Alma Mater Studiorum - Università di Bologna

Scuola di Scienze
Corso di Laurea in Informatica

Discussione sull'integrazione di strumenti di verifica formale nella programmazione quotidiana

Relatore:
Prof. Claudio Sacerdoti Coen

Presentata da:
Daniele Vito Ardito

Sessione II - Secondo Appello
Anno Accademico 2024/2025

Indice

Capitolo 1

Introduzione

Il presente studio propone un uso non convenzionale di Why3, uno strumento di verifica formale. Viene definita una strategia che guida il programmatore all'uso atipico di questo strumento. Il metodo proposto aiuterebbe a trovare un compromesso tra formalità ed efficienza nello sviluppo software. Questo approccio viene analizzato e studiato attraverso casi di studio reali: esercizi su problemi isolati ed un progetto. Queste attività hanno generato una serie di deduzioni e conclusioni, che sostanzialmente compongono l'elaborato.

1.1 Motivazione

La scrittura di questa tesi parte da una necessità. La necessità che ha un programmatore scarso di capire come migliorare il proprio stile di programmazione. In tutta la mia vita ho avuto molti consigli sullo stile di programmazione. Alcuni mi sono stati utili, altri meno, molti si contraddicevano. Studiando diversi linguaggi ho avuto modo di assorbire concetti, pratiche e stili di programmazione che mi hanno aiutato a programmare meglio ma non a capire esattamente come reagire agli ostacoli incontrati sviluppando codice. Inoltre, ho avuto spesso la sensazione di programmare in modo avventato, senza conoscere un'alternativa. Meglio, le alternative di cui ero a conoscenza, mi sembravano troppo deboli. Parlo di diagrammi di flusso, pseudocodice, diagrammi grafici come UML ecc...
Pensavo, quindi, che questi fossero gli unici strumenti per poter ragionare in modo strutturato su un problema, o meglio, per aiutarsi a pensare ad una soluzione prima di scrivere del codice considerabile definitivo.

1.1.1 La prima intuizione

Nel corso di studi, ho avuto la fortuna di poter seguire corsi di matematica e corsi come "logica per l'informatica", "algoritmi e strutture dati" e "informatica teorica" che mi hanno suggerito in varie forme l'uso della logica per ragionare sui problemi prima di scrivere del codice che implementasse una soluzione. Ho provato ad utilizzare questo metodo nello sviluppo dei progetti svolti per vari corsi. Il metodo consisteva semplicemente nel provare a ragionare sul problema scrivendo su carta delle proprietà matematiche che riguardassero il problema in analisi. Poi, provavo a scrivere dello pseudocodice, per lo più utilizzando notazioni matematiche. Successivamente, riportavo l'implementazione nel linguaggio di programmazione di destinazione. Non avevo risolto molto. Continuavo a commettere errori tipici, continuavo a scovare proprietà del mio codice dopo il ritrovamento di numerosi errori. Non sentivo di aver raggiunto un approccio strutturato alla programmazione. Tanto meno un metodo che mi permettesse di capire i problemi a pieno, o di avere fiducia nel codice che scrivevo. Questa era la mia più grande preoccupazione. Non riuscivo ad avere fiducia nelle mie implementazioni, sentivo sempre sfuggirmi qualcosa.
Nonostante ciò, rimanevo convinto del fatto che la matematica fosse la via giusta di approcciare certi tipi di problemi, più facilmente manipolabili con la logica. Nella mia mente era chiara la corrispondenza tra programmazione e dimostrazione, come la corrispondenza tra funzioni e teoremi. Studiando la matematica, leggere programmi diventava sempre più simile a capire teorie matematiche. Dunque, lo studio di programmi può avvenire in modo simile allo studio di una teoria matematica. Non mi era chiaro il processo con cui si scrive un programma quanto non mi era chiaro il processo con cui si compone una teoria. Sia l'informatica che la matematica, in certi aspetti, vengono spiegate in modo simile. Per capire la programmazione si studiano dei problemi e delle soluzioni algoritmiche a quei problemi. Si spiegano delle soluzioni e poi si chiede di risolvere un problema simile. Avviene la stessa cosa con la matematica. Si spiegano delle teorie, spiegandone gli assiomi, le definizioni, i teoremi, poi, alcuni teoremi simili si danno da dimostrare come esercizio. All'inizio si è spaesati, in entrambe le discipline. Scrivere una dimostrazione è molto più difficile che capirne una e, allo stesso modo, scrivere del codice è spesso più difficile che leggere del codice di cui si è intuita la correttezza. Facendo un po' di pratica, si tende a migliorare in questi processi. Si impara a sviluppare un certo intuito personale, che ci direziona nella formulazione di algoritmi e dimostrazioni.
A questo punto, l'intuizione che consisteva nell'uso della matematica per aiutarsi a

risolvere problemi in programmazione, continuava a reggersi sulla somiglianza che c'è tra queste discipline e sul rigore matematico, spesso assente nella programmazione. È un'intuizione e non una verità affermata perché, oltre alla somiglianza, esistono anche delle grandi differenze.
Equiparando una funzione informatica a un teorema, si notano delle discrepanze sostanziali. Per capire l'enunciato di un teorema, non serve leggere la dimostrazione. Leggere la dimostrazione aiuta a capire meglio il contesto in cui vive l'enunciato, permette di capire perché quell'enunciato sia vero. Questo perché l'enunciato descrive esattamente la verità che si vuole affermare.
Se si vede l'intestazione di una funzione informatica come l'enunciato, e il corpo della funzione come la dimostrazione, si nota che il processo di comprensione di una funzione è piuttosto diverso. L'intestazione della funzione, nei linguaggi di programmazione tradizionali, è troppo povera per descrivere lo scopo di quella funzione. Se il linguaggio è tipato, ci sarà scritto solo il nome della funzione, i nomi dei parametri, i tipi dei parametri, il tipo del valore di ritorno. L'intestazione della funzione, dunque, risulta quasi sempre insufficiente alla comprensione delle specifiche e ciò deriva dalla scarsa espressività dei tipi che si usano. Quindi, si cerca di dedurre l'enunciato dall'intestazione, dal nome (che è informale) e dalla dimostrazione (corpo). Risulta evidente che dedurre un enunciato dalla dimostrazione (spesso i corpi delle funzioni sono molto lunghi) è un'attività molto più complicata. La corrispondenza diventa totale con un sistema di tipi abbastanza ricco da poter specificare qualsiasi funzione con i parametri e il valore di ritorno. Tipizzare diventa però piuttosto dispendioso. I linguaggi di programmazione utilizzati nella maggior parte dei casi non dispongono di questi sistemi.

1.2 L'individuazione di un primo strumento

Continuavo a non sentirmi fiducioso riguardo al mio approccio. Sentivo un bisogno di imposizione di una serie di vincoli che mi avrebbero guidato in approcci più schematici e rigorosi. Ho pensato che ci potessero essere dei software che mi potessero aiutare a fare questo. Quindi, cercando, sono incappato in TLA+, un software sviluppato da Leslie Lamport, rilasciato nel 1999. TLA+ è uno strumento atto alla specifica di sistemi (o software) concorrenti. È basato sulla matematica e su una logica sviluppata da Lamport stesso, chiamata TLA (Temporal Logic of Actions) che è sostanzialmente un'estensione della logica LTL (Linear Temporal Logic). Studiandolo, ho letto una serie di sentenze che sentivo di condividere ma non avevo l'esperienza per verificarle. Ad esempio, una delle dichiarazioni che

echeggiava nei tutorial e nei libri su TLA+, diceva che dare specifiche formali di programmi e algoritmi sia fondamentale per scrivere codice corretto. Mi sembrava giusto ma non avevo avuto modo di sperimentarlo su di me. Quando l'oggetto di uno studio è il processo e non il prodotto, le sensazioni di coloro che conducono un processo sono fondamentali.

Conclusione

La comprensione del processo di programmazione può essere esplorata unicamente attraverso le testimonianze dei programmatori sulla loro attività, sulle loro sensazioni e sugli stati del prodotto, sotto determinate condizioni.

1.3 L'esplorazione di altri strumenti

Grazie al mio relatore, ho avuto modo di conoscere altri strumenti che potessero essere inerenti allo scopo della mia indagine, che quindi verte sull'individuazione di uno strumento che permetta di ragionare su problemi in modo strutturato, ad alto livello, senza utilizzare il linguaggio di programmazione di destinazione. Dunque il principio fondamentale vedrebbe lo studio di problemi (che concerne formalizzazione di specifiche e implementazione) come un processo circa separato e previo alla stesura del codice finale.
Oltre a TLA+, mi sono stati suggeriti altri software/ambienti di sviluppo. Questi sono Atelier B, Rocq, Iris, Why3.

1.4 Caratteristiche di TLA+, Atelier B, Rocq, Iris, Why3

1.4.1 TLA+

Come detto in precedenza, TLA+ è un software atto alla specifica di sistemi, utilizzando un'estensione della logica temporale lineare: TLA. Esso è stato concepito più per il design di sistemi concorrenti che per la scrittura di algoritmi. In sostanza, con TLA+ si scrivono una serie di predicati, la cui correttezza viene enunciata da invarianti e teoremi, i quali dovranno essere dimostrati, con dei prover automatici. L'attività per cui viene più utilizzato consiste nel formalizzare un sistema logicamente e verificarne la correttezza tramite l'individuazione di proprietà che dovranno essere dimostrate.

1.4.2 Atelier B

È un software molto ricco e complesso, molto diverso da tutti gli altri citati. Esso è nato per guidare un team nello sviluppo di un software in tutto il suo processo. Sostanzialmente, Atelier B implementa e valida un metodo in modo rigoroso, chiamato B method. È basato sulla logica classica e la teoria degli insiemi e prevede una scrittura formale dei requisiti attraverso macchine a stati, questi requisiti vengono poi rifiniti, implementati in un linguaggio intermedio, questa implementazione viene dimostrata (da prover automatici o interattivi) e infine tradotta in un linguaggio di programmazione finale.

1.4.3 Rocq

Rocq è uno dei proof assistant più famosi e utilizzati. È quindi un software che permette di formalizzare teorie matematiche e programmi informatici, basato sulla logica intuizionista e la teoria dei tipi (piuttosto teoria degli insiemi). È un proof assitant, quindi le dimostrazioni devono essere scritte dall'utente, nel linguaggio riconosciuto dall'ambiente, le quali potrebbero essere validate oppure no, a seconda della correttezza del ragionamento e della sintassi. Il suo linguaggio è piuttosto flessibile e permette di creare ulteriori linguaggi e tecniche di dimostrazione. Rocq è uno strumento utilizzatissimo per dimostrare la correttezza di codice e di teorie matematiche. La sua eccessiva flessibilità, però, rende praticamente necessaria (a meno di esercizi piuttosto semplici) l'uso di librerie che implementino un proprio linguaggio, una propria logica e utlizzando delle tecniche di dimostrazioni avanzate, adatte al linguaggio e alla logica stabiliti. Un esempio di framework per Rocq, è Iris.

1.4.4 Iris

Come annunciato precedentemente, è un framework per Rocq. In quanto tale, utilizza una logica: la separation logic, un'estensione della logica di Hoare che implementa il concetto di stato della memoria, separata tra stack e heap. Iris è un framework efficiente per la scrittura di codice dimostrato. Dispone di una serie di linguaggi adatti a vari scopi. Ci sono linguaggi più minimali, altri più complessi, adatti a determinati usi.
È un proof assistant, quindi le dimostrazioni di correttezza del codice devono essere scritte dall'utente.

1.4.5 Why3

Why3 è un software che utilizza un linguaggio per formalizzare teorie matematiche e programmi. Può essere utilizzato sia con prover automatici (uso comune) che con proof assistant (come Rocq). Il suo linguaggio è WhyML: un linguaggio funzionale arricchito da costrutti di formalizzazione logica basati sulla logica di Hoare.

1.5 La scelta di Why3

Sono stato obbligato alla scelta di uno di questi strumenti. Quest'obbligo deriva dal fatto che per ognuno di essi, la curva di apprendimento è piuttosto alta. Inoltre, ha poco senso testare questi strumenti su casi semplici, perché il mio obiettivo è verificabile solo in esempi pseudo-reali. Quindi, la cosa migliore si è rivelata scegliere uno strumento tra questi e diventare il più bravo possibile ad utilizzarlo, per fornire un'analisi completa dell'uso atipico che ne avrei fatto.
Ognuno di questi programmi, è adatto a determinati scopi. Non cercavo il software migliore, cercavo il software più adatto ai miei scopi. Quindi cercavo un software flessibile ma non troppo lontano da un normale ambiente di programmazione reale, che permettesse di specificare e implementare algoritmi nel modo più naturale possibile. Ho quindi analizzato le caratteristiche di ogni software, cercando di capirne non solo la natura, ma anche i contesti in cui vengono utilizzati. TLA+ ha il problema di essere troppo incentrato sui sistemi concorrenti e di essere basato solo sulle specifiche. Gli step da fare per passare dalle specifiche all'implementazione finale sono tanti e vaghi, quindi il rigore da me ambito non sarebbe pienamente soddisfatto.
Atelier B è stato concepito per creare un intero programma totalmente dimostrato. Il processo è piuttosto formale e rigoroso, ma non è uno strumento adatto a pensare su problemi isolati. Cercavo qualcosa di più flessibile.
Rocq, come anticipato, in molti casi necessita l'utilizzo con di librerie. La flessibilità eccessiva concerne l'obbligo di formalizzazione anche concetti basilari, che risultano triviali nella scrittura di codice tradizionale. I due candidati finali, quindi sono Iris e Why3. Iris permette di formalizzare, attraverso la separation logic, concetti relativi alla memoria molto più efficientemente rispetto a Why3. Iris è perfetto per scrivere codice di cui si vuole avere il controllo totale e la cui traduzione debba essere il più fedele possibile all'originale.
Mi serviva uno strumento che mi permettesse di ragionare al livello di astrazione che si sceglie e avere fiducia nei propri ragionamenti attraverso delle dimostrazio-

ni con dei tempi ragionevoli. Dunque, la flessibilità del linguaggio WhyML e la possibilità di dimostrazioni automatiche, mi ha spinto nella scelta di Why3.

Capitolo 2

La piattaforma Why3

2.1 Caratteristiche e funzionamento

Why3 è una piattaforma di verifica formale di programmi. Fornisce un ricco linguaggio di specifica e programmazione, chiamato WhyML e si basa su dimostratori di teoremi esterni (automatici o interattivi). Why3 è dotato di una libreria standard, che include teorie logiche (interi e aritmetica sui reali, operazioni booleane, insiemi, mappe, ecc...) e strutture dati base (array, code, hash tables, ecc...).
Si può interagire con Why3 sia da CLI che da un'IDE. Ho avuto modo di sperimentare entrambi e personalmente mi sono trovato meglio utilizzando Visual Studio Code per la scrittura del codice e l'IDE per la dimostrazione dei goal, per scovare errori di sintassi. Quando si apre un file .mlw o .why con l'IDE, si vedono tre schermate. A destra abbiamo nella parte superiore il codice a cui si fa riferimento, in basso messaggi di errori o successo. A sinistra abbiamo un pannello in cui vengono elencati tutti i goal generati a partire dal codice. Cliccando su di essi, si possono lanciare dei prover automatici. Si può scegliere di avviare un prover specifico, scegliere il tempo di esecuzione massima, come si può scegliere di utilizzare dei procedimenti predefiniti a diversi gradi di profondità, o applicare delle trasformazioni specifiche.
In alternativa, si può decidere di dimostrare interattivamente, c'è un'operazione che scrive una "traduzione" in un linguaggio adatto all'ambiente preferito (ad esempio Rocq), verrà creato un file completo del contesto utile alla dimostrazione. Il programmatore in quel caso dovrà scrivere una dimostrazione e farla verificare. Se questa verifica sarà andata a buon fine, il goal in questione risulterà vero anche nella piattaforma di Why3.
I goal vengono segnalati graficamente come non dimostrati, dimostrati, o falsi (il

prover ha trovato un controesempio).
I goal vengono inferiti automaticamente dal codice e vengono utilizzati come verità nei contesti dei goal successivi a prescindere dall'esito ricevuto dai dimostratori. Questa è una caratteristica molto importante da tenere a mente quando si sviluppa in Why3. Dei goal potrebbero risultare dimostrati perché sopra di loro potrebbero esserci degli enunciati falsi che hanno contribuito alla dimostrazione. Why3 dispone di un piccolo interprete che gli permette di eseguire il codice e di sistemi di traduzione automatica da WhyML a sottoinsiemi di linguaggi di programmazione come C, Python e OCaml.

2.2 Il linguaggio WhyML

Il linguaggio di Why3 è un linguaggio derivante dalla famiglia ML. È di tipo funzionale ma supporta anche costrutti imperativi. Inoltre, permette anche la creazione di funzioni non pure (quindi ci possono essere effetti collaterali, al contrario di altri linguaggi funzionali) e la gestione di puntatori. WhyML possiede una serie di costrutti logici e altri di programmazione classica. Alcuni di questi costrutti sono in comune e possono essere utilizzati in entrambi i contesti.
La divisione di contesto può avvenire anche nei moduli. Esistono due tipi di file, con estensioni diverse: .why e .mlw. Il linguaggio permesso del file .why è un sottoinsieme di WhyML. I file .why servono a scrivere logica pura. Servono a formulare specifiche di programmi e teorie logiche. All'interno di un file .why, le varie definizioni devono essere all'interno di un blocco `theory`, che viene chiuso dalla keyword `end`.
Il linguaggio legale nei file .mlw è WhyML. Quindi, in un file .mlw si possono scrivere sia delle `theory` che dei `module`. Un `module` deve essere chiuso anch'esso dalla keyword `end` e può contenere sia costrutti logici che di programmazione. Quindi, mentre nei file .why sono consentite solo delle specifiche, nei file .mlw si scrivono sia specifiche che implementazioni. Tra i vari costrutti vediamo i più importanti.

Nome: `axiom`
Contesto: logico
Descrizione: Serve a definire un assioma. All'interno di esso possono essere usati solo elementi puramente logici, come funzioni dichiarate con `function`, `val function` e `let function` e predicati (`predicate`).

Nome: `lemma`
Contesto: logico
Descrizione: Serve a definire un enunciato da dimostrare. Non ci sono altre keyword del tipo theorem o corollary. Può contenere gli stessi elementi che può contenere `axiom`

Nome: `function`
Contesto: logico (e possibilmente programmazione)
Descrizione: Serve a dichiarare una funzione pura. Se usata da sola (senza `let` e `val`) può ammettere solo costrutti logici o funzionali (ad esempio `for`, `while` e `ref` non sono permessi) e potrà essere utilizzata solo in contesti logici e non in contesti di programmazione. Nel caso venga affiancata a `let` o `val` ammette tutti i costrutti di programmazione e la funzione così dichiarata potrà essere usata sia in contesti logici che di programmazione. In questo caso, l'uso di function sta a indicare che la funzione non ha effetti collaterali, è quindi pura.

