13))

        ) //
  #HF <HF
  cut ( ∀i.
    (find_fun_stack H13) i = (find_fun_stack ((mk_Activation_Record H12 [])::H13))i
  )// #i @predicates_equality [||||//||//||//||//]//@(addr 0)] #QH @conj //
  -H -precond3 destruct
  lapply(fun_call_stack_variation ... compH H7 H10 ?? ) // *
  #HH1 #HH2 lapply(Rvar ... precond4 ?)[//|@(mem_search (mem h2))|]
  #HH3 @(predicates_equality ... HH3) /2/ [@(var 1)]/2/
qed.
theorem valid_return: ∀P, exp, retval.
                (∀s, s', f, f', h. mk_Hoare_Lookup s f h ⊩ P →
                                   s RETVAR = s' RETVAR →
                 mk_Hoare_Lookup s' f' h ⊩ P) →
       { λL. Exp_Eval exp retval (l_vars L)(l_mem L) ∧ L⊩(hoare_predicate_sub P ⟨RETVAR, retval⟩) }
      return_cmd exp  { P } .
  #P #exp #retval #H whd #s1 #h1 #s2 #h2 #Hcomp *#pre1 #pre2
```

```
  inversion Hcomp -Hcomp [#a #b #c destruct]try #H1 try #H2 try #H3 try #H4 try #H5
  try #H6 try #H7 try #H8 try #H9 destruct inversion H4 -H4 try #H7 try #H8 try #H9 try #H10
  try #H11 try #H12 try #H13 try #H14 try #H15 try #H16 try #H17
  try #H18 try #H19 try #H20 try #H21 try destruct [|@(Or_ind ... H11) #HH destruct]
  -H6 lapply (none_next_Computation ... H5) // -H5 #H5 lapply(states_destruct ... H5)
  **#HH1 #HH2 #HH3 -H5 destruct cut(H11 = retval)[
  @(expression_eval_injective ... H13 pre1)]#Hf destruct -H13
  @(H ... pre2) >(eq_var_id_trivial_case)
  cases stack_search[ whd in ⊢(??%?); /2/] #v whd in ⊢(??%?); /2/
qed.
theorem valid_free: ∀P, e, a, d.
       { λL. Exp_Eval e (ptr_value a)(l_vars L)(l_mem L) ∧ L ⊩ (P * (a ↦ [d]) ) }
      free e { P }
.
  #P #e #a #d whd #s1 #h1 #s2 #h2 #compH * #expH #starH
  inversion compH -compH [#H1 #H2 #H3 destruct]
  *#ss1 #hh1 #cc1 *#ss2 #hh2 #cc2 *#ss3 #hh3 #cc3 #stepH #compH
  #H1 #H2 #H3 destruct inversion stepH -stepH
  try #H2 try #H3 try #H4 try #H5 try #H6 try #H7 try #H8 try #H9
  try #H10 try #H11 try #H12 try #H13 try #H14 try #H15 try #H16
  destruct [|@(Or_ind ... H6) #HH destruct] -H1
  lapply(none_next_Computation ...compH) // -compH #compH destruct
  cases starH -starH #m1 * #m2 *** #disj #union #PH #pointsH
  lapply(expression_eval_injective ... H7 expH) #HH destruct -H7
  lapply (disj a) * -disj #disj1 #disj2
  whd in pointsH; lapply(disj2 ?) [@nmk #abs >abs in pointsH; #p
   destruct] #H lapply(free_success ... H8) #H8_1 @(predicates_equality ... PH) /2/
   [@a] <H8_1 >H >search_deallocated //
qed.
```

3.4.3 Gestione stack

```
lemma find_fun_invariant_ow_stack: ∀s, v, f_id.
              find_fun_stack (ow_stack s v) f_id
                                =
              find_fun_stack s f_id.
  #s elim s // -s #h #t #IH #v #f_id whd in ⊢(? ? (? % ?) %);
  cases (\snd (ow_store (var_store h) v)) whd in ⊢(? ? (? % ?) %);
  // whd in ⊢(? ? % ?); >(IH v f_id) //
qed.

