ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Type Checking in Rust per il kernel di Lean 4

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Mattia Albertini

Sessione II
Anno Accademico 2024/2025

Abstract

Questa tesi si propone di sviluppare un type checker in Rust per l'export del linguaggio Lean. L'obiettivo principale è quello di realizzare un sistema indipendente in grado di verificare la correttezza delle dimostrazioni prodotte da Lean. Infatti, poiché il kernel di Lean è responsabile della validazione logica delle prove, un eventuale bug al suo interno potrebbe compromettere l'affidabilità dei risultati. Reimplementando il kernel in modo autonomo si riduce il rischio di replicare gli stessi errori, aumentando così la probabilità di individuare eventuali incongruenze o bug nascosti nelle dimostrazioni. Inoltre, disporre di una versione indipendente del kernel consente una maggiore libertà di intervento sulle strutture dati e sugli algoritmi, facilitando l'esplorazione di possibili ottimizzazioni e miglioramenti prestazionali.
Il programma, come primo passo, deve leggere il file di export e memorizzare tutte le informazioni in esso contenute, suddividendole in base alla loro tipologia e verificando che ciascuna informazione sia valida. Successivamente, deve controllare che le dimostrazioni siano state definite correttamente, verificando i tipi delle dimostrazioni e, qualora presenti, che i valori al loro interno siano di tipo compatibile.

Indice

Capitolo 1

Introduzione

I dimostratori interattivi di teoremi sono solitamente suddivisi in due componenti principali: una parte esterna, che interpreta i comandi dell'utente e costruisce le dimostrazioni, e una parte interna, detta kernel, che ha il compito di verificare in modo rigoroso la correttezza logica delle prove secondo le regole del sistema formale.
Il kernel di un dimostratore è generalmente implementato in un linguaggio di programmazione tradizionale ed è progettato per essere il più semplice e affidabile possibile, poiché costituisce la base di fiducia dell'intero sistema. La parte esterna, invece, può essere scritta nel linguaggio del dimostratore stesso e si occupa delle euristiche, dell'interfaccia e della costruzione delle prove.

L'isomorfismo di Curry–Howard, che stabilisce una corrispondenza tra logica e programmazione, dove le dimostrazioni corrispondono a programmi e la loro verifica a un controllo di tipaggio, suggerisce una possibile direzione di lavoro: riprodurre la logica di un kernel all'interno di un linguaggio di programmazione distinto.
In questo modo si separa la componente di inferenza e costruzione delle prove da quella di verifica formale, affidando quest'ultima a un linguaggio con una gestione più esplicita della memoria e un maggiore controllo sulle strutture dati.

In questo progetto è stato scelto di lavorare con Lean come dimostratore di riferimento e con Rust come linguaggio di implementazione. Lean è stato selezionato per la semplicità del suo kernel e per la possibilità di generare un file di esportazione che rappresenta in modo esplicito le strutture interne del sistema. Rust è stato invece scelto per le sue caratteristiche di

efficienza, sicurezza e controllo della memoria, che lo rendono adatto alla realizzazione di un sistema di verifica robusto e affidabile.

1.1 Idea originale per la tesi

L'idea originale per la tesi era quella di modificare un type checker in Rust già esistente[6] per implementare i grafi come rappresentazione delle definizioni delle dimostrazioni. Tale scelta si basava sull'esistenza di algoritmi sui grafi che possono risultare utili per l'analisi di una definizione e per il controllo di uguaglianza tra due definizioni. Questi algoritmi risultano asintoticamente più efficienti su certi esempi, ma non sono mai stati testati su una libreria reale di un dimostratore interattivo per vedere come si comportano in pratica nei casi più frequenti.

1.2 Problemi riscontrati

Questa idea non è stata praticabile a causa di diversi problemi emersi durante l'analisi delle documentazioni e del programma.
Un primo problema è stato riscontrato nella fase di analisi della documentazione, in cui Lean descriveva la modalità di creazione del proprio file di export, utile per comprendere la logica dietro il type checking. Sono emerse, tuttavia, incongruenze tra quanto riportato nella documentazione e ciò che veniva effettivamente letto dal programma. Si è poi scoperto che le documentazioni di Lean relative alla definizione dell'export non erano state aggiornate, rendendole quindi non pienamente affidabili.
Altri problemi sono stati riscontrati nella logica stessa del programma. Uno di questi consisteva nel segnalare errori durante l'analisi di file in cui tali errori non erano presenti, in particolare controllando l'esistenza di strutture anche quando queste non venivano richieste. Inoltre, il programma funzionava leggendo prima un file di configurazione per poi procedere alla lettura del file di export; tuttavia, tale configurazione non veniva mai definita in modo chiaro, rendendo quindi difficile l'utilizzo del programma.

