Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Calcolo del Cost Model Esatto di un Programma in Solidity

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Mattia Guazzaloca

II Sessione
Anno Accademico 2020/2021

A Sofia, che non leggerà mai questa tesi,
ma mi ha aiutato a portarla a termine

A Peter, che forse prima o poi avrà il coraggio di leggerla,
ma avrebbe fatto di tutto per non farmela scrivere

Introduzione

Negli ultimi anni, il settore delle criptovalute, che ha visto per la prima volta la luce nel 2008 grazie a Bitcoin [18], ha ricevuto sempre maggiore attenzione e interesse tanto da portare alla nascita di molte altre valute digitali. Fra queste, una delle più diffuse ed utilizzate è senza dubbio Ether (ETH), la cui popolarità è seconda solo a Bitcoin. Tuttavia, a differenza di Bitcoin, il fattore determinante dietro al successo di Ether è legato al ruolo fondamentale che questa riveste all'interno della piattaforma Ethereum, ideata e sviluppata da Vitalik Buterin e altri nel 2013. L'idea rivoluzionaria degli autori di Ethereum è stata quella di impiegare la tecnologia alla base di Bitcoin, quella della blockchain, per permettere a chiunque di sviluppare applicazioni decentralizzate e quindi immuni a qualsiasi atto censorio. Queste applicazioni, che nel gergo di Ethereum prendono il nome di dApp o smart contracts, possono assolvere agli usi più disparati e consentono a chi le sviluppa di beneficiare della capacità computazionale dei migliaia di nodi presenti sulla rete di Ethereum sui quali queste applicazioni vengono eseguite. Purtroppo però, ciò apre le porte anche a possibili abusi. In particolare, grazie alla Turing-completezza del linguaggio in cui sono scritti gli smart contracts, un banale programma divergente potrebbe rendere rapidamente inutilizzabile l'intera piattaforma. Per tutelarsi da questo genere di attacchi, Ethereum associa a ciascuna istruzione del codice un certo consumo di *gas*, con l'obiettivo di misurare lo sforzo computazionale richiesto per l'esecuzione. Ogni utente dell'applicazione è pertanto obbligato a versare ai nodi esecutori una quantità di Ether pari al gas richiesto per completare l'esecu-

zione, da qui il ruolo centrale della criptovaluta. Pertanto uno dei problemi principali per gli sviluppatori di smart contracts è poter stimare in anticipo i consumi di gas dei loro programmi. Tuttavia, poiché gli smart contracts sono tipicamente scritti in linguaggi di alto livello e poi tradotti nelle istruzioni di basso livello, ottenere delle stime chiare e precise è un compito tutt'altro che triviale. In questo elaborato, presenteremo quindi il cosidetto metodo del *labelling*, sviluppato nell'ambito del progetto europeo CerCo come soluzione a questo problema, e ne illustreremo una possibile implementazione all'interno del compilatore di Solidity, il linguaggio più usato nello sviluppo di smart contracts. L'implementazione presentata non costituisce però una soluzione definitiva né tantomeno completa, pertanto nella parte conclusiva discuteremo alcuni dei miglioramenti che potrebbero essere realizzati in futuro.

Indice

Elenco dei listati

Elenco delle tabelle

Capitolo 1

Background

Scopo di questo capitolo è fornire un background agli argomenti che verranno trattati nell'elaborato. Verranno quindi illustrate le principali caratteristiche di Solidity e della piattaforma Ethereum, in particolare in relazione a come il funzionamento di questa abbia influenzato lo sviluppo del linguaggio.

1.1 Ethereum

Ethereum [6] è una piattaforma per lo sviluppo di applicazioni distribuite, denominate in gergo dApp, e basata su una tecnologia, quella delle blockchain, oramai ampiamente diffusa e studiata. Nonostante il primo e più noto esempio di utilizzo di questa tecnologia sia infatti quello dell'implementazione delle criptovalute come il Bitcoin, non è il solo e anzi col tempo sono state individuate sempre maggiori applicazioni per l'impiego di quest'ultima al di fuori del ramo delle criptovalute.
Il caso di Ethereum è però ancora più emblematico in quanto il suo elemento rivoluzionario consiste nella possibilità di permettere agli utenti di sviluppare dei veri e propri programmi, denominati smart contracts, che possono essere eseguiti all'interno della blockchain per realizzare una computazione distribuita. Per permettere la stesura degli smart contracts, Ethereum definisce un linguaggio composto da un insieme di istruzioni che hanno come target

la EVM, acronimo di Ethereum Virtual Machine, una macchina virtuale di cui il framework definisce la specifica e che i nodi della rete implementano.
All'interno di Ethereum la EVM rappresenta una sorta di computer distribuito dove gli smart contracts possono essere salvati ed eseguiti. Per poterlo modificare è quindi necessario avere il consenso di tutti gli altri partecipanti. Ciascun nodo ha infatti un duplice ruolo: in ogni momento può richiedere l'esecuzione di una qualsiasi computazione ma perché la richiesta abbia successo è necessario che questa sia verificata ed eseguita dagli altri membri del network. Quando ciò accade lo stato della EVM viene aggiornato e sia la richiesta iniziale che il nuovo stato sono salvati all'interno della blockchain. Queste richieste prendono il nome di *transazioni* e oltre a contenere il codice da eseguire (o un riferimento ad esso) specificano una serie di altri attributi fra cui i più rilevanti per questo elaborato sono:

- **gasPrice**: è il prezzo che il commitente della transazione è disposto a pagare al *miner*, ossia il nodo che la processerà, per unità di gas.
- **gasLimit**: è la quantità massima di gas che può essere utilizzata per portare a termine la transazione. Se risulta insufficiente questa verrà abortita e il gas consumato andrà perso insieme con l'Ether speso.

