Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Estrazione di codice da termini della Minimalist Type Theory

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Riccardo Caprari

Sessione II
Anno Accademico 2017/2018

Abstract

L'obiettivo di questa tesi è stato l'implementazione di un estrattore di codice per un dimostratore interattivo basato sulla *minimalist two-level foundation* (mTT), proposta da Maietti [Mai08].

Per raggiungere lo scopo, si è preso a modello l'estrattore di codice del dimostratore Coq ed è stato necessario acquisire conoscenze nell'ambito dell'eliminazione di codice inutile, la quale si occupa di rilevare tramite analisi statica quelle parti di codice che, se eliminate, lasciano invariato l'output riducendo l'uso di RAM e i passi di computazione del programma.

L'estrattore è stato implementato utilizzando il linguaggio λProlog, nel quale erano già stati prodotti precedentemente un *type checker* e una versione provvisoria del compilatore per mTT da Alberto Fiori [FS18] e Giacomo Molinari [Mol18] nelle loro rispettive tesi. L'interprete di λProlog che ho utilizzato durante il mio lavoro è ELPI [Dun+15], realizzato presso l'Università di Bologna.

Pur avendo avuto a disposizione risorse estensive su cui basare l'implementazione, è stato comunque necessario produrre un numero consistente di soluzioni originali: infatti, l'operazione di estrazione richiede spesso un adattamento o, in alcuni casi, una riscrittura dei procedimenti di manipolazione standard sulla base delle differenti idiosincrasie della fondazione logica di partenza.

L'estrattore prodotto come risultato finale del lavoro di tesi è completamente funzionante e in grado di estrarre ogni costrutto del livello intensionale della logica mTT, traducendolo in codice OCaml e Haskell.

Indice

Capitolo 1

Introduzione

L'estrazione di codice funzionale a partire da una dimostrazione formalizzata nella logica matematica può essere vista come una diretta conseguenza dell'isomorfismo di Curry-Howard, il quale mette in relazione di equivalenza le prove con i programmi e gli enunciati con i tipi di questi. In una fondazione costruttiva della matematica, che cioè rifiuti l'assioma del terzo escluso ($A \vee \neg A$), è dunque sempre possibile trasformare una dimostrazione in un programma e viceversa.

Esistono varie motivazioni dell'utilità di un estrattore di codice:

- il codice funzionale ottenuto può essere integrato in un programma esistente o essere utilizzato per creare una libreria, cosa che impatta in maniera estremamente positiva sull'usabilità della dimostrazione effettuata da parte della comunità di sviluppatori di software;

- la versione estratta viene ripulita da tutti i suoi elementi non informativi lasciando effettivamente un blocco di codice più leggibile e la cui esecuzione può avvenire in maniera molto più rapida, sfruttando la capacità di ottimizzazione del compilatore del linguaggio target.

In questo capitolo, tratterò degli argomenti teorici e degli studi esistenti sul problema dell'estrazione di codice che ho esaminato e utilizzato come

base per lo svolgimento del lavoro di tesi. Di seguito, una breve panoramica di questi argomenti:

- l'eliminazione di codice inutile, un insieme di tecniche che consentano di eliminare parti di codice non informative mantenendo intatta la semantica denotazionale del programma considerato;
- l'estrazione di codice nel dimostratore interattivo Coq, al fine di avere un termine di paragone e un modello sulla base del quale impostare il lavoro;
- la minimalist two-level foundation di Maietti [Mai08], la formulazione dei fondamenti della matematica dalla quale si vuole estrarre codice;
- λProlog, il linguaggio utilizzato per scrivere l'implementazione dell'estrattore.

1.1 Eliminazione di codice inutile

Durante la fase di estrazione si rende necessario eliminare tutte quelle parti delle prove che sono prive di contenuto informativo, le quali cioè non influenzano l'output prodotto: per esempio, le proposizioni hanno come unico scopo la descrizione di alcune proprietà vere e non calcolano effettivamente nulla. Per fare ciò è necessario eseguire un'analisi statica della dimostrazione che permetta di riconoscere e marcare le componenti non informative, per poterle successivamente eliminare. Questo compito in letteratura assume il nome di eliminazione di codice inutile.

Nella trattazione che segue faremo riferimento principalmente a quanto descritto in un articolo di Berardi [Ber+00] e parleremo esclusivamente di approcci basati sul tipo *(type-based code elimination)*, i quali per effettuare la marcatura delle componenti non rilevanti operano un'annotazione di tipo.

Un'alternativa all'eliminazione basata su tipi è l'eliminazione di variabili, nella quale la marcatura viene effettuata sulle singole variabili istanziate.

Questa tecnica risulta però meno potente di quella basata su tipi poiché, dato un termine composto da molti campi, solamente alcuni di essi potrebbero essere utili. L'eliminazione parziale è gestita correttamente nella versione basata sui tipi, mentre una variabile può essere solamente presa o scartata integralmente nell'altro metodo.

Per prima cosa, approcciandosi all'argomento dell'eliminazione di codice è necessario distinguere i due concetti di codice inutile *(useless code)* e codice morto *(dead code)*. Si dice inutile tutto quel codice che, se eliminato, lascia invariato il comportamento del programma, cioè agli stessi input vengono fatti corrispondere i medesimi output. Si definisce morto tutto quel codice che non viene mai eseguito.

Nell'esempio seguente, assumendo che sia implementato in un linguaggio di programmazione con valutazione call-by-name, l'introduzione e l'eliminazione della coppia sono inutili, mentre il secondo elemento della coppia è morto. Possiamo quindi operare la semplificazione S e ottenere un codice equivalente di partenza, ma privo di elementi inutili.

$$\Pi_1 < M, N > \quad \longrightarrow_S \quad M$$

Si noti che in un linguaggio di programmazione con valutazione call-by-value anche il secondo elemento della coppia, se garantito essere normalizzante, è codice inutile e non morto, poiché necessita di essere valutato prima di essere passato al costruttore della coppia. La distinzione tra i due tipi di codice da eliminare risulta rilevante poiché in fase di semplificazione essi assumono diverso comportamento.

Mentre l'eliminazione del codice morto porta solamente ad una semplificazione dei sorgenti, l'eliminazione di codice inutile può portare ad una variazione del tempo di esecuzione: in un linguaggio call-by-name questa variazione è sempre una diminuzione dei tempi di calcolo, mentre in un linguaggio call-by-value si potrebbero avere anche aumenti del tempo di valutazione. Si prenda l'esempio seguente:

$$(\lambda x.\ 3)(\lambda y : \mathbb{1}.\ loop)$$

dove *loop* è una funzione non terminante. Astrarre sull'insieme singoletto è un'operazione non informativa: in un linguaggio call-by-value, però, se l'astrazione sul singoletto venisse eliminata, *loop* sarebbe valutato prima di essere passato alla funzione costante facendo divergere il programma.

La prima versione dell'estrazione di codice in Coq, implementata da Mohring [Pau89], era incentrata intorno all'idea di assegnare a ogni tipo un kind, cioé un etichetta, tra Spec (utile) e Prop (inutile). Potendo assegnare solamente un kind ad ogni tipo, si rendeva necessario estrarre tutte le occorrenze di un certo tipo anche se solamente alcune erano utili. Per oltrepassare questo ostacolo si introdusse una relazione di sottotipaggio tra Spec e Prop: tutte le occorrenze marcate con Spec erano così utili ma potevano essere convertite a inutili in caso di necessità.

Inizialmente questo meccanismo di marcatura andava effettuato manualmente. Per quanto riguarda la determinazione di un algoritmo per la marcatura automatica i principali contributi sono di Berardi [Ber+00]. La tecnica consiste nel marcare i termini con due tipi δ e ω, di cui il primo denota termini che potrebbero essere utili mentre il secondo denota termini certamente inutili. L'algoritmo descritto è generale e può essere applicato sia a linguaggi tipati che a linguaggi non tipati ma, nel caso di linguaggi tipati, δ e ω diventano decorazioni incorporate nei tipi esistenti.

Le regole di inferenza per le annotazioni di tipo sono le seguenti:

$$\textbf{(Assioma per variabili)} \quad \Sigma, x : \alpha \vdash x^{\beta}(\alpha \sqsubseteq \beta)$$

$$\textbf{(Assioma per costanti)} \quad \Sigma \vdash k^{\beta}(\tau(k) \sqsubseteq \beta)$$

$$\textbf{(Introduzione } \rightarrow\textbf{)} \quad \frac{\Sigma, x : \alpha \vdash M^{\beta}}{\Sigma \vdash (\lambda x^{\alpha}.M^{\beta})^{\alpha \rightarrow \beta}}$$

$$\textbf{(Eliminazione } \rightarrow\textbf{)} \quad \frac{\Sigma \vdash M^{\alpha \rightarrow \beta} \quad \Sigma \vdash N^{\alpha}}{\Sigma \vdash (M^{\alpha \rightarrow \beta} N^{\alpha})^{\beta}}$$

$$\textbf{(Punto Fisso)} \quad \frac{\Sigma, x : \alpha \vdash M^{\alpha}}{\Sigma \vdash (\text{fix } x.M^{\alpha})^{\alpha}}$$

$$\textbf{(Condizionale)} \quad \frac{\Sigma \vdash N^{\delta} \quad \Sigma \vdash M_1^{\alpha} \quad \Sigma \vdash M_2^{\alpha}}{\Sigma \vdash (\text{if } N^{\delta} \text{ then } M_1^{\alpha} \text{ else } M_2^{\alpha})^{\alpha}}$$

dove

- α e β sono variabili di annotazione;
- $\sqsubseteq$ è una relazione d'ordine parziale tra tipi definita come

$$\textbf{(Riflessività)} \quad \alpha \sqsubseteq \alpha$$

$$\textbf{(Sottotipaggio rispetto a } \omega\textbf{)} \quad \alpha \sqsubseteq \rho(\omega)$$

con $\rho(\omega)$ tipo annotato con ω;

- $\tau(k)$ è l'annotazione di default del termine k, tipicamente assegna il decoratore utile δ a tutte le parti del termine.

Inoltre, dal sistema precedente è possibile derivare la seguente regola aggiuntiva per l'introduzione dell'applicazione:

$$\textbf{(Introduzione alternativa } \rightarrow\textbf{)} \quad \frac{\Sigma \vdash M^{\alpha} \quad M[\backslash x]}{\Sigma \vdash (\lambda x^{\rho(\omega)}.M^{\alpha})^{\rho(\omega)\rightarrow\beta}}$$

dove $M[\backslash x]$ denota che in M x non compare.

A partire da queste regole, si può definire una relazione di equivalenza tra termini basata sulla decorazione inutile. Due termini sono equivalenti se differiscono solo in quei sottotermini marcati come inutili. Inoltre, per ogni termine esiste una decorazione ottima tale che ogni sottotermine utile in essa è marcato come utile anche in tutte le altre decorazioni possibili.

La decorazione ottima è calcolabile con il seguente algoritmo:

1. si costruisce una prima decorazione completamente parametrica nella quale ad ogni variabile e costante viene associata una decorazione fresca;

$$((\lambda x^{a\rightarrow b}.\ (x^{c\rightarrow d}(s^{e\rightarrow f}1^{g})^{h})^{d})^{(a\rightarrow b)\rightarrow d}\ (\lambda y^{i}.\ 3^{l})^{i\rightarrow l})^{d}$$

2. si costruisce l'insieme dei vincoli di tipo imposti dalle applicazioni, se si assume che il programma sia ben tipato in un qualche sistema di tipi la corrispondenza è univoca;

$$\{e = g, c = h, a = i, b = l\}$$

3. si costruiscono insiemi di equazioni con guardia che vengono considerate valide solamente se la variabile messa in guardia viene tipata come codice utile, esse definiscono la propagazione del codice utile;

$$\{\{d\} \implies \{a = c, b = d\}, \{f\} \implies \{e = \delta, f = \delta\},$$
$$\{e\} \implies \{e = \delta\}, \{l\} \implies \{l = \delta\}, \}$$

4. si costruisce il vincolo che l'output finale del termine sia utile;

$$\{d = \delta\}$$

5. partendo dall'ultimo vincolo si cercano tutte le decorazioni forzate ad essere utili e le si assegna δ, mentre a tutte le decorazioni che rimangono libere si assegna ω.

$$((\lambda x^{\omega\to\delta}.\ (x^{\omega\to\delta}(s^{\omega\to\omega}1^{\omega})^{\omega})^{\delta})^{(\omega\to\delta)\to\delta}\ (\lambda y^{\omega}.\ 3^{\delta})^{\omega\to\delta})^{\delta}$$

Tutte le fasi dell'algoritmo operano in tempo lineare rispetto alla dimensione del termine valutato. Data una decorazione ottima, una funzione di semplificazione può essere definita come quell'operatore che cancella tutti i termini marcati come inutili e cancella l'astrazione delle applicazioni se il loro argomento è inutile.

$$Clean(((\lambda x^{\omega\to\delta}.\ (x^{\omega\to\delta}(s^{\omega\to\omega}1^{\omega})^{\omega})^{\delta})^{(\omega\to\delta)\to\delta}\ (\lambda y^{\omega}.\ 3^{\delta})^{\omega\to\delta})^{\delta}) = (\lambda x.\ x)\ 3$$

È possibile estendere questo sistema con la quantificazione universale [Boe94], marcando opportunamente le occorrenze inutili della variabile quantificata universalmente e sostituendole in fase di semplificazione con un opportuno valore dummy.

1.1.1 Slicing

Un concetto affine all'eliminazione di codice inutile è il *program slicing* [Tip94], che consiste nel selezionare un'istruzione all'interno di un programma e nel cancellare tutti gli statement che non influenzano l'esecuzione dell'istruzione selezionata. La principale differenza tra eliminazione e *slicing* è

che, mentre la prima tecnica produce un programma equivalente a quello di partenza e ha come fine l'ottimizzazione, la seconda ritorna un programma parziale che esegue solamente una parte della computazione originale e ha come principale applicazione il debugging.

Slicing statico

L'idea di *slicing* è stata formulata per la prima volta da Weiser, sulla base dell'assunzione che il sorgente, privato del codice che non contribuisce all'errore che si sta cercando di risolvere, abbia una maggiore corrispondenza con la rappresentazione mentale del programmatore e costituisca una risorsa utile per una più facile rilevazione dei *bug*. L'approccio di Weiser è statico: basandosi sulle dipendenze dei dati e delle istruzioni nei sorgenti del programma si costruisce a ritroso il grafo delle operazioni rilevanti, partendo dal punto di interesse.