1.3 La nuova idea

A causa di questi problemi, l'obiettivo della tesi è stato ridefinito come la creazione di un type checker in Rust per un file di export di Lean, utilizzando come riferimento una documentazione che descrive come costruire un sistema di type checking per Lean basandosi sulla versione più recente del file di export[1]; questo lavoro può essere un primo passo verso l'obiettivo originale: una volta ottenuto un kernel indipendente corretto che si basa sulla medesima logica di Lean, si potrà procedere con il testare variazioni nell'implementazione.
Poiché i tipi del kernel vengono associati a un indice identificativo, nel mio programma ho deciso di utilizzare per il loro salvataggio la struttura `IndexSet` che, similmente a una `HashMap`, memorizza i dati tramite l'associazione di un hash, ma mantiene anche l'ordine di inserimento degli elementi. Questa caratteristica risulta utile per verificare la corretta stesura dell'export, in quanto consente di controllare se un certo valore sia inserito nella posizione appropriata.
Per gestire al meglio il lavoro sui tipi, li ho implementati come `enum`, assegnando ai vari casi della loro grammatica un hash specifico, così da poter utilizzare l'`IndexSet`. Quest'ultimo viene generato "sommando" tutti i dati relativi a un elemento insieme a una costante ideata per ogni elemento di ciascuna grammatica, riducendo così la probabilità di collisioni con elementi già salvati.
Infine, ho creato una struttura denominata `Ptr`, di tipo generico `A`, composta da un indice e da un elemento `PhantomData`, che consente alla struttura di utilizzare i parametri e le funzioni legate al tipo `A`. Il `Ptr` è l'elemento restituito dall'`IndexSet` dopo il salvataggio e viene impiegato per memorizzare i dati che compongono gli elementi delle grammatiche.

Capitolo 2

Nomi

Il primo dei tipi primitivi del kernel è il Nome; esso viene utilizzato dagli elementi del kernel per indirizzare altri oggetti. La BNF di un nome è:

$$\textit{name} ::= \epsilon \mid \textit{name}.\textit{id} \mid \textit{name}.\texttt{nat}$$

che all'interno del file export diventa:

$$\textit{Name} ::= \texttt{Anonymous} \mid \texttt{Str}\ \textit{Name}\ \texttt{String} \mid \texttt{Num}\ \textit{Name}\ \texttt{Nat}$$

ed è definita come segue:

- `Anonymous` è il valore con indice 0;
- `Str` *Name* `String` aggiunge una `String` al *Name*; ad esempio, `Prod.mk` viene riscritto come: `Str (Str (Anonymous) "Prod") "mk"`;
- `Num` *Name* `Nat` aggiunge un `Nat` al *Name*; ad esempio, `foo.7` viene riscritto come: `Num (Str (Anonymous) "foo") 7`.

All'interno del file di export non viene mai definito il valore `Anonymous`, pertanto ho dovuto inizializzare l'`IndexSet` dei nomi con `Anonymous` per assegnargli l'indice 0.

Capitolo 3

Universi

Il secondo tipo primitivo del kernel è l'universo, utilizzato per rappresentare il livello dell'universo del tipo di un elemento.
La BNF di un universo è:

$$\texttt{univ} ::= \texttt{0} \mid \texttt{succ univ} \mid \texttt{max univ univ} \mid \texttt{imax univ univ} \mid \textit{name}$$

che nell'export diventa:

$$\textit{Universe} ::= \texttt{Zero} \mid \texttt{Succ}\ \textit{Universe} \mid \texttt{Max}\ \textit{Universe}\ \textit{Universe} \mid \texttt{IMax}\ \textit{Universe}\ \textit{Universe} \mid \texttt{Param}\ \textit{Name}$$

ed è definita come segue:

- `Zero` rappresenta il `Prop` (ovvero $\texttt{Type}_0$);
- `Succ` *Universe* rappresenta l'universo successivo a quello associato all'*Universe*;
- `Max` *Universe Universe* rappresenta il valore massimo tra i due *Universe*;
- `IMax` *Universe Universe* rappresenta `Zero` se il secondo valore è `Zero`, altrimenti svolge la stessa funzione di `Max`;
- `Param` *Name* è utilizzato per associare un *Universe* al *Name*.

All'interno del file di export non viene mai definito il valore `Zero`; per questo motivo ho dovuto inizializzare l'`IndexSet` degli universi con `Zero`, assegnandogli l'indice 0.
Poiché nel corso del lavoro è frequente la gestione di vettori di universi, ho deciso di implementare anche per questi vettori una struttura `Ptr`, denominata `UparamsPtr`, insieme a un `IndexSet`.

Semplificazione

Per poter lavorare più facilmente sugli universi, è possibile semplificarne il valore tramite le funzioni `semplifica` e `riduci`.

La funzione `semplifica` prende in input un universo e si comporta come segue:

- `Zero` ritorna `Zero`;
- `Succ` semplifica il valore precedente e ritorna il successivo del valore semplificato;
- `Max` semplifica i due *Universe* e poi chiama la funzione `riduci` sugli elementi semplificati;
- `IMax` semplifica i due *Universe*, controlla se il secondo valore è `Zero`; in tal caso ritorna `Zero`, altrimenti richiama la funzione `riduci`;
- `Param` ritorna se stesso.

La funzione `riduci` prende in input due universi e un valore booleano (inizialmente impostato a `false`) che serve a determinare quale dei due universi sia maggiore. Essa si comporta come segue:

- se il primo è `Zero`, ritorna il secondo;
- se il secondo è `Zero`, ritorna il primo;
- se entrambi sono `Succ`, richiama `riduci` sui due precedenti;
- se il primo è `Param`, richiama `semplifica` sul secondo universo e ritorna un `Max` composto dal `Param` e dal valore semplificato;

- se il secondo è `Param`, fa l'opposto del caso precedente;
- se uno dei due è `Max` o `IMax`, semplifica entrambi gli universi e, se il booleano è `false`, richiama `riduci` impostando il booleano a `true`; in caso contrario, ritorna il `Max` dei due valori.

Controllo dei parametri

Un'altra funzione utile consente di verificare se un certo universo è un parametro contenuto in un vettore di universi. Se l'universo analizzato è:

- `Zero`, ritorna `true`, poiché non utilizza parametri;
- `Succ`, richiama la funzione sul precedente;
- `Max` o `IMax`, richiama la funzione su entrambi i valori;
- `Param`, controlla se il parametro è presente nel vettore e ritorna `false` in caso contrario.