lemma find_fun_invariant_update_var_stack: ∀s, v, f_id.
  find_fun_stack (update_var_stack s v) f_id
                                      =
  find_fun_stack s f_id.
  #s elim s // -s #h #t #IH #v #f_id  whd in ⊢(? ? (? % ?) %);
  cases (stack_search (h::t) (\fst v)) // #value
  cut ( match Some Value value with
  [None ⟹
   mk_Activation_Record (v::current_var_store (h::t)) (current_fun_store (h::t))
   ::tail Activation_Record (h::t)
  |Some (d:Value) ⟹ ow_stack (h::t) v] = ow_stack (h::t) v

  ) // #H >H -H >find_fun_invariant_ow_stack //
qed.
lemma find_var_store_head_fail: ∀sh, st, i. eq_var_id (fst ? ? sh ) i = false →
      find_var_store (sh::st) i = find_var_store st i.
  #sh #st #i #H whd in ⊢(??%?); >H //
qed.
lemma stack_search_hit_split: ∀sh, st, v, i.
             stack_search (sh::st) i = Some ? v →
      find_var_store (var_store sh) i = Some ? v ∨ (stack_search st i = Some ? v ∧ find_var_store (var_store sh) i
  #sh #st #v #i #H whd in H:(??%?);
   lapply H inversion(find_var_store (var_store sh) i)
  -H[#H #inv whd in inv:(? ? % ?); /3/]
  #v2 #H1 #inv whd in inv:(? ? % ?); destruct /2/
qed.
lemma find_var_store_split: ∀sh,st, v, i.
             find_var_store (sh::st) i = Some ? v →
      (eq_var_id (fst ? ? sh) i = true ∧ (snd ? ? sh) = v)∨
      (find_var_store st i = Some ? v ∧ eq_var_id (fst ? ? sh) i = false ).
  #sh #st #v #i #H  whd in H:(??%?);lapply H  -H cases (eq_var_id (\fst sh) i)
  #H whd in H:(??%?); /3/ @or_introl  @conj // destruct //
qed.
lemma tail_hit_store: ∀sh, st, v_id, v.
             find_var_store (sh::st) v_id = Some ? v →
             eq_var_id (fst ? ? sh) v_id = false →
      find_var_store st v_id = Some ? v.
  #sh #st #v_id #v #H whd in H:(? ? % ?); #H2 >H2 in H; #H whd in H:(? ? % ?); //
qed.
lemma find_var_store_fail0: ∀sh, st, i, i'.
             eq_var_id i i' = true →
             find_var_store (sh::st) i = None ? →
      eq_var_id (fst ? ? sh) i' = false.
  #sh #st #i #i' #H1 #H2 whd in H2:(??%?); lapply H2 -H2
  @(b_elim ... (eq_var_id (\fst sh) i)) #b #H2 whd in H2:(??%?); destruct /3/
qed.
lemma find_var_store_fail1: ∀s, i, i'.
             find_var_store s i = None ? →
             eq_var_id i i' = true →
      find_var_store s i' = None ?.
  #s elim s -s // #sh #st #IH #i #i' #H1 #H2 lapply(find_var_store_fail0 ... H2 H1) #H3
  whd in ⊢(??%?); >H3 whd in ⊢(??%?); whd in H1:(??%?);
  lapply H1 @(b_elim ... (eq_var_id (\fst sh) i)) #b -H1 #H1 whd in H1:(??%?);
```