Nome: `predicate`
Contesto: logico (e possibilmente programmazione)
Descrizione: È esattamente una `function` che ha come valore di ritorno un booleano.

Nome: `let`
Contesto: programmazione
Descrizione: Può avere due usi: costrutto `let...in` (classico costrutto `let...in` usato nei linguaggi funzionali come OCaml) e nel contesto di funzioni. Serve a dichiarare una funzione di cui viene scritta anche l'implementazione, oltre alle specifiche. Nel caso venga usata con `function`, deve essere una funzione pura, altrimenti può non esserlo.

Nome: `val`
Contesto: programmazione
Descrizione: Può avere due usi: dichiarare costanti, usando `val constant`, oppure dichiarare funzioni di cui si scrivono solo le specifiche. La purezza della funzione viene determinata come in `let` (in questo caso solo in base al contratto).

Nome: `rec`
Contesto: programmazione

Descrizione: È il modo per indicare che una funzione è ricorsiva. La keyword va scritta subito dopo `let`

Nome: `ghost`
Contesto: programmazione
Descrizione: Serve a dichiarare funzioni (non necessariamente pure) che servono a enunciare proprietà che possano aiutare nella dimostrazione di un contratto di un'altra funzione. Sostanzialmente sono funzioni di servizio atte alla dimostrazione.

Nome: `requires`
Contesto: programmazione
Descrizione: È il modo per definire una precondizione. Si scrive dopo l'intestazione della funzione.

Nome: `ensures`
Contesto: programmazione
Descrizione: È il modo per definire una postcondizione. Si scrive dopo l'intestazione di una funzione.

Nome: `invariant`
Contesto programmazione
Descrizione: È il modo per definire un'invariante di ciclo. Si scrive subito dopo la riga del `for` o del `while`. Serve ad aiutare nella dimostrazione di correttezza di una funzione.

Nome: `assert`
Contesto: programmazione
Descrizione: È il modo per definire un'asserzione. Può essere scritta in qualsiasi punto del corpo di una funzione. Serve ad aiutare nella dimostrazione di correttezza di una funzione.

Nome: `variant`
Contesto: logico
Descrizione: È il modo per indicare una variante. Indica la terminazione di un ciclo o di ricorsione. Va scritta dopo l'intestazione della funzione per funzioni ricorsive, oppure dopo la dichiarazione di un ciclo `while` nel caso imperativo.

2.3 Studiare Why3

All'inizio, studiare Why3 è piuttosto difficoltoso. Approcciare uno strumento del genere richiede molto impegno anche per un programmatore con solidi background matematici. Oltre a questo, la documentazione è piuttosto scarsa. Le uniche risorse disponibili sono il manuale ufficiale, la libreria standard e una galleria di circa 200 esempi. Personalmente, ho trovato molto utile ricopiare pezzi di libreria standard, cercando di capire al meglio ogni singola definizione e provando ad estenderla. Credo sia molto importante perché serve a ereditare uno stile adatto alla piattaforma. La libreria standard, però, non è molto uniforme. Dei concetti simili spesso vengono gestiti in maniera diversa. Ad esempio, nel modulo `list.mlw` c'è una divisione molto granulare dei moduli. In `array.mlw` ci sono molti meno moduli ma le funzionalità coperte sono simili e in un numero paragonabile. I cambi di stile hanno delle implicazioni negative nella programmazione (bisogna consultare più frequentemente la libreria standard, c'è poca uniformità anche nel proprio codice), ma ha implicazioni positive sull'apprendimento di WhyML. Così si ha modo di capire più aspetti del linguaggio, in tutta la sua versatilità. Quindi, non basta studiare pochi moduli. È necessario studiarne tanti per avere un'idea più completa delle capacità del linguaggio e dell'uso che va fatto.
Gli esempi riguardano per lo più implementazioni di algoritmi già noti. Purtroppo non viene fornito molto contesto. Qui, inoltre, gli stili sono davvero numerosi, saltano fuori dei modi di programmare ancora diversi rispetto a quelli della libreria standard. Inoltre, quasi tutti gli esempi riguardano singoli algoritmi e non programmi interi, più complessi. I pochi software scritti interamente in Why3 riguardano per lo più l'ambito della verifica formale, o comunque contesti puramente logici o algoritmici. Non ci sono esempi di programmi simili a quelli di genere più comune: software che interagiscono con database, utenti, filesystem, system call.

2.4 Il mio uso di Why3

Il motivo per cui non ci sono esempi di software scritti in WhyML colmi di interazioni con utente, database, sistema, consiste nel fatto che Why3 è stato concepito per scrivere programmi interamente dimostrati, e questi elementi da me appena citati, sono piuttosto difficili da astrarre. Soprattutto, variano di linguaggio in linguaggio. Ad esempio, l'interazione con l'utente è estremamente difficile da formalizzare logicamente ma estremamente semplice da programmare. Le interazioni col database consistono spesso in query dichiarative, la cui formalizzazione

è molto macchinosa e la scrittura di queste non rappresenta un grande problema. Le interazioni col sistema, possono avvenire in modo molto diverso tra i vari linguaggi. Di nuovo, formalizzare le interazioni col filesystem, o la gestione di processi, o la gestione di dispositivi, può risultare estremamente complicato. In questo caso, però, i problemi possono essere molto più complessi rispetto a gestione dell'utente e query dichiarative.
Allora, perché credo che Why3 sia uno strumento utile? Per cosa si può utilizzare? Quando si scrive un software, le componenti esterne e meno pregne di logica, sono sempre presenti.
Il mio obiettivo, come detto in precedenza, non è quello di sostituire i test con Why3 in toto. Esistono già metodi per scrivere software totalmente dimostrato. Esistono già software totalemente verificati.
Io utilizzo Why3 per aiutarmi a ragionare e a verificare le proprietà del mio ragionamento. Piuttosto che pensare ad un'implementazione e cercare di esprimere le mie idee direttamente nel linguaggio di programmazione finale, in molti casi necessito di un passaggio intermedio. Spesso, questo passaggio avviene scrivendo su carta, il che rimane utile. A volte questo non basta. Quindi l'uso che faccio di questo strumento si allontana dallo scopo per cui è stato creato.
Questo è un uso diverso, di cui non ho trovato traccia in letteratura. A causa di ciò, sono stato costretto a una sperimentazione personale. Ho pensato che non bastasse fare un po' di pratica con Why3 nel modo in cui lo concepisco, bensì ho creduto fosse necessario esercitare questo uso in due tipi di casi: su problemi isolati e su un progetto reale. Da cui derivano i capitoli successivi.

Capitolo 3

Le esercitazioni su problemi isolati

3.1 La modalità con cui sono stati svolti

Ho deciso di praticare l'uso di Why3 svolgendo esercizi in due modalità:

- provando a risolvere quelli ufficiali e studiando le soluzioni;
- provando a risolvere problemi della piattaforma leetcode.

Credo che queste due modalità siano complementari ed entrambe necessarie.

3.1.1 Gli esempi ufficiali

Gli esempi ufficiali, seppur raccolti per argomenti, sono piuttosto discontinui. Non c'è una progressione lineare. Ci sono pochi esercizi per ogni argomento e la difficoltà tra questi varia enormemente. Studiare questi esempi, però, è essenziale per capire i propri errori, soprattutto se si è nuovi alla scrittura di specifiche matematiche del codice. All'inizio è difficile scrivere specifiche corrette. Inoltre, non si ha idea delle proprietà necessarie alla dimostrazione del codice. Ad esempio, all'inizio, sono stato bloccato nella scrittura di proprietà di un banalissimo insertion sort. Di seguito l'esempio ufficiale:

```
module InsertionSort

  (* Import vari omessi *)

  let rec insert (x: elt) (l: list elt) : list elt
    requires { sorted l }
    ensures  { sorted result }
```

```
    ensures  { permut (Cons x l) result }
    variant  { l }
  = match l with
    | Nil → Cons x Nil
    | Cons y r → if le x y then Cons x l else Cons y (insert x r)
    end

  let rec insertion_sort (l: list elt) : list elt
    ensures { sorted result }
    ensures { permut l result }
    variant { l }
  = match l with
    | Nil → Nil
    | Cons x r → insert x (insertion_sort r)
    end

end
```

Come si vede, le proprietà sono semplicissime. Sia le specifiche che il codice sono semplicissimi. Avevo già letto molti esempi, e di difficoltà superiore. Avevo ormai dimestichezza con la sintassi. Avevo capito tutti gli esempi letti, avevo persino esteso brevemente qualche modulo. Adesso mi ritrovavo a essere completamente spaesato. Ho sentito le stesse sensazioni che provavo quando programmavo per la prima volta. Sentivo di aver accumulato una serie di strumenti e di averne capito le funzionalità, ma non avevo idea di come utilizzarli. Non riuscivo a esprimermi, ne sapevo cosa dovessi esprimere. Qui ho capito che scrivere codice con proprietà è un'attività molto più difficile rispetto alla scrittura di codice tradizionale. Mi veniva naturale scrivere il codice ma non riuscivo a capire cosa dovesse fare quella funzione, esattamente. Notavo, così, che il mio modo di programmare era vago. Ero abituato a scrivere codice senza prima pensare cosa avrebbe dovuto fare esattamente la funzione che stavo sviluppando.

Ad esempio, per l'insertion sort, ho scritto l'implementazione corretta al primo colpo, ma non riuscivo a dimostrarla. Avevo perso pochissimo tempo per scrivere la funzione, tantissimo per formalizzarla e verificarla. Intuivo, quindi, che avrei dovuto imparare un'attività proprio diversa da quella che avevo sempre svolto. A mio avviso, c'è proprio un passaggio di classe. Stanco delle mie formalizzazioni errate (che ovviamente non venivano dimostrate), avevo provato a semplificare, consultando la libreria standard, per vedere se ci fossero predicati o funzioni che mi avrebbero aiutato. Trovando il predicato `sorted`, ho scritto una singola postcondizione, che indicava che la funzione fosse ordinata, e veniva dimostrata. A questo punto, ero convinto di aver fatto un buon lavoro, perchè la funzione

scritta risultava dimostrata. Guardando la soluzione ufficiale, però, mi ero reso conto che non avevo riportato la proprietà che diceva che la lista in output dovesse essere una permutazione della lista in input. A quel punto vedevo un risultato dimostrato, la cui implementazione era giusta, ma che non esprimeva le reali specifiche della funzione. Avevo dunque commesso un errore e la dimostrazione riuscita mi fuorviava.

Conclusione

Una dimostrazione di correttezza di una funzione implementata indica che il codice è sufficiente a dimostrare le precondizioni e le postcondizioni riportate, le quali, potrebbero essere un sottoinsieme di quelle che potrebbero essere dimostrate.

Si nota che leggere la soluzione dell'esempio ufficiale è stato fondamentale. Se non avessi visto esempi che indicassero i miei errori sulla deduzione delle proprietà, mi sarei fidato del risultato conferitomi dal prover automatico. Avrei potuto scrivere un'implementazione errata ma che era sufficiente a dimostrare che la lista in output sarebbe stata ordinata. Sarei stato convinto di un ragionamento errato e avrei tradotto il codice nel linguaggio di programmazione di destinazione convinto della correttezza del mio codice. È importante segnalare in che modo un codice risulti corretto.

Nota

Non esiste una correttezza assoluta del codice, quindi è errato dichiarare che del codice sia corretto o dimostrato. È giusto dichiarare che, nel caso lo sia, del codice sia corretto rispetto alla specifica, ovvero, il codice è sufficiente a dimostrarla.

Consiglio

Inizialmente, consiglio l'utilizzo di esempi molto semplici da implementare: permettono di concentrarsi totalmente sulla deduzione delle proprietà piuttosto che sulla scrittura del codice.

Un caso importante

C'è stato un caso abbastanza importante, che vale la pena riportare. Tra i vari esempi, mi sono imbattuto in una BFS per grafi. Ho ricopiato il codice Why3, completo di tutte le sue proprietà. Lanciando i prover, ho notato che alcuni dei risultati non venivano dimostrati. Non capivo il perché: nella pagina da cui

ho preso l'esempio, come nella maggior parte degli esempi, c'era il report che mostrava che tutti i goal generati erano stati dimostrati. I miei prover, invece, non riuscivano a dimostrare molti dei goal. Allora, ho analizzato in modo più approfondito il report, confrontandolo con i risultati ottenuti dall'esecuzione dei miei prover. Ho notato, prima di tutto, che sono state utilizzate delle tecniche di trasformazione avanzate, attraverso le quali i prover sono stati molto aiutati. Infine, una postcondizione della BFS, è stata addirittura dimostrata in modalità assistita con Rocq. Applicando le dimostrazioni segnalate passo passo, mi era rimasta da dimostrare, effettivamente, solo il risultato che era dimostrato con Rocq, che i prover non riuscivano a dimostrare in nessun modo.
È una semplice BFS, è un algoritmo di cui si conosce la correttezza ma è piuttosto difficile da dimostrare in Why3. A volte, vedendo il proprio risultato non dimostrato, si può pensare subito a degli errori commessi nell'implementazione o nelle specifiche. A volte, invece, i prover possono non essere abbastanza capaci. Provare a dimostrare gli esempi nel proprio ambiente risulta molto utile, permette di individuare i casi in cui i prover sono più in difficoltà.

3.1.2 Leetcode

Quando ho cominciato a sviluppare un certo intuito rispetto all'individuazione di proprietà (sia nelle precondizioni, che nelle postcondizioni che nelle invarianti), ho cominciato a testare le mie capacità su un percorso più lineare, attraverso la piattaforma leetcode. Leetcode è un sito in cui ci sono migliaia di problemi di vario genere. Si può scegliere il linguaggio di progammazione che si mastica più facilmente, ce ne sono tanti. La soluzione proposta verrà verificata tramite dei test lanciati dalla piattaforma.
WhyML è molto poco conosciuto e non è un normale linguaggio di programmazione, quindi non rientra tra i linguaggi disponibili da leetcode. Questo è un pro. Il mio obiettivo consiste nell'utilizzare WhyML e Why3 per ragionare, quindi ho approcciato i problemi leetcode in questo modo: scrivevo del codice in WhyML, lo verificavo e, una volta verificato, scrivevo una traduzione in un linguaggio a scelta.
Ad esempio, è quello che è avvenuto per il primo problema affrontato, che è stato semplicissimo da risolvere e da trasporre. Il problema richiedeva lo sviluppo di un algoritmo che trovasse duplicati in un array. Di seguito la soluzione WhyML verificata:

```
let function contains_duplicate (a: array int) : bool
```

```
  ensures { (exists i j: int. 0 ≤ i < j < length a ∧ a[i] = a[j]) ↔ result =
     true }
  ensures { (forall i j: int. 0 ≤ i < j < length a → a[i] ≠ a[j]) ↔ result
    = false }
=
  let values = empty () in
  for i = 0 to length a - 1 do
    invariant { forall j k: int. 0 ≤ j < k < i → a[j] ≠ a[k] }
    invariant { forall x: int. mem x values ↔ (exists k: int. 0 ≤ k < i ∧ a[
    k] = x) }
    if mem a[i] values then return true
    else add a[i] values
  done;
  return false
```

E la soluzione C++ che ha passato i test leetcode:

```
bool containsDuplicate(vector<int>& nums){
  set<int> values = {};
  for(int i = 0; i < nums.size(); i++){
    if(values.contains(nums[i])) return true;
    else values.insert(nums[i]);
  }
  return false;
}
```

L'unica difficoltà nella scrittura di questo codice è avvenuta nella traduzione nel linguaggio di destinazione. Si vedono delle differenze, naturalmente, tra le due implementazioni. Le operazioni sugli insiemi, ad esempio, vengono denotate in modo diverso. Quindi, nel caso saltino fuori degli errori, la prima cosa da controllare è la corrispondenza tra le due implementazioni. Bisogna verificare, in questo caso, che le specifiche delle proprie funzioni sugli insiemi corrispondano con quelle definite dalla libreria standard di C++.

3.2 Aiutarsi con gli LLM

In esempi successivi, mi sono ritrovato a non avere idea di come formalizzare un problema, pur avendo un'idea della soluzione finale. Dunque, ho provato ad interrogare degli LLM a riguardo. Ne ho utilizzati vari. Ho utilizzato sia versioni premium che versioni più leggere. Ho notato, che gli LLM sono piuttosto scarsi a scrivere codice verificato, anche per problemi molto semplici. Commettono errori grossolani. Gli errori riguardo la sintassi di WhyML sono comprensibili, perché è

un linguaggio poco utilizzato. I problemi più grandi riguardavano, però, le proprietà scovate, che erano spesso errate o insufficienti.
Quindi, concludo che l'esperienza avuta con gli LLM, sia stata piuttosto positiva. Grazie alle loro scarse capacità mi hanno permesso di sviluppare un senso critico spiccato nei loro confronti. In precedenza li ho utilizzati soprattutto in contesti in cui erano piuttosto bravi, sbagliavano raramente. Adesso mi fornivano soluzioni corrette molto raramente. Ciò mi ha permesso di utilizzarli come un interlocutore molto poco esperto, che mi potesse dare degli spunti di cui non ero a conoscenza o che non mi erano venuti in mente. Inoltre, quando non avevo idea di come continuare, scrivendo le domande, e leggendo le risposte, avevo modo di pensare per più tempo al problema, mi davo il tempo per pensare. Quando si vuole ottenere una soluzione, si è spesso inclini a cercare di trovarla subito, invece, spesso, è più utile capire le proprietà del problema, affinchè la soluzione venga fuori in modo più naturale. L'interazione con gli LLM, in questo ambito, aiuta il programmatore ad esplorare il problema in maniera semi-autonoma, piuttosto che affidarsi a del codice non capito.
Se gli LLM fossero molto bravi a scrivere codice verificato, il lavoro del programmatore muterebbe sostanzialmente. Come affermato in precedenza, il codice è corretto rispetto alle specifiche, quindi il ruolo del programmatore si riducerebbe alla scrittura delle specifiche, in modo più o meno granulare. Al momento mi sembra che siamo piuttosto distanti da questo scenario.

3.3 Difficoltà riscontrate nella risoluzione di un problema

Svolgendo molti esercizi, sono riuscito a impratichirmi nella scrittura delle specifiche. Finalmente, ho raggiunto un punto in cui mi capitava spesso di riuscire a scrivere il contratto di funzioni molto più velocemente rispetto alla loro implementazione. Questo, però, varia da problema a problema. Si può diventare più bravi a specificare che a programmare ma alcune funzioni, in certi contesti, richiedono una quantità di codice di specifica maggiore rispetto a quella richiesta dall'implementazione. Inoltre, alcune funzioni sono più comode da implementare scomponendole in sottofunzioni e, di alcune di queste, un'eccessiva formalizzazione può rappresentare una perdita di tempo.
Queste deduzioni nascono dallo svolgimento di un esercizio leetcode, chiamato group anagrams.