Uguaglianza

Nel kernel di Lean, l'uguaglianza tra due universi è definita tramite asimmetria: dati due universi x e y, essi sono uguali se $x \leq y$ e $y \leq x$. È quindi necessario definire la funzione che calcola $x \leq y$.
Questa funzione, denominata `leq`, prende in input due universi e un numero chiamato `deph_r`, utilizzato per salvare la profondità del secondo valore, ossia il numero di `Succ` aperti, utile per stabilire se il secondo valore sia maggiore del primo.
La funzione inizia semplificando i due universi, quindi controlla la coppia dei valori semplificati. In particolare:

- (`Zero`, _) e `deph_r` ≥ 0: ritorna `true`;
- (_, `Zero`) e `deph_r` < 0: ritorna `false`;
- (`Zero`, `Zero`): ritorna `true` se `deph_r`≥ 0;

- (`Param`, `Zero`): ritorna `false`;
- (`Zero`, `Param`): ritorna `true` se `deph_r`≥ 0;
- (`Param`, `Param`): ritorna `true` se rappresentano lo stesso nome e `deph_r`≥ 0, altrimenti `false`;
- (`Succ`, `Succ`): richiama `leq` sui precedenti;
- (`Succ`, _): richiama `leq` con il precedente del primo e il secondo universo, riducendo `deph_r` di uno;
- (_, `Succ`): richiama `leq` con il primo universo e il precedente del secondo, aumentando `deph_r` di uno;
- (`IMax`, _) o (_, `IMax`): se il secondo valore dell'`IMax` è `Zero`, richiama `leq` sostituendo `IMax` con `Zero`;
- (`IMax`, `IMax`) o (`Max`, `Max`): se i valori dei due `IMax` (o `Max`) sono uguali a coppie, ordinate o invertite, ritorna `true` se `deph_r`≥ 0;
- (`IMax`, _) o (_, `IMax`) in cui il secondo valore è un parametro: controlla il comportamento della `leq` sostituendo il `Param` con `Zero` o con il suo successivo;
- (`IMax`, _) o (`Max`, _): verifica che il secondo elemento sia maggiore di entrambi i valori del `IMax` (o `Max`);
- (_, `IMax`) o (_, `Max`): verifica che almeno uno dei due valori del `IMax` (o `Max`) sia maggiore del primo.

Inoltre, è stata definita la funzione `eq_many`, che prende in input due vettori di universi, verifica che abbiano la stessa lunghezza e non siano vuoti, e controlla che per ogni coppia (x, y) valga sia $x \leq y$ sia $y \leq x$.

Sostituzione degli universi

Un'ulteriore funzione riguarda la sostituzione di un certo parametro con un altro.
Questo compito è svolto dalla funzione `subst_universe`, che prende in input un universo e due vettori di universi (entrambi formati da parametri). La funzione analizza l'universo, richiama se stessa ricorsivamente su tutti i campi e ricrea l'elemento ottenuto; se trova un parametro, controlla le coppie dei due vettori e, se il parametro è presente nel primo elemento della coppia, ritorna il secondo, altrimenti mantiene quello originale.
Assieme a questa funzione è stata implementata `subst_universes`, che prende in input tre vettori di universi e, per ogni elemento del primo vettore, esegue `subst_universe`, restituendo infine un vettore contenente tutti gli universi "aggiornati".

Capitolo 4

Espressioni

Il terzo tipo primitivo del kernel è l'espressione, che serve a rappresentare la logica del programma che ha generato il file di export analizzato.
La BNF di un'espressione è:

$$\begin{aligned} \textit{expr} ::= {} & \textit{univ} \mid \textit{name} \mid \textit{var} \mid \lambda\, \textit{name} : \textit{expr}, \textit{expr} \mid \Pi\, \textit{name} : \textit{expr}, \textit{expr} \\ & \mid \textit{expr}\ \textit{expr} \mid \textit{name}.\texttt{nat}\ \textit{expr} \mid \texttt{let}\ \textit{name} : \textit{expr} := \textit{expr}\ \texttt{in}\ \textit{expr} \\ & \mid \textit{expr}.\textit{name} \mid \texttt{nat} \mid \texttt{string} \end{aligned}$$

che all'interno dell'export diventa:

$$\begin{aligned} \textit{Expr} ::= {} & \texttt{Sort}\ \textit{Universe} \mid \texttt{Const}\ \textit{Name}\ \textit{Universe}^* \mid \texttt{Var}\ \texttt{Nat} \\ & \mid \texttt{Lambda}\ \texttt{BinderInfo}\ \textit{Name}\ \textit{Expr}\ \textit{Expr} \mid \texttt{Pi}\ \texttt{BinderInfo}\ \textit{Name}\ \textit{Expr}\ \textit{Expr} \\ & \mid \texttt{App}\ \textit{Expr}\ \textit{Expr} \mid \texttt{Proj}\ \textit{Name}\ \texttt{Nat}\ \textit{Expr} \mid \texttt{Let}\ \textit{Name}\ \textit{Expr}\ \textit{Expr}\ \textit{Expr} \\ & \mid \texttt{NatLit}\ \texttt{Nat} \mid \texttt{StringLit}\ \texttt{Hex}^* \end{aligned}$$

$$\texttt{BinderInfo} ::= \texttt{Default} \mid \texttt{Implicit} \mid \texttt{InstanceImplicit} \mid \texttt{StrictImplicit}$$