Ai fini di questa tesi, non è eccessivamente interessante approfondire i dettagli dell'algoritmo di *slicing*, anche considerando che esso è stato pensato per operare su linguaggi imperativi che divergono dai nostri scopi, ma ne darò comunque una breve descrizione per confrontarlo con l'eliminazione di codice di Berardi. Per prima cosa, è necessario costruire un grafo di dipendenze tra le istruzioni, inserendo archi direzionati con queste regole:

- a partire da una struttura di controllo (`if-then-else` o `while`) verso ogni istruzione contenuta al suo interno;
- tra due istruzioni aventi almeno una variabile in comune, la direzione dell'arco è in ordine di esecuzione inverso.

Una volta costruito il grafo, il calcolo del codice da preservare è strutturato come un'iterazione di punto fisso costituita dalle seguenti fasi:

1. si raccoglie una lista di tutte le variabili utilizzate all'interno delle istruzioni già rilevate come utili;

2. per ognuna delle variabili collezionate al passo precedente, si marcano come utili le istruzioni che le producono in output e le istruzioni che dipendono da esse;

La marcatura ha termine se, al completamento delle due fasi, non sono state aggiunte nuove istruzioni rispetto all'iterazione precedente.

In conclusione, l'algoritmo di *slicing* e quello di eliminazione risultano essere molto simili, nonostante il primo abbia bisogno di una struttura di vincoli più sofisticata per tenere conto dei *side effects* imperativi.

Slicing dinamico

Una variante della tecnica precedentemente descritta è lo *slicing* dinamico, nel quale oltre ai sorgenti del programma vengono presi in considerazione anche gli input per una particolare esecuzione. In questo modo, vengono collezionate solamente quelle istruzioni che influenzano effettivamente la computazione di interesse nell'esecuzione corrente, eliminando alcuni rami morti che nell'approccio statico erano stati conservati.

In pratica, mentre nello *slicing* statico non si fanno assunzioni riguardo all'input, nello *slicing* dinamico si effettua la rimozione di codice sulla base di un input prefissato. Ciò consente di trattare in maniera diversa istanze della medesima istruzione dotate però di diversi parametri, le quali anche nell'eliminazione di codice avevano bisogno di un trattamento ad-hoc per massimizzare la rimozione di codice inutile. Ovviamente, lo slicing dinamico non si applica al nostro caso perché il codice estratto deve poter gestire molti input diversi.

1.2 Estrazione di codice in Coq

Coq è un dimostratore interattivo di teoremi basato sul calcolo delle costruzioni induttive *(calculus of inductive constructions)*, un'estensione del calcolo delle costruzioni elaborato da Thierry Coquand con forti influenze

dal calcolo di Martin-Löf. Coq supporta costrutti quali i prodotti, le astrazioni, i let in, l'eliminazione per casi e il punto fisso; il suo estrattore ha come linguaggi target Haskell, OCaml e Scheme.

L'estrattore di codice di Coq è stato preso come modello per la realizzazione di quello sviluppato in questa tesi poiché è un software particolarmente maturo per il quale sono stati effettuati sforzi estensivi in ambito di ricerca. In particolare, un articolo di Letouzey [Let03] e la sua tesi di dottorato [Let04] ne descrivono dettagliatamente il funzionamento e i problemi che si sono dovuti affrontare.

Nel prossimo capitolo tratterò come sono stati gestiti o se erano presenti questi medesimi problemi nell'estrazione della *minimalist foundation*: di seguito, sono elencate alcune criticità del codice Coq in fase di estrazione. Come piccola nota, nella trattazione seguente per evitare di appesantire il testo con molte notazioni diverse, userò la sintassi standard della logica e non quella specifica di Coq.

1.2.1 Problemi di eliminazione

Applicazione parziale

Un primo problema è l'applicazione parziale di funzioni dotate di argomenti non informativi. Per esempio, $f : (x : A) \to Px \to B$ che prende un argomento informativo x e una prova che esso soddisfi la proprietà Px può essere parzialmente applicato fornendo solamente x e in questo caso la computazione si blocca. Dopo l'estrazione però il codice non informativo è stato rimosso quindi la nuova f ha signature $f : (x : A) \to B$ rendendo un'applicazione parziale identica ad una completa.

Risulta così possibile applicare parzialmente un argomento z, per il quale non esiste una prova di Pz: la sua versione estratta verrà così considerata completa. Inoltre, il predicato Px cancellato potrebbe essere inabitabile e affermare l'uguaglianza di tipi distinti, facendo sì che il *type checker* sia ingannato accettando un corpo mal tipato e, se estratto, potrà causare un'ec-

cezione o anche divergenza a runtime. La soluzione in Coq è quella di rilevare questi casi e mantenere alcuni argomenti dummy quando necessario.

Ambiguità nell'assegnazione delle annotazioni

Coq permette la creazione di termini ambigui il cui tipo di ritorno diventa noto solo a tempo di esecuzione. Un esempio è (if e then nat else $True$) che, a seconda del valore della condizione e, può restituire il tipo dei naturali, che è informativo ($nat : Set$), o il tipo singoletto "vero", che è non informativo ($True : Prop$). Ciò è conseguenza dell'esistenza dell'universo $Type$ al quale sia nat che $True$ appartengono.

Le versioni di Coq precedenti il lavoro svolto da Letouzey rifiutavano di estrarre termini appartenenti al sort $Type$, mentre nella versione attuale si sostituiscono le parti non informative con una costante e si operano semplificazioni ad-hoc quando possibile.

1.2.2 Problemi di traduzione

Minore espressività del sistema di tipi

I linguaggi target dell'estrattore di Coq (OCaml e Haskell) hanno un sistema di tipi meno potente rispetto a quello del dimostratore. In particolare, l'assenza di universi e tipi dipendenti rende intraducibili questi costrutti.

Inoltre, anche l'espressività di costrutti presenti risulta essere differente tra Coq e il target, come nel caso del polimorfismo. Per esempio, in Coq è possibile definire un termine $f : Set \to T \to T$ che prende in input un tipo di sort Set, un termine di tipo $T : Set$ e ritorna un altro termine di quel tipo. La versione estratta non può prendere in input un tipo ed è quindi costretta ad eliminare il primo argomento, ottenendo una nuova signature per $f : T \to T$. Di conseguenza, il linguaggio target è costretto ad affidarsi completamente al proprio meccanismo di inferenza e non può sfruttuare la signature più precisa descritta in Coq.

Un esempio delle implicazioni di questo fenomeno è il seguente. Se si utilizza f per costruire in OCaml una coppia così formata:

```
let dp f = (f 0, f true)
```

questo termine non è tipabile, mentre in Coq dp ha tipo $(Set \to T \to T) \to Nat \times True$. Infatti, dp è tipabile in System-F ma non in Hindley-Milner. La soluzione implementata in Coq da Letouzey consiste nell'utilizzare la feature `Obj.magic` in OCaml e `unsafeCoerce` in Haskell, le quali assegnano un tipo generico ai termini passati. Questa operazione causa un'effettiva soppressione del controllo del tipo dell'argomento passato, per cui la nuova versione risulta tipabile:

```
let dp f = (Obj.magic f 0, Obj.magic f true)
```

Essendo il termine iniziale ben tipato in Coq e assumendo la correttezza del processo di estrazione, si ha garanzia dell'assenza di errori a *runtime*, anche in presenza di `Obj.magic`.

1.2.3 Riduzione

In seguito a questa fase di estrazione, si vuole garantire che un termine estratto, se calcolato, porti ugualmente ad un risultato significativo, cioè si vuole garantire l'equivalenza delle due computazioni.

Le regole di riduzione del calcolo delle costruzioni possono applicarsi senza variazione anche ai termini estratti, fuorché in quei casi in cui ad un redesso del calcolo delle costruzioni corrisponda un termine non più riducibile. Si dovrà dunque determinare una nuova strategia di riduzione per questi termini, i quali sono elencati di seguito:

1. $((\lambda x\!:\!X.\ t)\ u) \quad \longrightarrow_{extr} \quad (\Box\ u')$
2. $< \ldots >$ Cases e of $\ldots$ end $\quad \longrightarrow_{extr} \quad < \ldots >$ Cases $\Box$ of $\ldots$ end
3. $(\text{Fix } f_i \{\ldots\}\ u_1 \ldots u_n) \quad \longrightarrow_{extr} \quad (\Box\ u'_1 \ldots u'_n)$
4. $(\text{Fix } f_i \{\ldots\}\ u_1 \ldots u_n) \quad \longrightarrow_{extr} \quad (\text{Fix } f_i \{\ldots\}\ u_1 \ldots \Box)$

Caso 1 e 3

Il primo caso non si presenta mai in maniera diretta, ma solamente come conseguenza della riduzione di un'applicazione che prenda in input uno schema di tipo:

$$(\lambda f\!:\!A \to \mathit{Prop}.\ f\ 3)\ (\lambda x\!:\!A.\ \mathit{True}) \quad \longrightarrow_{extr} \quad (\lambda f\!:\!\Box.\ f\ 3)\ \Box \quad \longrightarrow_{\beta} \quad (\Box\ 3)$$

Per risolvere il problema è necessario aggiungere una nuova regola di riduzione ad-hoc:

$$(\Box\ u) \quad \longrightarrow_{\Box} \quad \Box$$

Il terzo caso ha la medesima forma del primo e viene automaticamente risolto dalla nuova regola introdotta.

Caso 2

Per come la logica di Coq è costruita, il secondo caso si presenta solo in due particolari situazioni:

1. un case vuoto scrutinato di tipo False (un tipo inabitabile);
2. un case con un solo costrutture il cui scrutinato è non informativo e il cui tipo, a seguito dell'eliminazione, è un singoletto.

Un case vuoto con condizione falsa non verrà mai eseguito e può quindi essere tradotto come un'eccezione. Invece, il case con singolo costruttore, non dipendendo dal dummy in input, potrebbe essere trasformato direttamente nel valore prodotto (per semplicità sintattica si assume che in questo caso il costruttore abbia zero parametri):

$$< \ldots > \text{Cases } \Box \text{ of } O \text{ end} \quad \longrightarrow_{\Box} \quad O$$

Questa operazione potrebbe però causare errori di tipaggio, poiché il tipaggio del ramo può dipendere da un'ipotesi falsa estratta dal case. Per impedire che ciò avvenga bisogna passare dalla riduzione forte a quella debole, impedendo

la riduzione di O. Il passaggio non è ovvio: esiste una prova a garanzia di questo fatto.

Inoltre, sostituendo gli argomenti logici con il dummy non è più possibile ricavare il numero di parametri, per questo motivo il costrutto case deve contenere l'informazione di quanti argomenti si aspetta di ricevere:

$$< \ldots > \text{Cases}_n \ \Box \text{ of } f \text{ end} \quad \longrightarrow_{\Box} \quad \underbrace{f \ \Box \ldots \Box}_{\text{n volte}}$$

Caso 4

Il quarto caso e il case con singolo costruttore sono molto simili: entrambi producono un termine informativo a partire da un termine non informativo. In particolare, nel quarto caso abbiamo un operatore di punto fisso che produce un risultato informativo, ma la cui guardia viene trasformata in dummy.

Una prima idea di riduzione di questo termine consisterebbe nel rimuovere la guardia ma, facendo questo, il termine diventerebbe un loop che potrebbe essere ridotto all'infinito. Il passaggio alla *call-by-value* unito al mantenimento del $\Box$ permette di raggiungere la forma normale.

1.2.4 Funzione di estrazione

La funzione di estrazione si occupa di eliminare le parti di codice non informativo. In particolare, tutti i termini di sort *Prop*, tutti i tipi e gli schemi di tipo vengono sostituiti con una costante di tipo dummy $\Box$.

La giustificazione dell'eliminazione di tutti i tipi consiste principalmente nel fatto che, nonostante sia possibile seppur raro che una particolare dimostrazione miri alla costruzione di un tipo, i linguaggi ML che sono l'output dell'estrazione non supportano l'uso di tipi come termini. Inoltre, come conseguenza ovvia della sostituzione si ha che l'applicazione della funzione di estrazione causa la perdita del tipaggio (es. vedi 1.2.2); i termini estratti sono quindi non tipati.

In fase di traduzione, si rende necessario determinare l'implementazione di □ per poterlo inserire nel codice del linguaggio target. In prima approssimazione, si potrebbe pensare di tradurlo come un elemento del tipo singoletto Unit ma, come vedremo nella prossima sottosezione, può capitare il caso in cui sia necessario passare a □ degli argomenti. È quindi necessario tradurlo come una funzione ricorsiva che semplicemente ignora tutti gli argomenti passati; risulta inoltre necessario sospendere il tipaggio poiché il termine così prodotto non sarebbe ben tipato.

Di seguito, riporto come esempio la traduzione in OCaml:

```
let rec □ x = Obj.magic □
```

1.2.5 Semplificazioni

Applicazioni parziali e argomenti inutili

Per ridurre il numero di □ presenti nel codice e aumentarne la leggibilità, le funzioni vengono estratte senza argomenti inutili. Le chiamate totali risultano quindi prive di □, mentre le chiamate parziali vengono riastratte su □ per bloccarne l'esecuzione.

1.3 Two-level minimalist foundation

La particolarità della minimalist foundation sviluppata da Maietti [Mai08], per la quale ho realizzato un estrattore di codice, è quella di essere una formulazione della logica a due livelli, uno intensionale e uno estensionale. Il livello intensionale (mTT) si fonda su due principali proprietà:

- minimalità, cioé le proprietà di questa formulazione della logica sono comuni a quelle di tutte le altre principali formulazioni in teoria degli insiemi, in teoria delle categorie e in teoria dei tipi;
- costruttività, cioé soddisfa il paradigma "proofs-as-programs" per il quale il valore di verità di un enunciato non è dato da booleani, ma

dall'esistenza di prove che abitano l'enunciato (Heyting algebra). Queste prove possono essere viste come programmi che costruiscono effettivamente gli oggetti descritti dall'enunciato.

Una teoria che rispetti il paradigma "proofs-as-programs" necessita di essere compatibile con l'assioma di scelta e con la tesi di Church:

- Assioma di scelta

  $$\forall x \in A.\exists y \in B.R(x,y) \implies \exists f \in (A \to B).\forall x \in A.R(x, f(x))$$

  dove R è una relazione tra due insiemi A e B.