3.3.1 Il problema

Dato un array di stringhe, raggruppare gli anagrammi.
Esempio
Input: `["eat","tea","tan","ate","nat","bat"]`
Output: `[["bat"],["nat","tan"],["ate","eat","tea"]]`
Quindi l'output risulta un array di array, in cui ogni array contiene degli anagrammi. Gli anagrammi, in realtà, in questo caso, vengono visti come delle semplici permutazioni, non è richiesto che ogni parola debba avere un senso in una data lingua.

3.3.2 La modalità di risoluzione

Per questo problema, ispirato da alcune soluzioni di problemi tra gli esempi ufficiali, ho deciso di definire prima le specifiche in un modulo separato, per poi ideare una soluzione, implementare dei sottoproblemi. Ho adottato quindi un approccio top-down. È il modo in cui ho agito più spesso in programmazione.

3.3.3 La formalizzazione

Per scrivere un contratto della funzione pulito e ben pensato, ho ritenuto necessario scomporre la specifica in funzioni e predicati. Sono servite quasi 50 righe di codice, solo per le specifiche. Non è stato affatto un processo difficoltoso, ma abbastanza lungo rispetto a ciò che ci si aspetta. Di seguito il modulo:

```
module GroupAnagramsSpec

  use int.Int
  use array.Array
  use array.ArrayPermut

  type str = array int

  predicate all_permut (a: array str) =
    forall i j: int, s1 s2: str. i ≠ j
      → 0 ≤ i < length a ∧ s1 = a[i]
      → 0 ≤ j < length a ∧ s2 = a[j] → permut_all s1 s2

  predicate all_all_permut (a: array (array str)) =
    forall i: int. 0 ≤ i < length a → all_permut a[i]

  use int.Sum
```

```
  function total_length (a: array (array str)) : int =
    sum (fun i → (length a[i])) 0 (length a)

  function offset (a: array (array str)) (i: int) : int =
    sum (fun k → length a[k]) 0 i

  val function flatten (a: array (array str)) : array str
    ensures { total_length a = length result }
    ensures { forall i j: int.
                0 ≤ i < length a →
                0 ≤ j < length a[i] →
                result[offset a i + j] = a[i][j] }

  predicate no_other_permut (a: array (array str)) =
    forall i k j t: int.  i ≠ k
      → 0 ≤ i < length a → 0 ≤ k < length a
      → 0 ≤ j < length a[i] → 0 ≤ t < length a[k]
      → not permut_all a[i][j] a[k][t]

  val function group_anagrams(strs: array str) : array (array str)
    ensures { all_all_permut result }
    ensures { permut_all strs (flatten result) }
    ensures { no_other_permut result }

end
```

In questo caso, il problema mi era già molto chiaro, scrivere così tante righe di specifica per un singolo contratto di funzione non mi ha aiutato a capire meglio il problema, perché il problema mi era già stato spiegato in modo esaustivo nella piattaforma leetcode. Si nota quindi come il processo di specifica sia necessario nei casi in cui non si siano capite a pieno le proprietà di un problema. In questo caso, però stavo risolvendo un problema che mi era stato assegnato. Spesso, invece, ai programmatori vengono assegnate delle task più o meno precise, da cui il programmatore individuerà il problema da risolvere. Non appena un problema venga individuato, risulta spesso vago, per cui risulta necessario formalizzarlo, per capirlo al meglio. Da qui deduco che per esercitarsi a scrivere specifiche, utilizzare piattaforme come leetcode, che danno una spiegazione esaustiva del problema, può essere molto utile. Ritengo sia un buon metodo per prepararsi, perché in questo caso l'esercitazione si limita ad una traduzione: dal linguaggio naturale (anche se schematico), al linguaggio logico.

3.3.4 La soluzione

Per questo problema, ho deciso di adottare una soluzione più macchinosa, che pensavo potesse avere un costo computazionale più basso.
L'idea che avevo era abbastanza vaga. La soluzione si reggeva sul fatto che una stringa può essere tradotta in un multiinsieme, quindi delle stringhe che erano tra loro degli anagrammi, venivano tradotte nello stesso multiinsieme. A questo punto pensavo, scorrendo l'array in input, di salvare in una lista delle mappe. Queste mappe avrebbero avuto come chiave un multiinsieme e come valore una lista contenente gli anagrammi. È una soluzione che in un linguaggio di programmazione normale avrebbe avuto una certa difficoltà nell'essere implementata, in WhyML è certamente ancora più difficile, perché ogni sottofunzione sarebbe dovuta essere dimostrata. Come anticipato, senza pensarci, ho utilizzato l'approccio top-down sempre utilizzato quando programmavo.
Il modulo sui multiinsiemi era piuttosto povero, quindi ho dovuto riformularlo. Per avere il giusto numero di operazioni a disposizione, ho dovuto scrivere circa 70 righe di specifiche:

```
module Multset

  use int.Int
  use bag.Bag
  type elt

  val eq (x y: elt) : bool
    ensures { result ↔ x = y }

  type multset = abstract {
    to_bag: bag elt;
  }
  meta coercion function to_bag

  val function nb_occ (x: elt) (b: multset) : int
    ensures { result = nb_occ x b }

  val function mem (x: elt) (b: multset) : bool
    ensures { result ↔ mem x b }

  val function (==) (a b: multset) : bool
    ensures { result ↔ a == b }

  val function empty_multset : multset
    ensures { result = empty_bag }
```

```
    ensures { card result = 0 }
    ensures { forall x: elt. nb_occ x result = 0 }

  val function singleton (x: elt) : multset
    ensures { result = singleton x }

  val function union (a b: multset) : multset
    ensures { result = union a b }

  val function add (x: elt) (b: multset) : multset
    ensures { result = add x b }
    ensures { card result = card b + 1 }
    ensures { forall y: elt. nb_occ y result = if y = x then nb_occ y b + 1
    else nb_occ y b }

  val function card (b: multset) : int
    ensures { result = card b }

  val function diff (a b: multset) : multset
    ensures { result = diff a b }

  val function inter (a b: multset) : multset
    ensures { result = inter a b }

  val function choose (b: multset) : elt
    ensures { result = choose b }

end

module IntMultset
  use int.Int
  use bag.Bag
  clone export Multset with type elt = int, val eq = Int.(=), axiom .
end

module MapImpMultset
  use IntMultset
  clone export fmap.MapImp with type key = IntMultset.multset, val eq =
    IntMultset.(==)
end
```

Ho voluto svolgere un lavoro coerente, quindi non ho scritto solo le funzioni che mi sarebbero servite in questo caso. Ho scritto le funzioni che possono essere utili quando si utilizzano dei multiisiemi.
Poi, ho cominciato a scrivere le sottofunzioni che pensavo potessero essere utili.

Per specificare, ho dovuto definire altre funzioni logiche. Inoltre, i sottoproblemi individuati, non sono stati impossibili da risolvere ma mi hanno fatto impiegare una discreta quantità di tempo. Di seguito le funzioni ausiliarie (di cui non riporto le funzioni logiche per pulizia):

```
let rec nb_occ_lu (x: int) (s: str) (l u: int) : int
  requires  { 0 ≤ l ≤ u ≤ length s }
  ensures   { result = num_occ_str_l x s l u }
  variant   { u - l }
=
  if u = l then 0
  else
    let count = nb_occ_lu x s (l+1) u in
    if s[l] = x then count + 1 else count


let string_to_multset (s: str) : multset
  ensures { forall i: int. 0 ≤ i < length s → IntMultset.mem s[i] result }
  ensures { forall x: int. nb_occ x result = num_occ_str x s }
=
  let ref m = empty_multset in
  assert { forall x: int. nb_occ x m = 0 };
  for i = 0 to length s - 1 do
    invariant { forall j: int. 0 ≤ j < i → IntMultset.mem s[j] m }
    invariant { forall x: int. nb_occ x m = num_occ_str_l x s 0 i }
    label L in
    m ← IntMultset.add s[i] m;
    done;
  m

let rec add_to_distinct_list (x: multset) (l: list multset) : list multset
  requires  { distinct l }
  ensures   { distinct result }
  ensures   { Mem.mem x l → Length.length result = length l }
  ensures   { not Mem.mem x l → Length.length result = length l + 1 }
  ensures   { Mem.mem x result }
  variant   { length l }
=
  match l with
  | Nil → Cons x Nil
  | Cons y r → if IntMultset.(==) y x
               then begin
                 assert { Mem.mem x l };
                 l
               end
```

```
                else begin
                  assert { not Mem.mem y r };
                  Cons y (add_to_distinct_list x r)
                end
  end
```

3.3.5 Una presa di coscienza

Adesso, però, non capivo bene come assemblare queste funzioni. Soprattutto, mi ero reso conto, che il costo computazionale non sarebbe stato inferiore a quello di una soluzione molto più semplice ed elegante. Avevo formalizzato un modulo, scritto funzioni e dimostrate, ma non avevo ancora risolto il mio problema, che, adesso, intuivo non avere neanche un costo computazionale ottimale. Avevo fatto un errore di calcolo, perché nella costruzione dell'output, avrei dovuto scorrere la struttura dati creata (la lista di mappe) una sola volta. Invece, adesso, mi rendevo conto che, per la creazione di questa struttura dati, avrei dovuto comunque scorrere la lista più volte, in cerca del multiinsieme corretto per la stringa trovata.
Avevo scritto circa 200 righe di codice, per risolvere un problema in un modo troppo e inutilmente complicato. Gran parte del tempo è stato speso a causa di Why3, perché avevo l'obbligo di formalizzare. La formalizzazione e la dimostrazione sono state le attività che hanno richiesto più tempo. A questo punto sono stato assalito da dubbi, che hanno generato una serie di domande, che sono state fondamentali per trovare un uso corretto di questo strumento.

Domanda

Perché utilizzare uno strumento come Why3?

Domanda

Quando utilizzarlo?

Domanda

Come va utilizzato?

Domanda

In che modo migliorano il processo di programmazione nella quotidianità?

Domanda

A lungo termine, si scrive codice più velocemente (prevenendo gli errori) o più lentamente (lo sforzo nella formalizzazione sarebbe superiore a quello nella risoluzione degli errori)?

3.4 L'ideazione del metodo

Tutte queste domande mi hanno costretto a cercare un uso più consono di questo strumento, che potesse realmente essere utile allo scopo che mi ero prefissato. Ho avuto modo di capire le vere potenzialità di Why3 proprio scoprendo i suoi limiti. Ho cercato quindi di ideare un metodo che provasse a soddisfare le domande formulate. Con esso, cerco di risolvere le difficoltà affrontate sfruttando delle peculiarità di Why3. Di seguito uno schema della strategia ideata:

1. Individuare un problema;
2. Scrivere le specifiche;
3. Cominciare a scrivere un'implementazione;
4. Se si incappa in un sottoproblema, scrivere, della funzione ausiliaria, solo il contratto, senza implementazione, utilizzando la notazione con `val`;
5. Finita l'implementazione della funzione principale, dimostrarla. Una volta dimostrata, implementare un sottoinsieme delle funzioni ausiliarie, seguendo il procedimento dal punto 3.

Esempio

1. Individuo il problema di ordinare una lista;
2. Scrivo le specifiche:

```
val insertion_sort (l: list elt) : list elt
  ensures { sorted result }
  ensures { permut l result }
```

3. Comincio a scrivere un'implementazione con gli strumenti che ho a disposizione:

```
let rec insertion_sort (l: list elt) : list elt
    ensures { sorted result }
    ensures { permut l result }
    variant { l }
  = match l with
    | Nil → Nil
    | Cons x r → insert x (insertion_sort r)
    end
```

4. Sono incappato in una funzione ausiliaria (`insert`), scrivo solo il contratto:

```
val insert (x: elt) (l: list elt) : list elt
  requires { sorted l }
  ensures  { sorted result }
  ensures  { permut (Cons x l) result }

let rec insertion_sort (l: list elt) : list elt
  ensures { sorted result }
  ensures { permut l result }
  variant { l }
= match l with
  | Nil → Nil
  | Cons x r → insert x (insertion_sort r)
  end
```

5. Dimostro la funzione principale. La funzione viene dimostrata. Ritengo sia necessario implementare anche la funzione ausiliaria:
   (a)

```
let rec insert (x: elt) (l: list elt) : list elt
  requires { sorted l }
  ensures  { sorted result }
  ensures  { permut (Cons x l) result }
  variant  { l }
= match l with
  | Nil → Cons x Nil
  | Cons y r → if le x y then Cons x l else Cons y (insert x r)
  end
```

   (b) Dimostro la funzione ausiliaria. Viene dimostrata. Non ho dovuto usare funzioni ausiliarie, ho finito.

Questo metodo è praticabile grazie a delle caratteristiche di WhyML e dell'ambiente Why3. WhyML permette di definire funzioni di cui è riportato unicamente

il contratto, con `val`. Queste funzioni possono essere utilizzate in funzioni successivamente definite, seppur prive di corpo. Il corpo delle funzioni è utile alla dimostrazione solo del contratto della funzione implementata, non contribuisce al contesto delle dimostrazioni di funzioni definite successivamente. Quindi, al contrario che nella programmazione classica, per validare un proprio ragionamento, basta formalizzare le funzioni ausiliarie piuttosto che implementarle. Questo approccio permette di concentrarsi su un problema alla volta, favorendo un approccio top-down. Questo è un approccio più difficile da seguire in programmazione classica. Non si possono testare funzioni che utilizzano delle sottofunzioni prive di implementazione. In questo senso, quindi, l'uso di Why3 rappresenta non solo un vantaggio nella formulazione di una soluzione corretta, ma anche nell'efficienza. Ricollegandoci alla corrispondenza che sussiste tra matematica e programmazione, vediamo che questo metodo segue un approccio top-down come avviene spesso nella costruzione di dimostrazioni. La lettura del codice, come la lettura della matematica, avviene spesso bottom-up, ma la scrittura e la lettura sono processi completamente diversi.
Ora, alcune delle domande formulate possono avere risposta.

1. Perchè utilizzare uno strumento come Why3?
   Per avere fiducia nei propri ragionamenti prima di scrivere codice.

2. Quando utilizzarlo?
   Quando si incappa in un problema di cui si intuisce una complessità difficile da risolvere direttamente.

3. Come va utilizzato?
   Usando il metodo sopra riportato

La risposta alla domanda 1 viene dall'intuizione iniziale. La risposta alla domanda 3 può avere un certo rigore, dato che segue un metodo abbastanza strutturato. La risposta 2, invece, risulta abbastanza arbitraria e la domanda è quella che bisogna porsi anche per ogni singolo sottoproblema in cui si incappa risolvendo un problema più grande. Ritengo sia giusto che la scelta attraverso la quale si decida di implementare (e quindi dimostrare) una funzione in Why3 debba essere decisa in base alle sensazioni del programmatore. Egli deve fare una stima mentale dell'importanza e del numero di errori generabile da un'implementazione errata e una stima sullo sforzo impiegato a formalizzare, implementare, dimostrare e trascrivere la stessa funzione. Deve sviluppare un certo fiuto per le difficoltà che la sua persona potrà affrontare su determinati problemi. Penso che questo

fiuto possa essere sviluppato più facilmente imparando e utilizzando strumenti di verifica formale.

3.5 Necessità di sperimentazione su un progetto reale

Le ultime due domande nei riquadri, invece, non trovano ancora risposta. Per trovarla, ho individuato la necessità di sperimentare questo uso di Why3 in un progetto reale. Scrivere un progetto è un processo più lungo e, le domande cui faccio riferimento, richiedono una sperimentazione più consistente nello stesso contesto. Ho quindi svolto dei progetti simili in due modalità diverse: la modalità classica, senza uso di strumenti di verifica formale e una modalità che integra Why3 nell'uso sopra descritto. Nel capitolo successivo, riporto informazioni più dettagliate riguardo i progetti.

Capitolo 4

Il Progetto

4.1 Requisiti

Che tipo di progetto avrei dovuto svolgere?
Il mio obiettivo consiste in un uso appropriato al mio scopo di Why3 e avevo bisogno di testarlo in un progetto che somigliasse ad un caso reale, che potesse essere il più versatile possibile. Nel capitolo 2 spiego che gli esempi di progetti online riguardano prettamente l'ambito logico, che è un caso abbastanza specifico e poco diffuso. Io volevo verificare la versatilità di Why3 in un progetto che avesse delle interazioni frequenti con l'utente (che avviene in molti software) e che avesse di certo delle componenti pregne di logica. Come descritto nel capitolo precedente, utilizzo Why3 solo nei problemi in cui ritengo che una dimostrazione preventiva possa essere più efficiente rispetto ad una riparazione di errori trovati in futuro. Mi interessa che la metodologia da me proposta possa essere più versatile possibile, quindi avevo bisogno di un esempio abbastanza variegato, in cui avessi la possibilità di incontrare le giuste difficoltà che mi avrebbero aiutato a raggiungere delle conclusioni significative. Infine, volevo che il mio progetto fosse comunque qualcosa di inerente allo sviluppo software, volevo un programma che aiutasse il programmatore in qualche modo.

4.2 Il progetto scelto

Spinto dalle esigenze espresse, ho individuato un problema, da cui avrei poi ricavato un software. Il problema consiste nella gestione di pezzi di codice che si ritengono significativi. A volte capita di dover scrivere algoritmi simili ad altri che si sono già scritti in precedenza. Si è spinti quindi alla ricerca di quegli snippet, che sono da qualche parte nei tanti progetti svolti in precedenza. A volte questa

ricerca può essere abbastanza lunga, a volte si decide di abbandonarla, dichiarandosi sconfitti. Il software da me proposto prova a mitigare questo problema.
Il programma è sostanzialmente uno snippet manager basato su metadati. Esistono già già degli snippet manager ma tutti quelli che ho trovato propongono un sistema di salvataggio a directory. Io volevo trovare i vari snippet di codice attraverso i concetti, da cui l'integrazione dei metadati.