che all'interno del programma di Rust diventa, con l'aggiunta delle `FreeVar`:

```
pub enum Expr<'a> {
    Var { hash: u64, dbj_idx: u32, },
```

```
    FreeVar {hash: u64, idx: u32, ty: ExprPtr<'a>, },
    Sort { hash: u64, universe: UniversePtr<'a>, },
    Const { hash: u64, name: NamePtr<'a>,
        universes: UparamsPtr<'a>, },
    App { hash: u64, fun: ExprPtr<'a>, arg: ExprPtr<'a>,  },
    Lambda { hash: u64, name: NamePtr<'a>,  ty: ExprPtr<'a>,
        body: ExprPtr<'a>, },
    Pi { hash: u64, name: NamePtr<'a>,  ty: ExprPtr<'a>,
        body: ExprPtr<'a>,  },
    Let { hash: u64, name: NamePtr<'a>, ty: ExprPtr<'a>,
        val: ExprPtr<'a>, body: ExprPtr<'a>,  },
    Proj { hash: u64, name: NamePtr<'a>, idx: u32,
        structure: ExprPtr<'a>,},
    NatLit { hash: u64, val: u128, },
    StrLit { hash: u64, val: String, },
}
```

Essa è definita nel seguente modo:

- `Sort`: associa un *Universe* a un'*Expr*;
- `Const`: associa una dichiarazione, identificata tramite il suo *Name*, e un insieme di universi usati come parametri universali per la dichiarazione;
- `Var`: identifica una variabile implementata con un numero secondo la logica degli indici de Bruijn;
- `FreeVar`: rappresenta un "riempitivo" che sostituisce una `Var`, indicando al sistema che in quel punto arriverà un valore di un certo tipo, definito dal valore di *Expr*. A ciascuna `FreeVar` è associato un indice che la rende univoca;
- `App`: è composta da una funzione e da un argomento, quest'ultimo usato per sostituire una `Var` all'interno della funzione;
- `Pi`: è composta da un *Name*, un `BinderInfo` (che identifica il tipo di parentesi usata), una *Expr* che rappresenta il tipo e una *Expr* che rappresenta il corpo della funzione;

- `Lambda`: è strutturata in modo analogo alla funzione `Pi`;

- `Let`: è composta da un *Name*, una *Expr* che rappresenta il tipo, una *Expr* che rappresenta il valore (analoga all'argomento dell'`App`) e una *Expr* che rappresenta il corpo in cui il valore viene sostituito a una `Var`;

- `Proj`: è formata da un *Name* (il nome della struttura), un indice che identifica il campo della struttura e una *Expr* che rappresenta la struttura stessa.

Istanziamento e astrazione

Due funzioni fondamentali per le *Expr* sono `inst` e `abstr`.

La funzione `inst` prende in input un'*Expr*, un valore (anch'esso un'*Expr*) e un numero, inizialmente impostato a 0. Essa legge ricorsivamente la struttura dell'espressione, sostituendo le `Var` il cui indice coincide con il numero dato in input. Ogni volta che viene letto il corpo di una funzione (`Lambda`, `Pi` o `Let`), il numero in input viene incrementato. Se si incontra una `Var`, essa viene:

- sostituita con il valore, se l'indice è uguale al numero in input;

- decrementata di uno, se l'indice è maggiore;

- lasciata invariata, altrimenti.

La funzione `abstr` opera in modo duale: data un'*Expr*, una `FreeVar` e un numero inizialmente 0, sostituisce le `FreeVar` dell'espressione che hanno lo stesso indice della `FreeVar` passata in input con una `Var` di indice pari al numero in input. Anche in questo caso il numero viene incrementato nei corpi di funzione. Se si trova una `Var`, essa viene:

- incrementata di uno, se il suo indice è maggiore o uguale al numero in input;

- lasciata invariata, altrimenti.

Sostituzione degli universi

La funzione `subst_expr_universes`, in modo analogo a `subst_universe`, prende in input un'espressione e due vettori di universi (composti da parametri). Essa attraversa ricorsivamente l'espressione e, quando trova:

- una `Sort`, richiama la funzione `subst_universe` sull'universo associato;
- una `Const`, richiama la funzione `subst_universes` sul vettore di universi.

Capitolo 5

Dichiarazioni

L'ultimo elemento del kernel è costituito dalle dichiarazioni, da cui si parte per eseguire i controlli di tipaggio. Tali controlli consistono nel verificare se il tipo della dichiarazione sia riducibile a un `Sort`.

La BNF di una dichiarazione è:

$$\begin{aligned}
\texttt{decl} ::= \ & \texttt{def}\ \textit{name}\ (\textit{name} : \textit{expr})^*\ : \textit{expr}\ :=\ \textit{expr} \\
| \ & \texttt{axiom}\ \textit{name} : \textit{expr}\ \mid\ \texttt{theorem}\ \textit{name} : \textit{expr} \\
| \ & \texttt{inductive}\ \textit{name}\ (\textit{name} : \textit{expr})^*\ : \textit{expr}\ \texttt{where}\ (\ \textit{name} : \textit{expr}\ )^+ \\
| \ & \texttt{structure}\ \textit{name}\ (\textit{name} : \textit{expr})^*\ \texttt{where}\ \textit{expr}
\end{aligned}$$

che all'interno del file export diventa:

$$\begin{aligned}
\texttt{Dichiarazione} ::= \ & \texttt{Axiom}\ \textit{Name Expr Name}^*\ \mid\ \texttt{Quotient}\ \textit{Name Expr Name}^* \\
| \ & \texttt{Opaque}\ \textit{Name Expr Expr Name}^*\ \mid\ \texttt{Theorem}\ \textit{Name Expr Expr Name}^* \\
| \ & \texttt{Definition}\ \textit{Name Expr Expr}\ \texttt{Hint}\ \textit{Name}^* \\
| \ & \texttt{Inductive}\ \textit{Name Expr}\ \texttt{Bool Bool Nat Nat Nat Nat} \\
& \quad \textit{Name}^*\ \texttt{Nat}\ \textit{Name}^*\ \textit{Name}^* \\
| \ & \texttt{Constructor}\ \textit{Name Expr Name}\ \texttt{Nat Nat Nat}\ \textit{Name}^* \\
| \ & \texttt{Recursor}\ \textit{Name Expr}\ \texttt{Nat}\ \textit{Name}^*\ \texttt{Nat Nat Nat Nat} \\
& \quad \texttt{Nat RecRule}^*\ \texttt{Bool}\ \textit{Name}^*
\end{aligned}$$

$$\texttt{Hint} ::= \texttt{O} \mid \texttt{A} \mid \texttt{R Nat}$$

$$\texttt{RecRule} ::= \textit{Name}\ \texttt{Nat}\ \textit{Expr}$$

Ogni dichiarazione è definita da un tipo, un *Name* identificativo e un elenco di parametri universali, elencati non come universi ma come *Name*.

Le dichiarazioni `Opaque`, `Theorem` e `Definition` contengono anche un valore: per verificarne la correttezza, si calcola il tipo del valore e si confronta con il tipo dichiarato, controllando che coincidano per uguaglianza definitoria.

Nel caso delle `Definition`, è presente anche il campo `Hint`, che può assumere i valori:

- `A`: abbreviazione;
- `O`: definizione opaca;
- `R Nat`: definizione riducibile `Nat` volte.

I casi particolari sono le dichiarazioni `Inductive`, `Constructor` e `Recursor`, che presentano numerosi campi, anche se non tutti vengono utilizzati nel programma.

La `Inductive` è definita da due valori booleani (per indicare se è riflessiva e ricorsiva), dal numero di annidamenti, di parametri e di indici, dal numero di `Inductive` e `Constructor` e dai relativi elenchi di nomi. Di questi, vengono salvati il numero di indici e l'elenco dei `Constructor`. Un caso particolare è rappresentato dalle strutture, cioè quelle `Inductive` che hanno un solo `Constructor` e nessun indice.

La `Constructor` è definita dal nome della `Inductive` di riferimento, dall'indice del costruttore, dal numero di parametri e di campi. In questo caso vengono salvati tutti i valori tranne l'indice.

La `Recursor` è definita dal numero di `Inductive` e dai loro nomi, dal numero di parametri, indici, motivi minori, regole e dal booleano associato. I valori salvati comprendono il numero di parametri, indici, motivi minori, i nomi delle `Inductive` e l'elenco delle regole di riduzione.

Le regole di riduzione (`RecRule`) sono definite da un *Name* (riferito a una `Constructor`), dal numero di parametri e dal loro valore.

Poiché le dichiarazioni sono identificate tramite *Name* e non tramite indice, vengono salvate in una `IndexMap` definita da una coppia *Name*-dichiarazione. Le `RecRule`, invece, essendo identificate tramite indice, vengono gestite tramite una `Ptr` e un `IndexSet` associato.

Capitolo 6

Inferenza di tipo

L'inferenza è la procedura che determina il tipo di un'espressione, rappresentando una delle funzioni fondamentali del kernel di Lean. La funzione che implementa questo comportamento è chiamata `infer` e prende in input una *Expr*. Essa lavora principalmente applicando le regole d'inferenza per ciascuna forma delle espressioni.

Le regole che ho seguito non sono syntax directed ma all'interno del programma le ho ridefinite come tali. Per poterlo fare sono andato a combinare le regole originali con questa regola:

$$\frac{\Gamma \vdash M : T_1 \quad T_1 \equiv T_2}{\Gamma \vdash M : T_2}$$

dove per controllare che $T_1 \equiv T_2$ in questo vado a riduzione di T_1 e verifico che sia uguale a T_2, dove in questo caso per uguale si intende secondo l'uguaglianza definitoria, elemento che verrà definito nel prossimo capitolo.
Questa cosa viene svola, anche se solo per i casi in cui T_2 sia una `Sort`, dalla funzione di supporto `infer_sort_of`. La funzione infatti calcola il tipo dell'espressione in input, andando a trovare ciò che sarebbe T_1, lo riduce e verifica se il risultato sia un `Sort`: in caso positivo restituisce l'universo, altrimenti genera un errore. Regola di inferenza di tipo per `Sort`:

$$\frac{}{\Gamma \vdash \texttt{Sort}_n : \texttt{Sort}_{n+1}}$$

per cui bisogna restituire un `Sort` con l'universo successivo rispetto a quello dell'espressione in analisi.

Regola di inferenza di tipo per `Const`:

$$\frac{(c : T) \in \Gamma}{\Gamma \vdash c : T}$$

per cui bisogna controllare l'esistenza della dichiarazione, verifica già effettuata in fase di salvataggio; pertanto è sufficiente recuperare il tipo associato al *Name* della dichiarazione, sostituendo i parametri universali della dichiarazione con quelli della `Const` mediante `subst_expr_universes`.