```
  [lapply(eq_var_id_transitivity i' i (fst ? ? sh) ? ?) // #abs >(eq_var_id_symm) in abs;
  #abs >abs in H3; #H destruct] @(IH ... H1 H2)
qed.
lemma find_var_store_fail: ∀sh, st, i, i'.
             find_var_store (sh::st) i = None ? →
             eq_var_id i i' = true →
      eq_var_id(fst ?? sh) i' = false ∧ find_var_store st i' = None ?.
  #sh #st #i #i' #H #H2 @conj /2/ whd in H:(??%?); lapply H −H
  @(b_elim ... (eq_var_id (\fst sh) i)) #b #H whd in H:(??%?); destruct
  @(find_var_store_fail1 ... H H2)
qed.
lemma var_defined_store_to_ow: ∀var_store, v_id, v, v2.
             find_var_store var_store v_id = Some ? v→
      \snd(ow_store var_store ⟨v_id, v2⟩) = true.
  #var_store #v_id #v #v2 elim var_store −var_store [#H whd in H:( ? ? % ?); destruct]
  #store_h #store_t #IH #H whd in ⊢ (? ? (? ? ? %) ?);
  @(b_elim ... (eq_var_id (\fst store_h) v_id)) #b /3/
qed.

lemma var_not_defined_store_to_ow: ∀s, i, i', v.
             find_var_store s i = None ? →
             eq_var_id i i' = true →
      \snd (ow_store s ⟨i', v⟩) = false.
  #s elim s −s //
  #sh #st #IH #i #i' #v #H #H2 lapply(find_var_store_fail ... H H2) −H * #H1 #H2
   whd in ⊢ (? ? (? ? ? %) ?); >H1 whd in ⊢ (? ? (? ? ? %) ?); /2/
qed.
lemma find_var_store_hit_head: ∀sh, st, i, v , i'.
             sh = ⟨i, v⟩ →
             eq_var_id i i' = true →
      find_var_store (sh :: st) i' = Some ? v.
  *#fsth #sndh #st #i #v #i' #H #H2 destruct whd in ⊢( ? ? % ?); >H2 //
qed.

lemma find_store_same_vars: ∀s, i, i2.
              eq_var_id i i2 = true →
      find_var_store s i = find_var_store s i2.
  #s elim s −s // #h #t #IH #i #i2 #H whd in ⊢(? ? % %);
  @(b_elim ... (eq_var_id (\fst h)i)) #b whd in ⊢(? ? % ?);
  [lapply(eq_var_id_transitivity i2 i (\fst h) ? ? ) // #trans
   <(eq_var_id_symm i2 (\fst h)) >trans normalize //]
  lapply(neq_var_id_transitivity i2 i (\fst h) ? ? ) // #trans
   <(eq_var_id_symm i2 (\fst h)) >trans whd in ⊢(? ? ? %);
  @IH //
qed.

lemma find_defined_var_store_ow_store: ∀s, i, v', v, i'.
             find_var_store s i = Some ? v' →
             eq_var_id i i' = true →
      find_var_store (fst ? ? (ow_store s ⟨i, v⟩)) i' = Some ? v.
  #s elim s −s [#h1 #h2 #h3 normalize #h4 #h5  destruct]
  #sh #st #IH #i #v' #v #i' #H1 #H2 whd in ⊢(??(?(???%)?)?); lapply(find_var_store_split ... H1)
  −H1 * * #HH1 #HH2 [>HH1 whd in ⊢(??(?(???%)?)?); @(find_var_store_hit_head ... H2) //]
    >HH2 whd in ⊢(??(?(???%)?)?); whd in ⊢(??%?);
   cut(eq_var_id (fst ? ? sh) i' = false) /3/ #hcut >hcut −hcut
   /2/
qed.
lemma ow_stack_and_search: ∀s, v', i, v, i'.
             eq_var_id i i' = true →
             stack_search s i = Some ? v' →
      stack_search (ow_stack s ⟨i, v⟩) i' = Some ? v.
  #s elim s −s [#v1 #i #v #H1 #H2 #H3 normalize in H3; destruct]
  #sh #st #IH #v' #i #v #i' #H1 #H2 whd in ⊢(??(?%?)?); lapply(stack_search_hit_split ... H2) −H2 *
  #H [>var_defined_store_to_ow //  whd in ⊢(??(?%?)?); whd in ⊢(??%?);
```

```
     >find_defined_var_store_ow_store //]
  @(And_ind ... H) -H #H1 #H2 >var_not_defined_store_to_ow // whd in ⊢(??(?%?)?);
   whd in ⊢(??%?); >find_var_store_fail1 /2/
qed.