4.2.1 Specifiche

L'utente deve poter inserire, modificare, eliminare e soprattutto recuperare snippet e metadati. Uno snippet ha un nome, un'estensione, un contenuto e un insieme di metadati associati. I metadati hanno un nome e una categoria. Gli snippet possono avere solo metadati della stessa categoria. Esistono due sole categorie: linguaggio e concetto. Gli snippet di concetto sono pezzi di codice che riguardano un concetto trasversale ai linguaggi e che ha quindi a che fare con la programmazione in generale, ad esempio: ordinamento, ricorsione, concorrenza, alberi, algoritmo. Gli snippet di linguaggio sono pezzi di codice che riguardano un concetto inerente ad un linguaggio, ad esempio: classe, monade, eccezione, decoratore, while.
Ho deciso di fare questa divisione perché a volte mi capita di dover cercare pezzi di codice che riguardano concetti che non uso da un po' di tempo; a volte capita di dover riutilizzare un certo linguaggio di programmazione dopo tanto tempo e, in questo caso, può essere utile recuperare del vecchio codice da cui è facile rispolverare certi concetti.
I metadati sono legati da una relazione gerarchica. Nel database, quindi, c'è una foresta di metadati. Allora, perché, ritengo che i dati vengano organizzati in modo diverso rispetto agli snippet manager tradizionali? Perché in questi, sono gli snippet ad essere organizzati in directory. Nel mio programma, gli snippet sono associati a una serie di metadati, i quali sono organizzati in maniera gerarchica. Il fine ultimo è cercare gli snippet e questa ricerca deve avvenire in modo concettuale. Perciò, la struttura gerarchica dei metadati serve solo a preservare una coerenza nell'astrazione dei concetti, che deve essere modellata in modo appropriata attraverso l'organizzazione dei metadati. Nel progetto, per cercare uno snippet, nel caso più comune, si inserisce una lista di metadati appropriati.
L'inserimento di snippet e metadati può avvenire singolarmente oppure, nel caso dei metadati, se ne può inserire una foresta. Questo deriva dalla natura gerarchica della relazione che c'è tra i metadati.
Ci sono diverse get, pensate per navigare efficientemente tra i concetti (uso con-

cetto come astrazione superiore di metadato). Dato un metadato, si possono ottenere tutti i concetti più specifici (get dell'albero), risalire a tutti i concetti più generali (get del percorso dalla radice al metadato), risalire a tutti i concetti allo stesso livello di genericità (get di tutti i fratelli). Per ottenere gli snippet, si può cercare attraverso il nome (che deve essere esatto), oppure attraverso i metadati. Data una lista di metadati, esistono due get diverse. Una get che restituisce tutti gli snippet che contengono tutti i metadati specificati; una get che restituisce tutti gli snippet che contengono almeno uno dei metadati specificati.
Questo software si rivela utile nel caso l'utente si impegni a organizzare in modo sensato i metadati e associ agli snippet i metadati corretti. Il focus principale è la ricerca, che concerne un impegno maggiore nell'organizzazione della foresta di metadati e nell'inserimento di snippet con caratteristiche appropriate.
Ho sviluppato il software in due forme e metodologie diverse. Ho avuto l'opportunità di creare, per il tirocinio, un MCP server con circa queste specifiche, senza l'uso di Why3. Successivamente, ho creato un altro snippet manager con specifiche molto simili ma con interazione da cli, usando Why3.

4.3 Il progetto svolto nel tirocinio

Lo snippet manager svolto nel tirocinio è un MCP server. Un MCP server è un server basato sul protocollo MCP (Model Context Protocol), sviluppato da Anthropic. È un protocollo che permette di sviluppare facilmente servizi con cui gli LLM siano in grado di interagire. Lo sviluppo del server consiste sostanzialmente nella creazione di una serie di tool e risorse (come se fossero endpoint di un'API) che possano essere invocati, in formato JSON, dagli LLM.
Ho utilizzato node.js con typescript, l'SDK ufficiale di Anthropic per gli MCP server e un database a grafo, neo4j. Sviluppare MCP server è piuttosto semplice. È molto simile a qualsiasi altro software. Esporre tool e risorse con l'SDK ufficiale è stata la parte più semplice del progetto, non ha mai generato problemi, ne errori difficili da trovare. La parte complicata del progetto, quindi, risiedeva nella logica, nella gestione di snippet e metadati nel db, nella formulazione di query sensate e ben concatenate.

4.4 Il progetto svolto per la tesi

Lo snippet manager sviluppato successivamente al tirocinio funziona da cli. L'interazione con l'utente avviene quindi in modo più "classico" e "manuale".

Il progetto è stato svolto in Python, usando un database a grafo: arangodb con adattatore networkx. Il sistema di persistenza scelto è molto diverso da neo4j. Mentre in neo4j si utilizza cypher, un linguaggio per scrivere query dichiarative (come SQL), con l'adattatore networkx per arangodb, ho dovuto gestire il grafo in maniera più "manuale". Non avevo a disposizione query dichiarative ma avevo la possibilità di gestire i dati in memoria come se fosse un grafo in RAM, con le classiche operazioni esposte da una libreria di un grafo diretto. Quindi ho dovuto scrivere da me gli algoritmi che mi permettessero di inserire ed eliminare i dati in modo appropriato, esplorare il grafo nelle giuste direzioni e selezionando i dati corretti.
Ho scelto questa metodologia di persistenza per due motivi:

- Essendo la gestione dei dati in memoria il fulcro di entrambi i progetti, ho avuto così la possibilità di svolgere due progetti diversi con le stesse e identiche specifiche. Così, sono più facilmente confrontabili.
- È il modo giusto per testare il mio uso di Why3. Si intuisce dalle specifiche che questo è un software che avrà dei pezzi in cui Why3 è inutile, in altri in cui può risultare vantaggioso.

4.5 Il diario

La scrittura del progetto è stata accompagnata dalla scrittura di un diario. Il diario riporta delle esperienze progettuali significative, da cui sono state dedotte delle conclusioni. Nel capitolo successivo, riporto una rielaborazione del diario.

Capitolo 5

Rielaborazione del diario

Le conclusioni riportate hanno l'utilità di sviscerare l'attività della programmazione (seguendo il metodo ideato) in modo completo, in modo da poter scegliere delle strategie di risoluzione appropriate alla propria persona in relazione al problema che si deve svolgere. Quindi, le conclusioni tratte, risponderanno alle domande del capitolo 3: "In che modo migliorano il processo di programmazione nella quotidianità?" e "A lungo termine, si scrive codice più velocemente o più lentamente?" nel mio singolo progetto.
Le risposte a queste domande possono essere considerate rigorose se fornite da uno studio in larghissima scala, su moltissimi programmatori, di diverso livello, su progetti di diversa matrice. Non disponendo di questa possibilità, ritengo che, con l'esperienza e le conclusioni riportate in questo capitolo, il singolo programmatore potrà capire autonomamente, in base alla sua persona e in base ai problemi che dovrà risolvere, se il metodo proposto sarà adatto in tali circostanze.
Inoltre, le conclusioni raggiunte cercano di preparare il programmatore che si approccia per la prima volta alla verifica formale ad un uso conscio di questo metodo. Nei prossimi paragrafi, si cercano di spiegare le insidie e le difficoltà che si devono affrontare, in quali contesti si trovano, come affrontarle e se vale la pena affrontarle.

5.1 I moduli del progetto

Fin dall'inizio, ho individuato delle parti del progetto in cui una formalizzazione sarebbe servita, altre parti in cui non avrebbe aiutato e, anzi, sarebbe stata limitante. Dunque, ho creato 3 file: `main.py`, `digraph.py` e `msm_digraph.py`. Mentre `digraph.py` e `msm_digraph.py` ricalcano dei moduli scritti in WhyML, `digraph.mlw` e `msm_digraph.mlw`, ho deciso di non formalizzare niente prima di scrivere `main.py`.

Mentre i file `digraph.py` e `msm_digraph.py` riguardano operazioni sui grafi, che riguardano operazioni pregne di logica, ho destinato `main.py` all'interazione utente. Formalizzare l'interazione con l'utente è piuttosto difficile e, nella maggior parte dei casi, non supera nessuna difficoltà, perché non ne sussiste alcuna. Nei test del codice si danno delle priorità. È importante testare il codice critico, che può essere più soggetto a errori logici. Gli errori di codice in cui c'è interazione utente sono estremamente semplici da scovare (basta qualche esecuzione del codice), lo sforzo della formalizzazione e della dimostrazione risulterebbe una perdita di tempo inutile.

5.1.1 Uno spunto sull'architettura esagonale

Per poter seguire il metodo descritto nel capitolo 3, è vantaggioso riuscire a individuare le differenze tra le funzionalità del programma che si sta scrivendo e le interazioni con sistemi esterni. Ad esempio, nell'architettura esagonale, un programma viene descritto separando il core (funzionalità del programma, la logica) dagli adattatori (vari tipi di agenti esterni). Questi adattatori possono essere il sistema operativo, il database, user interface, CLI, ed altri ancora.
Come detto in precedenza, il mio uso di Why3 consiste nella validazione di ragionamenti. Quindi, adottando il mio metodo, ognuno di questi adattatori sarebbe soggetto ad un livello di astrazione, che può essere variabile a seconda del software che si sta sviluppando. Ci sono adattatori la cui astrazione è più facile da formalizzare rispetto ad altri, come ci sono adattatori in cui la formalizzazione rappresenta un vantaggio sulla prevenzione degli errori ed altri no.
Anche all'interno degli adattatori, ci sono differenze. Ad esempio, riguardo il sistema operativo, ci sono system call più facili da formalizzare rispetto ad altre. Le `read/write` sui file sono molto più semplici da formalizzare rispetto a una fork, è intuitivo. Riguardo i database, è difficile trovare modelli difficili da astrarre nelle operazioni base, in quelli conosciuti: i più conosciuti sono i database relazionali, gerarchici, reticolari e chiave valore. Sono tutti facilmente astraibili, derivano direttamente da concetti matematici. Riguardo l'interazione utente da CLI, invece, modellare i comportamenti è quasi sempre molto complicato. Per non parlare di codice html/css. In quel caso i problemi non sono affatto logici, non vanno categoricamente affrontati in Why3.
L'esperienza di un programmatore e ulteriori studi in questo ambito potrebbero aiutare a individuare velocemente le parti di programma in cui è conveniente seguire un approccio formale piuttosto che empirico (d'ora in poi userò formale per

riferirmi a codice che viene prima dimostrato e poi scritto, empirico per riferirmi a codice che viene verificato tramite test).
L'architettura esagonale è solo un esempio di modularizzazione che può essere seguita per separare il codice in modo da poter separare problemi su cui è giusto ragionare in modo differente. Ad esempio, l'architettura a microservizi, che è un derivato, è un approccio paragonabile. Io non ho seguito nessuna architettura di modularizzazione prestabilita, individuare sezioni più logiche di altre (in cui conveniva un approccio formale), nello snippet manager, è risultato piuttosto semplice.

5.1.2 L'astrazione seguita nel mio progetto

Non ho diviso immediatamente il progetto in `main.py`, `digraph.py` e `msm_digraph.py`. Volevo separare dall'inizio le funzioni che agiscono sul database e le funzioni che interagiscono con l'utente. Come detto, volevo formalizzare solo le funzioni core, che interagissero sul grafo in memoria. Ho usato la libreria networkx. In Why3, però, non c'è un corrispondente della libreria networkx. Quindi, per poter adottare il mio metodo sulle funzioni importanti, sono stato forzato a scrivere un modulo Why3 che gestisse grafi direzionati in modo simile a networkx. Quindi, ho ricostruito e formalizzato il pezzo di networkx che mi serviva per poter costruire le mie funzioni. Da qui, è nato il modulo `digraph.mlw`. Questo modulo, è quasi del tutto privo di implementazioni, perché ricostruisce una teoria già fatta e che non devo costruire, che è appunto la libreria networkx. Da qui ho ricavato il modulo `digraph.py` per avere un'astrazione uniforme che corrispondesse più fedelmente possibile alla formalizzazione Why3 e che avesse una coerenza nel mio progetto. Così, ho implementato le mie funzioni (quindi ho costruito la mia teoria) in `msm_digraph.mlw`, clonando la teoria fornita da `digraph.mlw`; così come ho esteso la classe di `digraph.py` per poter scrivere le funzioni effettive in `msm_digraph.py`. La scrittura di `msm_digraph.mlw` è stata quindi completamente diversa rispetto a alla scrittura di `digraph.mlw`. In `msm_digraph.mlw` ho composto una teoria, in `digraph.mlw` ho ricostruito una teoria.

5.2 Comporre o ricostruire teorie?

Questa terminologia può risultare atipica. Ho scelto di usarla ugualmente perché ritengo che sia più estensiva. Sono concetti che ho individuato nella programmazione ma che si possono ritrovare in altri ambiti come logica, matematica, fisica. Intendo per *ricostruire una teoria*: lo studio di una teoria già esistente, che sia un

programma, una teoria matematica, una teoria fisica. È paragonabile al processo di analisi.
Intendo per *comporre una teoria*: la composizione di una teoria nuova (o di un pezzo nuovo di teoria), che sia un programma, una teoria matematica, una teoria fisica. È paragonabile al concetto di sintesi.

Adesso, svolgere delle task e creare un progetto sono entrambe attività che richiedono la scrittura di codice. Quindi possono sembrare molto simili. Utilizzando Why3 mi sono reso conto che hanno dei pezzi in comune ed altri diametralmente opposti. Quando si crea un progetto nuovo, si compone una teoria, aggiungendo dei tasselli pian piano. Quando si svolge una task bisogna ricostruire la teoria già composta da altri, per poter poi aggiungere qualche tassello. Un programmatore può essere incaricato di creare un progetto nuovo (caso molto raro, in cui certamente e principalmente avviene la composizione di una teoria), oppure di integrare un software già esistente. L'integrazione di un software già esistente può avvenire creando dei moduli circa indipendenti, utili al software principale (quindi componendo una teoria), oppure implementando dei moduli integranti del software principale. In questo ultimo caso bisogna scrivere del codice in cui la comprensione del codice pregresso è fondamentale, quindi si ricostruisce una teoria. In realtà, si svolgono entrambi i processi in proporzioni diverse. Ad esempio, come spiegato, ho dovuto ricostruire la "teoria" di networkx, ovvero ho dovuto capire le funzioni della libreria nel dettaglio, per essere sicuro degli effetti che esse avranno sul mio codice. Facendolo, però ho anche svolto un processo di composizione, perché capendo le funzioni disponibili, ho scritto le specifiche di funzioni e dei lemmi che erano coerenti con la teoria dei grafi diretti e che sarebbero servite in futuro. Quindi poi ho scritto un'interfaccia Python (un wrapper) che congiungesse networkx alle specifiche da me formulate (sia quelle ricostruite che quelle composte). In seguito, col modulo `digraph.mlw`, ho già disponibili i pezzi di teoria che mi permettono di lavorare in modo "indipendente", quindi, scrivendo `msm_digraph.mlw`, sto solo componendo. È evidente, in questo ultimo modulo, che le proporzioni sono molto spostate sulla composizione che sulla ricostruzione.
Durante la risoluzione di un task, invece, si è costretti a ricostruire il contesto intorno al singolo problema da risolvere. È prevedibile, che una volta ricostruito il contesto, serva comporre poca teoria per la risoluzione del singolo problema. Ci sono però, sicuramente, dei casi in cui il problema si rivela più difficile (o meno affine al resto del codice) del previsto e prevede la formulazione di un grosso pezzo di teoria, che diventerà integrante a quella principale. Tutto ciò deriva dalle

similitudini che ci sono tra la programmazione e la matematica e, nello specifico, dalla domanda millenaria che recita: La matematica viene creata o viene scoperta? La risposta di questa domanda è un'opinione. Nel software può esserlo solo in certi e pochi casi, perchè il codice che viene creato è un componimento che ha una vita e che fa qualcosa (distinzione programma-processo, il programma ha vita attraverso il processo), un programma serve e fa qualcosa: se non esiste, va creato. La teoria di un programma è atta a un'esecuzione, è dinamica. La teoria matematica è statica, perché tenta di modellare dei concetti che troviamo nella realtà. Fornisce delle risposte sotto determinate assunzioni, che possono essere utili alla creazione e alla scoperta di qualcos'altro. Quindi, mentre nella programmazione c'è una differenza nel contesto, che ci permette di distinguere più facilmente composizione e ricostruizione, nella matematica diventa molto più difficoltoso (e in toto impossibile). Detto ciò, in programmazione, tradizionalmente, la teoria di un programma si ricostruisce mentalmente, leggendo il codice; e si compone scrivendo. Con Why3, si ha l'opportunità sia di ricostruire che di comporre. Essere coscienti di ciò che si sta facendo (ricostruire o comporre) può essere utile per individuare dei metodi per svolgere l'attività in questione nel modo più efficiente possibile. Questi metodi possono essere anche individuati in letteratura. Il binomio ricostruzione e composizione di una teoria può essere associato al binomio modello descrittivo e normativo.

I modelli descrittivi servono a descrivere un sistema esistente, estrapolando un modello (reverse engineering, ricostruisco una teoria). La valutazione del modello creato si basa sull'accuratezza.

I modelli normativi servono a stabilire come dovrebbe essere un sistema, estrapolando un modello che dovrà poi essere implementato (progettazione, compongo una teoria). La valutazione del modello creato si basa sulla correttezza, completezza e coerenza.

Nel mio contesto, una volta scritto `digraph.py` non ho più dovuto consultare la documentazione di networkx, seppure non avessi mai usato la libreria prima d'ora. Utilizzare le funzioni ponte da me sviluppate è stato piuttosto organico, avevo piena padronanza nella scrittura del software. Quindi, la descrizione del sistema preesistente, è avvenuta in modo solido e accurato.

Conclusione

Si può utilizzare Why3 per leggere e capire software già esistenti, scrivendone le specifiche formalmente. Questo uso porta a una comprensione solida e completa del software in analisi, permettendo un'integrazione efficiente.

Riguardo invece la correttezza, la completezza e coerenza del modello creato (sostanzialmente il modulo `msm_digraph.mlw`), verranno discusse nelle sezioni successive.

5.3 Testare le specifiche

Dicevo che in `digraph.mlw` sono state scritte per lo più specifiche di funzioni. Per poter scrivere queste specifiche ho dovuto prima modellare il concetto di grafo nel modo più vicino possibile alla libreria che avrei usato in seguito (la teoria dei grafi formalizzata in Why3 aveva formalizzazione povera e distante da networkx.).

5.3.1 Testare gli assiomi

Ho dovuto quindi definire il tipo grafo:

```
type v  (* Tipo vertice *)
type e  (* Tipo arco (edge) *)

val function src (e: e) : v     (* Sorgente di un arco *)
val function dst (e: e) : v     (* Destinazione di un arco *)

(* Il grafo è una coppia: insieme di vertici e insieme di archi *)
type graph = { vertices: fset v; edges: fset e }

val function memv (v: v) (g: graph) : bool    (* Membership di un vertice *)
  ensures { result =  mem v g.vertices }
val function meme (e: e) (g: graph) : bool    (* Membership di un arco *)
  ensures { result = mem e g.edges }
```

Da qui, ho dovuto scrivere degli assiomi, che modellassero il tipo grafo ulteriormente:

```
(* Se un arco è nel grafo, anche sorgente e destinazione sono nel grafo *)
axiom edges_well_formed:
  forall g: graph, e: e.
    meme e g → memv (src e) g ∧ memv (dst e ) g
(* Non ci sono archi uguali *)
axiom no_duplicate_edges:
  forall g: graph, e1 e2: e.
    meme e1 g → meme e2 g → src e1 = src e2 → dst e1 = dst e2 → e1 = e2
```

Una volta scritti, però, ho avuto un dubbio. Come faccio a sapere che siano corretti?