Regola di inferenza di tipo per `Pi`:

$$\frac{\Gamma \vdash S : \texttt{Sort} \quad \Gamma, (x : S) \vdash T : \texttt{Sort}' \quad \texttt{Sort}'' = \texttt{IMax}(\texttt{Sort}, \texttt{Sort}')}{\Gamma \vdash \Pi(x : S) \to T : \texttt{Sort}''}$$

Nel codice questa regola si traduce in una sequenza di operazioni che riflettono le varie parti della derivazione formale: si ottiene l'universo del tipo, per poterlo fare si deve sfruttare la funzione `infer_sort_of` per verificare che il tipo sia riconducibile ad una `Sort` da cui si ottiene l'universo; si istanzia una `FreeVar` nel corpo con il tipo del `Pi`, si calcola l'universo del corpo istanziato e infine si restituisce `IMax` dei due universi come nuovo `Sort`.

```
Pi {ty, body, ..} => {
    let l = self.infer_sort_of(ty);
    let free = self.free_var(ty);
    let b = self.inst(body, free, 0);
    let r = self.infer_sort_of(b);
    let imax = self.imax(l, r);
    self.sort(imax)
}
```

Regola di inferenza di tipo per `Lambda`:

$$\frac{\Gamma \vdash S : \texttt{Sort} \quad \Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x : S, M : \Pi(x : S) \to T}$$

per cui si controlla che il tipo della `Lambda` sia un `Sort`, si istanzia una `FreeVar` nel corpo, si calcola il tipo del corpo istanziato e lo si astrae nuovamente. Il risultato è un `Pi` con lo stesso tipo della `Lambda` e corpo dato dall'espressione astratta.

Regola di inferenza di tipo per `App`:

$$\frac{\Gamma \vdash M : \Pi(x : S) : T \quad \Gamma \vdash N : S}{\Gamma \vdash M\ N : T[N/x]}$$

per cui si calcola il tipo della funzione e lo si riduce. Se il risultato è un `Pi`, si controlla che il tipo del suo parametro e il tipo dell'argomento siano uguali per uguaglianza definitoria, quindi si istanzia il corpo con l'argomento e si restituisce il risultato. Se non si trova un `Pi`, si genera un errore.

Regola di inferenza di tipo per `FreeVar`:

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

anche in questo caso non è necessario verificarne l'esistenza, poiché le `FreeVar` sono dichiarate direttamente nel programma; si può quindi restituire il loro tipo.

Regola di inferenza di tipo per `Var`: le `Var` devono far restituire al sistema un errore, poiché tali variabili non sono ancora istanziate e quindi non hanno un tipo determinabile.

Regola di inferenza di tipo per `Let`:

$$\frac{\Gamma \vdash S : \texttt{Sort} \quad \Gamma \vdash N : S \quad \Gamma, x : S \vdash M : T}{\Gamma \vdash \texttt{Let}\ \ S\ N\ M : T[N/x]}$$

per cui si verifica che il tipo sia un `Sort`, si controlla che il tipo di `Let` e quello del valore coincidano per uguaglianza definitoria, quindi si istanzia il valore nel corpo e si restituisce il tipo risultante.

Regola di inferenza di tipo per `Proj`:

$$\frac{\Gamma \vdash e : (f_i : T_i \ldots f_n : T_n)}{\Gamma \vdash (\#i\ e) : T_i}$$

per cui si verifica che il valore sia una struttura, ottenendo e riducendo il suo tipo. Se è un'`App`, si utilizza `unfold_app`. Si controlla che l'espressione corrisponda a una struttura, quindi si ottiene il `Constructor` e si sostituiscono i parametri universali con gli universi della `Const`. Si riduce il tipo del costruttore, istanziando progressivamente gli argomenti e le `Proj`; se a un certo punto non si trova un `Pi`, si segnala un errore. Infine, si riduce nuovamente e, se il risultato è un `Pi`, se ne restituisce il tipo; altrimenti, si genera un errore.

Regole di inferenza di tipo per `NatLit` e `StrLit`:

$$\frac{}{\Gamma \vdash \texttt{NatLit} : \texttt{Nat}} \qquad \frac{}{\Gamma \vdash \texttt{StrLit} : \texttt{String}}$$

queste regole rappresentano l'uso diretto dei tipi `Nat` e `String`. Pertanto, il loro tipo è una `Const` corrispondente al tipo di base, senza parametri.

Poiché la funzione `infer` può essere richiamata più volte sullo stesso valore, è stata introdotta una `IndexMap` da utilizzare come cache. Essa consente di memorizzare un'espressione e il suo tipo, riducendo così il numero di iterazioni necessarie.

Capitolo 7

Uguaglianza definitoria

Con uguaglianza definitoria si intende la verifica che due espressioni rappresentino lo stesso concetto, pur essendo scritte in maniera differente.
Questo controllo viene effettuato tramite la funzione `def_eq`, che inizialmente verifica se le due definizioni siano esattamente uguali, ovvero se coincidano nella loro forma testuale. In caso contrario, le due espressioni vengono ridotte tramite la `whnf`, riduzione che viene spiegata nel prossimo capitolo; se una delle due risulta essere una `Var`, viene sollevato un errore.

Successivamente, il controllo procede confrontando le due espressioni a coppie. Nei vari casi si esegue quanto segue:

- per due `FreeVar`, si verifica che i rispettivi indici coincidano;
- per due `Sort`, si controlla che i due universi siano equivalenti, verificando sia $x \leq y$ che $y \leq x$;
- per due `Const`, si verifica che i nomi siano uguali e che gli universi corrispondano a coppie, tramite la funzione `eq_many`;
- per due `App`, si controlla che le due funzioni e i rispettivi argomenti siano uguali per uguaglianza definitoria;
- per due `Lambda` o due `Pi`, si verifica che i tipi siano uguali e, se lo sono, si crea una `FreeVar` con il tipo di una delle espressioni; la variabile viene poi istanziata nei corpi di entrambe, che vengono infine confrontati per verificare l'uguaglianza;

- nel caso in cui solo una delle due espressioni sia una `Lambda` bisogna andare ad implementare al volo una eta-espansione, ciò è possibile solo perché avere da un lato una lambda-astrazione e nell'altro no impedisce la divergenza. Per farlo si infera il tipo dell'altra espressione, se questo tipo è un `Pi`, si verifica che il tipo della `Lambda` e quello del `Pi` coincidano. In tal caso si crea una `FreeVar` con uno dei due tipi, la si istanzia nella `Lambda` e si costruisce una `App` la cui funzione è l'espressione non `Lambda` e il cui argomento è la `FreeVar`. Infine, si confrontano i due risultati;

- in tutti gli altri casi la funzione restituisce `false`.