1.1.1 L'ether e il Gas

L'ether, spesso abbreviato in ETH, è la criptovaluta di Ethereum. Come le altre criptovalute può essere utilizzata per i pagamenti e immagazzinata nei portafogli digitali all'interno della rete, ma in Ethereum il suo ruolo è più ampio e risulta di fondamentale importanza per mantenere l'intero ecosistema al riparo da attacchi mirati a sfruttarne in maniera incodizionata l'elevatissima capacitá computazionale o a limitarne o impedirne l'utilizzo da parte degli altri utenti (attacchi di tipo DDoS).
Chi volesse richiedere una qualsiasi computazione è infatti obbligato ad associare alla sua richiesta anche una certa quantità di ether, in maniera non dissimile da come per poter usufruire dei servizi offerti dalle piattaforme di

cloud computing è necessario pagare una certa quota sulla base della durata della computazione e/o dello spazio occupato.
Chiaramente l'ether richiesto per effettuare una certa transazione non è sempre lo stesso ed è anzi strettamente correlato al codice dello smart contract di cui la transazione richiede l'esecuzione, oltre che dal numero e dalla dimensione degli eventuali parametri associati alla chiamata. A ciascuna delle istruzioni che compongono il bytecode della EVM corrisponde infatti un certo costo, il cui valore è espresso in unità di gas. Il *gas*, come suggerisce il nome, permette di misurare lo sforzo computazionale necessario per portare a termine una specifica operazione. Nella Tabella 1.1 sono riportati i consumi di gas di alcune delle istruzioni della EVM, per la lista completa si veda [7] e l'appendice G di [21].

1.2 Solidity

Solidity è un linguaggio di programmazione di alto livello per lo sviluppo di smart contracts. Il suo scopo è semplificare la scrittura dei contratti da parte degli sviluppatori, che possono così fare affidamento su un linguaggio moderno simile a molti dei più noti linguaggi di programmazione abitualmente utilizzati dagli sviluppatori come C++, Java o JavaScript (al quale la sintassi di Solidity è ispirata). Una volta completato il codice del contratto viene compilato tramite il compilatore di Solidity [10] che produce in output il bytecode che può essere eseguito dalla EVM.
Di seguito illustreremo alcune delle caratteristiche principali di Solidity, soffermandoci in particolare su quelle che lo contraddistinguono come linguaggio per smart contracts.

1.2.1 Tipi

Solidity è un linguaggio di programmazione staticamente tipato, pertanto il tipo di ciascuna variabile deve essere noto a tempo di compilazione in modo che il type checker possa effettuare i controlli necessari a garantire la

Istruzione	Gas	Descrizione
`ADD`	3	Prende due elementi dalla cima dello stack e vi sostituisce la loro somma
`MUL`	5	Prende due elementi dalla cima dello stack e vi sostituisce il loro prodotto
`PUSHn`	3	Aggiunge allo stack un elemento di dimensione pari a `n` byte
`POP`	2	Rimuove un elemento dallo stack
`JUMP`	8	Salto incondizionato
`JUMPI`	10	Salto condizionato
`AND`	3	Prende due elementi dalla cima dello stack e vi sostituisce il loro and logico
`OR`	3	Prende due elementi dalla cima dello stack e vi sostituisce il loro or logico
`SLOAD`	800	Legge un valore dallo *storage*
`ADDRESS`	3	Indirizzo del contratto in esecuzione
`BALANCE`	700	Saldo del contratto in esecuzione in *Wei* (un sottomultiplo dell'Ether)

Tabella 1.1: Consumi di gas per alcune instruzioni della EVM

coerenza dei tipi, per quanto questa sia purtroppo facilmente aggirabile come evidenziato da [4, 5]. Solidity contiene i tipi di dato primitivi più comuni come booleani e interi, di cui sono presenti diverse varianti con e senza segno, oltre che diversi tipi utili nel contesto delle dApp come diverse tipologie di array di byte e il tipo speciale `address`, usato per ottenere l'indirizzo di un account Ethereum. Grazie ai costrutti `struct` e `enum` è poi possibile definire i propri tipi di dato come composizione dei tipi di dato primitivi come nella maggior parte dei linguaggi di programmazione e inoltre ciascun contratto definisce un tipo a sé stante, come le classi in Java. Un elemento particolarmente distintivo è invece la possibilità (che in alcuni casi diventa

un obbligo) di poter specificare, tramite keyword corrispondenti alle diverse aree di memoria della EVM, dove memorizzare una variabile.
Queste sono:

- `storage`: si tratta di uno store per coppie di chiavi-e-valori persistente, pertanto non viene resettato fra una chiamata di funzione e l'altra o fra una transazione e la successiva. È però costoso sia da leggere che da scrivere perciò tipicamente conviene limitare il suo utilizzo esclusivamente ai dati propri del contratto, come le variabili di stato (§1.2.4)

- `memory`: si tratta di un'area di memoria lineare indirizzabile tramite *word* di lunghezza pari a 256-bit. Inoltre le scritture possono avvenire anche a livello di un singolo byte (tramite l'istruzione `MSTORE8`). Se viene effettuato un accesso ad una parola di memoria che non era stata precedentemente interessata da qualche operazione, la memoria viene allargata della quantità richiesta per completare l'accesso e il costo per l'espansione viene pagato in termini di consumo di gas. Infine, a differenza di `storage`, quest'area non è persistente e il contratto in esecuzione ne riceve una nuova istanza ad ogni chiamata.

- `calldata`: è l'area della EVM dove vengono memorizzati gli argomenti delle funzioni. Si comporta in modo simile a `memory` ma non è modificabile.

1.2.2 Funzioni e Modifiers

Tipicamente le funzioni vengono definite all'interno dei contratti, analogamente ai metodi delle classi in Java, ma Solidity supporta anche la definizione di free-standing functions, ossia funzioni non legate ad uno specifico contratto bensì parte dello scope globale. Una carattestica peculiare delle funzioni in Solidity è data dalla possibilitá di poter indicare nella definizione il livello di state mutability richiesto, ovvero in che modo la funzione accede allo stato interno del contratto. Si distinguono pertanto tre tipi di funzioni:

- `pure`: non interagiscono in alcun modo con lo stato interno, né in lettura né in scrittura. Sono tipicamente semplici funzioni utilizzate per esempio per calcoli matematici.

- `view`: possono accedere in lettura allo stato interno ma non possono in alcun modo modificarlo. In pratica tutte le funzioni getter o quelle che calcolano un qualche valore derivabile dallo stato interno.

- tutte le funzioni che non presentano i due modifcatori precedenti possono leggere e modificare liberamente lo stato del contratto

Un altro componente caratteristico di Solidity sono i *modifiers*. Questi rappresentano blocchi di codice che possono essere applicati alle funzioni per dotarle di una certa funzionalitá (come per esempio il controllo di una qualche pre o post condizione) senza ripetere sempre lo stesso codice.
Solidity supporta inoltre, come pressoché tutti i linguaggi di alto livello, il meccanismo della ricorsione.

1.2.3 Gestione degli errori

Il meccanismo di creazione e gestione degli errori è analogo a quello di molti altri linguaggi orientati agli oggetti ed è basato sulle eccezioni. Queste possono essere dichiarate sia internamente che esternamente ad un contratto attraverso la keyword `error` e vengono individuate e gestite attravero il costrutto *try/catch*, mentre per sollevarle è necessario utilizzare un *revert-statement* il cui compito è quello di annullare tutte le modifiche effettuate dalla transazione corrente riportando lo stato a quello precedente.

