Program Transformations in the Delay Monad

A Case Study for Coinduction via Copatterns and Sized Types

Edoardo Marangoni

Università Statale di Milano

Introduzione

Contesto

- Trasformazioni:
  - Funzioni con input e output listati di linguaggi di programmazione
  - Se input e output sono nello stesso linguaggio si chiamo *source to source*
  - Altrimenti *source to target*
- Compilatori:
  - *Trasformano* i programmi in input
  - Ottimizzano, riducono o modificano il listato del programma in input

Contesto

- GCC: decine di milioni di LoCs, ~30 ottimizzazioni
- LLVM: centinaia di migliaia di LoCs, ~40 ottimizzazioni
- Ottimizzazioni, passaggio critico:
  - Zhou, Ren, Gao and Jiang nel 2021 trovano ~57K bug in GCC e ~22K in LLVM
  - Yang, Chen, Eide and Regehr nel 2011 trovano input per cui alcuni compilatori generavano output sbagliato

Contesto

- Le trasformazioni non devono cambiare il significato del programma
- Non basta la “buona fede” nel programmatore
- E’ necessario **dimostrare** che la trasformazione è *safe*
- Stato dell’arte: CompCert
  - Leroy et al
  - Compilatore C
  - 16 trasformazioni e 10 linguaggi intermedi
  - Ogni passaggio verificato formalmente in Coq, un proof assistant

Contesto

Dati:

1. Linguaggio di programmazione (e.g. C)
2. Trasformazioni sui programmi del linguaggio di programmazione

Vogliamo:

3. Strumento formale per dimostrare che la trasformazione non cambia il *significato* del programma
4. Mantenimento dei comportamenti *osservabili*: convergenza, I/O..

Semantiche e comportamenti

Semantiche

- Strumento matematico per modellare il comportamento di un programma
- Per i nostri obiettivi, secondo Leroy (co-autore di CompCert): **operazionale big-step**, simile ad un interprete che mette in relazione risultato finale e programma iniziale

Comportamenti osservabili

- Vogliamo modellare (tra i vari) anche **fallimento** (crash, e.g. divisione per zero, identificatore non inizializzato a runtime), e **divergenza**
- Possiamo mostrare che se `p` diverge allora anche `t(p)` diverge e che se `p` fallisce anche `t(p)` fallisce, non solo che se `p` converge allora anche `t(p)` converge (allo stesso risultato)
- Già fatto in letteratura, usando definizioni non equazionali e regole che portavano a dimostrazioni complesse
- Scopo della tesi è farlo utilizzando un'unica semantica espressa come funzione che modella i tre comportamenti in maniera che faciliti le dimostrazioni

Proof assistants

- Uso generale: aiutare utenti a fare dimostrazioni, a volte in maniera (semi-)automatica
- Sono anche linguaggi di programmazione: i tipi sono proposizioni e i termini sono dimostrazioni (corrispondenza di Curry-Howard)
- Internalizzano una logica, ma per mantenere la consistenza della logica tutte le definizioni devono essere **terminanti**
- All'interno possiamo definire la semantica come relazione (ad uso *deduttivo*) o come funzione (per *calcolare*)

Effetti

- Possiamo modellare i comportamenti come *effetti*
- Nei linguaggi funzionali gli effetti sono rappresentati tramite **monadi**
  - **Convergenza** l'interprete produce un risultato
  - **Fallimento** monade `Maybe`, effetto del fallimento
  - **Divergenza** (da Capretta: *"divergence is an effect"*) monade `Delay`, effetto della divergenza

Effetti

- La semantica è espressa come un interprete che avrà una segnatura simile a

```
eval : Program -> Store -> Delay (Maybe Store)
```

- `Maybe` è un tipo induttivo, i valori di tipo `Maybe A` sono oggetti (matematici) finiti
- `Delay` deve rappresentare valori infiniti e non può essere induttivo

serve uno strumento matematico che permetta di formalizzare oggetti infiniti

Coinduzione e Agda

Coinduzione

- Duale all'induzione
- Molte interpretazioni: categorie, linguaggi formali e automi, punti fissi..
- In generale, se l'induzione definisce l'insieme più piccolo che soddisfa un insieme di regole, la coinduzione definisce l'insieme più grande
- Esempi di definizioni coinduttive sono stream, parole finite ed infinite in un linguaggio formale, alberi infiniti...
- Induzione richiede terminazione, coinduzione richiede **produttività**

Agda

- Proof assistant e linguaggio di programmazione (non general-purpose)
- Infrastruttura migliore per la coinduzione tra i proof-assistant maturi attualmente
- Vari approcci per gestire la coinduzione: *musical notation*, *guardedness* e **size-types**
  - Terminazione e produttività sintattiche
  - *size* è un numero naturale associato ai tipi, controllo terminazione e produttività nel type system

Sviluppo formale

Linguaggio e semantica

- **Imp** è un linguaggio imperativo molto semplice ma Turing-completo

aexp	:=	n | id | plus $a_1 a_2$
bexp	:=	b | le $a_1 a_2$ | and $b_1 b_2$ | not b
comm	:=	skip | assign id a | if b then c^t else c^f | seq c_1 c_2 | while b do c'

- Semantica – `Store` rappresenta la memoria durante l'esecuzione

```
ceval : ∀ {i} (c : Command) → (s : Store) → Delay (Maybe Store) i
```

`ceval` è **un'unica funzione** e modella tutti e tre i comportamenti

Trasformazioni

- Un'analisi e una trasformazione
- *Definite initialization analysis*: controlla se vengono utilizzate variabili inizializzate
- *Constant folding*: esegue calcoli staticamente prima dell'esecuzione del programma

Definite initialization analysis

Teorema

Per ogni comando `c`, per ogni insieme di variabili `v` e `v'`, per ogni store `s`, se vale `Dia v c v'` e `v ⊆ s` allora l'esecuzione di `c` diverge o converge, ma non fallisce.

In Agda:

```
dia-safe : ∀ (c : Command) (s : Store) (v v' : VarsSet)
  (dia : Dia v c v') (v⊆s : v ⊆ dom s) → (h-err : (ceval c s) ↯) →  ⊥
```

Constant folding

- E' una trasformazione, quindi una funzione che opera sugli elementi sintattici di Imp
- In Agda:

```
cpfold : (c : Command) → Command
```

- Vogliamo dimostrare che `cpfold` non cambia la semantica del programma

Constant folding

Teorema

Per ogni comando `c` e per ogni store `s`, la valutazione di `c` in `s` è *uguale* alla valutazione di `(cpfold c)` in `s`.

In Agda:

```
cpfold-safe : ∀ (c : Command) (s : Store) →
 ∞ ⊢ (ceval c s) ≋ (ceval (cpfold c) s)
```

Conclusioni

Semantica, delay e size

- Abbiamo definito la semantica come un'unica funzione che modella i tre effetti
- Abbiamo usato il tipo coinduttivo `Delay` utilizzando il type system come termination checker grazie alle size
- Abbiamo dimostrato che le trasformazioni che abbiamo scelto non cambiano la semantica del programma

Lavori futuri

- Modellare altri effetti: I/O, altri tipi di fallimento
- Considerare linguaggi più ampi
- Implementare trasformazioni più complesse