In programmazione, si è abituati a testare tutto. Per testare basta spesso una funzione di print sul caso che non si ha ben chiaro. In Why3 non si esegue il codice, si dimostra. Gli assiomi non possono essere eseguiti. Per le funzioni ci sono modi per eseguire il codice e per scrivere dei test ma risulta parecchio macchinoso perché non è uno scopo dello strumento. Quindi, quando si fa il design della logica, per "testare", per avere delle prove (di cui spesso si sente il bisogno), come si fa?
Si possono scrivere dei teoremi (in Why3 viene usata solo la parola lemma). Questi teoremi devono essere delle proprietà immancabili nel contesto che si sta modellando e devono essere dimostrabili a partire dagli assiomi. Di seguito il lemma (che è stato dimostrato) a partire dagli assiomi riportati:

```
(* Caso specifico dell'assioma edges_well_formed *)
lemma vertex_existence:
  forall v1 v2: v, e: e, g: graph.
    memv v2 g ∧ meme e g ∧ src e = v1 ∧ dst e = v2
        → memv v1 g
```

5.3.2 Testare i contratti di funzione

Anche per le funzioni ausiliarie, se non si è sicuri delle specifiche che si sono date, se queste sono definite con `val function` (quindi oltre a essere funzioni che si possono usare in un programma, sono funzioni logicamente pure, prive di effetti collaterali), si possono utilizzare dei lemmi per verificare la correttezza delle specifiche. Se questi lemmi verranno dimostrati, si avranno sia dei risultati che potrebbero essere utili a dimostrare le correttezza di funzioni future, sia la sicurezza che queste specifiche siano state formulate correttamente.
Quando un concetto risulta già visto e semplice, si tende a non avere bisogno di verificarne la correttezza immediatamente. Nel caso opposto, si vuole testare ogni riga. Quindi ciò che accade nella programmazione classica per le implementazioni (spesso avendo un'idea vaga delle specifiche), accade in Why3 nelle specifiche. La correttezza del codice viene dimostrata rispetto alle specifiche date, le quali possono essere errate.
Riporto un esempio. Modellando la specifica della funzione di insert di un vertice, ho ritenuto necessario scrivere dei lemmi che verificassero il suo comportamento su certe classi di casi:

```
val function insertv (v: v) (g: graph) : graph
  ensures { result.edges = g.edges }
  ensures { result.vertices = Fset.add v g.vertices }
```

```
(* Inserimento di vertice nel vuoto *)
lemma insertv_empty:
  forall g1 g2: graph, v: v. empty_graph g1
    → g2 = insertv v g1
    → memv v g2 ∧ cardinalv g2 = 1

(* Inserimento di vertice in grafo generico, cardinalità maggiore e membership
    *)
lemma insertv_memv_cardinalv:
  forall g1 g2: graph, v: v, n: int. cardinalv g1 = n ∧ not (memv v g1)
    → g2 = insertv v g1
    → memv v g2 ∧ cardinalv g2 = (n+1)
```

In questo caso le specifiche erano corrette dall'inizio, però è successa una cosa interessante. Il primo lemma, è stato dimostrato immediatamente. Il secondo no, perchè non avevo scritto la condizione `not (memv v g1)`. Il fatto che non venisse dimostrato mi ha subito fatto venire in mente la condizione mancante. Quindi, grazie al "test" della specifica, ho pensato ad una caratteristica che non avevo pensato prima, quindi ho prevenuto dei dubbi che sarebbero venuti più in la. Attraverso questo lemma ho preso subito coscienza del fatto che questa funzione era corretta ma poco chiara. L'operazione `Fset.add` non aggiunge duplicati, quindi nel grafo non ci saranno nodi duplicati. Però, la funzione è invocabile su un duplicato. Questo, mi ha portato, in futuro, a cambiare la funzione, aggiungendo una precodizione:

```
val function insertv (v: v) (g: graph) : graph
  requires  { forall u:v. memv u g → u ≠ v }
  ensures   { result.edges = g.edges }
  ensures   { result.vertices = Fset.add v g.vertices }
```

Quando ho dovuto cambiare la funzione (è avvenuto molto tempo dopo la sua definizione), avevo piena coscienza dei cambiamenti che andavano fatti e perché andavano fatti, perché ero stato costretto a scovare delle proprietà.

Conclusione

Il modo giusto per testare specifiche è scrivere delle proposizioni che dovranno essere dimostrate. Scrivere queste proposizioni risulta anche utile a dedurre proprietà che probabilmente si riveleranno utili a dimostrazioni future.

Conclusione

Scrivere proposizioni che vengono dimostrate è utile in due sensi: dedurre e dimostrare proprietà di un problema e ampliare il contesto utile alla dimostrazione di risultati futuri.
Capire le prorpietà di un problema è vantaggioso nella prevenzione degli errori. L'effetto è diretto perché è utile alla programmazione.
Ampliare il contesto è tendenzialmente utile a non perdere tempo nelle dimostrazioni future. L'effetto è indiretto perché è utile alla modellazione.

5.3.3 Difficoltà nella dimostrazione di lemmi

Stavo specificando la funzione di eliminazione di un vertice. Per testare il contratto della funzione, ho scritto dei lemmi:

```
val function deletev (v: v) (g: graph) : graph
  ensures { result.vertices = Fset.remove v g.vertices }
  ensures { result.edges = Fset.filter g.edges (fun (e: e) → src e ≠ v ∧ dst
     e ≠ v) }

(* L'eiliminazione di un nodo da un grafo vuoto restituisce un grafo vuoto *)
lemma deletev_empty:
  forall g1 g2: graph, v: v. empty_graph g1 → g2 = deletev v g1 →
    empty_graph g2

(* Alla cardinalità degli archi va sottratto il grado del vertice eliminato *)
lemma deletev_card:
  forall g1 g2: graph, v: v, n: int.
    n = cardinale g1 ∧ g2 = deltev v g1
    → cardinale g2 = n - (degree v g1)
```

Il primo lemma è stato dimostrato, il secondo, non riusciva a essere dimostrato. A questo punto non sapevo se le specifiche fossero sbagliate, il lemma fosse sbagliato, oppure se i prover non riuscissero a dimostrare questo risultato. Mi sembrava che il contratto della funzione fosse corretto e che anche il lemma fosse corretto. Quindi, ho cercato di concentrare i miei sforzi nell'aiutare i prover a dimostrare questo lemma. Ci sono tanti modi per farlo. Si possono aggiungere dei lemmi (o assiomi) ausiliari, aggiungere postcondizioni che si ritenevano superflue, utilizzare delle tecniche di trasformazione opportune, riformulare l'enunciato da dimostrare o riformulare il contratto della funzione protagonista del risultato. Successivamente, ho provato a usare diverse di queste tecniche, riuscendo a dimostrare una versione debole dell'enunciato problematico:

```
val function deletev(v: v) (g: graph) : graph
  ensures { result.vertices = Fset.remove v g.vertices }
  ensures { result.edges = Fset.filter g.edges (fun (e: e) → src e ≠ v ∧ dst
    e ≠ v) }
  (* Il grado del nodo eliminato è 0 *)
  ensures { degree v result = 0 }
  (* stesso grado per gli altri nodi *)
  ensures { forall u: v. u ≠ v → memv u g → degree u g = degree u result }

lemma deltev_cardinal_vertices_weak:
  forall g1 g2: graph, v: v, n: int. n = cardinalv g1 ∧ g2 = deletev v g1
    → cardinale g2 ≤ n
```

Questo lemma è stato dimostrato, grazie alle postcondizioni inserite. Quando bisogna unicamente specificare una funzione senza implementazione, per poterla usare in altre funzioni, aggiungere troppe postondizioni per dimostrare certi lemmi rappresenta un rischio. Le postcondizioni, in funzioni prive di implementazioni che dimostrano il contratto, rappresentano degli assiomi. Quindi, allargare le specifiche di una funzione significa dare per vero più risultati, alcuni dei quali potrebbero essere falsi. È buona pratica, dato un numero di postcondizioni, verificare che alcune di queste siano dimostrabili a partire da altre. È un risultato più solido e meno rischioso.
Ecco un esempio in cui ho scritto inizialmente, intuitivamente, una postcondizione che sono riuscito a dimostrare successivamente in un lemma:

```
val function fset_to_list (s: fset 'a) : list 'a
  ensures { forall x: 'a. Fset.mem x s → LM.mem x result }
  ensures { cardinal s = length result }
  ensures { forall x y: 'a. LM.mem x result ∧ Lm.mem y result → x ≠ y }
```

Ho eliminato la seconda postcondizione, riuscendo a dimostrarla come lemma:

```
val function fset_to_list (s: fset 'a) : list 'a
  ensures { forall x: 'a. Fset.mem x s → LM.mem x result }
  ensures { forall x y: 'a. LM.mem x result ∧ Lm.mem y result → x ≠ y }

lemma cardinal_eq_length:
  forall s: fset 'a, l: list 'a.
    l = fset_to_list s
      → cardinal s = length l
```

5.4 Rifattorizzazione di specifiche

Ci si può rendere conto, a volte, di aver formalizzato dei pezzi di teoria in modo poco intelligente. Ci si può rendere conto che gli assiomi sono troppo deboli, o che esprimono proprietà false, o che esprimono proprietà dimostrabili. Può capitare di aver formalizzato male i tipi su cui si fonda la teoria. Questo è stato il mio caso. Dopo le prime funzioni, ho provato a clonare la teoria, per cominciare a formalizzare il modulo del grafo che mi sarebbe servito. Inizialmente, gli archi erano definiti così: `type e = (v,v)`. Per riuscire a formalizzare snippet e metadati, ho dovuto rendere generico il tipo arco, accedendo agli elementi di un arco tramite le funzioni `src` e `dst`, come definito nella sezione 5.3.1. Così facendo, ho dovuto rifattorizzare gran parte della teoria creata. Ho dovuto riformulare lemmi, assiomi, contratti di funzione. Così facendo, ho dovuto ripensare al senso di ogni enunciato, di ogni predicato. È stata un'occasione per rafforzare la mia conoscenza riguardo i problemi affrontati e per riformulare gli enunciati per renderli più coerenti. Quando si fa un refactoring della propria teoria, però, si perde la dimostrazione dei risultati ottenuti e tutti quelli che dipendono da essi. Quindi, se si riformula, ad esempio, l'ultimo lemma scritto, sarà l'unico risultato a dover essere ri-dimostrato. Se, invece, riformulo un assioma fondante, che è definito prima di qualsiasi risultato, dovrò ri-dimostrare tutti i risultati successivi. Si vede quindi che il refactoring impone uno sforzo non banale. Dovendo ri-dimostrare, si potrebbe non riuscire a dimostrare dei risultati che era stati già provati. Oppure potrebbe avvenire il caso opposto. È ciò che è avvenuto nel mio caso. Nella sezione 5.3.3 spiegavo che non riuscivo a dimostrare la funzione `deletev`. Rifattorizzando il codice (per altri motivi), rilanciando i prover, il lemma che prima non veniva dimostrato, è stato provato. Ciò è avvenuto grazie al fatto che stessi usando uno strumento che utilizzasse prover automatici. In quel momento non pensavo più al lemma che non riuscivo a dimostrare, pensavo di occuparmene più tardi. Dovendo lanciare i prover su tutti i goal generati (dato che avevo rifattorizzato), questi hanno avuto modo di riprovare a dimostrare il lemma, ottenendo successo. In uno strumento di dimostrazione assistita, ciò non avviene, perché le dimostrazioni in cui la fattorizzazione non influisce, rimangono intatte. Quelle in cui influisce, vanno riformulate. I risultati che non si sono riusciti a dimostrare, rimangono tali. Con i proof assistant si ha più controllo, quindi, dopo una fattorizzazione, potrebbero non esserci molti cambiamenti, dal punto di vista dei risultati ottenuti. In un sistema basato su prover automatici, invece, bisogna sperare che tutto ciò che era stato dimostrato venga confermato. È un sistema certamente più instabile.

Conclusione

Sia M un modulo, che dispone di un insieme di enunciati E.
Sia $E_1 \subseteq E$ l'insieme di enunciati dimostrati di M.
Sia M' il modulo ottenuto da una rifattorizzazione di M.
Sia $E_2 \subseteq E$ l'insieme di enunciati dimostati di M'.
Allora $E_2 \subseteq E_1 \vee E_1 \subseteq E_2 \vee E_2 = E \vee E_2 = \emptyset$.
Ovvero, l'insieme degli enunciati dimostrati dopo la fattorizzazione potrebbe variare rispetto a quello precedente.

Come descritto, la fattorizzazione è stata necessaria, ed è avvenuta perché ho provato a clonare la il modulo `digraph.mlw` nel mio caso specifico, `msm_digraph.mlw`. Da qui, ricavo un consiglio.

Consiglio

Se si sta sviluppando un modulo Why3 il cui è scopo principale è essere clonato (usato), è utile clonarlo (usarlo) in un caso specifico già quando si ottiene un insieme minimale di funzioni, in modo da poter individuare prontamente eventuali fattorizzazioni necessarie. Rimandare la fattorizzazione è controproducente, perché, tutto il codice accumulato dovrà essere rifattorizzato.

Questa fattorizzazione è stata molto importante, perché mi ha fatto capire delle proprietà riguardo a questa attività, ma, soprattutto, mi ha causato dei grandi problemi, grazie ai quali ho avuto delle esperienze significative, descritte nel capitolo successivo.

5.5 Teoria inconsistente

La fattorizzazione era avvenuta in un momento in cui stavo sviluppando contemporaneamente i moduli `digraph.mlw` e `msm_digraph.mlw`, motivo per il quale ho avuto necessità della rifattorizzazione.
Per testare la coerenza della teoria del grafo specifico del mio progetto, ho provato a scrivere un lemma che verificasse la presenza di archi che non potevano esistere:

```
lemma wrong_relation:
  forall g: graph. exists e: e. meme e g ∧ (is_metadata (src e) ∧ is_snippet (
    dst e))
```

Quindi, speravo che il lemma non venisse dimostrato. Poi avrei scritto il lemma opposto, sperando venisse dimostrato, perchè effettivamente, nel grafo, non possono esserci archi da metadato a snippet. Con mia grande sorpresa, il lemma è stato dimostrato.
Pensandoci, il lemma non solo non avrebbe dovuto essere dimostrato in questo contesto, ma non dovrebbe risultare dimostrabile neanche in un contesto senza gli assiomi che avrebbero dovuto ostacolarlo. Questo perchè è falso a prescindere che un grafo abbia almeno un arco con sorgente un metadato e destinazione uno snippet, un grafo può essere vuoto!
Ho scritto la negazione del lemma:

```
lemma not_wrong_relation:
  forall g: graph, e: e. meme e g → (is_metadata (src e) → not (is_snippet (
    dst e)))
```

Veniva dimostrato anch'esso. Ormai, avevo capito che la mia teoria fosse inconsistente. Ho sentito una sensazione di fallimento, e sentivo che scovare l'errore sarebbe stato ancora più umiliante. Ho chiesto aiuto a un LLM, preferivo che fosse qualcun'altro (o meglio, qualcos'altro) a scoprire il motivo dell'inconsistenza. Gemini ha trovato subito il problema, ahimè, era l'assioma fondante della teoria DiGraph (per fortuna gli avevo fornito anche DiGraph come contesto!). Quello stesso assioma che pensavo fosse la chiave per dimostrare altri lemmi, in realtà rendeva la mia teoria inconsistente, creando contraddizioni con altri assiomi presenti successivamente.
L'inconsistenza avviene tra l'assioma `edges_well_formed`, assolutamente sbagliato che implica che ci possa essere al massimo un nodo. Tutte le funzioni successive avevano specifiche che permettevano di avere più di un nodo nel grafo. Le postcondizioni delle funzioni di cui sono state scritte solo le specifiche rappresentano degli assiomi. Questi assiomi, quindi, erano in contraddizione con l'assioma principale.
Durante la fattorizzazione, avevo appunto cambiato l'assioma fondante commettendo un errore madornale. Di seguito l'assioma che generava l'inconsistenza:

```
axiom edges_well_formed:
  forall v1 v2: v, e: e, g: graph. meme e g ∧ src e = v1 ∧ dst e = v2 ↔ memv
    v1 g ∧ memv v2 g
```

In questo caso, quindi l'LLM ha scovato prontamente l'inconsistenza nella mia teoria, soprattutto l'errore madornale che era quell'assioma. Questo è rassicurante. Quando si trova un'inconsistenza è già frustrante accettarla e ammetterla, inizialmente ero riluttante a cercarla. Un LLM può aiutare a superare questi

vincoli affettivi che si provano per il proprio codice. Bisogna però notare anche che ho fornito il codice (incluso l'assioma sbagliato) più volte allo stesso Gemini o altri LLM per scopi diversi e loro non mi hanno mai segnalato questo errore gravissimo. Se avessi chiesto qualcosa riguardo un altro lemma ad un umano mediamente preparato, questi, leggendo l'assioma avrebbe molto probabilmente segnalato l'errore. Gli LLM cercano sempre di dare una risposta soddisfacente a ogni domanda, non conoscono/capiscono il vero obiettivo dell'interlocutore.
Sistemato l'assioma, ho trovato altre contraddizioni. Soprattutto, molte delle contraddizioni trovate, erano in `digraph.mlw`.
Adesso, volevo trovare un procedimento per riuscire a scovare l'inconsistenza in modo efficiente. Pensando al modo in cui è stata trovata la prima volta, ovvero da una contraddizione, ho capito che sarebbe stato molto semplice.
Basta aggiungere, in fondo al modulo, un lemma che semplicemente dichiara il falso: `lemma lemma_false: false`.
Se questo verrà dimostrato, avremo la certezza che la teoria sia inconsistente.
Ma nel caso non venga dimostrato? Vuol dire che la teoria è consistente? Potrebbe capitare che i prover non riescano a dimostrare il falso, ma che il falso sia effettivamente un risultato ottenibile nel contesto di riferimento. Allora, si può avere certezza sull'inconsistenza ma non sulla consistenza della teoria che si sta costruendo? La risposta alla domanda è sì, non si può essere certi della consistenza della propria teoria. Allora, come si fa? È vero che non si può avere la certezza, ma si può avere una discreta sicurezza. Mi rendevo conto, che, quando dimostravo risultati in una teoria inconsistente, questi venivano dimostrati molto velocemente, così come veniva dimostrato velocemente il falso. Quindi, questo mi ha fatto avere delle intuizioni sulla natura dei prover automatici, che sono state confermate dalle documentazioni che spiegano il loro funzionamento. I prover automatici provano a dimostrare tutto per assurdo, raggiungono i risultati molto più velocemente. Quindi, quando devono dimostrare il falso, suppongono il vero e cercano una contraddizione. Questo permette di scovare l'inconsistenza in modo estremamente veloce. Possono essere rallentati solo dalla larghezza del contesto. Quindi l'efficienza dei prover nel dimostrare il falso, un fallimento nella dimostrazione di questo conferisce molta sicurezza sulla coerenza della propria teoria.