  Un'interpretazione di questo enunciato è che per ogni relazione sia possibile costruire una mappa che la descrive.

- tesi formale di Church

  $$\forall f \in (\mathbb{N} \to \mathbb{N}).\exists e \in \mathbb{N}.\forall x \in \mathbb{N}.\exists y \in \mathbb{N}.(T(e,x,y) \wedge U(y) = f(x))$$

  dove T è il predicato di Kleene che determina se la funzione indicizzata con e, eseguita su una macchina di Turing, termina prendendo in input x e raggiungendo lo stato finale y, mentre U è una funzione che decodifica y nel risultato della chiamata.

  L'enunciato dà una definizione precisa di funzione effettivamente calcolabile.

Il livello estensionale (emTT) è dotato di una maggiore capacità espressiva e facilità d'uso, poiché permette di lavorare utilizzando le classi di equivalenza e l'uguaglianza estensionale. Questi costrutti consentono di considerare come indifferenti tutte quelle strutture che hanno i medesimi elementi e i medesimi comportamenti.

Una teoria dotata dell'uguaglianza estensionale per le funzioni è però incompatibile con l'assioma di scelta e la tesi di Church, poiché l'uguaglianza è una proprietà indecidibile.

Questa incompatibilità tra le proprietà volute è la motivazione alla base della divisione della logica in due livelli. Il passaggio dal livello intensionale a quello estensionale è fatto sulla base del principio *forget-restore* di Sambin [SV98], per il quale i concetti estensionali sono astrazioni operate sui concetti intesionali "dimenticando" una parte non informativa della prova, la quale può essere ricreata successivamente. In pratica, costruendo le classi quozionte nel livello intensionale è possibile implementarvi il livello estensionale. Esiste dunque, anche se non decidibile, una fase di compilazione dal livello estensionale a quello intensionale che permette, in casi semplici, di ricostituire l'informazione tralasciata, cioé ricreare le prove mancanti e simulare i quozienti in un linguaggio che ne è privo.

In questo quadro, la fase di estrazione di codice si pone al termine: una volta espressa una dimostrazione al livello estensionale e compilatala al livello intensionale, è possibile estrarre un programma in un linguaggio di programmazione a partire dall'espressione intensionale. Per via della scarsa correlazione che hanno il livello estensionale e la fase di compilazione con il mio lavoro, mi limiterò a descrivere in dettaglio il livello intensionale.

1.3.1 mTT

Il livello intensionale risulta molto simile alla teoria dei tipi intensionale di Martin-Löf, ma è dotato di alcuni ampliamenti per l'interpretazione del livello estensionale. mTT è una logica many-sorted, i cui tipi possono essere di quattro kind:

- proposizioni piccole ($prop_s$);
- *set*;
- proposizioni (*prop*);
- collezioni (*col*).

Già da questa divisione in kind è possibile vedere come la distinzione tra proposizioni e termini informativi sia completamente esplicita: *set* e *col* con-

tengono oggetti matematici, rispettivamente calcolabili e potenzialmente non calcolabili, mentre $prop_s$ e $prop$ contengono le proposizioni che li descrivono.

$$type \in \{prop_s, set, prop, col\}$$

Il tipaggio avviene seguendo i giudizi elencati di seguito, i quali si occupano di definire in un contesto Γ rispettivamente l'istanza di tipo, l'uguaglianza di tipo, l'associazione tra termine e tipo e l'uguaglianza tra termini. La logica supporta i tipi dipendenti, per cui un tipo può essere parametrizzato su un termine di altro tipo.

$$A\ type\ [\Gamma] \quad A = B\ type\ [\Gamma] \quad a \in A\ type\ [\Gamma] \quad a = b \in A\ type\ [\Gamma]$$

I set sono definiti induttivamente, specificando le regole di formazione dei loro elementi, mentre le collezioni non devono sottostare a questo vincolo. A partire dalla divisione precedente, se ne opera una simile anche sulle proposizioni: le $prop_s$ sono tutte le proposizioni quantificate unicamente sui set. Inoltre, le proposizioni possono essere viste come una collezione di prove e la medesima relazione vale anche tra proposizioni piccole e set.

$$\textbf{Sub1)}\ \frac{A\ prop_s}{A\ prop} \quad \textbf{Sub2)}\ \frac{A\ set}{A\ col} \quad \textbf{Sub3)}\ \frac{A\ prop_s}{A\ set} \quad \textbf{Sub4)}\ \frac{A\ prop}{A\ col}$$

In più, esiste anche la collezione di tutte le proposizioni piccole che permette di trasformare il tipo di una proposizione piccola in un termine, elemento della collezione, e viceversa.

$$\textbf{F)}\ \mathrm{prop_s}\ col \quad \textbf{I)}\ \frac{A\ prop_s}{A \in \mathrm{prop_s}} \quad \textbf{E)}\ \frac{A \in \mathrm{prop_s}}{A\ prop_s}$$

All'interno delle regole di inferenza viene mantenuta la distinzione forte tra proposizioni e collezioni. In particolare, le regole di eliminazione delle proposizioni producono solamente proposizioni, anche nel caso in cui si stia manipolando un *set* o una *col*; il testimone viene incapsulato necessariamente all'interno della proposizione di output e non può essere ritornato singolarmente.

Di seguito, trascrivo a titolo di esempio le regole di introduzione ed eliminazione dell'uguaglianza intensionale e dell'esistenziale, annotate esplicitamente con i kind dei termini di input e output:

$$\mathbf{I)}\ \frac{b \in B\ col \qquad \begin{array}{c} C(x)\ prop\ [x \in B] \\ c \in C(b) \end{array}}{\langle b, c\rangle_{\exists} \in (\exists x \in B.\ C(x))\ prop}$$

$$\mathbf{E)}\ \frac{\begin{array}{c} M\ prop \\ m(x,y) \in M\ [x \in B, y \in C(x)] \end{array}}{Elim_{\exists}(\langle b, c\rangle_{\exists}, m) = m(b,c) \in M\ prop}$$

Quantificatore esistenziale: introduzione seguita da eliminazione

$$\mathbf{I)}\ \frac{a \in A\ col}{id(a) \in ID(A,a,a)\ prop} \qquad \mathbf{E)}\ \frac{\begin{array}{c} C(x,y)\ prop\ [x \in A, y \in A] \\ c(x) \in C(x,x)\ [x \in A] \end{array}}{Elim_{ID}(id(a), c) = c(a) \in C(a,a)\ prop}$$

Uguaglianza proposizionale: introduzione seguita da eliminazione

Il completo isolamento esistente tra proposizioni e collezioni non è casuale, ma discende da motivazioni pratiche:

- per definizione, le proposizioni sono un costrutto a sé stante e solo a posteriori vengono riconosciute anche come una collezione di prove;
- la presenza di questo vincolo previene la validità dell'assioma di scelta. Infatti, se valesse l'assioma di scelta allora i due costrutti dell'esistenziale e della coppia collasserebbe nell'esistenziale costruttivo della logica intuizionista. La distinzione avviene per mantenere la compatibilità con alcune formulazioni della logica nelle quali non è presente l'assioma di scelta.

Esiste un altro importante dettaglio relativo all'interazione tra kind nelle regole di inferenza. L'eliminazione della lista e della somma disgiunta è sempre una *col*, anche se esse possono avere solamente kind *set*.

Ciò diventa particolarmente rilevante per l'estrazione poiché, in prima approssimazione, si potrebbe pensare di estrarre solamente i *set* e di denotare come codice inutile tutti gli altri kind, comprese le *col*. Ma ovviamente l'estrazione di elementi da una lista sembra essere un'operazione informativa, per cui risulta necessario estrarre sia *set* che *col*, considerando solo le proposizioni come codice inutile.

(Introduzione lista vuota) $$\frac{List(C)\ set}{\varepsilon \in List(C)\ set}$$

(Introduzione lista) $$\frac{s \in List(C)\ set \quad c \in C\ set}{cons(s,c) \in List(C)\ set}$$

(Eliminazione lista vuota)

$$\frac{\begin{array}{c}\varepsilon \in List(C)\ set \quad a \in L(\varepsilon)\ col \quad L(z)\ col\ [z \in List(C)] \\ l(x,y,z) \in L(cons(x,y))\ col\ [x \in List(C)\ set, y \in C\ set, z \in L(x)\ col]\end{array}}{Elim_{List}(\varepsilon,a,l) = a \in L(\varepsilon)\ col}$$

(Eliminazione lista)

$$\frac{\begin{array}{c}tl \in List(C)\ set \quad hd \in C\ set \quad a \in L(\varepsilon)\ col \quad L(z)\ col\ [z \in List(C)] \\ l(x,y,z) \in L(cons(x,y))\ col\ [x \in List(C)\ set, y \in C\ set, z \in L(x)\ col]\end{array}}{Elim_{List}(cons(tl,hd),a,l) = l(tl,hd,Elim_{List}(tl,a,l)) \in L(cons(tl,hd))\ col}$$

1.3.2 Differenze tra mTT e Coq

Ora avendo a disposizione cognizioni riguardo sia l'estrazione di codice di Coq che la teoria di mTT, risulta necessario valutare le differenze tra le due logiche per capire quali problematiche possono avere in comune.

1. Per cominciare, in Coq è sempre possibile passare da tipi a termini e viceversa in maniera completamente svincolata, tale per cui la distinzione tra i due concetti risulta particolarmente labile. In mTT invece questo passaggio può essere effettuato solamente nel caso specifico dei

tipi delle proposizioni piccole e solamente in direzione della collezione degli stessi. Inoltre, questo passaggio avviene tramite un operatore esplicito, cosa che consente di gestire agevolmente questo caso.

2. Un'altra fonte di promiscuità in Coq è la possibilità di passare equivalentemente proposizioni e termini informativi ai costruttori delle strutture del linguaggio. Diversamente, in mTT, ogni costrutto accetta solo uno specifico sottoinsieme dei kind disponibili e per la maggior parte dei costrutti esistono due versioni: una informativa alla quale si può passare quel che si vuole e una non informativa che accetta in input solamente proposizioni.

3. I tipi induttivi di Coq vengono eliminati tramite *pattern matching*. In mTT il *pattern matching* viene usato per eliminare la setSum, che implementa la somma logica binaria. Questo comporta una differenza nell'arietà del costrutto che è tutto sommato marginale; in entrambe le formulazioni è importante mantenere intatta la condizione per poter determinare quale ramo della computazione venga imboccato.

4. A differenza di Coq, in mTT non è presente l'operatore di punto fisso.

In particolare, la distinzione più marcata tra kind e tra tipi e termini in mTT rende l'estrazione di codice sensibilmente più agevole.

1.4 λProlog

L'estrattore di codice per mTT è stato implementato utilizzando il linguaggio λProlog. Anche il type checher e il compilatore dal livello estensionale a quello intensionale, precedentemente implementati sotto la supervisione di Claudio Sacerdoti Coen da Alberto Fiori [FS18] e Giacomo Molinari [Mol18] nelle loro rispettive tesi, sono stati realizzati utilizzando il medesimo linguaggio.

λProlog nasce come estensione di Prolog elaborata da Dale Miller e Gopalan Nadathur [MN86]: è un linguaggio logico di ordine superiore con *binder*. Essendo di ordine superiore permette di utilizzare funzioni come dati, consentendo di manipolare strutture sintattiche contenenti *binder*. Questo orientamento risulta di grande utilità nell'ambito del mio lavoro di tesi poiché l'estrazione consiste in una serie di fasi di program transformation, le quali necessitano di poter manipolare agevolmente non solo le strutture sintattiche di mTT ma anche eventuali ipotesi espresse direttamente in λProlog.

I *binder* di λProlog permettono di ottenere un costrutto equivalente alla quantificazione universale, il quale consente di parametrizzare gli argomenti dei predicati relativamente a una o più variabili. Inoltre, il linguaggio è tipato ed è possibile dichiarare predicati polimorfi definendo manualmente la struttura delle signature. In coda, notiamo che il paradigma logico ereditato da Prolog fornisce al linguaggio un elevato grado di espressività che permette di descrivere succintamente il comportamento voluto.

Il primo compilatore realizzato per λProlog è Teyjus, progetto supervisionato dai creatori del linguaggio stesso, Miller e Nadathur. Nello svolgimento della mia tesi ho utilizzato invece ELPI, un interprete per λProlog sviluppato principalmente da Claudio Sacerdoti Coen ed Enrico Tassi [Dun+15].

Capitolo 2

Implementazione

In questo capitolo, descriverò in dettaglio l'estrattore di codice sviluppato per questa tesi e le problematiche che ho dovuto affrontare. L'implementazione dell'estrazione di codice è divisa in tre fasi:

1. **starificazione**: in cui si marcano come inutili tutti i termini trivialmente privi di informazione;

2. **eliminazione**: in cui si esaminano tutte le componenti dei termini contenuti nel codice per determinare se contengano informazione e, in caso contrario, collassarli;

3. **traduzione**: in cui si traducono i termini restanti in programmi OCaml o Haskell.

Ogni fase di trasformazione è definita per ricorsione strutturale sul termine di mTT da estrarre. La starificazione mappa termini ben tipati di mTT in termini ben tipati di mTT, il risultato viene passato alla fase di eliminazione che produce a sua volta termini ben tipati di mTT. L'ultimo passo da eseguire è la traduzione che produce in output una stringa contenente il codice nel linguaggio target.

La struttura dell'algoritmo ricalca superficialmente sia l'approccio di Berardi che l'estrazione in Coq descritti nel capitolo precedente, ma ci sono importanti differenze. Nell'estrazione di mTT i tipi vengono mantenuti e sono

sottoposti alle medesime trasformazioni che avvengono per i termini, i quali rimangono dunque tipati nelle varie fasi, a differenza di Coq che eliminava tutte le informazioni di tipo non potendo preservare il tipaggio. Mantenendo anche i tipi, diventa necessario sostituire la trasformazione dei termini inutili in $\square$ con una manipolazione più strutturata. I termini inutili e il loro tipo vengono mappati diversamente, rispettivamente nella stella ($\star$) e in un nuovo tipo singoletto ($\mathbb{1}$) abitato da $\star$. mTT supporta già al proprio interno il tipo Unit, ma nell'estrattore ne utilizziamo una nuova ridefinizione. Infatti, il tipo Unit di mTT ha kind *set*, mentre quello utilizzato per l'estrattore ha kind $prop_s$.

$$\star_e \in \mathbb{1}_e : prop_s$$

La necessità della ridefinizione precedente è una conseguenza diretta della relazione di sottotipaggio tra kind: essendo $prop_s$ convertibile a tutti gli altri kind, i suoi elementi possono occupare liberamente qualsiasi posizione all'interno di un costrutto, consentendo di utilizzarli come sostituzione univoca.