1.2.4 Programmazione Orientata agli Oggetti

In Solidity gli oggetti sono rappresentati dai contratti, i quali possono essere istanziati, distrutti e modificati attraverso i metodi che mettono a disposizione. La ragion d'essere della OOP in Solidity è principalmente legata al riuso del codice che è reso possibile dal meccanismo dell'ereditarietà.

Quando si definisce un contratto è infatti possibile specificare quelli (supporta l'ereditarietà multipla) da cui il primo eredita metodi e variabili di stato, analoghe alle variabili di istanza dei linguaggi orientati agli oggetti, le cui definizioni possono poi essere sovrascritte.
Vi è poi un ulteriore elemento tipico della programmazione a oggetti di cui Solidity fa ampio uso: l'information hiding. Quando si dichiara un metodo o una variabile di stato è possibile indicarne il livello di accesso tramite uno dei quattro modificatori definiti dal linguaggio:

- `external`: una funzione esterna è parte dell'API del contratto e può quindi essere invocata da altri contratti. Non è però possibile chiamare internamente una funzione esterna, a meno di non utilizzare la sintassi `this.someExternalFunction()`.

- `public`: simili alle funzioni `external` ma possono essere chiamate internamente senza utilizzare `this`. La natura di questa differenza verrà trattata in §3.5.1. Dichiarando una variabile come pubblica il compilatore genererà inoltre automaticamente una funzione getter.

- `internal`: a funzioni e variabili marcate con questo modificatore è possibile accedere solo internamente, ossia dal contratto corrente o dai suoi derivati.

- `private`: la visibilità di funzioni e variabili private è limitata al contratto in cui sono definite.

Capitolo 2

Certified Complexity

L'obiettivo di questo capito è quello di illustrare il framework teorico sviluppato dal progetto europeo CerCo [17] per la realizzazione di un compilatore capace di riportare nel codice sorgente le informazioni relative al costo dell'esecuzione del corrispondente codice oggetto. Il lavoro svolto da CerCo ha posto le basi per lo sviluppo di questo elaborato e ne costituisce il background teorico.

2.1 Il compilatore di CerCo

Il contributo principale del progetto CerCo consiste nell'aver realizzato, e verificato formalmente, un cost annotating compiler ossia un compilatore capace di riportare nel codice sorgente delle indicazioni circa il costo di esecuzione di un certo blocco di codice e chiamate pertanto annotazioni di costo. Formalmente, un cost annotating compiler, preso in input un programma P è in grado di produrre come output un programma $An(P)$ "annotato" funzionalmente equivalente al precedente ma modificato oppurtamente per tenere traccia del costo di esecuzione. Vediamo un esempio:

```
1 int fib(int n) {
2   if (n < 2) {
3     return 1;
4   } else {
5     return fib(n - 1) + fib
          (n - 2);
6   }
7 }
```

Listing 2.1: Sorgente originale

```
1  int _cost = 0;
2  int fib(int n) {
3    __cost2:
4    _cost = _cost + 571;
5    if (n < 2) {
6      __cost0:
7      _cost = _cost + 179;
8      return 1;
9    } else {
10     __cost1:
11     _cost = _cost + 375;
12     return fib(n - 1) + fib
           (n - 2);
13   }
14 }
```

Listing 2.2: Codice annotato

Appare immediatamente evidente come il compilatore abbia aggiunto al programma originale la variabile `_cost` il cui compito è intuitivamente quello di tracciare il costo dell'esecuzione del programma in ogni sua parte. Questo avviene attraverso incrementi successivi della variabile inseriti all'inizio dei blocchi il cui costo computazionale è O(1), ossia costante. Come si può notare infatti l'incremento della variabile `_cost` per il ciclo for viene fatto all'interno del corpo del ciclo e non all'esterno. Ciò è determinante per ottenere delle annotazioni di costo che siano *sound* e *precise*.

Definition 1. Una cost annotation è *sound* se il suo valore costituisce un upper bound al costo di esecuzione reale

Definition 2. Una cost annotation è *precise* quando la differenza fra il costo reale dell'esecuzione e quello espresso dall'annotazione è limitata dal valore di una costante che dipende solo dal programma.

Una volta attribuito il costo ai blocchi costanti sarà sufficiente sommare i loro costi per ottenere quello totale, esattamente come accade per l'esempio in figura.

Scopo di queste annotazioni è quindi quello di creare un mapping quanto più possibile preciso fra un frammento di codice scritto nel lingauggio sorgente, tipicamente di alto livello, preso in input dal compilatore e il prodotto della compilazione ossia il corrispettivo codice oggetto per una qualche architettura. Grazie ad esse è quindi possibile formulare una descrizione esatta del modello di costo del codice oggetto emesso dal compilatore che può quindi essere successivamente impiegata per verificare che alcune importanti proprietà del sorgente siano soddisfatte. È il caso, ad esempio, dei programmi real-time per i quali conoscere staticamente con precisione il tempo necessario per eseguire una certa sequenza di istruzioni sarebbe determinante per decidere se il codice riuscirà o meno a rispettare le deadline richieste.

2.2 Labelling

Il *labelling* è il metodo proposto dai membri di CerCo per trasformare un normale compilatore in uno capace di generare annotazioni di costo [2]. L'idea alla base del metodo è quella di *etichettare* ciascuno dei blocchi O(1) in maniera particolare in modo tale che il suo costo venga associato all'etichetta e sia così possibile ragionare sul cost model del programma a un livello di astrazione più alto rispetto a quello delle singole istruzioni presenti nel codice oggetto. Una descrizione sommaria del metodo è riportata nello schema seguente.

$$\begin{array}{ccccccc} L_1 & \xleftarrow[\mathcal{I}]{} & L_{1,l} & \xrightarrow{\mathcal{C}_1} & L_{2,l} & \cdots \xrightarrow{\mathcal{C}_k} & L_{k+1,l} \\ & & \mathcal{L}\uparrow\downarrow er_1 & & \downarrow er_2 & & \downarrow er_{k+1} \\ & & L_1 & \xrightarrow{\mathcal{C}_1} & L_2 & \cdots \xrightarrow{\mathcal{C}_k} & L_{k+1} \end{array}$$