Conclusione

Quando il falso viene dimostrato, è certo che la teoria sia inconsistente.
Quando il falso non viene dimostrato, non è certo che la teoria sia consistente, ma è molto probabile.

Adesso, per capire il punto in cui avviene la contraddizione, basta spostare in alto il lemma che dichiara il falso, per vedere fino a che punto viene dimostrato. Quando si troverà un punto in cui non viene dimostrato, vuol dire che, in teoria, nell'assioma successivo (o specifica di funzione successiva), avviene la contraddizione. Dico in teoria e non sicuramente perché, può capitare, in casi remoti, che una contraddizione avvenga prima ma venga rilevata solo dopo dei risultati dimostrati.
Ecco una descrizione più formale. Siano A e B i due assiomi in contraddizione. Può capitare che quest'ultima non sia evidente, quindi, il falso non venga dimostrato. Viene dimostrato un lemma C a partire da B. Il falso viene dimostrato dopo C. Quindi sembrerà che l'inconsistenza avvenga a causa di C, ma in realtà C avrà solo palesato la contraddizione tra A e B. A questo punto, se il falso risulta dimostrato dopo un risultato provato, per capire dove avvenga effettivamente la contraddizione, bisogna intuire come sia stato dimostrato quel risultato.
Trovata, l'inconsistenza, ho dovuto rifattorizzare la teoria, per risolvere le varie contraddizioni. È curioso notare che una rifattorizzazione aveva generato l'inconsistenza, dimostrando un risultato che non riusciva a essere verificato: un lemma su `deletev`. Successivamente, trovata l'inconsistenza, e rifattorizzata la teoria per eliminare qualsiasi tipo di contraddizione, `deletev` è stata dimostrata correttamente. Da quel momento in poi, ho spesso cercato di scovare inconsistenze inserendo il `lemma_false` in fondo al modulo. È importante ricordarsi, di togliere quel lemma, però, quando si è sicuri della consistenza. Altrimenti il falso, per come è fatto Why3, servirà da contesto per i risultati successivi anche se non dimostrato.

Consiglio

Per verificare la consistenza della propria teoria, scrivere, nell'ultima riga, un lemma che dichiari il falso.
Nel caso venga dimostrato, spostarlo in alto finché non venga più a dimostrato. Molto probabilmente, l'enunciato successivo, sarà la causa dell'inconsistenza.

5.6 Premeditazione involontaria di funzioni future

Scrivendo le specifiche delle prime funzioni del grafo, pensavo già a quelle successive, quindi scrivere queste risultava più semplice. Scrivendo le successive, pensavo già alle specifiche del grafo nel mio programma specifico, che è risultato più semplice. Capito come fare queste, pensavo già alle implementazioni delle funzioni del progetto finale, in che modo sarebbero state implementate sia in Why3 che in Python. Tutto ciò avveniva mentre facevo altro. Ho quindi sperimentato quanto sia forte l'attività di specifica in un progetto grande. Invece non ho notato la potenza delle specifiche quando ho risolto singoli problemi come esercitazione. Quindi, secondo la mia esperienza, noto che l'attività di specifica possa risultare lunga e tediosa nella soluzione di un singolo problema isolato, può sembrare anche inutile se la soluzione che si ha in mente dall'inizio si riveli perfetta alla fine. Quando si crea un programma più complesso, specificare aiuta a pensare al problema prima di produrre codice che si è riluttanti a buttare. In questo caso, diventa un'attività molto granulare e soddisfacente. La potenza dell'attività di specifica sta proprio nel pensare involontariamente alle soluzioni di problemi che si devono risolvere più avanti e questo avviene solo in contesti in cui il problema da risolvere è complesso, in cui bisogna fare delle scelte. Ad esempio, al posto di leggere le specifiche di funzioni che avevo a disposizione, ho dovuto specificarle per poterle usare, come quella dei successori e dei predecessori in `digraph.mlw`:

```
val function edges_srcv (v: v) (g: graph) : fset e
  ensures { result = Fset.filter g.edges (fun (p:e) → src p = v) }

val function successors (v: v) (g: graph) : fset v
  ensures { result = map (fun (p: e) → dst p) (edges_srcv v g) }

val function successors_list (v: v) (g: graph) : list v
  ensures { result = fset_to_list (successors v g) }

lemma members_successors:
  forall g: graph, v: v, succ: fset v. succ = successors v g →
    (forall u: v. mem u succ ↔ (exists e: e. meme e g ∧ src e = v ∧ dst e = u
    ))

val function edges_dstv (v: v) (g: graph) : fset e
  ensures { result = Fset.filter g.edges (fun (p:e) → dst p = v) }

val function predecessors (v: v) (g: graph) : fset v
```

```
  ensures { result = map (fun (p: e) → src p) (edges_dstv v g) }

val function predecessors_list (v: v) (g: graph) : list v
  ensures { result = fset_to_list (predecessors v g) }

lemma members_predecessors:
  forall g: graph, v: v, pred: fset v. pred = predecessors v g →
    (forall u: v. mem u pred ↔ (exists e: e. meme e g ∧  src e = u ∧ dst e =
    v) )
```

Mentre scrivevo gli enunciati di lemmi relativi ho pensato all'implementazione di una funzione che dovrò svolgere molto più in la, `get_siblings` (la funzione che prende i fratelli dei metadati). In fare ciò ero privo di intenzione, in quel momento il mio unico scopo era di specificare. Questo è solo un esempio, l'implementazione di `get_siblings` è sicuramente piuttosto semplice, ma questo è avvenuto di continuo, e, molte delle intuizioni avute, si sono rivelate corrette. Ho pensato involontariamente alle specifiche e alle implementazioni delle funzioni successive molto frequentemente. Questo rappresenta un gran vantaggio.
Adesso, questo avveniva nella mia mente e non conosco la mente degli altri. Riporto semplicemente la mia esperienza. Non so se sia così per la maggior parte delle persone. Ciò non toglie che delle persone in cui questo avvenga ci siano e quindi credo valga la pena provare, data la potenza del vantaggio.

5.7 Sulla ricostruizione della teoria di networkx. Una nota sul tempo

Scrivendo l'interfaccia che riguarda i grafi, in modo che abbia circa le stesse funzionalità che sono fornite dalla libreria networkx, bisogna fare un lavoro abbastanza importante. Ho dovuto scrivere circa 250 righe di specifiche (tra tipizzazione, specifiche di funzioni, lemmi, assiomi). Nonostante il tempo impiegato sia notevole, credo ne valga la pena per vari motivi:

- Così facendo si è forzati a conoscere bene la libreria di riferimento, le funzionalità e gli edge cases sono molto più chiari
- Si deducono, prima di scrivere il programma effettivo, le funzioni che possono essere utili al programma, prima ancora di svilupparlo
- Si deducono già delle funzioni di servizio che potrebbero servire sia al programma nel linguaggio finale, che nelle specifiche finali

- Il tempo impiegato è ben investito perchè, in un contesto lavorativo, può essere frequente l'utilizzo della stessa libreria in più progetti. Spesso le aziende tendono a individuare delle tecnologie di riferimento, quindi formalizzare una libreria sarebbe un lavoro che si farebbe solo una volta (al massimo con degli aggiornamenti sporadici) e che si potrebbe riutilizzare nei progetti successivi

- Per restare aggiornati su una libreria, nelle nuove release, può essere utile formalizzare le specifiche delle nuove funzioni piuttosto che leggere semplicemente la documentazione. Spesso capita di affezionarsi a vecchie versioni di linguaggio o una libreria semplicemente perché ci si è abituati. Dunque si fa fatica ad integrare le nuove funzionalità nella programmazione di tutti i giorni. Formalizzare queste funzionalità può aiutare a capirle meglio e quindi a capirne meglio il potenziale

5.8 Aiutarsi con gli LLM, assiomi al posto di lemmi

Scrivendo `msm_digraph.mlw`, ho notato che i prover fanno molta fatica a dimostrare determinati tipi di lemmi. Spesso non riuscivano a essere dimostrati dei risultati (piuttosto triviali) sulla cardinalità. Ho deciso di convertire alcuni lemmi in assiomi (annotandolo nel codice), che poi avrebbero aiutato a dimostrare risultati successivi. Questo perchè sono molto sicuro che questi siano veri e li ritengo piuttosto importanti. Inoltre, ho notato che gli LLM non riescono quasi mai ad aiutare nelle dimostrazioni automatiche (trovando lemmi e proprietà del codice) ma (soprattutto quelli con deepthinking) sono riusciti spesso a trovare dei miei errori, se chiedevo esplicitamente di trovarli. Sono fatti per soddisfare la risposta dell'utente, quindi non scovano quasi mai errori se non chiedo di farlo. Quando invece chiedo di farlo "si impegnano" in quel compito. Quando hanno trovato degli errori, ci sono stati due casi:

- Segnalavano dei lemmi falsi quando in realtà erano dimostrati. In questo caso, le loro segnalazioni erano errate e non mi interessavano (abbastanza frequente);

- Segnalavano dei lemmi falsi che non erano stati dimostrati. In questo caso ci sono stati suggerimenti preziosi, riguardo il perchè i lemmi fossero sbagliati e riguardo incoerenze dell'intera teoria.

Quindi, in questo caso, dato che ho individuato dei punti deboli dei prover automatici sul mio contesto, ho deciso di chiedere a 3 LLM diversi (Gemini, Claude, ChatGPT) cosa ne pensassero riguardo la correttezza di questi lemmi e hanno tutti dato un riscontro positivo, fornendo una dimostrazione informale.
Questo è un uso molto poco rigoroso di Why3. Si dovrebbe cercare di avere meno assiomi possibile. Utilizzare gli LLM per dare delle dimostrazioni informali, o provare a farle autonomamente, su carta, per convincerci della correttezza di un enunciato, credo si possa fare solo in circostanze particolari:

1. i lemmi che si vogliono convertire hanno delle proprietà in cui i prover fanno spesso molta fatica;

2. i lemmi che si vogliono convertire sono di fondamentale importanza. Ovvero, sono utili a dimostrare risultati successivi (non deve essere un mero test di specifica);

3. si scrive (o si legge da un LLM) una dimostrazione informale che ci dà molta sicurezza sulla correttezza del risultato.

5.9 Corrispondenza tra digraph.mlw e digraph.py

L'applicazione del metodo che avevo ideato nel capitolo 3 ha avuto luogo dopo molte righe di codice. Ho dovuto prima scrivere `digraph.mlw`, per poter scrivere le basi della teoria di `msm_digraph.mlw`. Inoltre, per poter scrivere codice in modo organico, prima di scrivere le prime funzioni significative, ho scritto `digraph.py`. In `digraph.py` ho scritto le funzioni che sono state specificate nel corrispondente modulo formale. Quindi ho scritto delle implementazioni nel linguaggio di programmazione di destinazione, che non erano state dimostrate in Why3. È avvenuto perché, appunto, `digraph.py` rappresenta semplicemente un ponte tra la libreria networkx e il core del mio programma. Le implementazioni delle funzioni sono estremamente triviali, perché utilizzano una libreria. Le specifiche formali sono state scritte proprio per ricostruire la teoria dedotta dalla libreria. Ad esempio, la specifica della funzione deletee, è stata scritta così in Why3:

```
val function deletee (e: e) (g: graph) : graph
  ensures { result.edges = Fset.remove e g.edges }
  ensures { result.vertices = g.vertices }
```

L'implementazione Python è questa:

```
def deletee(self, source_key: str, target_key: str):
```

```
    """Delete edge from graph."""
    if not self.G.has_edge(source_key, target_key):
        raise KeyError(f"Attempted to delete non-existent edge ('{source_key}'
     -> '{target_key}').")
    self.G.remove_edge(source_key, target_key)
```

Così, il programmatore che vorrà capire esattamente le specifiche della funzione Python, potrà riferirsi al modulo scritto in WhyML.
Dunque le prime implementazioni sensate, scritte in WhyML e tradotte in Python, hanno avuto luogo in `msm_digraph.mlw`. Chiaramente, è stato applicato il metodo desctitto nel capitolo 3.

5.10 Le prime applicazioni del metodo nel progetto

La prima funzione a dover essere implementata è stata quella di inserimento di uno snippet. Ho quindi scritto la funzione, ho utilizzato una funzione ausiliaria, di cui ho specificato solo il contratto:

```
val function insert_metadata_list_for_snippet (s: v) (metadata_list: list v) (
    g: graph) : graph
  requires  { is_snippet s }
  requires  { memv s g }
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m ∧ memv m g
    }
  ensures   { result.vertices = g. vertices }
  ensures   { forall m: v. LM.mem m metadata_list → exists e: e. meme e
    result ∧ src e = s ∧ dst e = m }

let function insert_snippet (s: v) (metadata_list: list v) (g: graph) : graph
  requires  { is_snippet s }
  requires  { not (memv s g) }
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m ∧ memv m g
    }
  ensures   { result.vertices = Fset.add s g.vertices }
  ensures   { forall m: v. LM.mem m metadata_list → exists e: e. meme e
    result ∧ src e = s ∧ dst e = m }
= let g_s_inserted = insertv s g in
  insert_metadata_list_for_snippet s metadata_list g
```

A questo punto, la correttezza della funzione è stata dimostrata immediatamente. Si nota che il problema più difficile consiste nella risoluzione della funzione ausi-

liaria. Valeva ovviamente la pena implementarla e cercare di dimostrarla, quindi ho fornito un'implementazione:

```
let rec function insert_metadata_list_for_snippet (s: v) (metadata_list: list
    v) (g: graph) : graph
  requires  { is_snippet s }
  requires  { memv s g }
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m ∧ memv m g
    }
  ensures   { result.vertices = g. vertices }
  ensures   { forall m: v. LM.mem m metadata_list → exists e: e. meme e
    result ∧ src e = s ∧ dst e = m }
= match metadata_list with
    | Nil → g
    | Cons m l →
        let g_sm_edge = inserte (s,m) g in
        insert_metadata_list_for_snippet s l g_sm_edge
    end
```

Di essa, non riuscivo a dimostrare l'ultima postcondizione. Non capivo come fare. Privo di idee, ho chiesto a ChatGPT e Claude. Entrambi mi hanno dato una risposta simile, cercavano di aiutarmi aggiungendo delle asserzioni:

```
let rec function insert_metadata_list_for_snippet (s: v) (metadata_list: list
    v) (g: graph) : graph
  requires  { is_snippet s }
  requires  { memv s g }
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m ∧ memv m g
    }
  ensures   { result.vertices = g. vertices }
  ensures   { forall m: v. LM.mem m metadata_list → exists e: e. meme e
    result ∧ src e = s ∧ dst e = m }
= match metadata_list with
    | Nil → g
    | Cons m l →
        let g_sm_edge = inserte (s,m) g in
        let g_final = insert_metadata_list_for_snippet s l g_sm_edge in
        begin
          assert { exists e: e. e = (s,m) ∧ meme e g_sm_edge ∧ src e = s ∧
    dst e = m };
          assert { exists e: e. meme e g_final ∧ src e = s ∧ dst e = m };
        end
    end
```

Le postcondizioni sono state dimostrate ma il problema è diventato dimostrare la seconda asserzione. Poi, senza l'aiuto di AI, ho capito fosse opportuno provare

a scrivere diversamente le postcondizioni. Dato che avevo notato che i prover facevano più fatica negli enunciati con il quantificatore esistenziale, ho provato a riformulare le postcondizioni usando il quantificatore universale. Prima di tutto, ho ridimostrato la funzione principale con le nuove postcondizioni di quella ausiliaria, ed è andata a buon fine. Successivamente, ho provato a dimostrare la funzione senza asserzioni, convinto che il prover non sarebbe riuscito a dimostrare alcune delle postcondizioni, così avrei usato altri assert che avrebbero potuti aiutarli in queste dimostrazioni. Invece, il contratto della funzione è stato dimostrato correttamente, senza asserzioni o lemmi aggiuntivi. Da qui si nota quanto sia potente e importante l'eleganza con cui bisogna scrivere la logica, e quanto risulti più facile dimostrare gli stessi concetti formulati diversamente. Di seguito la versione dimostrata delle funzioni:

```
let rec function insert_metadata_list_for_snippet (s: v) (metadata_list: list
    v) (g: graph) : graph
  requires  { is_snippet s }
  requires  { memv s g }
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m ∧ memv m g
    }
  ensures   { result.vertices = g.vertices }
  ensures   { forall e: e. meme e g → meme e result }
  ensures   { forall m: v. LM.mem m metadata_list → meme (s,m) result }
= match metadata_list with
  | Nil → g
  | Cons m l →
      let g_sm_edge = inserte (s,m) g in
      insert_metadata_list_for_snippet s l g_sm_edge
  end

let function insert_snippet (s: v) (metadata_list: list v) (g: graph) : graph
  requires  { is_snippet s }
  requires  { not (memv s g) }
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m ∧ memv m g
    }
  ensures   { result.vertices = Fset.add s g.vertices }
  ensures   { forall m: v. LM.mem m metadata_list → exists e: e. meme e
    result ∧ src e = s ∧ dst e = m }
= let g_s_inserted = insertv s g in
  insert_metadata_list_for_snippet s metadata_list g_s_inserted
```

Da notare che la seconda postcondizione della funzione principale ha mantenuto la notazione con l'esiste ed è stata dimostrata ugualmente. Per preservare coerenze, ho deciso di riformulare anche quella postcondizione, che ripeto essere

equivalente.

```
let function insert_snippet (s: v) (metadata_list: list v) (g: graph) : graph
  requires  { is_snippet s }
  requires  { not (memv s g) }
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m ∧ memv m g
    }
  ensures   { result.vertices = Fset.add s g.vertices }
  ensures   { forall m: v. LM.mem m metadata_list → meme (s,m) result }
= let g_s_inserted = insertv s g in
  insert_metadata_list_for_snippet s metadata_list g_s_inserted
```

Il contratto è stato ri-dimostrato con successo.

Conclusione

Per dimostrare la correttezza di un'implementazione, si può:

- aggiungere dei lemmi;
- aggiungere asserzioni;
- modificare l'implementazione stessa;
- riformulare il contratto in un altro equivalente.