Nelle prossime sezioni, discuterò in dettaglio le tre fasi dell'estrazione: quando si renderà necessario parlare della struttura del codice mTT cercherò di trascrivere gli esempi usando la sintassi standard della logica; mostrerò frammenti di codice mTT solamente se strettamente necessario alla comprensione del problema discusso.

2.1 Starificazione

Questa prima fase compie un numero molto limitato di operazioni: vengono trasformati in stella tutti e soli quei termini base, privi di contenuto informativo. In pratica, gli unici a soddisfare questa proprietà sono i seguenti:

- la collezione delle proposizioni piccole (**propS**);
- le costanti e i tipi primitivi di kind proposizione.

Tutto il resto viene lasciato intatto. La ragione concettuale di questa scelta è che questa fase sia intesa come un preprocessing, nel quale vengono operate solo trasformazioni elementari. Poiché per determinare la necessità della trasformazione in stella di un costrutto sarebbe necessaria un'analisi dei suoi sottotermini e delle relazioni tra di essi, questa operazione viene posticipata alla fase successiva.

L'esistenza di questa fase è principalmente collegata alla crescita organica dell'estrattore: nelle prime versioni, la starificazione si occupava di compiere un più ampio gruppo di operazioni, le quali col tempo si sono rivelate non necessarie o comunque operabili con minor sforzo in altri momenti dell'estrazione e sono state quindi soppresse. In linea teorica, dovrebbe essere possibile accorpare questa fase a quella di eliminazione lasciando inalterato il comportamento del programma.

2.1.1 Ricorsione strutturale su chiamate di libreria

Un problema rilevante che ho dovuto affrontare in questo lavoro di tesi è stata l'estrazione delle definizioni presenti in libreria, la quale non era contemplata nella formulazione di Maietti ma è stata introdotta da Claudio Sacerdoti Coen e Giacomo Molinari [Mol18] per realizzare la fase di compilazione. Discuterò più avanti di come ho operato il collasso parziale o totale delle chiamate, per il momento mi limiterò a illustrare come ho dovuto operare per fare in modo che la semplice ricorsione strutturale funzionasse correttamente.

L'anatomia di una definizione di libreria, in questo caso la definizione di un tipo, è la seguente:

$$\forall B : set.\ \forall C(b \in B) : set.\ \forall bs \in setoid(B).$$
$$setoidDep(B, C, bs) : col = \ldots$$

dove è possibile parametrizziare sia su tipi (B, C) che su termini (bs) ed è necessario definire sia il kind che il corpo della chiamata di libreria. Nel caso della definizione di un termine, è necessario indicare il tipo e il corpo

della chiamata. Inoltre, durante lo sviluppo, è stato introdotto il vincolo che i parametri compaiano nella testa della definizione nel medesimo ordine in cui appaiono nelle astrazioni per semplificare il lavoro di alcune parti dell'estrazione.

Sintatticamente, in λProlog una chiamata di libreria viene formulata nel modo seguente:

```
call_name arg0 arg1 ... argN
```

Il problema che sorge quando si cerca di andare in ricorsione strutturale su un termine come il precedente è l'impossibilità di sapere a priori il nome della chiamata e, di conseguenza, il numero di argomenti.

Numero degli argomenti

Non avendo a disposizione un meccanismo di pattern matching che consenta l'uso di argomenti variadici e non potendo sapere a priori quali chiamate di libreria siano state effettivamente dichiarate, non è nemmeno possibile effettuare la ricorsione. L'impossibilità di accedere alla struttura delle chiamata preclude la possibilità di operare le trasformazioni di estrazione sui singoli parametri, i quali potrebbero essere arbitrariamente complessi. Per risolvere questo problema, è stato introdotto un operatore `lib_app`, il quale ha lo scopo di applicare uno alla volta gli argomenti alla chiamata di libreria.

L'esempio seguente di codice:

```
setoidDep Bi (x \ singleton) bS
```

è stato quindi sostituito da:

```
(lib_app (lib_app (lib_app setoidDep Bi) (x \ singleton)) bS)
```

In questo modo, il pattern matching viene eseguito sulla testa `lib_app`, la quale ha sempre due argomenti.

Tipo degli argomenti

Per completare la ricorsione sulle chiamate di libreria rimane però un ultimo ostacolo, poiché gli argomenti passati potrebbero essere tipi, termini o le loro controparti dipendenti e ognuno di essi va processato utilizzando metodi diversi.

Non essendovi modo di estrarre questa informazione dal contesto, risulta necessario mantenere una lista di tutte le chiamate di libreria precedentemente definite in modo da poter ricavare l'informazione richiesta dalla definizione.

L'overhead necessario a mantenere questa lista è tutto sommato trascurabile, poiché non si necessita l'intera definizione ma solo il nome della chiamata e le astrazioni che lo precedono. Inoltre, come vedremo, questa informazione sarà richiesta anche durante alcune operazioni delle fasi successive.

Dalla lista viene estratta la definizione della chiamata corrispondente a quella attualmente in fase di manipolazione e le viene applicata la lista degli argomenti passati alla chiamata corrente. In questo modo, è possibile estrarre sia il tipo in ELPI degli argomenti che il tipo in mTT di un eventuale parametro dei tipi dipendenti, in modo da poter effettuare correttamente la ricorsione strutturale.

Per esemplificare questo procedimento, applichiamolo ad un'ipotetica chiamata di *setoidDep*, la quale prende in input il tipo singoletto, un tipo dipendente che ignora il proprio argomento e produce un singoletto e il termine stella:

$$setoidDep(\mathbb{1}_e, \mathbb{1}_e(x), \star_e)$$

Per prima cosa, estraiamo dalla lista la definizione della chiamata:

$$\forall B : set.\ \forall C(b \in B) : set.\ \forall bs \in setoid(B).$$
$$setoidDep(B, C, bs) : col = \ldots$$

Da qui, recuperiamo l'informazione che il primo argomento è un tipo di kind *set*, quindi effettuiamo la ricorsione strutturale su $\mathbb{1}_e$ di conseguenza. Poi,

applichiamo il valore $\mathbb{1}_e$ al primo schema di tipo, ottenendo:

$$\forall C(b \in \mathbb{1}_e) : set.\ \forall bs \in setoid(\mathbb{1}_e).$$
$$setoidDep(\mathbb{1}_e, C, bs) : col = \ldots$$

Da ciò, ricaviamo che il secondo argomento è un tipo dipendente parametrizzato su un termine di tipo $\mathbb{1}_e$. Ripetiamo quindi il procedimento descritto al passo precedente, ottenendo:

$$\forall bs \in setoid(\mathbb{1}_e).$$
$$setoidDep(\mathbb{1}_e, \mathbb{1}_e(x), bs) : col = \ldots$$

Ci è quindi noto che il terzo argomento è un termine di tipo $setoid(\mathbb{1}_e)$, per cui andiamo in ricorsione strutturale su di esso completando la manipolazione della chiamata di libreria.

2.1.2 Ricorsione strutturale su applicazione di variabili

Il medesimo problema rilevato sulle chiamate di libreria si presenta anche sull'applicazione di variabili. Per esempio, il termine seguente che applica la prima proiezione della variabile `z` alla variabile dipendente `Ci`:

```
(Ci (p1 z))
```

non può essere esaminato con la ricorsione strutturale. Per eliminare questo problema, si adotta la medesima soluzione utilizzata nel caso delle chiamate di libreria: è stato introdotto un nuovo operatore `#` che ha la funzione di esplicitare l'applicazione di una variabile.

L'esempio precedente assume dunque la nuova forma:

```
(# Ci (p1 z))
```

Ovviamente, l'introduzione di questo nuovo operatore obbliga a modificare anche il modo in cui viene gestita l'applicazione delle variabili nel codice esistente del *type checker* e del compilatore tra i due livelli. In particolare, si vuole che `#` sia presente e utilizzabile solamente durante l'estrazione, mentre il tipaggio e la compilazione non dovrebbero prenderlo in considerazione.

Si procede dunque a scrivere un predicato che vada in ricorsione strutturale su tutti i costrutti di mTT, compresa la libreria e le ipotesi, il quale cancelli ogni occorrenza del nuovo operatore. Il caso base di questa passata è il seguente:

$$\#(A, B) \quad \longrightarrow_{erase} \quad (A\ B)$$

dove A è una variabile e B è un termine arbitrariamente complicato. Il predicato così definito si applica ai termini prima di effettuarne il tipaggo o la compilazione, ottenendo una versione ripulita da `#`.

Invece, nell'estrazione si utilizza l'operatore `#` per effettuare correttamente la ricorsione strutturale e si ha quindi bisogno di mantenerlo durante tutte le fasi. Sono comunque presenti alcuni casi in cui è necessario richiedere la cancellazione dell'operatore per un sottotermine: quando si richiede il tipaggio di un'espressione e quando si premettono delle ipotesi ad una definizione di libreria. Questo fenomeno è determinato dal fatto che i due termini che differiscono per l'inserimento di `#` non risultano convertibili. Occorre quindi operare una cancellazione *ad-hoc* nei casi in cui è noto che avverrà un tentativo di conversione.

2.1.3 Ricorsione strutturale su variabili

Poiché nella libreria di mTT è possibile astrarre su termini dipendenti, può capitare di trovare una semplice variabile al posto di un termine parametrico. Si vorrebbe che ELPI fosse in grado di gestire la ricorsione strutturale su questo caso come quella del caso standard, senza la scrittura di codice aggiuntivo. Purtroppo, per come è strutturato il sistema di *binder* in λProlog, si ottiene un comportamento indesiderato.

Preso un predicato p che effettui una ricorsione strutturale, nel caso in cui si incontri un termine con *binder* si deve operare nel modo seguente:

$$p(binder\ F) :- \Pi z.\ p\ (F\ z)$$

dove F è un termine dipendente e Π opera una quantificazione universale in λProlog che dichiara una variabile fresca z da applicare al termine

dipendente. Applicando l'operazione precedente per $F = (\lambda x.\ G[x])$ si ottiene:

$$p(binder\ \lambda x.\ G[x]) :- \Pi z.\ p\ (G[z])$$

il quale ha la medesima forma del termine di partenza. Se si utilizza invece una variabile dipendente $F = y$ si ottiene:

$$p(binder\ y) :- \Pi z.\ p\ (y\ z)$$

il quale è operativamente equivalente a y ma, se venisse utilizzato nella costruzione di un termine di output, ne sporcherebbe la sintassi con un'astrazione inutile. Il nuovo termine sarebbe meno leggibile e non sarebbe possibile processarlo correttamente durante le fasi successive. Infatti, come descritto nella sottosezione precedente, le varie fasi dell'estrattore si aspettano che le applicazioni di variabili siano precedute da `#`, cosa che non può avvenire in questo caso.

Per ovviare a questo inconveniente, è necessario determinare tutti i casi in cui potrebbe venire passata una variabile al posto di un termine dipendente, rilevare manualmente se il valore passato sia una variabile e, in tal caso, non effettuare la ricorsione strutturale su di esso ma mapparlo con l'opportuno predicato.

2.2 Eliminazione

In questa parte della trasformazione, i termini e i tipi vengono mappati nella loro controparte privata del codice inutile. L'eliminazione è a sua volta divisa in tre sottofasi, le quali vengono eseguite in maniera alternata per ogni sottotermine:

1. mappa di variabili e chiamate di libreria;
2. scorporo di eventuale codice informativo morto;
3. collasso di codice inutile.

2.2.1 Mappa di variabili e chiamate di libreria

Questa parte si occupa di mappare occorrenze di termini e tipi, sulla cui definizione sia già stata precedentemente eseguita l'estrazione, nella loro controparte semplificata. Istanze parametriche, la cui estrazione sia già completata, possono comparire solamente in presenza di quantificatori universali ed esistenziali, non di mTT, ma del suo metalivello. In particolare, essi sono rispettivamente le variabili e le chiamate di libreria.

Variabili

In mTT è possibile costruire termini e tipi parametrizzati, cioè dipendenti, su di un termine, l'analisi dei quali richiede la dichiarazione di una variabile tramite quantificazione universale. In fase di estrazione diventa necessario chiedersi se il tipo di questa variabile sia o meno semplificabile. In caso affermativo, è necessario collassarlo, dichiarare una nuova variabile del tipo semplificato e richiedere che nel corpo del tipo/termine dipendente si operi la sostituzione della vecchia variabile con la nuova.

Per esempio, supponiamo di avere una variabile il cui tipo sia l'identità sul singoletto. L'estrazione di questo tipo è la seguente:

$$ID(\mathbb{1}, \star, \star) \quad \longrightarrow_{extr} \quad \mathbb{1}_e$$

Ora, dato un M dipendente da un x del tipo sopra specificato, la sua versione mappata è M stesso, nel quale a x si sia sostituito y del tipo estratto:

$$\forall x \in ID(\mathbb{1}, \star, \star).\ M(x) \longrightarrow_{map} \forall y \in \mathbb{1}_e.\ M[x \leftarrow y](y)$$

Chiamate

Come illustrerò nel dettaglio più avanti, dopo essere state estratte, le definizioni delle chiamate di libreria potrebbero vedere cambiato non solo il tipo dei propri argomenti, ma anche il numero.

Diventa quindi necessario estendere la lista di definizioni di libreria utile alla fase di starificazione, trasformandola in una mappa che metta in corrispondenza la definizione originale con il proprio estratto. Inoltre, è necessario

tenere traccia di quali argomenti siano stati eliminati poiché questa informazione andrebbe normalmente persa e non sarebbe più possibile determinare cosa passare alla chiamata estratta.

Durante la fase di mappatura delle chiamate si ha anche bisogno di applicare la fase di eliminazione a tutti gli argomenti conservati, in modo da mantenere il corretto tipaggio dell'espressione. Per di più, è necessario estrarre anche il tipo dei parametri dei tipi dipendenti per poterne effettuare correttamente la mappatura all'interno dei corpi.