Il processo di compilazione è rappresentato dalla sequenza di linguaggi $L_1...L_{k+1}$ intervallati dalle rispettive funzioni di compilazione $C_1...C_k$. I compilatori sono infatti programmi molto complessi e pertanto sono tipicamente

suddivisi in più fasi, a cui corrispondono le funzioni $C_1...C_k$. Ciascuna delle fasi ha come output una certa rappresentazione che, per quanto eventualemente arrichita di informazioni, deriva più o meno direttamente dal codice sorgente e prende in input quella generata dalla fase precedente. La fase di parsing, per esempio, processa la lista di token prodotta dalla fase di analisi lessicale e restituisce come output un albero di sintassi astratta (AST). Una rappresentazione può quindi assumere forme e strutture molto diverse: lista, albero e, nei casi più complessi, anche veri e propri linguaggi intermedi. Nello schema ciascuna di queste rappresentazione viene identificata da uno dei linguaggi $L_1...L_k$.
Per ognuno dei linguaggi $L_1...L_k$ viene poi definito un nuovo linguaggio $L_{i,l}$ identico a L_i ma con l'aggiunta di un nuovo costrutto per permettere l'etichettatura dei blocchi tramite *label*. Chiaramente per supportare $L_{i,l}$ la corrispettiva funzione di compilazione C_i deve essere opportunamente modificata. Inoltre, per ciascuno dei linguaggi *etichettati* è naturalmente definita una funzione er_i il cui compito è cancellare tutte le label presenti nel codice scritto in $L_{i,l}$ in modo da ottenere il suo corrispettivo nel linguaggio senza label. Con queste premesse è possibile definire le due operazioni principali del metodo proposto da CerCo.

Definition 3. Un *labelling* L di un linguaggio sorgente Li è una funzione tale che $er_{L_i} \circ L$ è la funzione identità

Definition 4. Una *instrumentazione* I di un linguaggio etichettato $L_{1,l}$ è una funzione che rimpiazza le label di $L_{1,l}$ con, per esempio, incrementi di una variabile di costo

Un esempio di codice sottoposto a labelling è riportato nel Listato 2.3 mentre il risultato dell'instrumentazione è esattamente quello riportato nel Listato 2.2.

```
1 int fib(int n) {
2   __cost2:
3   if (n < 2) {
4     __cost0:
```

```
5       return 1;
6     } else {
7       __cost1:
8       return fib(n - 1) + fib(n - 2);
9     }
10  }
```

Listing 2.3: Sorgente etichettato da CerCo

Date le operazioni definite da 3 e 4 un'annotazione di costo può essere definita molto semplicemente come la composizione, e quindi in pratica come un'esecuzione in sequenza, dell'operazione di labelling e di quella di instrumentazione, in simboli $An = I \circ L$.

Tuttavia, giunti a questo punto è lecito chiedersi se il metodo proposto sia realmente in grado di fornire delle annotazioni che siano sia *sound* che *precise* e la risposta è affermativa. Le annotazioni prodotte dal metodo del labelling risultano infatti *sound* se tutto il codice sorgente è coperto da almeno una label e, allo stesso tempo, se tutti i percorsi che vanno da una label alla successiva hanno lo stesso costo allora sono anche *precise*. Le prove formali di questi risultati sono anch'esse parte integrante del lavoro svolto dai membri di CerCo [17].

Capitolo 3

Implementazione

3.1 Overview

L'implementazione per un nuovo linguaggio del framework sviluppato dal progetto CerCo può avvenire sostanzialmente in due modi:

1. Realizzazione *ex novo* dell'implementazione del linguaggio, ossia di un nuovo compilatore, ma con le estensioni descritte nel capitolo 2

2. Modifica di un compilatore esistente per trasformarlo in un *cost annotating compiler*

L'approccio seguito dagli autori di CerCo è il primo, in quanto i moderni compilatori C, come `gcc` [13], poco si prestano all'implementazione delle estensioni necessarie per il metodo del labelling. Inoltre, poiché uno degli obiettivi principali del progetto era quello di provare formalmente la correttezza dell'implementazione, realizzarne una da zero avrebbe permesso di costruirla in modo da semplificarne le prove di correttezza.
Tuttavia, lo sforzo richiesto per implementare un compilatore è molto elevato e pertanto nello svolgimento di questo elaborato si è preferito impiegare il secondo approccio. Inoltre, mentre è ampiamente noto come il linguaggio C sia stato progettato per essere semplice e quanto più possibile vicino al linguaggio macchina [20], è evidente come le caratteristiche peculiari di Solidity

[10], soprattutto in quanto linguaggio orientato agli oggetti, e il suo ambiente di riferimento, Ethereum e gli smart contracts, lo rendano un linguaggio intrinsecamente complesso. Tale complessità ha influenzato quindi inevitabilmente anche il compilatore, il cui compito principale è proprio quello di tradurre i complessi meccanismi di alto livello di Solidity nel bytecode della EVM.
Per questo motivo abbiamo cercato di minimizzare il numero di modifiche effettuate alla codebase del compilatore di Solidity, *solc*, optando invece per la creazione di un tool aggiuntivo esterno al compilatore denominato, in onore al progetto originale, `Cerco2`[15] con l'obiettivo di demandare ad esso lo svolgimento della gran parte delle verifiche e delle procedure necessarie per l'implementazione del metodo del labelling.
Nelle sezioni seguenti effettueremo quindi una panoramica di entrambi gli strumenti, ponendo l'accento in particolare su come sono fra loro legati e come dovrebbero quindi essere usati da un utente terzo.

3.1.1 Il compilatore di Solidity: solc

Le interfacce utente

Il **Sol**idity **C**ompiler[10], *solc* in breve, è l'implementazione di riferimento, oltre che la sola al momento, del linguaggio Solidity ed è disponibile per Windows, Macos, e Linux. La modalità consigliata dagli sviluppatori per interagire con il compilatore è la cosidetta *JSON Input/Output Interface*. In questa modalità, solc si aspetta di ricevere in input un file JSON contente i nomi dei file sorgente da compilare e altri metadati ed emette in output un altro file JSON contente i risultati della compilazione e quanto specificato dall'utente nel file di input (la specifica completa del formato dei file di input e output è riportata nella documentazione di Solidity [9]). La JSON Input/Output Interface è pensata per favorire l'integrazione del compilatore all'interno di setup automatici complessi o altri tool come Remix [8], un ambiente di sviluppo integrato (IDE) per smart contract, o Truffle [12], una

suite di software per lo sviluppo e il testing degli smart contract. Per questo motivo durante lo sviluppo di questo elaborato abbiamo preferito utilizzare direttamente l'interfaccia a riga di comando messa a disposizione dal compilatore, di cui è possibile visualizzare tutte le opzioni invocandolo con il flag `--help`.
Nella Tabella 3.1 sono riportate in particolare quelle più usate durante lo sviluppo dell'elaborato.