Quando non si riesce a dimostrare un risultato, si tende a pensare che:

- bisogna fornire più contesto, aggiungendo lemmi o asserzioni, oppure
- l'implementazione è sbagliata, oppure
- il contratto è sbagliato.

Quando si dimostra con prover automatici, invece, capita spesso che l'uso di predicati equivalenti, ma scritti in modo diverso, permetta una verifica che prima non avveniva. Questo è legato al modo in cui funzionano la maggior parte dei prover automatici basati su logica classica. Intuitivamente, si può pensare al fatto che dimostrare un risultato basato su un quantificatore esistenziale sia più difficoltoso rispetto a dimostrarne uno basato su quantificatore universale, perché nel caso del quantificatore esistenziale, bisogna ottenere un testimone.
Nel dettaglio funziona così. I prover automatici, come detto in precedenza, utilizzano quasi sempre dimostrazioni per contraddizione. Quindi, quando il goal consiste in: $\forall x P(x)$, vengono svolti questi step da prover:

- Viene assunta la negazione dell'enunciato: $\neg(\forall x P(x))$
- L'assunzione viene semplificata con le leggi di De Morgan, ottenendo $\exists x(\neg P(x))$
- Viene applicata una tecnica chiamata skolemizzazione, sostituendo $\exists x$ con una costante c, ottenendo $\neg P(c)$

Dimostrare un risultato simile è più semplice perché lo spazio di ricerca è più basso. Il prover deve solo confrontare la costante ottenuta con altri enunciati nel contesto, per ottenre una contraddizione. Quando, invece, il goal consiste in: $\exists x P(x)$, vengono svolti questi step:

- Viene assunta la negazione dell'enunciato: $\neg(\exists x P(x))$
- L'assunzione viene semplificata con le leggi di De Morgan, ottenendo $\forall x \neg P(x)$

Non può essere applicata skolemizzazione, quindi il massimo risultato ottenuto dalla contraddizione è che per qualsiasi cosa nell'universo, P è falso. Per ottenere una contraddizione, bisogna trovare una costante che renda vera P. Lo spazio di ricerca può essere grandissimo, spesso infinito.
Mentre, nella dimostrazione di un per ogni, lo spazio di ricerca consiste negli enunciati, nella dimostrazione di un esiste, lo spazio di ricerca consiste negli abitanti dei tipi coinvolti. È ovvio che, quasi sempre, il numero di possibili abitanti di un tipo è estremamente maggiore del numero di enunciati accumulati.

Consiglio

Se possibile, negli enunciati, preferire quantificatori universali a quantificatori esistenziali.

Questa, è chiaramente una nota negativa sui prover automatici. In generale, dovrebbe bastare riuscire a scrivere delle specifiche corrette, nel modo più comprensibile possibile. In questo modo, invece, bisogna sviluppare un intuito sulla formulazione di enunciati, che deve essere corretta, ma in cui la difficoltà di gestione da parte di prover automatici è una priorità rispetto all'eleganza.

5.11 Cosa fare prima: testare le specifiche o implementare?

Scrivendo le implementazioni delle funzioni, ho avuto questo dubbio. Cosa è meglio fare? Quando si scrivono i soli contratti di funzioni, questi si testano con

dei lemmi. È l'unico modo. Quando bisogna implementarle, si scrive il contratto, poi si ha un dubbio: scrivo prima dei lemmi che fungano da test del contratto, o implemento la funzione direttamente?
Cosa è più giusto fare? Credo che ciò sia dibattibile. Poniamo i due scenari:

- Scrivo prima i lemmi. Sono più sicuro di non dover cambiare le specifiche, ho più fiducia nella loro correttezza. Quando implementerò la funzione, sarò molto sicuro del fatto che il contratto non vada cambiato, quindi, se ci saranno problemi nella dimostrazione, saranno probabilmente nell'implementazione. Potrebbe capitare, però, che la verifica non avvenga perché le specifiche sono state formulate in un modo tale che i prover facciano fatica a maneggiarle. In questo caso andrebbero riformulate, probabilmente anche i lemmi precedentemente verificati. C'è anche il rischio di scrivere troppi lemmi inutili, che potrebbero sovraccaricare il contesto e allungare i tempi di dimostrazione.

- Scrivo prima l'implementazione con le proprietà (asserzioni, invarianti). L'implementazione può essere vista come una "dimostrazione". Quando si programma normalmente, è come se il corpo della funzione fosse la dimostrazione, da cui si deduce l'enunciato, che non è ben formalizzato dall'intestazione della funzione. Quindi, scritta un'implementazione, che viene dimostrata rispetto alle specifiche, si è abbastanza sicuri della correttezza di quest'ultime. Ho sperimentato che quando ho scritto implementazioni corrette dall'inizio, ero riluttante a scrivere i lemmi, perché mi sembrava inutile, ero già sicuro della correttezza della funzione, che è l'obiettivo principale. Non scrivere lemmi potrebbe portare all'impossibilità di dimostrare la correttezza di implementazioni future, dato che potrebbero risultare utili. C'è una probabilità, inoltre, che quelle specifiche siano errate, quindi, oltre ad aver perso tempo, potremmo illuderci di aver scritto una funzione corretta, quando in realtà è corretta rispetto a specifiche errate, rendendola inutile.

Conclusione

Quando si scrive codice verificato, è dibattibile se sia giusto scrivere e dimostrare direttamente la correttezza di un'implementazione rispetto a un contratto o scrivere prima delle proprietà derivanti dalle specifiche.

Quindi, come decidere cosa fare, in questi casi? Dipende dalle proprie sensazioni riguardo al problema da risolvere e dalle caratteristiche di quest'ultimo. Ad

esempio, un problema può essere molto difficile da implementare ma molto semplice da formalizzare; può avere molte proprietà che derivano dalle sue specifiche, oppure molto poche. Credo che la scelta vada fatta, per ogni, funzione, in base a queste caratteristiche e alle proprie capacità nell'implementare funzioni e dedurre proprietà.

Consiglio

- Se una funzione risulta piuttosto semplice da specificare, non si intuiscano delle proprietà importanti e si intuisca un'idea di implementazione, scrivere direttamente il corpo della funzione.
- Se una funzione risulta molto difficile da specificare, scrivere prima dei lemmi che verifichino il contratto.
- Se una funzione risulta piuttosto difficile da implementare, la causa di ciò potrebbe variare. Può capitare che si abbiano bene in mente le proprietà di un problema ma non si sappia bene come sfruttarle, incapaci di ideare un algoritmo. Oppure, può succedere che non si abbia idea delle proprietà di un problema ma si abbia in mente una strategia di risoluzione. Nel primo caso, sarebbe meglio concentrarsi sull'implementazione, nel secondo potrebbe aiutare la scrittura di lemmi.

5.12 Sostituire lemmi con postcondizioni in funzioni implementate

Quando le funzioni non sono implementate, è assolutamente raccomandato scrivere un contratto minimale e dimostrare le proprietà successivamente. Nel caso le funzioni vadano implementate, questa raccomandazione non è così forte. Spiego attraverso una mia esperienza.
Ho scritto una funzione che serve a filtrare una lista di vertici, in una lista di snippet. Ho scritto una sola postcondizione, così poi da scrivere un lemma che venisse dimostrato di conseguenza. Il contratto della dimostrazione è stato dimostrato mentre il lemma no:

```
let rec function filter_snippets_from_vertices (l: list v) : list v
  ensures { forall s: v. LM.mem s result ↔ is_snippet s ∧ LM.mem s l }
= match l with
  | Nil → Nil
```

```
  | Cons x r → if (is_snippet x)
                then Cons x (filter_snippets_from_vertices r)
                else (filter_snippets_from_vertices r)
  end

lemma filter_snippets_from_vertices_length:
  forall l1 l2: list v. l2 = filter_snippets_from_vertices l1 → length l2 ≤
    length l1
```

Al che ho deciso di spostare il risultato del lemma in una postcondizione della funzione. La postcondizione è stata dimostrata con facilità:

```
let rec function filter_snippets_from_vertices (l: list v) : list v
  ensures { forall s: v. LM.mem s result ↔ is_snippet s ∧ LM.mem s l }
  ensures { length result ≤ length l }
= match l with
  | Nil → Nil
  | Cons x r → if (is_snippet x)
                then Cons x (filter_snippets_from_vertices r)
                else (filter_snippets_from_vertices r)
  end
```

In questo caso, sono autorizzato a spostare dei lemmi nella postcondizione, perché questo risultato è stato dimostrato comunque. Se nella scrittura delle specifiche (quindi funzioni senza implementazioni), le postcondizioni risultano degli assiomi, che, oltre a poter essere sbagliati, possono rendere la teoria inconsistente. In questo caso le postcondizioni vengono dimostrate. È sicuramente un approccio migliore avere un insieme minimale di postcondizioni, per dimostrare dei lemmi a partire dal contratto, piuttosto che dal corpo della funzione, però, l'approccio di aggiunta di postcondizioni, quando la funzione viene implementata, è comunque un approccio robusto. Inoltre, questa esperienza evidenzia un altro aspetto. Usando Why3, prima scrivo le specifiche di una funzione, poi scrivo il corpo della funzione per soddisfare queste specifiche. L'implementazione, però, potrebbe garantire un'altra serie di proprietà, che altrimenti verrebbero dedotte difficilmente. Quindi, il processo di scrittura di una funzione si allarga. Oltre a scrivere specifiche, scegliere se testarle e nel caso aggiungere lemmi, poi scrivere l'implementazione con le giuste asserzioni, si aggiunge l'attività di deduzione di proprietà dall'implementazione, piuttosto che dal problema. Quest'ultima operazione può essere anche tralasciata, però, dedurre altre proprietà, può risultare utile a dimostrazioni successive.

Ad esempio, in questo caso, se dovessi scrivere, successivamente, una funzione a cui serve sapere `length result <= length l` per essere dimostrata, questa proprie-

tà è stata già dedotta.
Quindi, anche quando il contratto di una funzione non viene dimostrato, può essere utile, oltre alle strategie già citate, provare a dedurre delle proprietà a partire dalle implementazioni di funzioni precedentemente scritte. Così facendo, si allarga il contesto.

Conclusione

L'implementazione di una funzione potrebbe avere più proprietà significative di quelle espresse nelle postcondizioni. Nel caso siano proprietà rilevanti, sarebbe utile aggiungerle al contratto.

5.13 Implementazioni Python. Traduzione del codice

Nel codice Why3 ho creato una distinzione tra snippet e metadati ma non ho modellato i loro campi. Ho deciso che avrebbero complicato l'astrazione inutilmente, perché non erano questioni difficili su cui ragionare. O meglio, utilizzare Why3 non avrebbe aiutato nella creazione di soluzioni migliori, sono questioni più inerenti al linguaggio che alla logica. Ci sono delle questioni riguardo alla categoria dei metadati e snippet, però, che risultano importanti e vanno gestite. Quindi, la scelta di utilizzo di un'astrazione superiore porta uno svantaggio: il codice scritto in WhyML non è del tutto aderente al codice Python. Però, confrontando le funzioni scritte nei linguaggi diversi, risulta molto semplice capire quali sono i pezzi formalizzati e dimostrati precedentemente e quali no.
Di seguito il codice WhyML:

```
let function insert_freemetadata (m: v) (g: graph) : graph
  requires  { is_metadata m }
  requires  { not (memv m g) }
  ensures   { result = insertv m g }
= insertv m g

let function insert_metadata (m p: v) (g: graph) : graph
  requires  { is_metadata m ∧ is_metadata p }
  requires  { not (memv m g) ∧ memv p g }
  ensures   { result.vertices = Fset.add m g.vertices }
  ensures   { result.edges = Fset.add (p,m) g.edges }
= let g_with_m = insert_freemetadata m g in
  inserte (p,m) g_with_m
```

E il codice Python:

```
def insert_freemetadata(self, metadata: Metadata) -> str:
    key = self._format_metdata(metadata)

    if self.memv(key):
        raise KeyError("Metadata already exists")

    # Convert metadata to dict for storage
    data = metadata.model_dump()
    self.insertv(data, key)
    return key

def insert_metadata(self, metadata: Metadata, parent: Metadata, category:
    Category) -> str:

    if metadata.category != category or parent.category != category:
        raise ValueError(f"Category mismatch: child:{metadata.category},
    parent:{parent.category} category to insert:{category}")

    new_metadata_key = self._format_metdata(metadata)
    parent_key = self._format_metdata(parent)

    if not (self.is_metadata(parent_key)):
        raise ValueError("Parent metadata doesn't exist")

    if metadata.category != parent.category:
        raise ValueError(f"Parent category ({parent.category}) and child
    category ({metadata.category}) must match")

    if self.memv(new_metadata_key):
        raise ValueError("Metadata already exists")

    data = metadata.model_dump()
    self.insertv(data, new_metadata_key)
    self.inserte(parent_key, new_metadata_key, RelationType.METADATA_PARENT)
    return new_metadata_key
```

Nel codice Python si notano una serie di if con delle try-except. È facile notare che questi corrispondono, insieme al tipaggio pydantic, alle precondizioni in Why3. Dopo queste "precondizioni", l'implementazione effettiva è molto simile. Si nota che nel codice Python, c'è un controllo in più, sulla categoria. Notare le differenze con il codice Why3 risulta semplice, quindi, nel caso di errori, si capisce bene quali parti sono simili alla formalizzazione e quali no.

5.13.1 Espressione non banale di precondizioni

Adattando il codice di una funzione che permette l'inserimento di una lista di metadati per uno snippet (è la funzione ausiliaria di `insert_snippet`), ho riscontrato una prima versione di conversione di requires a controllo con eccezione non banale. Di seguito il codice WhyML:

```
let rec function insert_metadata_list_for_snippet (s: v) (metadata_list: list
    v) (g: graph) : graph
  requires  { is_snippet s }
  requires  { memv s g }
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m ∧ memv m g
    }
  ensures   { result.vertices = g.vertices }
  ensures   { forall e: e. meme e g → meme e result }
  ensures   { forall m: v. LM.mem m metadata_list → meme (s,m) result }
  ensures   { forall m r. metadata_list = Cons m r → meme (s,m) result }
  ensures   { metadata_list = Nil → (exists m: v. is_metadata m ∧ memv m g ∧
    meme (s,m) result) }
= match metadata_list with
  | Nil → g
  | Cons m l →
    let g_sm_edge = inserte (s,m) g in
    insert_metadata_list_for_snippet s l g_sm_edge
  end
```

E il codice Python:

```
def insert_metadata_list_for_snippet(self, snippet_key: str, metadata_list:
    List[str]):
    if not self.is_snippet(snippet_key):
        raise ValueError(f"Snippet {snippet_key} isn't in db")

    for m in metadata_list:
        if not self.is_metadata(m):
            raise ValueError(f"Metadata {metadata_list} isn't in db")

    match metadata_list:
        case []: return
        case [m, *l]:
            self.inserte(snippet_key, m, RelationType.HAS_METADATA)
            self.insert_metadata_list_for_snippet(snippet_key, l)
```

Come si vede, il controllo sui metadati (presenti e coerenti), è specificato con un forall. Quindi, in Python avviene il controllo con un ciclo for. Questo viene da un'implementazione che sia più vicina possibile alle specifiche. Questo risulta

però estremamente inefficiente. Dunque si potrebbe pensare che il controllo possa essere inserito più in basso, prima dell'insirimento dell'arco, mutando la funzione così:

```
def insert_metadata_list_for_snippet(self, snippet_key: str, metadata_list:
    List[str]):
    if not self.is_snippet(snippet_key):
        raise ValueError(f"Snippet {snippet_key} isn't in db")

    match metadata_list:
        case []: return
        case [m, *l]:
            if not self.is_metadata(m): # Precondizione spostata
                raise ValueError(f"Metadata {metadata_list} isn't in db")
            self.inserte(snippet_key, m, RelationType.HAS_METADATA)
            self.insert_metadata_list_for_snippet(snippet_key, l)
```

Adesso risulta meno evidente la corrispondenza tra la precondizione in WhyML e il controllo effettuato in Python. A questo punto i commenti giocano un ruolo fondamentale, sono il mezzo che permettono di distinguere codice implementativo da quello di controllo (più descrittivo). Da notare che programmando normalmente, la differenza tra codice implementativo a controlli è molto più sfocata. Spesso si scrive prima il codice implementativo per poi scoprire degli errori di controllo più tardi. Inoltre, viene considerata programmazione l'attività che concerne entrambi i processi (programmazione e specifica), che, in Why3, notiamo essere distinti.

Da ciò traggo che imparare questo strumento può portare a sviluppare un certo intuito nel capire che "tipo di programmazione" si sta svolgendo in un determinato momento, così da sviluppare anche una capacità superiore nella scrittura dei commenti. Ad esempio, sopra l'intestazione di una funzione, si può scrivere l'equivalente di un contratto Why3, un "enunciato discorsivo" e nell'implementazione, in corrispondenza di un controllo, scrivere un commento che rimanda a quell'enunciato. Da notare che i controlli avvengono circa scrivendo un `if` con eccezione, dove l'`if` valuta la negazione di una precondizione.

Si vede che nell'implementazione di una funzione nel mio metodo, avviene un processo di raffinamento. Si sceglie un'astrazione, da cui deriva la specifica di una funzione in Why3. Si scrive un'implementazione, la cui correttezza viene dimostrata rispetto alla specifica. Qui la bisimulazione è rigorosa. Poi si passa all'implementazione nel linguaggio di destinazione. La bisimulazione qui non è affatto rigorosa: non c'è un algoritmo per passare da un'implementazione all'altra, oppure una dimostrazione che provi che l'implementazione finale corrisponda

all'implementazione Why3.

Conclusione

Si tende ad accomunare processi di natura distinta con lo stesso termine: programmazione. Attraverso il mio metodo, si impara a separare le attività in un processo di raffinamento più completo. Dalla specifica ad un'implementazione coerente dimostrata. Da un astrazione validata al codice nel linguaggio di programmazione di destinazione.