Si prenda, per esempio, la seguente definizione:

$$\forall B : set.\ \forall C(b \in B) : set.\ \forall bs \in setoid(B).$$
$$setoidDepCast(B, C, bs) \in \ldots = \ldots$$

che ha come estratto:

$$\forall B : set.\ \forall C(b \in B) : set.$$
$$setoidDepCast_{extr}(B, C) \in \ldots = \ldots$$

si potrebbe avere la sottostante estrazione per un'ipotetica chiamata:

$$\ldots setoidDepCast(ID(\mathbb{1}, \star, \star), \forall b \in ID(\mathbb{1}, \star, \star).\ \mathbb{1}, setoid_{ID}) \ldots$$
$$\downarrow_{extr}$$
$$\ldots setoidDepCast(\mathbb{1}_e, \forall b' \in \mathbb{1}_e.\ \mathbb{1}) \ldots$$

in cui il primo argomento è stato collassato poiché non informativo; il secondo è stato riparametrizzato da b in b'; infine, il terzo è stato eliminato poiché il suo estratto è sempre inutile. Come nota, utilizziamo $setoid_{ID}$ per abbreviare il corpo del setoide dell'identità, il quale è particolarmente verboso e poco pertinente con la trattazione attuale.

2.2.2 Scorporo di codice informativo morto

Come abbiamo visto nel primo capitolo, quando si effettua l'introduzione di una proposizione come l'identità o l'esistenziale, i termini che occorrono al suo interno non possono essere estratti tramite l'eliminazione per entrare

a far parte di termini informativi. Poiché la proposizione introdotta è codice inutile, anche il termine informativo incorporato andrebbe perso durante l'estrazione.

Supponendo di voler conservare questo codice, si potrebbe pensare di scorporarlo dal termine e ritornarlo tra gli output della fase di trasformazione. Purtroppo, questo primo tentativo risulta essere un fallimento, poiché il termine da scorporare potrebbe contenere un numero arbitrario di variabili quantificate universalmente al di fuori di esso, le quali sarebbero semplicemente libere senza le informazioni di contesto. Perciò, risulta necessario operare una riastrazione del termine che si desidera scorporare per ognuna delle variabili che contiene.

Per esempio, volendo portare fuori la chiamata di f dal blocco seguente, occorre effettuare due riastrazioni:

$$\forall x.\, \forall y.\, \forall z.\ \ldots f(x, z) \ldots \quad \longrightarrow_{abstr} \quad \forall x_0.\, \forall x_1.\, f(x_0, x_1)$$

Una volta accertato che il termine sia informativo, scorporo comincia determinando tutte le variabili contenute al suo interno. Il predicato extralogico `names` ha un unico parametro tramite il quale ritorna la lista di tutte le variabili attualmente dichiarate nel contesto di ELPI. A partire dalla lista di variabili, l'intersezione con il termine può avvenire agevolmente utilizzando un ulteriore predicato extralogico: `occurs`, il quale prende in input una costante e un termine di ELPI e fallisce se la prima non è contenuta nel secondo.

Purtroppo, la lista di variabili è insufficiente, poiché per utilizzarle con successo nella seconda parte del procedimento sarà necessario sapere anche se ognuna di esse rappresenta un tipo, un termine o la versione dipendente di uno dei due. Si rende quindi necessario costruire manualmente una lista di tutte le variabili astratte, mano a mano che si procede nella ricorsione strutturale del termine.

Costruire la lista delle variabili contenute nel termine da riastrarre però non basta, poiché anche i tipi delle variabili potrebbero essere a loro volta variabili o tipi compositi, i quali contengono variabili al proprio interno.

Perciò, l'algoritmo deve essere trasformato in un'iterazione di punto fisso: durante la prima iterazione si aggiungono alla lista tutte le variabili contenute nel termine, poi all'iterazione n-esima vengono aggiunte tutte le variabili contenute nei tipi delle variabili che si trovavano nella lista all'iterazione precendente. L'algoritmo termina quando la lista in input è uguale a quella in output.

Per chiarire questo procedimento, vediamo un esempio. Volendo scorporare la coppia seguente:

$$\langle x_3, x_4 \rangle_\Sigma$$

nel contesto Γ così definito:

$$\Gamma = \{x_0 : set,\ \forall y \in x_0.\ x_1 : set,\ x_2 \in x_0,\ x_3 \in (x_1\ x_2),\ x_4 \in \mathbb{1}\}$$

la determinazione della lista di variabili produrrebbe i seguenti passi:

$$\begin{aligned}
&\textbf{Start)}\ L = \{\} \\
&\textbf{Iter 0)}\ L = \{x_3,\ x_4\} \quad \leftarrow \quad \langle x_3, x_4 \rangle_\Sigma \\
&\textbf{Iter 1)}\ L = \{x_1,\ x_2,\ x_3,\ x_4\} \quad \leftarrow \quad x_3 \in (x_1\ x_2) \\
&\textbf{Iter 2)}\ L = \{x_0,\ x_1,\ x_2,\ x_3,\ x_4\} \quad \leftarrow \quad x_2 \in x_0 \\
&\textbf{Iter 3)}\ L = \{x_0,\ x_1,\ x_2,\ x_3,\ x_4\}
\end{aligned}$$

Successivamente, è necessario riastrarre il termine relativamente ad ogni variabile contenuta nella lista così costruita. Una volta terminata la riastrazione, sarà necessario innestare il termine dipendente ottenuto all'interno di una astrazione che leghi la variabile parametrica. Nel caso in cui questa variabile fosse un termine o un termine dipendente, si ha bisogno di effettuare anche l'eliminazione del suo tipo, in modo da poterne indicare la versione correttamente ridotta nell'incapsulamento.

Inoltre, la variabile riastratta potrebbe non essere un termine, ma un tipo, un tipo dipendente o un termine dipendente. Esistendo in mTT solamente l'astrazione sui termini (`setPi`), è stato necessario introdurre tre nuovi costrutti di astrazione che permettessero di parametrizzare anche su

tipi, termini dipendenti e tipi dipendenti. I termini contenenti questa nuova estensione sono ovviamente non tipabili in mTT e vengono conservati in una lista a parte, separata dall'output principale dell'estrattore.

Al di là dell'interesse che questa fase può generare per i suoi dettagli tecnici, non è facilmente determinabile in quali casi il codice che viene messo da parte possa essere effettivamente utile. Infatti, per come è costruita mTT, le collezioni contenute all'interno delle proposizioni non devono necessariamente essere calcolabili e costrutti quali l'esistenziale permettono di emulare le proprie controparti nella logica classica: in molti casi, il testimone potrebbe anche non essere una computazione, dipendendo da variabili libere di tipo non abitabile. Inoltre, nel caso in cui l'utente del dimostratore interattivo volesse richiedere esplicitamente la conservazione di quel codice, potrebbe isolarlo manualmente in un lemma del quale richiedere l'estrazione.

2.2.3 Collasso di codice inutile

Quest'ultima sottofase è dove viene effettivamente svolto il cuore del lavoro di eliminazione. Preso un termine composito, una volta applicata la trasformazione di eliminazione a tutti i suoi sottotermini, questo viene analizzato per valutarne la collassabilità.

L'operazione di collasso è descritta da una serie di regole che valutano il contenuto informativo degli argomenti del costrutto e, in caso esso sia nullo, trasformano l'intero blocco di codice in una stella. Questa procedura viene applicata ricorsivamente ad ogni nodo dell'albero del termine, riducendo solamente i sottoalberi inutili. In alcuni contesti, oltre al collasso totale, è possibile operare un collasso parziale, trasformando il costrutto in un altro più semplice ma dal comportamento equivalente.

Di seguito, analizzerò tutti i costrutti presenti in mTT, esponendo in che modo ognuno di essi possa essere ridotto.

Astrazione (`setPi`)

Il tipo dell'astrazione sui termini in mTT è $\Pi x \in B.\ C(x)$ con tipo di ritorno dipendente dal termine in input. Le regole di collasso sono le seguenti:

$$\textbf{A} \to \textbf{1 - F)}\ \frac{C(x) =_c \mathbb{1}_e}{\Pi x \in B.\ C(x) \mapsto \mathbb{1}_e}$$

$$\textbf{1} \to \textbf{A - F)}\ \frac{B =_c \mathbb{1}_e}{\Pi x \in B.\ C(x) \mapsto C(\star_e)}$$

dove $=_c$ indica uguaglianza a meno di conversione, cioè i due tipi sono identici una volta ridotti a forma normale.

L'astrazione sui termini viene scritta $\lambda x.\ c(x)$. Per mantenere coerenza tra tipo e termine, le regole di collasso sono determinate da quelle precedentemente illustrate per il tipo:

$$\textbf{A} \to \textbf{1 - I)}\ \frac{\lambda x.\ c(x) \in \Pi x \in B.\ C(x) \quad C(x) =_c \mathbb{1}_e}{\lambda x.\ c(x) \mapsto \star_e}$$

$$\textbf{1} \to \textbf{A - I)}\ \frac{\lambda x.\ c(x) \in \Pi x \in B.\ C(x) \quad B =_c \mathbb{1}_e}{\lambda x.\ c(x) \mapsto c(\star_e)}$$

dove $c(\star_e)$ viene sottoposto nuovamente alla fase di eliminazione. Ciò avviene poiché le nuove stelle inserite tramite l'applicazione potrebbero causare un ulteriore collasso all'interno di c.

L'applicazione ripetuta del processo di eliminazione dopo un collasso rende il costo computazionale di questa fase $O(n^2)$. All'atto pratico, il degrado di performance non è percepibile poiché, in genere, regole di questo tipo vengono applicate di rado e spesso a termini di piccole dimensioni. Volendo, sarebbe comunque possibile rendere l'algoritmo nuovamente $O(n)$ tentando l'applicazione di questa regola dopo il collasso di B, ma prima del collasso di $c(x)$. Questa modifica avrebbe però come effetto collaterale la perdita della regolarità del codice della fase di collasso.

Nonostante nella trattazione seguente non sarà specificato caso per caso, la reiterazione dell'eliminazione viene fatta ogni volta che una regola produca in output un nuovo termine costruito combinando i termini in ipotesi.

L'applicazione assume la forma $App_\Pi(f, b)$ e può essere collassata con le seguenti regole:

$$\textbf{1 - E)}\ \frac{f \in F \quad b \in B \quad F =_c \mathbb{1}_e}{App_\Pi(f, b) \mapsto \star_e}$$

$$\textbf{A} \to \textbf{1 - E)}\ \frac{f \in F \quad b \in B \quad F =_c \Pi x \in B.\ \mathbb{1}_e}{App_\Pi(f, b) \mapsto \star_e}$$

$$\textbf{1} \to \textbf{A - E)}\ \frac{f \in F \quad b \in B \quad B =_c \mathbb{1}_e}{App_\Pi(f, b) \mapsto f}$$

Si noti come la seconda regola di collasso dell'introduzione e l'ultima regola sopra esposta siano coerenti tra di loro, in quanto l'una opera la cancellazione dell'astrazione mentre l'altra elimina la sua applicazione, ottenendo un termine comunque ben tipato.

Coppia (`setSigma`)

Le coppie di mTT hanno tipo $\Sigma x \in B.\ C(x)$ con tipo del secondo elemento dipendente dal primo elemento. L'unica regola di collasso è la seguente:

$$\textbf{1} \to \textbf{1} \to \textbf{1} \times \textbf{1 - F)}\ \frac{B =_c \mathbb{1}_e \quad C(x) =_c \mathbb{1}_e}{\Sigma x \in B.\ C(x) \mapsto \mathbb{1}_e}$$

Non è possibile collassare coppie di cui un solo elemento sia di tipo $\mathbb{1}_e$, poiché sarebbe necessario creare un costrutto ad-hoc per mantenere l'informazione relativa a quale elemento sia stato conservato. Essendo tra gli obiettivi dell'estrattore il mantenimento della tipabilità in mTT dei termini estratti, si accetta la possibile presenza di alcuni $\star_e$ nei termini estratti, anche considerando che vi sono altri casi in cui non si può fare a meno di mantenerli.

In alternativa, si potrebbe ricavare l'informazione di quale elemento della coppia sia stato preservato a partire dal tipo del termine non semplificato ma, per come è stata strutturata la fase di collasso, essa prende in input solamente la versione già semplificata del termine.

In conseguenza alla regola sul tipo, quella sul termine coppia $\langle b, c \rangle_\Sigma$ diventa:

$$\mathbf{1 \to 1 \to 1 \times 1 \text{ - } I)}\ \frac{\langle b, c\rangle_\Sigma \in \Sigma x \in B.\, C(x) \quad B =_c \mathbb{1}_e \quad C(x) =_c \mathbb{1}_e}{\langle b, c\rangle_\Sigma \mapsto \star_e}$$

Di seguito, descrivo il collasso per $\pi_1(p)$ e $\pi_2(p)$, rispettivamente la prima e seconda proiezione della coppia. Questi due operatori non sono presenti nella definizione della logica data da Maietti [Mai08], ma sono stati implementati come costrutto *core* a causa di alcune idiosincrasie della fase di compilazione dal livello estensionale a quello intensionale. Le proiezioni possono essere considerate casi particolari della normale eliminazione della coppia definita da Maietti. Visto che i due costrutti sono pressoché identici, elencherò le regole solamente per $\pi_1(p)$:

$$\mathbf{1 \to 1 \text{ - } E_1)}\ \frac{p \in P \quad P =_c \mathbb{1}_e}{\pi_1(p) \mapsto \star_e}$$

$$\mathbf{1 \times 1 \to 1 \text{ - } E_1)}\ \frac{p \in \Sigma x \in B.\, C(x) \quad B =_c \mathbb{1}_e \quad C(x) =_c \mathbb{1}_e}{\pi_1(p) \mapsto \star_e}$$

$$\mathbf{A \times 1 \to A \text{ - } E_1)}\ \frac{\langle b, \star_e\rangle_\Sigma \in \Sigma x \in B.\, \mathbb{1}_e}{\pi_1(\langle b, \star_e\rangle_\Sigma) \mapsto b}$$

Si noti che la seconda regola è ridondante e non sarà mai applicata, poiché se p ha già subito l'estrazione e, se il suo corpo fosse stato composto da una coppia di stelle, sarebbe già stato collassato e ricadrebbe nel caso trattato dalla prima regola.