lemma update_stack_and_search: ∀s, i, i', v.
              eq_var_id i i' = true →
      stack_search (update_var_stack s ⟨i, v⟩) i' = Some ? v.
  #s elim s -s [#i #i' #v #H whd in ⊢(??(?%?)?); whd in ⊢(??%?);
   >(find_var_store_hit_head ... H) //]
  #sh #st #IH #i #i' #v #H whd in ⊢(??(?%?)?); inversion (stack_search (sh::st) i)
  [#inv whd in ⊢(??(?%?)?); whd in ⊢(??%?); >(find_var_store_hit_head ... H) // ]
  #v2 #inv cut(match Some Value v2
   in option
  returnλ_:(option Value).(list Activation_Record)
  with
  [None ⇒
   mk_Activation_Record (⟨i,v⟩::current_var_store (sh::st))
   (current_fun_store (sh::st))
   ::tail Activation_Record (sh::st)
  |Some (d:Value) ⇒ ow_stack (sh::st) ⟨i,v⟩] = ow_stack(sh::st)⟨i,v⟩) // #Hcut >Hcut -Hcut
  /2/
qed.

lemma stack_search_fail: ∀sh, st, i.
             stack_search (sh::st) i = None Value →
      find_var_store (var_store sh) i = None ? ∧ stack_search st i = None ?.
  #sh #st #i #H @conj whd in H:(??%?); lapply H -H inversion(find_var_store (var_store sh) i)
  //[#v2 #inv #H whd in H:(??%?); destruct
  | #inv #H whd in H:(??%?); //] #v2 #inv #H whd in H:(??%?); destruct
qed.
lemma find_var_store_fail_head: ∀st, i, i', v.
             eq_var_id i i' = false →
      find_var_store  (⟨i, v⟩::st) i' = find_var_store st i'. /3/
qed.

lemma find_var_store_to_fail_ow_store: ∀s, i, i', v.
             eq_var_id i i' = false →
             find_var_store s i = None ? →
      find_var_store (fst ? ? (ow_store s ⟨i', v⟩)) i = None ?.
  #s elim s -s // #sh #st #IH #i #i' #v #H1 #H2 whd in ⊢(??(?(???%)?)?);
  lapply (find_var_store_fail ... H2 ?) [|@i] // * -H2 #H2_1 #H2_2
  @(b_elim ...(eq_var_id (\fst sh) i')) #b whd in ⊢(??(?(???%)?)?); whd in ⊢(??%?);
  [ >eq_var_id_symm >H1 /2/] >H2_1  /2/
qed.

lemma find_var_store_to_fail_ow_stack: ∀s, i, i', v.
             eq_var_id i i' = false →
             stack_search s i = None ? →
      stack_search(ow_stack s ⟨i', v⟩) i = None ?.
  #s elim s -s // #sh #st #IH #i #i' #v #H1 #H2 lapply(stack_search_fail ... H2) -H2 * #H2_1 #H2_2
  whd in ⊢(??(?%?)?); cases (\snd (ow_store (var_store sh) ⟨i',v⟩))
  whd in ⊢(??(?%?)?); whd in ⊢(??%?); >(find_var_store_to_fail_ow_store ... H1 H2_1) /2/
  >H2_1 /2/
qed.

lemma update_and_search_fail: ∀i, v_fst, v_snd, s.
             eq_var_id v_fst i = false →
             stack_search s i = None ? →
      stack_search(update_var_stack s ⟨v_fst, v_snd⟩) i = None ? .
  #i #v #v_snd #s elim s -s [#H1 #H2 whd in ⊢(??(?%?)?); whd in ⊢(??%?);
    >find_var_store_fail_head //]
  #sh #st #IH #H1 #H2
  whd in ⊢(??(?%?)?); cases (stack_search (sh::st) v)
```

```
  [whd in ⊢(??%?); >current_var_store_unfold >find_var_store_fail_head //
    lapply(stack_search_fail ... H2) -H2 * #H2_1 #H2_2 >H2_1 whd in ⊢(??%?); /2/]
  #v2 >find_var_store_to_fail_ow_stack /2/
qed.