Gruppo	**Flag**	**Descrizione**
Informazioni generali	`--help`	Mostra il messaggio di aiuto.
	`--version`	Mostra la versione.
Opzioni di output	`--evm-version`	Permette di selezionare la versione della EVM su cui verrà eseguito il bytecode.
	`--experimental-via-ir`	Abilita la compilazione attraverso IR, ossia Yul.
Componenti di output	`--asm`	Emette l'assembly EVM del contratto.
	`--bin`	Emette il bytecode del contratto.
	`--ir`	Emette la rappresentazione intermedia (Yul) del contratto.
	`--ir-optimized`	Emette la rappresentazione intermedia (Yul) ottimizzata del contratto.
Ottimizzatore	`--optimize`	Abilita l'ottimizzatore.
	`--yul-optimization-steps`	Permette di specificare la sequenza di ottimizzazione.
Output aggiuntivo	`--gas`	Stampa una stima del massimo consumo di gas per ogni funzione.

Tabella 3.1: Opzioni interfaccia a riga di comando di solc

Inoltre sono state aggiunte tre opzioni per la generazione di particolari output:

- `--lb`: genera le definizioni di tutti i contratti in seguito alla fase di labelling (3.2)
- `--asm-costs`: analogo di `--asm` ma con l'aggiunta del consumo del gas degli opcode (3.3)
- `--instrumented`: genera la versione parzialmente instrumentata dei contratti (3.5)

La struttura

La struttura di solc corrisponde a quella che si ritrova tipicamente in tutti i compilatori moderni per linguaggi di alto livello ed è composta da una serie di layer che corrispondono, più o meno direttamente, alle diverse fasi del processo di compilazione la cui suddivisione, così come descritta in [14, 3], è riportata brevemente di seguito:

- **Analisi lessicale**: a partire dalla stringa del codice sorgente il *lexer* (o scanner), genera una lista di *token*, ossia gli elementi atomici che compongono la sintassi del linguaggio. Ad esempio, il codice seguente in Solidity, `uint v;` produce tre token: l'identificatore del tipo `'uint'`, la variable `'v'` e l'operatore di concatenazione dei comandi `';'`.
- **Analisi sintattica**: dalla lista dei token il *parser* deriva un albero di sintassi astratta, *AST*, il quale rappresenta la struttura logica del programma e verrà usato nelle fase successive. Inoltre, un fallimento nella costruzione dell'AST indica un errore di sintassi nel programma.
- **Analisi semantica**: durante questa fase l'AST viene sottoposto ai controlli relativi ai cosidetti *vincoli contestuali* come la coerenza dei tipi, la validità delle dichiarazioni, ecc. L'output di questa fase è un AST aumentato, ossia arrichito con le informazioni ottenute tramite i controlli effettuati.
- **Ottimizzazione**: in questa fase il codice viene ottimizzato attraverso molteplici tecniche con l'obiettivo di renderlo più "snello" in termini

di spazio, per esempio attraverso la rimozione del *dead code*, ossia il codice irragiungibile, o più veloce, attraverso *inlining*, sostituzione delle chiamate alle funzioni col corpo delle funzioni stesse, *loop unrolling*, l'eliminazione dei cicli con un numero di iterazioni staticamente note in favore della ripetizione per lo stesso numero di volte del corpo del ciclo, e altre.

- **Generazione del codice oggetto**: viene generato il codice oggetto, tipicamente in formato binario, per l'architettura target.

Come si può facilmente notare ogni fase genera una qualche rappresentazione intermedia, ad eccezione dell'ultima il cui output è il codice oggetto che nel caso di solc corrisponde al bytecode della EVM, e riceve in input quella emessa dalla fase precedente, tranne per l'analisi lessicale il cui dato di ingresso è chiaramente il sorgente stesso. Queste rappresentazioni intermedie possono essere liste, come l'insieme dei token, alberi, come nel caso dell'AST, e in alcuni casi veri e propri linguaggi intermedi caratterizzati da una sintassi e da una semantica ben precise, come [19]. I linguaggi intermedi hanno diversi scopi ma quello per cui vengono impiegati più spesso è quello di separare due parti fondamentali del compilatore: il *frontend*, che comprende le diverse fasi di analisi condotte dal compilatore, e il cosidetto *backend*, composto invece dalle fasi di ottimizzazione e generazione del codice oggetto. Questa suddivisione può risultare fondamentale nel caso in cui si voglia implementare lo stesso linguaggio sorgente su target diversi o, al contrario, diversi linguaggi sorgenti sul medesimo target poiché sarebbe sufficiente modificare solamente il frontend, rispettivamente il backend, lasciando inalterata l'altra parte. Ciò è reso possibile però solo dalla presenza di un qualche linguaggio che funge da target univoco per le ottimizzazioni e può essere compilato su diverse architetture indipendentemente dal linguaggio sorgente.

Il linguaggio intermedio Yul

Recentemente, proprio con l'intento di aggiungere a Solidity il supporto per molteplici target, gli sviluppatori di solc hanno ideato *Yul* [11], un linguaggio intermedio che può essere compilato nel bytecode di diverse versioni della EVM (`EVM 1.0`, `EVM 1.5` e `Ewasm`). Lo scopo principale di Yul è per l'appunto quello di astrarre i dettagli delle singole piattaforme in particolare ponendosi come linguaggio target della fase di ottimizzazione. Non a caso infatti, in seguito alla sua introduzione, gli sviluppatori hanno completamente riscritto l'ottimizzatore che è adesso in gran parte basato proprio su di esso. Per quanto sia ancora in fase sperimentale quindi, gli investimenti per renderlo uno standard *de facto* sono concreti e continui, per questo motivo abbiamo deciso di includerlo nell'elaborato. Fra gli obiettivi che ne hanno guidato il design vi sono poi:

- Leggibilità: la sintassi è molto più simile a quella di un linguaggio di alto livello che a quella di un tipico linguaggio assembly
- Semplicità del controllo di flusso: anche in questo caso si è preferita una sintassi ispirata ai linguaggi di alto livello e pertanto sono presenti direttamente costrutti come `if`, `switch` e `for` ed è ammessa perfino la ricorsione.
- Facilità di traduzione in bytecode: nonostante i costrutti e la sintassi di alto livello è abbastanza facile intuire i loro corrispettivi in bytecode osservando il codice in Yul.