La terza regola, invece, costituisce un caso particolare: è un'ottimizzazione che viene effettuata solamente quando viene costruita una coppia con un elemento inutile, la quale viene immediatamente eliminata in direzione dell'elemento utile. Un'operazione del genere non verrà probabilmente mai scritta da un utente umano, ma questo caso potrebbe effettivamente apparire dopo alcune fasi di collasso che eliminino il codice compreso tra l'introduzione e l'eliminazione.

Per come è stato costruito l'estrattore, è possibile separare completamente le regole di collasso dal resto della logica del programma, rendendole completamente estendibili. Per esempio, un eventuale futuro manutentore

dell'estrattore potrebbe creare un nuovo file in cui inserire ulteriori ottimizzazioni, ordinate per priorità, consentendo miglioramenti arbitrari della fase di estrazione. In particolare, la fase di compilazione genera automaticamente alberi di codice molto complicati, per i quali potrebbero essere scritte ottimizzazioni ad-hoc che ne riducano anche di molto la dimensione.

La generica eliminazione si scrive $Elim_\Sigma(p, m)$ con p coppia ed m termine dipendente dai due elementi di p, che produce l'output dell'eliminazione. L'eliminazione è soggetta alle seguenti regole di collasso:

$$\mathbf{1 \to A\text{ - }E)}\ \frac{p \in P \quad m \in M(p) \quad P =_c \mathbb{1}_e}{Elim_\Sigma(p, m) \mapsto m(\star_e, \star_e)}$$

$$\mathbf{1 \times 1 \to A\text{ - }E)}\ \frac{\begin{array}{c} p \in \Sigma x \in B.\ C(x) \quad m \in M(p) \\ B =_c \mathbb{1}_e \quad C(x) =_c \mathbb{1}_e \end{array}}{Elim_\Sigma(p, m) \mapsto m(\star_e, \star_e)}$$

$$\mathbf{A \times B \to 1\text{ - }E)}\ \frac{p \in P \quad m \in M(p) \quad M(p) =_c \mathbb{1}_e}{Elim_\Sigma(p, m) \mapsto \star_e}$$

Le due regole seguenti operano invece un collasso parziale, trasformando l'eliminazione della coppia in un'astrazione che prende in input solamente l'elemento della coppia contenente informazione e passando l'altro come costante (*currying*):

$$\mathbf{A \times 1 \to B\text{ - }E)}\ \frac{\langle b, \star_e \rangle_\Sigma \in \Sigma x \in B.\ \mathbb{1}_e \quad m \in M(\langle b, \star_e \rangle_\Sigma)}{Elim_\Sigma(\langle b, \star_e \rangle_\Sigma, m) \mapsto App_\Pi(\lambda x \in B.\ m(x, \star_e), b)}$$

$$\mathbf{1 \times A \to B\text{ - }E)}\ \frac{\langle \star_e, c \rangle_\Sigma \in \Sigma x \in \mathbb{1}_e.\ C(x) \quad m \in M(\langle \star_e, c \rangle_\Sigma)}{Elim_\Sigma(\langle \star_e, c \rangle_\Sigma, m) \mapsto App_\Pi(\lambda x \in C(\star_e).\ m(\star_e, x), c)}$$

Somma disgiunta (`setSum`)

Il tipo della somma disgiunta, indicato con $B + C$, e i suoi elementi sinistri e destri, denotati rispettivamente come $inl(b \in B)$ e $inr(c \in C)$, non possono mai essere collassati poiché l'informazione della loro provenienza è sempre codice utile. Infatti, l'eliminazione potrebbe eseguire codice differente a seconda che un elemento provenga dall'insieme destro o da quello

sinistro, indipendentemente dal suo valore. L'eliminazione viene indicata con $Elim_+(w, al, ar)$, dove w è un elemento della somma, al è il termine dipendente che opera l'eliminazione sinistra, mentre ar opera l'eliminazione destra. Come conseguenza della struttura della somma, l'eliminazione può essere collassata in un unico caso:

$$\textbf{A + B} \to \textbf{1 - E)} \ \frac{\begin{array}{c} w \in B + C \quad al \in A(b \in B) \quad ar \in A(c \in C) \\ A(b) =_c \mathbb{1}_e \quad A(c) =_c \mathbb{1}_e \end{array}}{Elim_+(w, al, ar) \mapsto \star_e}$$

Esistono però regole di ottimizzazione nel caso in cui si conosca la provenienza di w. Di seguito, trascrivo le regole per il caso $w = inl(b)$; si tenga presente l'esistenza di regole equivalenti per $w = inr(c)$:

$$\textbf{1 + A} \to \textbf{B - E)} \ \frac{\begin{array}{c} inl(b) \in B + C \quad al \in A(b \in B) \quad ar \in A(c \in C) \\ B =_c \mathbb{1}_e \end{array}}{Elim_+(inl(b), al, ar) \mapsto al(\star_e)}$$

$$\textbf{A + B} \to \textbf{1 - E)} \ \frac{\begin{array}{c} inl(b) \in B + C \quad al \in A(b \in B) \quad ar \in A(c \in C) \\ A(b) =_c \mathbb{1}_e \end{array}}{Elim_+(inl(b), al, ar) \mapsto \star_e}$$

$$\textbf{A + B} \to \textbf{C - E)} \ \frac{inl(b) \in B + C \quad al \in A(b \in B) \quad ar \in A(c \in C)}{Elim_+(inl(b), al, ar) \mapsto App_\Pi(\lambda x.\ al(x), b)}$$

Lista (`list`)

Oltre alla gestione in fase di estrazione, per questo costrutto ho scritto anche l'implementazione del type checking la quale non era ancora presente nel progetto. Il costrutto lista, come la somma disgiunta, non può mai essere collassato poiché, anche se gli elementi della lista sono codice inutile, la sua dimensione è comunque un'informazione che può essere utilizzata nella computazione del codice utile. Perciò, esiste un unica regola di collasso:

$$\textbf{A} \to \textbf{1 - E)} \ \frac{list \in List(C) \quad a \in L(\varepsilon) \quad l \in L(list) \quad L(list) =_c \mathbb{1}_e}{Elim_{List}(list, a, l) \mapsto \star_e}$$

Singoletto (`singleton`)

L'eliminazione del singoletto potrebbe essere soggetta a collasso tramite la regola seguente:

$$\mathbf{A \to 1 \text{ - } E)}\ \frac{c \in C(\star) \quad C(\star) =_c \mathbb{1}_e}{Elim_{\mathbb{1}}(\star, c) \mapsto \star_e}$$

Identità proposizionale (`propId`)

Passiamo ora a considerare i costrutti di kind proposizione, per i quali le regole di collasso vengono sempre applicate poiché il contenuto informativo delle proposizioni è sempre nullo. Iniziamo dalle regole dell'identità, notando *en passant* che questo è l'unico costrutto a fare realmente uso dell'infrastruttura dei tipi dipendenti:

$$\mathbf{A \to 1 \text{ - } F)}\ \frac{}{ID(A, a, a) \mapsto \mathbb{1}_e}$$

$$\mathbf{A \to 1 \text{ - } I)}\ \frac{}{id(a) \mapsto \star_e}$$

$$\mathbf{A \to 1 \text{ - } E)}\ \frac{c \in C(\star_e, \star_e) \quad C(\star_e, \star_e) =_c \mathbb{1}_e}{Elim_{ID}(\star_e, c) \mapsto \star_e}$$

$$\mathbf{A \to 1 \text{ - } E)}\ \frac{c \in C(a, a) \quad C(a, a) =_c \mathbb{1}_e}{Elim_{ID}(id(a), c) \mapsto \star_e}$$

Altre proposizioni (`and, or, implies, forall, exist`)

I rimanenti costrutti hanno un comportamento estremamente uniforme, per cui mi limiterò a descrivere le regole di collasso per uno di essi: le regole per gli altri potranno essere facilmente inferite. Tra i costrutti proposizionali l'unico ad avere un comportamento leggermente diverso dagli altri è il quantificatore esistenziale.

Infatti, pur volendo sempre collassare il costrutto, si potrebbe avere un testimone contenente codice significativo in grado di bloccare la riduzione. Per questa ragione, se il primo elemento dell'esistenziale è un termine informativo, esso viene scorporato come descritto nella sottosezione precedente e

viene sostituito da $\star_e$. Per gli altri costrutti proposizione, eventuali termini informativi sono semplicemente sostituiti con $\star_e$, come accade per l'identità.

Vediamo dunque le regole per il collasso del quantificatore esistenziale:

$$\mathbf{1 \to 1 \to 1 \times 1 \text{ - F)}}\ \frac{B =_c \mathbb{1}_e \quad C(x) =_c \mathbb{1}_e}{\exists x \in B.\, C(x) \mapsto \mathbb{1}_e}$$

$$\mathbf{1 \to 1 \to 1 \times 1 \text{ - I)}}\ \frac{\langle b, c\rangle_\exists \in \exists x \in B.\, C(x) \quad B =_c \mathbb{1}_e \quad C(x) =_c \mathbb{1}_e}{\langle b, c\rangle_\exists \mapsto \star_e}$$

$$\mathbf{1 \to 1 \text{ - E)}}\ \frac{e \in E \quad m \in M \quad E =_c \mathbb{1}_e \quad M =_c \mathbb{1}_e}{Elim_\exists(e, m) \mapsto \star_e}$$

$$\mathbf{1 \times 1 \to 1 \text{ - E)}}\ \frac{\begin{array}{c} e \in \exists x \in B.\, C(x) \quad m \in M \\ B =_c \mathbb{1}_e \quad C(x) =_c \mathbb{1}_e \quad M =_c \mathbb{1}_e \end{array}}{Elim_\exists(e, m) \mapsto \star_e}$$

Si noti come sia implicito nel funzionamento dell'estrattore che queste regole vengano sempre applicate. Infatti, se il procedimento di eliminazione ha operato correttamente su tutti i sottotermini del costrutto, allora esso, al momento del collasso, conterrà solamente $\star_e$. Questa struttura delle regole di collasso si ripete invariata anche per la congiunzione, la disgiunzione, l'implicazione e il quantificatore universale.

Estrazione estensionale e uguaglianza (`propEq`)

Poiché il passaggio di compilazione dal livello estensionale a quello intensionale ha il compito di ricreare alcune prove mancanti, supponendo che queste prove non contengano codice informativo si potrebbe fare estrazione di codice direttamente dal livello estensionale. Questo metodo avrebbe il vantaggio di non dover attendere il completamento della fase di compilazione. Inoltre, se fosse nota a priori l'assenza di informazione all'interno dell prove generate si potrebbero evitare all'estrattore calcoli inutili.

Ovviamente per verificare la correttezza della dimostrazione effettuata sarebbe necessario compilarla ma, una volta eseguito questo passaggio, is potrebbe distribuire la dimostrazione al livello estensionale e lasciare che siano eventuali utenti terzi ad estrarla in un linguaggio target.

Per permettere di verificare questa speculazione quando il codice del compilatore sarà più maturo, ho parametrizzato l'estrazione sul livello considerato. Infatti, la sintassi dei costrutti che abbiamo esaminato è la stessa ad entrambi i livelli e non è necessario ulteriore lavoro per estrarre codice dalla loro versione estensionale. L'unica eccezione è l'identità, la quale non è presente al livello estensionale, poiché viene sostituita dall'uguaglianza.

Ho dunque implementato l'estrazione anche per l'uguaglianza, il quale generando una proposizione deve in ogni caso essere collassato, e mi è stato possibile estrarre codice da alcuni piccoli segmenti di mTT estensionale.

2.2.4 Collasso delle definizioni di libreria

I termini e i tipi di mTT di cui ho predecentemente descritto il collasso vanno a inserirsi all'interno di una più ampia struttura che permette di dichiarare chiamate di libreria, che consentono di definire nuovi tipi e termini. Le definizione possono essere quantificate universalmente e possono contenere ipotesi aggiuntive e meccanismi di conversione descritti direttamente in λProlog. Anche le chiamate di libreria vengono manipolate durante la fase di eliminazione, poiché il tipo e il numero dei loro argomenti potrebbe variare.

Schemi di tipo (`univPi`, `univDepPi`, `univPiT`, `univDepPiT`)

La libreria consente di astrarre sia su tipi e termini che sulle loro controparti dipendenti, ma l'astrazione operata implementa uno schema al contrario di ciò che accade in di Coq dove si ricorre alla quantificazione universale. La verifica che le proprietà richieste valgano sulla variabile quantificata viene effettuata solamente a posteriori, quando un qualche oggetto viene passato come un parametro.

Per prima cosa, vediamo in quali casi è possibile collassare l'astrazione sui tipi:

$$\mathbf{1 \to A\text{ - }I)}\ \frac{}{\forall X : prop_s.\ F(X) \mapsto F(\mathbb{1}_e)}$$

$$\textbf{A} \to \textbf{B - I)}\ \frac{X \notin F}{\forall X : K.\ F \mapsto F}$$

Ovviamente la seconda regola sarebbe raramente usata in una definizione scritta da un utente, ma la fase di eliminazione potrebbe effettivamente cancellare tutte le occorrenze di X da F, rendendo dunque inutile la parametrizzazione. Non viene fornito l'equivalente della prima regola per le *prop*, poiché la quantificazione avviene su un preciso kind, non a meno di conversione, e l'implementazione corrente della libreria non consente di astrarre sulle proposizioni.

Esiste un ulteriore dettaglio da tenere in considerazione: per come sono state formalizzate le definizioni di libreria, X compare sempre in F. Infatti, la testa di una definizione contiene il nome della chiamata e tutti i parametri ordinati per cui, anche se X non dovesse occorrere mai nel corpo della definizione, comparirà sempre nella testa. Per ovviare a questo inconveniente, è sufficiente operare la medesima riduzione anche sulla testa della definizione, cancellando gli argomenti che non compaiono nel corpo.

Tralascio di elencare le regole di riduzione per le astrazioni sui tipi dipendenti poiché sono praticamente identiche a quelle già illustrate per i tipi. Lo stesso può essere detto dei termini dipendenti, per cui descriverò di seguito solamente le regole di collasso per i termini:

$$\textbf{1} \to \textbf{A - I)}\ \frac{T =_c \mathbb{1}_e}{\forall x \in T.\ F(x) \mapsto F(\star_e)}$$

$$\textbf{A} \to \textbf{B - I)}\ \frac{x \notin F}{\forall x \in T.\ F \mapsto F}$$

Ipotesi e Conversione (`hyp`, `cut`)

Il costrutto di ipotesi locale viene utilizzato per estendere il predicato di conversione. Se, per esempio, si vuole che l'applicazione di un certo F a zero sia convertibile con zero si aggiunge un'ipotesi ($F\ 0 =_c 0$). Nell'istanza specifica si verifica poi che questa proprietà valga. L'introduzione di queste ipotesi rompe però la decidibilità del processo di conversione. È quindi necessario introdurre un `cut` che viene utilizzato per dimostrare la conversione tra due

termini sulla base di una catena di ipotesi. Il *type checker* usa poi i lemmi definiti da questi due costrutti per effettuare correttamente il tipaggio.