Ma torniamo al nostro esempio. I più attenti avranno notato un'imprecisione nell'ultima versione Python proposta per la funzione di inserimento di metadati per uno snippet. In realtà il controllo proposto non funziona bene, perché nel caso uno solo dei metadati non fosse presente, voglio che lo snippet non venga inserito e nessun arco relativo, quindi, il controllo va fatto per tutti i metadati della lista in input. Nel caso si voglia accettare un risultato parziale, quel codice andrebbe benissimo. Non era ciò che volevo, quindi ho dovuto cambiare l'implementazione così:

```
def _metadata_present_same_cat(self, metadata_list: List[Metadata], category:
    Category):
        match metadata_list:
            case []:
                return
            case [m, *l]:
                if not self.is_metadata(self._format_metdata(m)):
                    raise ValueError(f"Metadata {m} isn't in db")
                if m.category != category:
                    raise ValueError(f"Metadata category: {m} doesn't match
    requested category:{category}")
                self._metadata_present_same_cat(l, category)

def _insert_metadata_list_for_snippet_rec(self, snippet_key: str,
    metadata_list: List[Metadata], category: Category):
    match metadata_list:
        case []:
            return
        case [m, *l]:
            self.inserte(snippet_key, self._format_metdata(m), RelationType.
    HAS_METADATA)
            self._insert_metadata_list_for_snippet(snippet_key, l, category)

def _insert_metadata_list_for_snippet(self, snippet_key: str, metadata_list:
    List[Metadata], category: Category):
```

```
    if not self.is_snippet(snippet_key): # Precondizione
        raise ValueError(f"Snippet {snippet_key} isn't in db")
    self._metadata_present_same_cat(metadata_list, category) # Precondizione

    self._insert_metadata_list_for_snippet_rec(snippet_key, metadata_list,
    category) # Corpo della funzione
```

Come si vede, per preservare il controllo in maniera coerente, ho complicato il codice. Questa complicazione, però, ha permesso una semplificazione nella funzione madre: `insert_snippet`. Di seguito il codice WhyML:

```
let function insert_snippet (s: v) (metadata_list: list v) (g: graph) : graph
  requires  { is_snippet s }
  requires  { not (memv s g)}
  requires  { metadata_list ≠ Nil }
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m ∧ memv m g
    }
  ensures   { result.vertices = Fset.add s g.vertices }
  ensures   { forall e: e. meme e g → meme e result }
  ensures   { forall m: v. LM.mem m metadata_list → meme (s,m) result }
  ensures   { snippet_metadata_outdegree s result > 0 }
= let g_s_inserted = insertv s g in
  insert_metadata_list_for_snippet s metadata_list g_s_inserted
```

E il codice Python:

```
def insert_snippet(self, snippet: Snippet, metadata_list: List[Metadata],
    category: Category):
    if self.memv(snippet.name): # Precondizione
        raise ValueError(f"Snippet {snippet.name} already exists")
    if len(metadata_list) < 1:  # Precondizione
        raise ValueError("There should be at least one metadata name")

    data = snippet.model_dump(mode='json')
    self.insertv(data, snippet.name)
    self._insert_metadata_list_for_snippet(snippet.name, metadata_list,
    category)
```

Non ho riportato i controlli sui metadati, espressi nelle precondizioni, perché questi vengono già eseguiti nella funzione ausiliaria. Quindi, traducendo, in un caso ho dovuto complicare una funzione, nell'altro ho potuto semplificarla. Ho ottenuto una discrepanza col codice WhyML in entrambi i casi.
Ciò deriva dal fatto che Why3 è comunque un linguaggio, non è pseudocodice. Quindi è più naturale ragionare con costrutti più vicini a un linguaggio funzionale. Allontanandoci dallo stile nel linguaggio di destinazione rimaniamo fedeli

all'implementazione Why3, dimostrata. Quindi sacrifichiamo l'efficienza per rimanere conformi a un'implementazione che sappiamo essere dimostrata. D'altro canto, se siamo coerenti con lo stile del linguaggio di programmazione finale (perdiligendo leggibilità ed efficienza), tendiamo ad allontanarci dal codice originale dimostrato Why3, perdendo fiducia sulla correttezza del proprio codice.

5.13.2 Problemi di ottimizzazione

Scrivendo il codice Python, ho provato a essere il più fedele possibile alle implementazioni scritte in Why3. Quindi ho scritto prevalentemente codice funzionale, modulare, ho usato molto il costrutto di pattern matching. Ma se questo approccio, garantisse la correttezza del codice, ma ottimizzazioni piuttosto scarse? Per preservare l'ottimizzazione, credo sia giusto farlo come procedimento separato. Credo sia giusto avere prima un passaggio in cui si risolva un problema nel modo più elegante, dimostrabile e modulare possibile, per poi cercare di ottimizzare il costo computazionale. Quando si vuole risolvere un problema nel modo più ottimizzato possibile dall'inizio, si fa spesso l'errore di innamorarsi di una "scorciatoia computazionale", dimenticando i vincoli a cui deve sottostare un problema. Invece, se l'ottimizzazione viene trattata come un procedimento separato, si affronta un problema con molta più consapevolezza. Si ha un'idea molto più ampia dei vincoli, quindi non si perde tempo di implementazioni la cui idea è sbagliata da principio, a causa di un errore logico. L'ottimizzazione avverrebbe in maniera molto più analitica.

5.14 Utilizzo di scorciatoie di linguaggio in WhyML

I linguaggi hanno dei limiti, designati da vincoli. Questi vengono creati per guidare il programmatore, secondo certe filosofie di sviluppo. Esistono poi, delle tecniche consigliabili e delle strategie da evitare. Questo avviene in WhyML come avviene in tutti i linguaggi. In un esempio che riporto, ho utilizzato una strategia da evitare, che rappresenta una scorciatoia.
Stavo scrivendo una funzione che prenda tutti gli snippet raggiungibili da una lista di metadati, eccola:

```
let rec function get_snippets_union (metadata_list: list v) (g: graph) : fset
    v
  requires  { forall m: v. LM.mem m metadata_list → is_metadata m }
  ensures   { forall m: v. LM.mem m metadata_list → (Fset.subset (
    get_all_snippets_from_metadata m g) result) }
```

```
= match metadata_list with
  | Nil → Fset.empty
  | Cons m r → Fset.union (get_all_snippets_from_metadata m g) (
    get_snippets_union r g)
  end
```

Questa ha un problema, sto utilizzando gli fset, che sono oggetti logici che posso utilizzare solo in contesti appunto logici (lemmi, postcondizioni, precondizioni, assiomi e ghost code) e non nel corpo di funzioni eseguibili. Perciò, avevo un problema di compilazione.
Per rimediare avrei dovuto usare gli applicative sets, un modulo della libreria standard, che permette di usare la versione degli insiemi eseguibile per codice classico. Però, per poterli usare, per loro costituzione, dovrei clonare il modulo sui vertici, per poi poterlo utilizzare. Prima di farlo, ho comunque voluto verificare che il codice fosse corretto, quindi ho segnalato la funzione in questione come ghost, così da rendere legale l'utilizzo di fset e poter verificare la correttezza della funzione. Ho riscontrato che la funzione era corretta.
Quindi, adesso, convinto del ragionamento, avrei potuto investire tempo nell'adattamento. In questo caso, però, non l'ho fatto. È un uso errato del linguaggio, ma non è grave.
Non bisogna mai confondere le funzioni ghost con quelle normali, perché è giusto separare i due piani. Soprattutto, se si scrive una funzione normale, che si segna ghost per comodità, questa potrà essere utilizzata solo in contesti ghost, quindi, le funzioni che la utilizzeranno dovranno essere ghost anch'esse, a cascata. Così si andrebbero a perdere totalmente tutti i vantaggi che offre il ghost code. Se in questo caso mi permetto di barare, è perché sono sicuro che la funzione che ho scritto non verrà utilizzata da altre funzioni. Essa è una funzione finale, che utilizzerà l'utente e non è utile a nessuna funzione che dovrò implementare in futuro. Sto facendo questo per risparmiare tempo. Quindi questo è un esempio di scorciatoia illegale ma tollerabile, esplicativo sul grado degli errori. Soprattutto, se mi sono permesso di farlo, è perché uso Why3 come strumento per ragionare e validare i miei ragionamenti e non per scrivere codice finale, tradotto automaticamente. In questo caso, utilizzare scorciatoie di linguaggio, preserva efficienza nella programmazione, permette di non dilungarsi in formalizzazioni poco utili ai problemi che si vogliono affrontare. L'obiettivo principale è sempre quello di scrivere codice corretto nel linguaggio di programmazione di destinazione.

Capitolo 6

Conclusioni

Tutto il lavoro è stato svolto per rispondere ad una necessità: trovare un metodo o delle approssimazioni che regalassero più fiducia nella scrittura del proprio codice. Questo è avvenuto, per me. La necessità è stata soddisfatta ma, soprattutto, sono stati riportati dei mezzi per poter capire al meglio in che modo andrebbe sfruttata la metodologia proposta, con i suoi vantaggi e le sue limitazioni. Ci sono, però, ancora delle considerazioni da riportare.

6.1 Considerazione sul tempo e sul codice prodotto

Nel progetto, ho scritto circa 1200 righe di codice Python e 650 righe di codice Why3. Il codice Why3 potrebbe risultare tanto, considerando che riguarda solo due moduli su tre. Dunque, è stato scritto circa quanto codice Why3 quanto quello Python. Si ipotizzerà che ci sia stato uno spreco di tempo. Questo dipende sempre. Nel caso del mio progetto, credo che non ci sia stato uno spreco di tempo, anzi credo che il tempo investito sia stato di qualità superiore, rispetto a un normale processo di programmazione. Non mi è capitato quasi mai di dover modificare codice Python, che è il codice finale, quello più importante. Non ci sono stati bug impossibili da trovare. Le difficoltà sono state sempre nella risoluzione del singolo problema. Il tempo speso a svolgere questo progetto è stato circa equivalente al progetto svolto nel tirocinio (senza test, usando un database a grafo con query dichiarative). Il progetto attuale, presentava il vantaggio di conoscere bene le specifiche, data la similitudine con quello precedente, però usava una tecnologia molto più difficile da gestire. Qui ho dovuto gestire i grafi manualmente, nell'altro progetto ho semplicemente dovuto scrivere delle query

dichiarative. Inoltre, conoscevo di più il linguaggio usato nel progetto precedente (typescript) rispetto al linguaggio usato adesso (Python). Nel progetto precedente, come spesso è accaduto, mi è capitato più volte di modificare funzioni vecchie, inoltre, c'era poco riuso di funzioni. Quindi, il tempo che ho spesso impiegato cercando di dimostrare delle funzioni qui, scrivendo in un altro linguaggio, nell'altro progetto è stato spesso impiegato nella ricerca degli errori. Nella risoluzione di un singolo problema, però, il contesto di riferimento è più piccolo. Non riuscire a dimostrare/specificare/implementare un singolo problema può essere frustrante ma si sta gestendo un problema singolo. È meno frustrante rispetto a cercare un errore che può essere in qualsiasi funzione. Spesso cercare un errore in un programma è come cercare l'ago in un pagliaio, il che è molto frustrante. Questo influisce sulla produttività del programmatore, la cui lucidità è importantissima nello sviluppo giornaliero.
Il fatto che il codice Why3 venga dimostrato è importante, ma non credo sia causa principale della qualità del codice prodotto e del tempo speso. Credo che la causa principale sia che uno strumento come Why3 impone al programmatore un freno, egli non può scrivere codice per eseguirlo a più non posso, per accontentarsi di test su casi isolati.
Il programmatore, quando usa questi strumenti, ha l'obbligo di pensare meglio al problema in generale e carpirne le proprietà, il che rende molto meno frenetico lo sviluppo. Con Why3 si è più inclini ad avere un approccio sobrero, rispetto ad un approccio magician hat. Col primo, si tende a dover capire un problema in modo più estensivo, le funzioni importanti risultano più semplici da creare, svilupparle diventa un effetto collaterale dello sviluppo di una teoria coerente.
Normalmente, si è inclini a voler scrivere solo il codice che si ritiene utile, avendo una scarsa conoscenza sulla teoria su cui si regge il problema che si vuole risolvere. Alle volte il programmatore è preso dalla frenesia di risolvere problemi, spesso sottostimati. Mitigare questa frenesia permette di avere un approccio in cui si capisce un problema e poi si risolve, piuttosto che l'approccio di produrre codice che risolva un problema il più velocemente possibile. Inoltre, strumenti simili, puniscono molto l'errore. Non vedere il proprio codice dimostrato, permette di rivalutare molto più facilmente il proprio operato. La sensazione che si ha quando si lancia un prover è quella di avere un giudice di autorità superiore alla propria. Così, ci si abitua a vedere i propri errori dall'inizio, quando il codice è fresco. Grazie a questa esperienza, si normalizza il fatto che si sbaglia di continuo.

6.1.1 Sul peso del codice

Utilizzare uno strumento simile col metodo da me ideato, permette di attribuire un peso al codice che si scrive. Ci sono programmi che richiedono la scrittura di molte righe di codice ma molto più semplice da scrivere e viceversa. L'uso di Why3 come strumento per formalizzare determinati pezzi di codice, permette di attribuire un peso maggiore al codice che si sceglie di formalizzare. Più nel dettaglio, il peso può essere attribuito in base al livello di astrazione che si sceglie di dare a una certa formalizzazione. Ad esempio, nel mio progetto, ho scelto di non astrarre le categorie, perché rappresentavano un problema semplice. Documentare i livelli di astrazione che si attribuiscono alle formalizzazioni, permette, a chi leggerà il codice, di capire quale sarà il codice più complicato da capire e quale meno.

6.2 Integrazione in un contesto aziendale

Se si dovesse decidere di utilizzare un metodo di sviluppo software che si ispira a quello proposto, si dovrebbe capire come integrarlo in un team di sviluppo che conosce Why3 e WhyML. Apporterei dei cambiamenti minimi. Ad ogni programmatore, normalmente, vengono assegnate delle task, più o meno specifiche, più o meno rigorose. Sulle implementazioni di queste task, andrebbero fatti dei test. I test vengono fatti normalmente dopo lo sviluppo delle implementazioni, a volte prima (test driven programming). Why3 porterebbe un cambiamento minimo. Colui che avrà assegnato la task e il programmatore potrebbero accordarsi sul sistema di verifica di quella specifica task. Concorderanno se valga la pena modellare logicamente (e a che livello di astrazione) il problema per dimostrarlo e poi implementarlo nel linguaggio specifico, senza test; oppure se programmare nel modo totalmente classico, con verificazione empirica.
Questo permetterebbe di individuare i pezzi di software più adatti ai diversi tipi di verifica, su cui, nel tempo, si potrebbe sviluppare un certo intuito. Il metodo di verifica andrebbe concordato perchè si deve adattare non solo alle specifiche del software ma anche alle capacità e alle caratteristiche del programmatore. Tutti i programmatori che lavorerebbero ad un software con tecniche di verifica miste dovrebbero però conoscere entrambi i metodi.

6.3 Alla fine? Ne vale la pena?

Questa è una domanda a cui è impossibile rispondere.
È una domanda estremamente personale. Il metodo proposto ha il problema di avere delle componenti arbitrarie. Il programmatore deve scegliere il livello di astrazione da usare in Why3 e che funzioni valga la pena implementare. Queste scelte sono prive di rigore. Il processo è formale solo nello sviluppo delle singole funzioni. Dunque, le scelte potrebbero rivelarsi, in futuro, poco efficienti o poco formali. Si potrebbe scegliere un livello di astrazione troppo alto, che trascurerebbe troppi dettagli importanti: il raffinamento da WhyML al linguaggio finale potrebbe non essere affatto scontato, si perderebbero i vantaggi di Why3. Un livello di astrazione troppo dettagliato potrebbe invece risultare estremamente complicato da formalizzare e implementare in Why3: si perderebbe troppo tempo in formalizzazioni inutili.
Le scelte, quindi, sono fondamentali e determinanti. Non esiste un modo per fare scelte giuste, questo metodo richiede esercitazione ed esperienza.
Il programmatore che avrà un certo intuito nella scelta di un'astrazione consona sarà probabilmente avvantaggiato dall'uso anomalo che propongo di Why3. Per capire se ne valga la pena l'unico modo è provare e analizzare la propria esperienza.
A questo punto, a prescindere dal metodo sviluppato, capire se sia più giusto utilizzare strumenti come Why3 (seppur a un più alto livello di astrazione) in alternativa ai test è pressappoco impossibile. Infatti, è poco utile dare una risposta a questa domanda. Posso rispondere per me, ritengo che per me sia valsa la pena. Invece, ritengo sia utile che si possa rispondere facilmente alla domanda: vale la pena provare?
Ho cominciato questo percorso con una scarsa conoscenza in questo ambito. Non sapevo neanche cosa fosse la logica di Hoare. La documentazione e le risorse erano scarse. Dunque, tutte le conclusioni riportate sono state dedotte dall'esperienza, che consisteva nel programmare con l'ausilio di questi strumenti. Grazie a questi ho capito proprietà sull'attività di programmazione che in tanti anni non avevo mai capito. Ci sono cose che forse non avrei mai capito, altre che probabilmente avrei interiorizzato negli anni.
La programmazione è un'attività logica, quindi, utilizzare strumenti che hanno un approccio logico è essenziale per capire al meglio le insidie che ci sono dietro ad uno dei processi più complicati a cui ci si può sottoporre.
Quindi ritengo che valga la pena provare a usare strumenti di verifica formale. Io ho capito che sono adatti alla mia persona, e l'ho capito solo provandolo. Se,

in alternativa, ci si vede rallentati con l'uso di questi strumenti, ritengo comunque che il loro utilizzo possa essere illuminante per migliorare il proprio stile di programmazione.

Bibliografia

[1] François Bobot et al. *The Why3 Platform: User Manual*. Manuale ufficiale: riferimento per la sintassi WhyML e la libreria standard. Inria. 2024. URL: https://why3.lri.fr/doc/.

[2] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Manuale di riferimento per TLA+. Addison-Wesley, 2002.

[3] ClearSy. *Atelier B User Manual*. Documentazione ufficiale consultata per il funzionamento del B-Method. ClearSy. 2024. URL: https://www.atelierb.eu/en/documentation-atelier-b/.

[4] Charles Antony Richard Hoare. «An axiomatic basis for computer programming». In: *Communications of the ACM* 12.10 (1969). L'articolo breve originale che introduce le precondizioni e postcondizioni, pp. 576–580.

[5] The Iris Team. *The Iris Project Documentation*. Documentazione ufficiale e tutorial consultati per lo studio del framework Iris. MPI-SWS. 2024. URL: https://iris-project.org/.

[6] Philip Wadler. «Propositions as types». In: *Communications of the ACM* 58.12 (2015). Una introduzione alla corrispondenza Curry-Howard, pp. 75–84.

[7] Nikolaj Bjørner, Leonardo De Moura et al. *Programming Z3*. Guida pratica e tutorial interattivo per l'uso di Z3 e SMT solvers. Microsoft Research. 2024. URL: https://microsoft.github.io/z3guide/.

[8] NetworkX Developers. *NetworkX Documentation*. Consultata per l'analisi delle API e la ricostruzione della teoria dei grafi. NetworkX. 2024. URL: https://networkx.org/documentation/stable/.

[9] Alistair Cockburn. *Hexagonal Architecture*. https://alistair.cockburn.us/hexagonal-architecture/. Riferimento per la separazione tra logica core e adattatori esterni. 2005.

[10] Anthropic. *Model Context Protocol (MCP) Documentation*. https://modelcontextprotocol.io/. Specifiche tecniche per il progetto di tirocinio. 2024.