A sostegno di quanto detto mostriamo di seguito due esempi della funzione `power` per calcolare la potenza ennesima di una fuzione la prima in versione iterativa e la seconda ricorsiva.

```
1 {
2     function power(base, exponent) -> result {
3         result := 1
4         for { let i := 0 } lt(i, exponent) { i := add(i, 1) }
5         {
```

```
6               result := mul(result, base)
7           }
8       }
9   }
```

Listing 3.1: Definizione iterativa della funzione `power` in Yul

```
1   {
2       function power(base, exponent) -> result {
3           switch exponent
4           case 0 { result := 1 }
5           case 1 { result := base }
6           default {
7               result := power(mul(base, base), div(exponent, 2)
                    )
8               switch mod(exponent, 2)
9                   case 1 { result := mul(base, result) }
10          }
11      }
12  }
```

Listing 3.2: Definizione ricorsiva della funzione `power` in Yul

3.1.2 Cerco2

Cerco2 è un tool da riga di comando sviluppato durante lo svolgimento dell'elaborato in aggiunta alle modifiche apportate al compilatore di Solidity. Così come spiegato in §3.1, la sua realizzazione si è resa necessaria per via della complessità di solc. I controlli necessari a verificare che le annotazioni di costo prodotte risultino *sound* e *precise* così come l'algoritmo per l'attribuzione dei costi, descritto in 3.4, sono infatti tutt'altro che triviali. Il secondo, per esempio, si basa sulla visita di una struttura dati particolare chiamata *Control Flow Graph* la cui implementazione in C++, il linguaggio nel quale solc è scritto, avrebbe rappresentato un inutile ostacolo aggiuntivo.
Per ovviare alla complessità di C++, si è quindi deciso di scrivere il tool in JavaScript in modo tale che potesse essere eseguito su Node.js e risultare così

comunque disponibile per tutte le piattaforme supportate da solc. Così come per solc, è possibile ottenere la lista di tutte le opzioni disponibili per la CLI passando il flag `--help` (o `-h` in breve). Le principali sono riportate nella Tabella 3.2.
Nonostante sia uno strumento esterno, Cerco2 è pensato direttamente come estensione di solc. Internamente infatti, una volta elaborate le opzioni ricevute dall'utente (fra le quali deve essere presente almeno anche un file sorgente) invoca il compilatore con un set di opzioni predefinito in modo da ottenere i file necessari per le successive elaborazioni. Fra questi i più importanti sono il codice assembly dei sorgenti compilati con l'aggiunta dei costi delle istruzioni e le definizioni parzialmente instrumentate di ciascun contratto (ottenuti grazie ai flag aggiunti a solc, per i dettagli si faccia riferimento a §3.1.1).

Flag	**Descrizione**
`-h,--help`	Mostra il messaggio di aiuto.
`-e,--example <path>`	Processa l'esempio con percorso `path`.
`--stdout`	Mostra lo *standard output* della compilazione per gli esempi processati.
`--stderr`	Mostra lo *standard error* della compilazione per gli esempi processati.
`-g,--graph`	Emette in output il CFG dell'esempio processato.
`-c,--costs`	Emette in output le associazioni *etichetta-costo* in un file JSON.

Tabella 3.2: Opzioni interfaccia a riga di comando di Cerco2

3.2 Labelling

Nelle fasi iniziali del progetto, l'operazione di labelling descritta nel paragrafo §2.2 veniva effettuata manualmente. Tuttavia, è evidente come ef-

fettuare un processo così delicato in questo modo possa condurre facilmente ad errori e non è comunque sostenibile nel caso di programmi di grandi dimensioni. In seguito abbiamo quindi deciso di implementarla direttamente all'interno di solc, in modo tale da renderlo un processo automatico e, idealmente, corretto.

A prescindere dal linguaggio considerato l'implementazione del labelling passa in primis necessariamente per la definizione di un costrutto per etichettare i blocchi di codice. Questa operazione corrisponde alla definizione del linguaggio etichettato $L_{i,l}$ a partire dal linguaggio L_i. Per supportare questo nuovo costrutto è stato sufficiente aggiungere una nuova tipologia di nodi all'AST di Solidity. L'aggiunta di un nodo avviene attraverso la definizione di una nuova classe all'interno del file `AST.h` in cui è possibile trovare le definizioni di tutti i possibili nodi dell'albero. Nel nostro caso la classe è stata denominata `LabelStatement` poichè eredita dalla classe `Statement`, la classe base per tutti gli elementi che, a differenza delle espressioni, non producono alcun valore. La classe `Statement` eredita a sua volta dalla classe astratta `ASTNode` che definisce i metodi necessari per realizzare la visita di un particolare nodo. In seguito, abbiamo modificato il parser affichè potesse utilizzare questo nuovo nodo per rappresentare le etichette. Infatti, poiché la generazione delle label è un compito esclusivo del compilatore, queste possono essere aggiunte direttamente dal parser durante la costruzione dell'albero, mentre sarebbe un errore implementarne il parsing in quanto non esiste un modo per lo sviluppatore di specificarle nel sorgente. Ciò nonostante, per favorire il debugging e l'ispezione del codice, nel caso in cui venga richiesto in output il sorgente etichettato i nodi di tipo `LabelStatement` verrano trasformati in semplici commenti multilinea contenenti il nome della label generata come nell'esempio seguente.

```
1 contract Fibonacci {
2     function fibonacci(uint256 n) public pure returns (
      uint256 b) {
3         /* __cost0 */
4         if (n == 0) {
```

```
5               /* __cost1 */
6               return 0;
7           }
8           /* __cost2 */
9           uint256 a = 1;
10          b = 1;
11          for (uint256 i = 2; i < n; i++) {
12              /* __cost3 */
13              uint256 c = a + b;
14              a = b;
15              b = c;
16          }
17          /* __cost4 */
18          return b;
19      }
20  }
```

Listing 3.3: Esempio di codice etichettato

L'algoritmo impiegato dal parser per aggiungere le label all'albero è il seguente (si veda ad esempio il Listato 3.3):

1. L'inizio di ciascun blocco è etichettato con una label.

2. Tutte le sequenze di istruzioni precedute da un qualche costrutto per il controllo di flusso (che apre quindi uno o più blocchi all'interno di quello principale) sono etichettate con una label posta subito dopo la fine del costrutto

Ciò garantisce che tutte le istruzioni siano etichettate con almeno una label e, se assumiamo che un'etichetta sia valida fino a quella successiva, allora questo bound diventa pari a uno e pertanto ciascuna istruzione risulta coperta da esattamente una label.