Inizialmente, il contenuto di questi costrutti veniva scartato. In seguito, mi sono accorto che non solo è necessario mantenere queste ipotesi ma anche generare la loro controparte estratta, nella quale le variabili e le chiamate di libreria sono state mappate nel loro estratto, per preservare il tipaggio. Per fare ciò, è sufficiente andare in ricorsione strutturale sul sottoinsieme di costrutti di ELPI e di predicati del type checker utilizzato in queste proposizioni ed effettuare la mappa alla stessa maniera dei termini.

Al momento, i predicati utilizzati all'interno di queste ipotesi aggiuntive sono di numero molto ridotto e vengono principalmente utilizzati per specificare uguaglianze a meno di conversione. Nel caso in cui la variabilità delle ipotesi dovesse aumentare o venisse concesso agli utenti di utilizzare predicati arbitrari al loro interno, sarebbe necessario riflettere su come si possa effettuarne la mappa correttamente.

Definizioni (`locDefL, locTypeDefL`)

L'eliminazione delle definizioni si occupa di ridurne il corpo, costituito da un tipo o da un tipo e un termine, utilizzando le regole illustrate nella sottosezione precedente. Inoltre, viene effettuata la cancellazione di tutti quegli argomenti della testa, i quali siano stati trasformati in $\star_e$ e $\mathbb{1}_e$ o che non siano presenti nel corpo.

Un dettaglio problematico è che anche le ipotesi fanno parte del "corpo" che è necessario controllare per determinare la possibilità della cancellazione di un parametro. Si deve quindi trasportare le ipotesi fino alla fase di collasso della definizione per poterle prendere in considerazione. Inoltre, le variabili che compaiono solamente nelle ipotesi non sono di alcuna utilità ai linguaggi target e dovranno quindi essere eliminate in fase di traduzione.

2.2.5 Differenze di eliminazione tra OCaml e Haskell

Idealmente, si vorrebbe applicare la fase di eliminazione indipendentemente dal linguaggio target, ottenendo termini semplificati tipabili in mTT a partire dai quali sia possibile effettuare la traduzione.

Purtroppo, come descritto nella sezione di eliminazione di codice inutile del primo capitolo, le differenze presenti tra la strategia di valutazione *lazy* di Haskell e quella *eager* di OCaml potrebbero causare una variazione nella complessità computazionale del tradotto. Per evitare che ciò avvenga, è necessario introdurre delle nuove regole di collasso per OCaml, le quali impediscano di eliminare quei costrutti che prevengono l'esecuzione di alcuni blocchi di codice.

In particolare, l'unico caso di interesse è quello in cui ad una computazione richiedente dei parametri venga cancellata un'astrazione. Se il corpo viene passato come argomento ad un'astrazione, esso viene calcolato prima di sapere se effettivamente sia o meno codice morto:

$$(\lambda x.\ 3)\ (\lambda y.\ loop) \quad \longrightarrow_{extr} \quad (\lambda x.\ 3)\ loop$$

Si aggiungono dunque le seguenti regole di collasso per l'astrazione, bandando a inserirle prima delle loro controparti più generali che effettuano, invece, la riduzione:

$$\mathbf{1 \to A\text{ - }F)}\ \frac{B =_c \mathbb{1}_e \quad target = OCaml}{\Pi x \in B.\ C(x) \mapsto \Pi x.\ C(x)}$$

$$\mathbf{1 \to A\text{ - }I)}\ \frac{\lambda x.\ c(x) \in \Pi x \in B.\ C(x) \quad B =_c \mathbb{1}_e \quad target = OCaml}{\lambda x.\ c(x) \mapsto \lambda x.\ c(x)}$$

$$\mathbf{1 \to A\text{ - }E)}\ \frac{b \in B \quad B =_c \mathbb{1}_e \quad target = OCaml}{App_\Pi(f, b) \mapsto App_\Pi(f, b)}$$

Queste regole per l'applicazione non sono però sufficienti, poiché lo stesso fenomeno si manifesta anche nel caso del collasso delle eliminazioni costanti. È dunque necessario aggiungere anche le seguenti regole:

$$\mathbf{1 \to A\text{ - }E)}\ \frac{p \in P \quad P =_c \mathbb{1}_e \quad target = OCaml}{Elim_\Sigma(p, m) \mapsto Elim_\Sigma(\langle \star_e, \star_e \rangle_\Sigma, m)}$$

$$\textbf{1} \times \textbf{1} \to \textbf{A - E)} \ \frac{\begin{array}{c} p \in \Sigma x \in B.\ C(x) \\ B =_c \mathbb{1}_e \quad C(x) =_c \mathbb{1}_e \quad target = OCaml \end{array}}{Elim_\Sigma(p, m) \mapsto Elim_\Sigma(p, m)}$$

$$\textbf{1} + \textbf{A} \to \textbf{B - E)} \ \frac{inl(b) \in B + C \quad B =_c \mathbb{1}_e \quad target = OCaml}{Elim_+(inl(b), al, ar) \mapsto Elim_+(inl(b), al, ar)}$$

Si tenga presente che della terza regola esiste anche la versione simmetrica per gli elementi dell'insieme destro. Inoltre, si noti che la prima regola trasforma una stella in una coppia di stelle per ripristinare il corretto tipaggio dell'espressione. Un'alternativa sarebbe non collassare mai la coppia di stelle, cosa che potrebbe portare ad avere un estratto contenente una maggiore quantità di codice inutile.

Volendo utilizzare un altro metodo per risolvere il medesimo problema si potrebbe pensare di inibire, invece che sovrascrivere, le regole di collasso che potrebbero causare la variazione. In questo caso, il set di regole di OCaml diventerebbe più piccolo di quello di Haskell. Si noti che sarebbe però comunque necessario aggiungere la prima regola dell'eliminazione della coppia alla lista di regole OCaml, al fine di ottenere un termine di output ben tipato.

Considerato che il problema della variazione del costo computazionale si presenta solamente in casi specifici e non ad ogni occorrenza delle espressioni sopra descritte, si potrebbero determinare dei criteri più stringenti per l'applicazione delle summenzionate regole. Si faccia attenzione che un raffinamento delle ipotesi potrebbe essere applicato senza problemi alle eliminazioni, ma non all'astrazione.

Infatti, supponendo di decidere se collassare o meno l'applicazione sulla base del contenuto del suo corpo, non sarebbe possibile operare il medesimo procedimento sul suo tipo, il quale non contiene questa informazione. La diversità di riduzione tra un termine ed il suo tipo diventa un problema nel caso in in cui l'astrazione sia passata come argomento di un altro costrutto: si otterrebbe infatti un incongruenza tra il tipo dell'argomento e quello specificato nel costrutto.

Per esempio, supponendo di inserire una regola per collassare applicazioni con parametro inutile in OCaml nel caso in cui il corpo sia costante, si otterrebbe la seguente estrazione che produce un termine mal tipato:

$$(\lambda x \in (\Pi y \in \mathbb{1}_e.\ \mathbb{N}).\ x)\ (\lambda z \in \mathbb{1}_e.\ 3)$$
$$\downarrow_{extr}$$
$$(\lambda x \in (\Pi y \in \mathbb{1}_e.\ \mathbb{N}).\ x)\ 3$$

2.3 Traduzione

Una volta completata l'eliminazione dei termini, rimane solamente da tradurli nel linguaggio target. Al momento, l'estrattore supporta la traduzione verso OCaml e Haskell: si vuole dunque costruire una mappa da un termine ben tipato di mTT estratto ad una stringa di codice del linguaggio target.

2.3.1 Requisiti della traduzione

Per poter effettivamente cominciare ad elaborare una fase di traduzione è prima di tutto necessario determinare quali costrutti devono essere tradotti nei linguaggi target e come il sistema di tipi di mTT interagisca con quelli di output.

Al termine della fase di eliminazione, rimangono nei termini estratti solamente quei costrutti che potrebbero contenere codice informativo. In particolare, tutti i costrutti proposizione sono stati cancellati: rimangono dunque da tradurre solamente l'infrastruttura di libreria, le astrazioni, le coppie, le somme disgiunte, le liste, i singoletti e i *dummy* dell'estrazione che non è stato possibile eliminare.

Tipi dipendenti da termini

Pur essendo mTT dotato di tipi dipendenti, come abbiamo visto, essi compaiono effettivamente solo nell'identità proposizionale, la quale è stata completamente eliminata dall'estrazione. I tipi rimasti sono quindi tutti già non dipedenti. Inoltre, poiché in mTT non vi è quantificazione sui tipi (alla

System-F), tutte le costanti sono tipabili nel sistema di tipi di Hindley-Milner. Pertanto, il codice OCaml/Haskell non necessita di `Obj.magic`/`unsafeCoerce` come nel caso di Coq.

Poiché è impossibile per un termine comparire in un tipo, si può semplificare ulteriormente anche le definizioni di tipo nella libreria, eliminando tutti i termini passati come argomento. Per fare ciò, è necessario avere a disposizione un contesto contenente le informazioni di tipaggio, come quello utilizzato nelle fasi precedenti. Si noti che in assenza della rimozione degli argomenti termine dai tipi e degli argomenti tipo dai termini, la fase di traduzione potrebbe essere operata senza costruire il contesto.

Termini parametrizzati su tipi

Nelle definizioni di libreria oltre a tipi parametrizzati su termini possono comparire anche termini che prendano in input dei tipi. Poiché sia Haskell che OCaml supportano il polimorfismo e la *type inference* è possibile ignorare completamente tutte le informazioni di tipo contenute nei termini e lasciare che sia il compilatore a ricalcolare i tipi. Inoltre, poiché le definizioni contengono esplicitamente il tipo che dovrebbero avere, è possibile tradurlo come *type signature* e lasciare che sia il compilatore del linguaggio target a verificare che il tipo fornito corrisponda a quello inferito. Di conseguenza, non c'è bisogno di mantenere i parametri di tipo nella testa delle definizioni dei termini ed è possibile cancellarli. Si noti che effettuare questa operazione prima della fase di traduzione porterebbe alla produzione di definizioni di libreria contenenti varibili libere.

2.3.2 Codifica dei costrutti

Considerati tutti i requisiti illustrati nella sottosezione precendete, passiamo ora ad esaminare la codifica vera e propria dei costrutti di mTT nei linguaggi target.

Codifica OCaml

La traduzione avviene in maniera naturale utlizzando i corrispettivi dei costrutti mTT in OCaml. In particolare, si rappresenta l'astrazione con una funzione unaria polimorfa di OCaml; la coppia con un record dotato di due campi; la somma disgiunta con un ADT (*algebraic data type*) dotato di un costruttore per l'insieme destro e uno per l'insieme sinistro; la lista e il singoletto con l'equivalente costrutto primitivo di OCaml; le definizioni di libreria di tipi e termini rispettivamente con alias di tipo e definizioni di funzione.

$\Pi x \in A.\, B(x) \to_{tr}$ `'a -> 'b`

$\Sigma x \in A.\, B(x) \to_{tr}$ `type ('a, 'b) set_sigma = { l: 'a; r: 'b };;`

$A + B \to_{tr}$ `type ('a, 'b) set_sum = Inl of 'a | Inr of 'b;;`

$List(A) \to_{tr}$ `'a list`

$singleton \to_{tr}$ `unit`

$\forall X.\, f(X) = B \to_{tr}$ `type 'x f = B;;`

$\forall x.\, f(x) = B \to_{tr}$ `let f x = B;;`

Inoltre, per le definizioni di nuovi termini vengono anche estratte le *type signature* corrispondenti:

$\forall x.\, f(x) \in T \to_{tr}$ `val f : T`

Codifica Haskell

La codifica in Haskell non ha bisogno di particolare commento, poiché segue i medesimi principi di quella effettuata per OCaml:

$\Pi x \in A.\, B(x) \to_{tr}$ `a -> b`

$\Sigma x \in A.\, B(x) \to_{tr}$ `data SetSigma a b = SetSigma { l :: a, r :: b }`

$A + B \to_{tr}$ `data SetSum a b = Inl a | Inr b`

$List(A) \to_{tr}$ `[a]`

$singleton \to_{tr}$ `()`

$\forall X.\, f(X) = B \to_{tr}$ `type F x = B`

$\forall x.\, f(x) = B \to_{tr}$ `f x = B`

$\forall x.\, f(x) \in T \to_{tr}$ `f :: T`

Dettagli implementativi

Come è possibile notare dalla codifica, in OCaml gli argomenti delle dichiarazioni di tipo sono in posizione prefissa; inoltre, in caso ve ne sia più di uno si utilizza una notazione dotata di parentesi e inframezzata da virgole, come nell'esempio seguente:

```
type ('a, 'b, 'c) g = ...;;
```

Per ragioni di semplicità e uniformità con il resto della traduzione, le definizioni vengono tradotte inizialmente con gli argomenti postfissi. In seguito, la stringa ottenuta viene manipolata tramite espressioni regolari per spostare gli argomenti in posizione prefissa. Nel caso di Haskell, invece, i parametri rimangono in posizione postfissa, ma le definizioni dei tipi devono essere modificate in modo da avere la prima lettera del nome maiuscola.

Un ulteriore dettaglio del quale è necessario tenere conto nella fase di traduzione è che ogni termine dipendente potrebbe essere costituito da una singola variabile, necessitando dunque di una traduzione più semplice. Per risolvere il problema, è necessario controllare la forma di tutti i termini dipendenti, traducendo solamente quelli in forma standard con funzioni anonime.