Se il controllo precedente non restituisce un risultato positivo, vengono eseguite alcune verifiche aggiuntive.

Nel primo controllo si deve implementare la eta per le strutture; per farlo una delle due espressioni viene aperta come una serie di `App`, per verificare se il valore di base sia un `Constructor` di una struttura. Se lo è, e se il numero di argomenti della serie corrisponde alla somma dei parametri e dei campi del `Constructor`, si inferano i tipi delle due espressioni e si verifica che siano uguali. In tal caso si costruisce una serie di `Proj` pari al numero di campi, dove l'indice è sommato al numero di parametri e la struttura è l'espressione originaria. Infine, si controlla che ogni funzione corrisponda all'argomento che si trova nella posizione indicata dall'indice della `Proj`.

Se anche questo controllo restituisce esito negativo, si verifica se una delle due espressioni sia una `Const` riferita a una struttura; in tal caso si inferano i tipi di entrambe le espressioni e, se risultano uguali, si controlla che il numero di universi coincida con quello dei parametri del `Constructor` della struttura.

Qualora neppure questo controllo produca esito positivo, si prendono i tipi delle due espressioni, si inferano i tipi di tali tipi e si verifica se siano riconducibili a un `Sort` a `Zero`. Se lo sono, si controlla che i due tipi delle espressioni siano uguali.

Se anche quest'ultimo controllo fallisce, significa che non esistono altri modi per dimostrare l'uguaglianza tra le due espressioni, e viene quindi restituito un risultato negativo.

Capitolo 8

Riduzione di un'espressione

La riduzione implementata nel programma è la `weak head normal form`. Questa forma di riduzione è innanzitutto debole, ovvero riduce soltanto il livello più esterno delle espressioni, senza entrare all'interno dei binder (lambda-astrazioni e pi-astrazioni). Inoltre, è una riduzione estremamente lazy: a ogni passo viene ridotta esclusivamente la testa dell'espressione, ossia il redex in posizione leftmost-outermost del livello più esterno. Pertanto, gli argomenti di un'applicazione non vengono mai ridotti.

La funzione che realizza questa riduzione è denominata `whnf`. Essa costruisce un array in cui vengono salvati gli argomenti provenienti dalle `App` e imposta un ciclo di riduzione, che prosegue finché l'espressione non raggiunge la forma normale. Al termine, se sono ancora presenti argomenti nell'array, viene costruita una concatenazione di `App` per esaurirli. All'interno del ciclo vengono effettuati dei controlli basandosi sulle regole di riduzione.

Regola di riduzione di `Let`:

$$\frac{}{\Gamma \vdash \texttt{Let}\ \ S\,N\,M : T[N/x]}$$

per cui si istanzia il valore nel corpo e si continua la riduzione su questo nuovo valore.

Regola di riduzione di `App`:

$$\frac{\Gamma \vdash M \to M' \quad M \in \Gamma}{\Gamma \vdash M\ N \to M'}$$

per cui si aggiunge l'argomento all'array e si prosegue la riduzione sulla funzione.

Regola di riduzione di `Lambda`:

$$\frac{}{\Gamma \vdash \lambda x : S, M \to M[\alpha/x]}$$

per cui si verifica se l'array degli argomenti è vuoto: in tal caso si interrompe il ciclo; altrimenti si istanzia il corpo con l'ultimo argomento inserito e si continua la riduzione su di esso.

Regola di riduzione di `Proj`:

$$\frac{\Gamma \vdash e : (f_i : T_i \dots f_n : T_n)}{\Gamma \vdash (\#i\ e) : T_i}$$

per cui si riduce la struttura, si apre la catena di `App` e, sull'ultima funzione, si verifica se essa corrisponde a un `Constructor`. Se sì, si preleva l'argomento (dalla serie di `App`) alla posizione data dall'indice del `Proj` sommato al numero di parametri del `Constructor`, e si continua la riduzione su tale valore. Se invece non viene trovato un `Constructor`, si interrompe il ciclo.

Regola di riduzione di `Const`

$$\frac{c.val \to v}{\Gamma \vdash c \to v}$$

per cui si controlla se la dichiarazione ha un valore, si sostituiscono i parametri universali del valore con gli universi della `Const` tramite la funzione `subst_expr_universes`, e si prosegue la riduzione sul valore ottenuto.