lemma find_var_store_hit_tail: ∀st, i, i', v, v'.
             eq_var_id i i' = false →
             find_var_store st i = Some ? v →
      find_var_store ((i', v')::st) i = Some ? v.
  /2/
qed.

lemma find_defined_var_store_ow_store2: ∀s, i, i', v, v'.
             eq_var_id i i' = false →
             find_var_store s i = Some ? v →
       find_var_store (fst ? ? (ow_store s ⟨i', v'⟩)) i = Some ? v.
  #s elim s -s  [#h1 #h2 #h3 #h3 #h5 #h6 normalize in h6; destruct]
  #sh #st #IH #i #i' #v #v' #H1 #H2 whd in ⊢(??(?(???%)?)?); lapply(find_var_store_split ... H2)
  -H2 * * #HH1 #HH2  @(b_elim ... (eq_var_id (\fst sh) i') ) #b whd in ⊢(??(?(???%)?)?);
  [lapply(eq_var_id_transitivity i (fst ?? sh) i' ? ?) // #abs >abs in H1; #H1 destruct]
  whd in ⊢(??%?); [ >HH1 /2/ | >eq_var_id_symm >H1 /2/] >HH2 /2/
qed.


lemma ow_stack_and_search2: ∀s, i, i', v, v'.
             eq_var_id i i' = false →
             stack_search s i = Some ? v →
      stack_search (ow_stack s ⟨i', v'⟩) i = Some ? v.
  #s elim s -s [#h1 #h2 #h3 #h3 #h5 #h6 normalize in h6; destruct]
  #sh #st #IH #i #i' #v #v' #H1 #H2 lapply(stack_search_hit_split ... H2) * -H2 #H2
  whd in ⊢(??(?%?)?); cases(\snd (ow_store (var_store sh) ⟨i',v'⟩))
  whd in ⊢(??(?%?)?); whd in ⊢(??%?); [
  >(find_defined_var_store_ow_store2 ... H1 H2) //
  | >H2 //] @(And_ind ... H2) -H2 #HH1 #HH2
  [ >(find_var_store_to_fail_ow_store ... H1 HH2) /2/
  ] >HH2 /2/
qed.