2.4 Un esempio di estrazione

Terminata la descrizione di tutte le fasi, pare opportuno mostrare l'input e l'output dell'estrazione per una singola chiamata della libreria al fine di esemplificare i concetti precedentemente esposti. Si prenda la seguente definizione contenuta all'interno della libreria del singoletto:

```
(univPi Bi \
 (univPiT (lib_app setoid Bi) bS\
  (locDefL (lib_app (lib_app singleton_setoidDep Bi) bS)
   (lib_app(lib_app(lib_app setoidDep Bi) (x\ singleton)) bS)
   (pair (setSigma (setPi Bi _\ lib_app setoid singleton) _\
    (setPi Bi x \ setPi Bi x' \ setPi (decode (app (app (app
```

```
(lib_app setoid_rel Bi) bS) x) x')) _ \
setPi singleton _ \ singleton))
(cc\ (setPi Bi x1\ setPi Bi x2\
setPi singleton y1\ setPi singleton y2\
setPi (decode (app (app (app (lib_app setoid_rel Bi) bS)
x1) x2)) d1\ setPi (decode (app (app (app
(lib_app setoid_rel singleton) (app (elim_setSigma cc
(_\ (setPi Bi _\ lib_app setoid singleton))
(x\y\x)) x1)) y1) y2)) d2\ (decode (app (app (app
(lib_app setoid_rel singleton) (app (elim_setSigma cc
(_\ (setPi Bi _\ lib_app setoid singleton)) (x\y\x)) x2))
(app (app (app (app (elim_setSigma cc
(_\ (setPi Bi x \ setPi Bi x' \ setPi
(decode (app (app (app (lib_app setoid_rel Bi)
bS) x) x')) _\ setPi singleton _\ singleton))
(x\y\y)) x1) x2) d1) y1))
(app (app (app (app (elim_setSigma cc
(_\ (setPi Bi x \ setPi Bi x' \ setPi (decode
(app (app (app (lib_app setoid_rel Bi) bS) x) x'))
_ \ setPi singleton _ \ singleton))
(x\y\y)) x1) x2) d1) y2)))))
(pair (setPi Bi _\ lib_app setoid singleton)
(_\ (setPi Bi x \ setPi Bi x'\ setPi (decode
(app (app (app (lib_app setoid_rel Bi) bS) x) x'))
_ \ setPi singleton _ \ singleton))
(lambda Bi x \ singleton_setoid)
(lambda Bi x \ lambda Bi x' \ lambda (decode (app (app (app
(lib_app setoid_rel Bi) bS) x) x'))
_\ lambda singleton y \ y))
(lambda Bi x1\ lambda Bi x2\ lambda singleton
y1\ lambda singleton y2\ lambda (decode (app (app (app
```

```
    (lib_app setoid_rel Bi) bS) x1) x2)) d1\ lambda
    (decode (app (app (app (lib_app setoid_rel singleton)
    (app (lambda Bi x \ singleton_setoid) x1)) y1) y2))
    d2\ d2)))))
```

della quale non si pretende dal lettore che ne interpreti la semantica, ma solo che faccia caso alla sua dimensione. Infatti, al momento, la compilazione dal livello estensionale a quello intensionale causa un aumento esponenziale nella dimensione dei termini.

Una volta eliminato il codice inutile, nel caso in cui il linguaggio target sia *lazy* (Haskell), il termine assume la forma seguente:

```
univPi x0 \
 locDefL (lib_app extr_singleton_setoidDep x0)
  (lib_app (lib_app extr_setoidDep x0) x1 \ singleton)
  (pair (setSigma extractor_singleton x1 \ setPi x0
   x2 \ setPi x0 x3 \ setPi singleton x4 \ singleton)
  (x1 \ extractor_singleton)
  (pair extractor_singleton (x1 \ setPi x0 x2 \ setPi x0
  x3 \ setPi singleton x4 \ singleton)
  extractor_star (lambda x0 x1 \ lambda x0 x2 \
  lambda singleton x3 \ x3)) extractor_star)
```

Per un linguaggio *eager*, come OCaml, la versione semplificata del termine è invece:

```
univPi x0 \
 locDefL (lib_app extr_singleton_setoidDep x0)
  (lib_app (lib_app extr_setoidDep x0) x1 \ singleton)
  (pair (setSigma extractor_singleton x1 \ setPi x0
  x2 \ setPi x0 x3 \ setPi extractor_singleton
  x4 \ setPi singleton x5 \ singleton)
  (x1 \ extractor_singleton)
  (pair extractor_singleton (x1 \ setPi x0 x2 \ setPi x0
```

```
  x3 \ setPi extractor_singleton x4 \ setPi singleton
  x5 \ singleton)
  extractor_star (lambda x0 x1 \ lambda x0 x2 \
  lambda extractor_singleton x3 \ lambda singleton
  x4 \ x4)) extractor_star)
```

nella quale è possibile notare una astrazione aggiuntiva su $\mathbb{1}_e$, che non è stata rimossa per preservare la complessità computazionale.

Infine, di seguito sono trascritte le traduzioni rispettivamente in Haskell e OCaml del termine estratto con relativa *type signature*.

Haskell:

```
extr_singleton_setoidDep :: Extr_setoidDep x0 ()
extr_singleton_setoidDep = (SetSigma (SetSigma ()
 (\x1 -> (\x2 -> (\x3 -> x3))))
 ())
```

OCaml:

```
val extr_singleton_setoidDep : ('x0, unit) extr_setoidDep
let extr_singleton_setoidDep = { l = { l = ();
 r = (fun x1 -> (fun x2 -> (fun x3 -> (fun x4 -> x4)))) };
 r = () };;
```

Conclusioni

Nel primo capitolo di questa tesi, ho esposto i principali risultati presenti in letteratura nell'ambito dell'estrazione di codice inutile e ho descritto l'implementazione di queste tecniche nell'estrattore di codice del dimostratore interattivo Coq. Inoltre, ho presentato la *minimalist two-level foundation* di Maietti, la cui peculiarità consiste nell'essere una formulazione a due livelli nella quale un livello estrensionale viene compilato in uno intensionale. La formulazione intensionale è il punto di partenza per l'estrazione di codice da me implementata.

Infine, ho descritto le caratteristiche principali di λProlog, il linguaggio utilizzato per l'implementazione dell'estrattore, il cui interprete ELPI è stato realizzato presso l'Università di Bologna. ELPI è uno strumento di relativamente recente sviluppo ed, essendo io stato uno dei primi utenti esterni ad aver fatto un uso estensivo della maggior parte delle sue funzionalità, posso dire di averne avuto una buona impressione complessiva: è uno strumento molto versatile e dalla grande potenza espressiva, unico neo l'assenza di infrastrutture di debugging sofisticate che forzano l'utilizzo di stampe dello stato, dalle quali può essere a volte problematico estrarre informazioni significative.

Nel secondo capitolo, ho trattato nel dettaglio la struttura dell'estrattore che ho implementato per mTT, il quale si divide in tre fasi: la starificazione che opera una prima cancellazione di termini foglia inutili, l'eliminazione che collassa le strutture non informative e la traduzione che trasporta i termini mTT semplificati nei linguaggi target, Haskell e OCaml. Per effettuare correttamente l'estrazione è stato necessario modificare parzialmente la sintassi

esistente del livello intensionale per consentire la ricorsione strutturale sulle chiamate di libreria e sulle applicazioni di variabili.

Lo sforzo complessivo per questa tesi è stato costituito da circa 5000 righe di codice λProlog scritte durante un periodo di 8 mesi per un totale stimato di 600 ore di lavoro.

2.5 Confronto con Coq

Pur avendo assunto a modello l'estrattore di Coq, dovrebbe risultare evidente dalla trattazione precedente che il problema dell'estrazione di codice dipenda largamente dalla formulazione della logica che si sta cercando di estrarre. Infatti, sulla base delle proprietà della logica le strategie di manipolazione applicate, nonché le possibili codifiche nei linguaggi target, possono differire anche radicalmente. Ritengo dunque opportuno ricapitolare ed esplicitare le principali differenze tra l'estrattore per mTT e quello per Coq:

1. la forte divisione tra termini e tipi in mTT consente di estrarre anche i secondi senza particolari difficoltà, mentre in Coq i tipi vengono sempre cancellati;

2. una conseguenza ulteriore del punto precedente è che in mTT possiamo estrarre mantenendo i termini ben tipati durante tutte le fasi del procedimento, mentre i termini intermedi di Coq non sono tipabili;

3. il problema della possibile variazione della complessità computazionale a seguito dell'eliminazione in linguaggi *eager* come OCaml, compare in maniera uguale in mTT e Coq, obbligando a mantenere una parte del codice inutile quando si estrae verso di essi;

4. il limitato utilizzo dei tipi dipendenti in mTT, i quali compaiono solamente nelle proposizioni, permette di avere un codice tradotto ben tipabile dal sistema di tipi di OCaml e Haskell, mentre in Coq era necessario utilizzare `Obj.magic` e `unsafeCoerce` in punti specifici per aggirare la minore espressività dei linguaggi target;

5. l'assenza di applicazioni parziali in mTT permette di evitare ambiguità sul numero di argomenti applicati all'interno del codice estratto, mentre si aveva invece bisogno di applicare un trattamento speciale in Coq;

6. in mTT la rigidità del costrutto di somma disgiunta permette di avere tipo di ritorno variabile solamente nel caso delle proposizioni, le quali avendo il medesimo kind sono ugualmente soggette a eliminazione, mentre in Coq poteva essere necessario distinguere casi con output informativo da casi con output inutile;

7. in seguito alla forte gerarchizzazione dei kind, la seguente espressione, che in Coq richiede una gestione particolare poiché il suo estratto diverge, in mTT viene correttamente estratta in una semplice stella senza manipolazioni aggiuntive:

$$U : prop_s \vdash \ldots \in (enc(U) = enc(U \to U)) \implies loop \in U$$

dove *enc* è l'operatore esplicito che in mTT trasforma un tipo in un termine.

2.6 Lavori futuri

L'estrattore sviluppato in questa tesi è completo e in grado di tradurre ogni costrutto di mTT in codice OCaml e Haskell funzionante. I programmi così estratti potrebbero però ancora avere una struttura eccessivamente complicata o poco naturale per via della diversa sintassi di mTT rispetto ai linguaggi target e per la meccanicità della generazione di prove effettuata dal compilatore. In particolare, abbiamo visto come a causa della valutazione *eager* di OCaml sia necessario conservare una parte del codice inutile per non alterare la complessità computazionale del programma.

È dunque possibile cercare di mitigare questi due problemi attraverso la scrittura di ottimizzazioni *peephole*, in grado di rilevare parti di codice inefficienti o verbose e sostituirle con un equivalente più rapido e conciso. Conscio

di questa necessità, ho strutturato la fase di eliminazione di codice in modo che sia facilmente estendibile: nuove regole di collasso scritte in λProlog possono essere aggiunte, anche in un file a parte, senza alcuna modifica al codice esistente; all'interno di questi predicati è possibile fare *pattern matching* su strutture arbitrariamente complicate di mTT e definirne l'eventuale semplificazione.

In conclusione, l'implementazione di un estrattore di codice si pone al servizio di un *software* più ampio, un dimostratore interattivo, senza il quale l'estrazione da sola risulta priva di significato. Al completamento del dimostratore basato su mTT mancano due requisiti fondamentali:

- il completamento della fase di compilazione, per la quale risultano ancora assenti alcune codifiche e l'intera fase di effettiva traduzione dal livello estensionale a quello intensionale;
- un'interfaccia, che consenta di scrivere agevolmente enunciati e dimostrazioni, utilizzando la sintassi standard della logica matematica.

Una volta terminata l'implementazione di queste altre due parti sarà possibile per un utente trascrivere una dimostrazione, compilarla, accertandone così la correttezza, ed ottenere da essa un programma effettivamente eseguibile ed utilizzabile da tutti.

Bibliografia

[Ber+00] Stefano Berardi, Mario Coppo, Ferruccio Damiani e Paola Giannini. «Type-Based Useless-Code Elimination for Functional Programs Position Paper». In: *Semantics, Applications, and Implementation of Program Generation*. A cura di Walid Taha. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 172–189. ISBN: 978-3-540-45350-5.

[Boe94] Luca Boerio. «Extending pruning techniques to polymorphic second order λ-calculus». In: *Programming Languages and Systems — ESOP '94*. A cura di Donald Sannella. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 120–134. ISBN: 978-3-540-48376-2.

[Dun+15] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen e Enrico Tassi. «ELPI: fast, Embeddable, λProlog Interpreter». In: *Proceedings of LPAR*. Suva, Fiji, nov. 2015. URL: `https://hal.inria.fr/hal-01176856`.

[FS18] Alberto Fiori e Claudio Sacerdoti Coen. «Towards an implementation in lambdaProlog of the two level Minimalist Foundation». In: *Joint Proceedings of the CME-EI, FMM, CAAT, FVPS, M3SRD, OpenMath Workshops, Doctoral Program and Work in Progress at the Conference on Intelligent Computer Mathematics co-located with the 11th Conference on Intelligent Computer Mathematics (CICM 2018)*. (http://ceur-ws.org/). Ago.

2018. URL: https://www.cicm-conference.org/2018/infproc/index.html.

[Let03] Pierre Letouzey. «A New Extraction for Coq». In: *Types for Proofs and Programs*. A cura di Herman Geuvers e Freek Wiedijk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 200–219. ISBN: 978-3-540-39185-2.

[Let04] Pierre Letouzey. «Certified functional programming : Program extraction within Coq proof assistant». Tesi di dott. Lug. 2004.

[Mai08] Maria Emilia Maietti. «A minimalist two-level foundation for constructive mathematics». In: *ArXiv e-prints* (nov. 2008). arXiv: 0811.2774 [math.LO].

[MN86] Dale A. Miller e Gopalan Nadathur. «Higher-order logic programming». In: *Third International Conference on Logic Programming*. A cura di Ehud Shapiro. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 448–462. ISBN: 978-3-540-39831-8.

[Mol18] Giacomo Molinari. *Towards an implementation in Lambda-Prolog of Maietti's two-level minimalist foundation*. Tesi di laurea. Ott. 2018.

[Pau89] C. Paulin-Mohring. «Extracting F_ω's Programs from Proofs in the Calculus of Constructions». In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: ACM, 1989, pp. 89–104. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75285. URL: http://doi.acm.org/10.1145/75277.75285.

[SV98] Giovanni Sambin e Silvio Valentini. «Building up a toolbox for Martin-Löf's type theory: subset theory». In: *Twenty-five years of constructive type theory, Proceedings of a Congress held in Venice, October 1995,* a cura di G. Sambin e J. Smith. 1998, pp. 221–244.

[Tip94] Frank Tip. *A Survey of Program Slicing Techniques.* Rapp. tecn. Amsterdam, The Netherlands, The Netherlands, 1994.