3.2.1 Da Solidity a Yul

Come riportato in §2.2, la presenza di linguaggi intermedi influenza in maniera importante il processo di labelling perché le etichette aggiunte al

sorgente devono essere trasferite a ciascuno di questi fino a raggiungere il codice oggetto. Diventa quindi necessario estendere anche questi linguaggi con costrutti opportuni affiché sia possibile tenere traccia delle label emesse. Tuttavia, quanto descritto nel paragrafo precedente non può essere applicato a Yul perché le etichette non devono essere generate ma tradotte. Perciò in questo caso è stato necessario estendere la sintassi del linguaggio con una nuova keyword `emit`. Questa, insieme con un identificatore, definisce un nuovo nodo per l'AST di Yul denominato `EmitLabel`. Nell'esempio 3.4 è riportato un estratto della traduzione in Yul della funzione `fibonacci` dell'esempio 3.3. Come si può notare le label emesse sono le stesse che compaiono nel sorgente in Solidity.

```
1  function fun_fibonacci(var_n) -> var_b {
2      emit __cost0
3      if iszero(var_n)
4      {
5          emit __cost1
6          var_b := 0x00
7          leave
8      }
9      emit __cost2
10     var_b := 0x01
11     let var_a := 0x01
12     let var_i := 0x02
13     // ...
```

Listing 3.4: Esempio di codice etichettato in Yul

3.2.2 Da Yul all'Assembly

Il passaggio finale del processo di labelling corrisponde alla traduzione da Yul all'assembly della EVM. Per quanto questo non rappresenti il vero e proprio codice oggetto della virtual machine, che ricordiamo essere il bytecode, esiste però fra i due una corrispondeza diretta. L'assembly è infatti una semplice rappresentazione testuale degli opcode e dei dati contenuti nel

bytecode, ma è chiaramente più leggibile e facilmente analizzabile. Inoltre, grazie a questo corrispondenza, è possibile effettuare il calcolo e l'attribuzione dei costi direttamente sull'assembly (come vedremo nei paragrafi §3.3 e §3.4) senza estendere in alcun modo il bytecode. L'unica modifica fatta ha riguardato la generazione del file assembly per il quale si è chiaramente reso necessario aggiungere il supporto per la traduzione del nodo `EmitLabel` di Yul in un corrispettivo in assembly. Abbiamo deciso quindi di optare, similmente a quanto fatto per Solidity, per un banale commento. Nel listato 3.5 ritroviamo un estratto della traduzione in assembly del codice Yul dell'esempio 3.4.

```
1   // emit __cost0
2   ISZERO
3   PUSH [tag] 9
4   JUMPI
5   // emit __cost1
6   PUSH 0x00
7 tag_9:
8   // emit __cost2
```

Listing 3.5: Esempio di codice etichettato in Assembly

3.3 Calcolo dei Costi

Completata la fase di labelling è necessario stabilire il gas utilizzato da ciascuna istruzione contenuta nell'assembly. Al contrario di quanto si potrebbe pensare infatti, non tutti gli opcode del bytecode consumano una quantità di gas sempre costante, come invece accade per le istruzioni riportate nella Tabella 1.1. La motivazione principale dietro a questa disparità è dovuta alla necessità, da parte degli sviluppatori di Ethereum, di fornire un bound non solo al tempo impiegato dalla virtual machine per eseguire un certo contratto ma anche allo spazio richiesto per l'esecuzione. Anche la memoria costituisce infatti una risorsa finita per i nodi della rete pertanto non è ammissibile che un contratto ne faccia un uso indiscriminato. Le istruzioni come `MSTORE`

e `CODECOPY` che interagiscono con l'area *memory* della EVM, o l'istruzione `SSTORE`, che scrive nello *storage*, è quindi evidente che consumeranno tanto più gas quanto maggiore sarà la dimensione dei dati letti, scritti, o copiati.

3.3.1 La Funzione di Costo del Gas

La funzione di costo del gas permette di stabilire la quantità di gas consumata da una specifica istruzione contenuta nel bytecode di un dato smart contract nel contesto di un preciso stato dell'esecuzione. Di seguito ne riportiamo la definizione che tuttavia, per motivi di leggibilità e brevità, non copre tutte le istruzioni del bytecode. Per le istruzioni mancanti rimandiamo a [21].

Definition 5. La generica funzione di costo del gas C è così definita:

$$C(\boldsymbol{\sigma}, \boldsymbol{\mu}, w) \equiv C_{\text{mem}}(\boldsymbol{\mu}_{\text{i}}') - C_{\text{mem}}(\boldsymbol{\mu}_{\text{i}}) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SSTORE} \\ G_{\text{verylow}} + G_{\text{copy}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w \in W_{\text{copy}} \\ C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w \in W_{\text{call}} \\ C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SELFDESTRUCT} \\ G_{\text{create}} & \text{if} \quad w = \text{CREATE} \\ G_{\text{jumpdest}} & \text{if} \quad w = \text{JUMPDEST} \\ G_{\text{zero}} & \text{if} \quad w \in W_{\text{zero}} \\ G_{\text{base}} & \text{if} \quad w \in W_{\text{base}} \\ G_{\text{verylow}} & \text{if} \quad w \in W_{\text{verylow}} \\ G_{\text{low}} & \text{if} \quad w \in W_{\text{low}} \\ G_{\text{mid}} & \text{if} \quad w \in W_{\text{mid}} \\ G_{\text{high}} & \text{if} \quad w \in W_{\text{high}} \end{cases} \tag{3.1}$$

dove:

- $\boldsymbol{\sigma}$ è lo stato globale della blockchain;
- $\boldsymbol{\mu}$ è lo stato attuale della EVM, $\boldsymbol{\mu}_s$ lo stack, e $\boldsymbol{\mu}_i$ e $\boldsymbol{\mu}_i'$ rappresentano la dimensione (in parole) della memoria rispettivamente prima e dopo l'esecuzione dell'istruzione w;
- w è un'instruzione del bytecode;

Inoltre, gli insiemi W_{zero}, W_{base}, $W_{verylow}$, W_{low}, W_{mid}, W_{high} raggruppano fra loro le istruzioni del bytecode che hanno lo stesso costo, mentre l'insieme W_{copy} contiene tutte le operazioni che copiano porzioni di memoria come `CODECOPY` (ad eccezione di `EXTCODECOPY`).