lemma update_stack_and_search2: ∀s, i, i', v, v2.
               eq_var_id i i' = false →
               stack_search s i = Some ? v2 →
      stack_search (update_var_stack s ⟨i', v⟩) i = Some ? v2.
  #s elim s -s [ #i #i' #v #v2 #H1 #H2  whd in H2:(??%?); destruct]
  #sh #st #IH #i #i' #v #v2 #H1 #H2 lapply(stack_search_hit_split ... H2) * #H2_1
  whd in ⊢(??(?%?)?); inversion (stack_search (sh::st) i')
  [#inv whd in ⊢(??(?%?)?); whd in ⊢(??%?); >current_var_store_unfold
   >(find_var_store_hit_tail ... H1 H2_1) //
  | #v3 #inv /2/] @(And_ind ... H2_1) -H2_1 #HH1 #HH2 [
   #inv whd in ⊢(??(?%?)?); whd in ⊢(??%?); >current_var_store_unfold
   >(find_var_store_fail_head  ) // >HH2 /2/
   ]#v3 #inv /3/
qed.
```

Conclusioni

Il progetto, come si é potuto evincere dalla divisione in capitoli, é stato diviso nelle 2 parti: **implementazione linguaggio** e **implementazione logica di Hoare**. A questo aggiungo un ulteriore "layer" (di difficoltá): l'utilizzo di Matita. Con "utilizzo di Matita" non intendo denotare la sua sintassi. Intendo il paradigma di programmazione che sta alla base del software. Non avendo mai usato paradigmi di programmazione logica, funzionale né tantomeno strumenti di dimostrazione interattiva ho trovato un notevole ostacolo nell'usare il software. I primi 2 mesi di lavoro sono stati interamente incentrati sullo studio e sull'apprendimento di Matita e ciononostante ho dovuto, fino a pochi giorni dalla consegna della tesi, chiedere delucidazioni sull'utilizzo di alcune tattiche al mio relatore. A tal proposito, non ho inserito una sezione ringraziamenti nel documento, quindi approfitto di questo spazio dedicato alle conclusioni per ringraziare il professor Coen per la tempestivitá e per la disponibilitá nel rispondere ai miei (talvolta sciocchi, devo riconoscerlo) dubbi.

Continuando sul progetto, la parte relativa all'implementazione del linguaggio é stata la parte meno tediosa dato che, come esercizio, si é scelto di non ottimizzare, di non effettuare type-checking e, in generale, di mantenere tutto il piú semplice possibile. La parte relativa alla logica di Hoare, invece, é stata la piú complessa, complice soprattutto la mia inesperienza, ancora una volta, con le tattiche e le dimostrazioni in Matita. Per motivi di tempo e per le diverse ristrutturazioni e modifiche apportate al codice, la parte relativa alla logica di Hoare resta da ultimare, in

particolare quella relativa alla separation logic, mentre é da implementare da 0 la parte sulla concorrenza. Per quanto riguarda la separation logic sono state individuate le proprietá da dimostrare nella memoria per soddisfare le regole d'inferenza rimaste "sospese" e per dimostrare la *Framing Rule*.
Occorre, in seconda battuta, estendere quanto visto sulla logica di Hoare alla concorrenza, quindi dimostrare la regola vista alla sezione 1.4 e pensare a un *framework* per verificare la sincronizzazione dei thread.
Ultimate queste dimostrazioni si potrá implementare un'interfaccia lato utente, per permettere a questo la creazione di programmi nel linguaggio presentato e, utilizzando le regole d'inferenza di Hoare come "tattiche", dimostrarne la correttezza.
Concludo dicendo che é stato veramente interessante lavorare "alla maniera di Euclide", vale a dire usando definizioni e assiomi per procedere formalmente, in maniera quanto piú rigorosa possibile. Personalmente, penso che questo iter sia quello piú corretto per scrivere qualunque genere di programma e spero che strumenti simili a Iris[16] (e ringrazio, ancora, il relatore per avermelo fatto conoscere) diventino sempre piú conosciuti.

Bibliografia

[1] *Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, Enrico Tassi. The Matita Interactive Theorem Prover. matita.cs.unibo.it*

[2] *Hoare, C. A. R. An Axiomatic Basis for Computer Programming. Communications o f the ACM 1969*

[3] *J. W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In IFIP Congress, 1959.*

[4] *Gordon D. Plotkin. A Structural Approach to Operational Semantics. (1981) Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark.*

[5] *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*

[6] *M. O. Rabin and D. Scott, "Finite Automata and their Decision Problems", IBM Journal of Research and Development, 3:2 (1959) .*

[7] *Reynolds, John C. (2002). Separation Logic: A Logic for Shared Mutable Data Structures. LICS*

[8] *Reynolds, John C. (2008). Separation Logic: An Introduction to Separation Logic. ITU University, Copenhagen*

[9] *R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic.*

[10] *Peter W. O'Hearn, Theoretical Computer Science 375(1-3), May 2007, Sections 6-7*

[11] *Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, Matita Tutorial, Journal of Formalized Reasoning Vol. 7, No. 2, 2014, Pa*

[12] *Strachey, C. (2000). Currying. Higher Order and Symbolic Computation, 13(1), 11-49*

[13] *M. Gabbrielli, S. Martini Linguaggi di programmazione: principi e paradigmi McGraw-Hill Italia, 2005*

[14] Moschovakis, Joan, "Intuitionistic Logic", The Stanford Encyclopedia of Philosophy (Summer 2023 Edition), Edward N. Zalta, Uri Nodelman (eds.)

[15] *Logica 4: Sintassi, Claudio Sacerdoti Coen, Universitá di Bologna, 2017*

[16] Iris-project, Logic and Semantics group, Computer Science @ Aarhus University