Se si trova una `Const` di una `Recursor` si recupera l'argomento dall'array posizionato dopo un numero di argomenti pari alla somma dei parametri, dei motivi, dei minori e degli indici della `Recursor`. Si apre quindi la serie di `App`, da cui si distinguono tre casi:

- se viene trovato un `Constructor`, si applica questa regola:

  $$\frac{\Gamma \vdash \texttt{Recursor}(\texttt{Constructor}(a_1 ... a_n))}{\Gamma \vdash \texttt{Recursor}(\texttt{Constructor}(a_1 ... a_n)) \to rule[a_1 ... a_n]}$$

  per cui si recupera la `RecRule` a esso associata, cercandola nell'elenco delle regole della `Recursor`. Su tale valore si costruisce una catena di `App` utilizzando gli argomenti saltati dell'array (escludendo gli indici) e quelli trovati nell'apertura della `App`. Infine, si riduce questa `App` e si continua il ciclo sulla riduzione ottenuta;

- se non si trova un `Constructor`, ma la `Recursor` ha una sola `Inductive` che è una struttura, allora si applica questa regola:

  $$\frac{\Gamma \vdash S \to r}{\Gamma \vdash S \to (...((r\ \texttt{Proj}_1(S))\texttt{Proj}_2(S))...)\texttt{Proj}_n(S)}$$

  per cui si costruisce una catena di `App` come nel caso precedente, aggiungendo eventualmente delle `Proj` nel caso in cui vi siano parametri. Il numero di `Proj` corrisponde al numero di campi della struttura. Si riduce poi la `App` risultante e si prosegue il ciclo;

- in tutti gli altri casi, si termina il ciclo di riduzione.

In ogni altro caso, si esce dal ciclo.

Anche per la funzione `whnf` è stata introdotta una cache (una `IndexMap`) per memorizzare le associazioni tra espressioni e la sua riduzione, riducendo così il numero di elaborazioni ripetute.

Una funzione di supporto utile, impiegata anche nel capitolo successivo, è `unfold_app`. Questa funzione consente di scomporre una "catena" di applicazioni `App` nella sua testa ereditaria di applicazioni imbricate a sinistra e nel vettore dei relativi argomenti. In altre parole, un'espressione della forma $(\ldots((f\ E_1)\ E_2)\ldots)E_n$ viene scomposta nella coppia $(f, [E_1, E_2, \ldots, E_n])$, dove f non è a sua volta un'applicazione.

Capitolo 9

Lavori futuri e conclusioni

Questo progetto non è riuscito a raggiungere l'obiettivo di realizzare un type checker completamente funzionante, a causa di alcune problematiche riscontrate nella fase di riduzione, in particolare nei casi che coinvolgono una `Const` associata a una `Recursor`. Tali difficoltà derivano dal fatto che la documentazione utilizzata per definire la costruzione di un type checker per il linguaggio Lean non risulta chiara in merito alla gestione di questo tipo di riduzione e, più precisamente, alla modalità con cui devono essere trattati gli argomenti coinvolti. Un possibile approccio per risolvere questi problemi consisterebbe nell'approfondire il modo in cui Lean gestisce internamente la riduzione di una `Recursor` e, soprattutto, nel comprendere con maggiore precisione come il file di export rappresenti la definizione e gli argomenti di tale costrutto, dal momento che la documentazione attuale è priva di una descrizione adeguata.

Nel caso in cui questa problematica venisse risolta, vi sarebbero ulteriori ambiti in cui intervenire per migliorare la sicurezza, la coerenza logica e l'efficienza complessiva del type checker. Tali miglioramenti, non implementati in questa versione a causa della limitata disponibilità di tempo e di informazioni, potrebbero comprendere:

- l'introduzione di un controllo per le dichiarazioni `Inductive`, `Constructor` e `Recursor`, poiché esse possono andare a rappresentare dei tipi complessi definiti nel file originale; questi controlli svolgerebbero una verifica sulle dichiarazioni per vedere se siano coerenti fra di loro e se anche i dati delle dichiarazioni lo siano;
- l'applicazione di strutture a grafo per la rappresentazione delle dichiarazioni — ossia l'idea originaria della tesi — al fine di migliorare l'organizzazione e le prestazioni del

programma;

- l'estensione del sistema di riduzione e di uguaglianza, includendo regole non specificate nella documentazione ufficiale ma utilizzate nell'ambiente di riduzione a forma normale.

Anche se non sono riuscito a realizzare un type checker completamente funzionante, questo progetto mi ha permesso di approfondire in modo concreto il funzionamento del kernel di Lean e le logiche che ne regolano la verifica dei tipi. Lo sviluppo ha richiesto circa tre mesi di lavoro continuativo, che hanno portato alla scrittura di 1630 righe di codice suddivise in 8 file, durante i quali ho potuto mettere in pratica le competenze acquisite durante il percorso triennale e consolidarne di nuove, in particolare nella programmazione in Rust e nella gestione di strutture dati complesse.
In generale, questo progetto mi ha permesso di comprendere meglio le difficoltà legate alla realizzazione di strumenti di analisi formale e di acquisire un approccio più critico e strutturato alla progettazione di software. Guardando al percorso svolto, considero questa esperienza non solo un importante traguardo personale, ma anche un punto di partenza per continuare ad approfondire il rapporto tra logica, linguaggi formali e sviluppo del software.

Bibliografia

[1] Type Checking in Lean 4
(ammkrn.github.io/type_checking_in_lean4/)

[2] documentazione per lean4export
(github.com/leanprover/lean4export)

[3] documentazione export per lean3
(github.com/leanprover/lean3/blob/master/doc/export_format.md)

[4] Documentazioni sul sito di Lean
(lean-lang.org/learn/)

[5] documentazione di Rust
(https://doc.rust-lang.org/core/index.html)

[6] nanoda_lib
(github.com/ammkrn/nanoda_lib)

[7] codice del programma
(github.com/mattiaAlbertini03/TypeCheck-Rust)

Ringraziamenti

Desidero ringraziare la mia famiglia per il sostegno, la pazienza e l'incoraggiamento che mi hanno sempre dato in questo percorso.
Un grazie anche all'università e ai docenti per le opportunità di crescita e di apprendimento che mi hanno offerto.