La funzione C_{mem} calcola, sulla base della dimensione attuale, di quanto sia necessario estendere la memoria per far sì che gli indirizzi a cui accederà l'istruzione siano validi. La sua definizione è la seguente:

$$C_{\text{mem}}(a) \equiv G_{\text{memory}} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor \tag{3.2}$$

Come si nota è una funzione polinomiale il cui coefficiente di grado più alto viene diviso e arrotondato per difetto e pertanto risulta lineare fino a circa 724B di utilizzo della memoria, dopodiché il suo costo aumenta e può diventare molto significativo.
Allo stesso tempo, si osserva come per alcune delle istruzioni tipicamente più utilizzate nella stesura degli smart contract la funzione di costo dipenda non solo dallo stato della EVM ma anche da quello globale in maniera tutt'altro che ovvia: è il caso per esempio di `SELFDESTRUCT` o `CALL`. Per queste istruzioni in 5 sono riportati solo i nomi delle specifiche "sottofunzioni" di costo, come $C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$, mentre la loro definizioni complete si trovano nell'appendice H di [21]. Nella Tabella 3.3 sono riportati i valori delle costanti che appaiono in 5 (la versione completa della stessa tabella è presente nell'appendice G di [21]).

3.3.2 L'approccio di Cerco2

La definizione della funzione di costo (5), specialmente per via della funzione C_{mem} (3.2) e delle "sottofunzioni" di costo di alcune istruzioni come `SSTORE`, costituisce la ragione principale per cui non è possibile, impiegando esclusivamente tecniche di analisi statica, ottenere delle stime del consumo di gas precise, a meno che non si considerino casi triviali. Ciò vale, ad esempio, anche per il compilatore di Solidity. Fra le opzioni descritte nella Tabella 3.1

Nome	Valore	Descrizione
G_{copy}	3	Pagamento parziale per le operazioni del tipo `*COPY` (da moltiplicare per il numero di parole e arrotondare)
G_{memory}	3	Quantità di gas da pagare per ogni parola aggiuntiva durante l'espansione della memoria
G_{create}	32000	Quantità di gas usata dall'operazione `CREATE`
$G_{jumpdest}$	1	Quantità di gas usata dall'operazione `JUMPDEST`
G_{zero}	0	Quantità di gas usata dalle operazioni dell'insieme W_{zero}
G_{base}	2	Quantità di gas usata dalle operazioni dell'insieme W_{base}
$G_{verylow}$	3	Quantità di gas usata dalle operazioni dell'insieme $W_{verylow}$
G_{low}	5	Quantità di gas usata dalle operazioni dell'insieme W_{low}
G_{mid}	8	Quantità di gas usata dalle operazioni dell'insieme W_{mid}
G_{high}	10	Quantità di gas usata dalle operazioni dell'insieme W_{high}

Tabella 3.3: Valori e descrizioni delle costanti nella definizione 5

vi è il flag `--gas` che permette di ottenere una stima del gas necessario per ognuna delle funzioni definite dal contratto. Il Listato 3.6 riporta l'output prodotto da solc per l'esempio 3.3 che definisce una versione iterativa della funzione di Fibonacci.

```
1 ======= examples/simple/For.sol:Fibonacci =======
2 Gas estimation:
3 construction:
4    171 + 126600 = 126771
```

```
5 external:
6    fibonacci(uint256):  infinite
```

Listing 3.6: Output del flag `--gas` di solc per la funzione di Fibonacci

Come si può notare, solc non riesce a fornire un bound al consumo di gas della funzione `fibonacci(uint256)` e restituisce pertanto `infinite`.
Per calcolare questi valori il compilatore si affida alla classe `PathGasMeter` che data un lista di istruzioni (o la definizione di una funzione) ne simula parzialmente l'esecuzione. Purtroppo però questa simulazione presenta non pochi limiti. In primis, analizzando il codice contenuto in `PathGasMeter`, è possibile osservare che la simulazione impiegata non ammette salti all'indietro, pertanto tutte le funzioni che contengono un qualche tipo di costrutto iterativo daranno come risultato `infinite`. In secondo luogo, sempre ispezionando il codice, ci si accorge che la classe `PathGasMeter` per calcolare il consumo di gas di una certa istruzione si appoggia alla classe `GasMeter`, all'interno della quale troviamo la definizione di C_{mem}. Tuttavia, anche in questo caso, è presente una limitazione: se durante la simulazione non è possibile determinare a quale parola di memoria verrà effettuato l'accesso, `GasMeter` restituisce `infinite`, in quanto risulta ovviamente impossibile calcolare la funzione C_{mem}. Infine, è importante notare che nel momento in cui il consumo di gas diventa pari a `infinite`, l'intera simulazione viene arrestata, perciò anche se fosse possibile determinare il costo di altri opcode, ciò non avviene.
Nonostante queste importanti limitazioni, la simulazione condotta dalla classe `PathGasMeter` rimane, quantomeno per alcune istruzioni, la stima più precisa che è possibile ottenere senza effettuare un'esecuzione completa. Per questo motivo abbiamo deciso di sfruttarla parzialmente per attribuire il costo alle istruzioni. Fra i file richiesti da Cerco2 al compilatore è infatti presente anche una versione dell'assembly (ottenuta col flag `--asm-costs`) che riporta per ogni opcode il consumo di gas calcolato da `PathGasMeter`. Successivamente, durante la fase di parsing dell'assembly, Cerco2 assegna a tutte le istruzioni prive di costo, il loro valore numerico se costante (è il caso

per esempio delle istruzioni aritmetiche) oppure una forma simbolica rappresentata da una particolare stringa. Questa forma simbolica corrisponde pressoché alla definizione della funzione di costo del gas (5) dell'istruzione in questione. Per i dettagli relativi a come questa particolare forma venga poi gestita dalle fasi di attribuzione dei costi e instrumentazione si faccia riferimento ai paragrafi dedicati (rispettivamente §3.4 e §3.5). Appare comunque evidente come questo approccio non risolva il problema poiché non permette di ottenere una stima reale del consumo di gas e la sua utilità è quindi, almeno attualmente, limitata esclusivamente a fornire allo sviluppatore alcune indicazioni in merito a quali siano le istruzioni che influenzano maggiormente il costo del programma. Al termine della trattazione verranno presentati alcuni possibili miglioramenti che consentano di trasformare queste indicazioni in vere e proprie stime dell'utilizzo di gas.

3.4 Attribuzione dei Costi

Una volta determinato il consumo di gas di ciascuna istruzione, a prescindere che questo sia dato in forma numerica o simbolica, è necessario stabilire a quali etichette associarlo. Intuitivamente possiamo dire che il costo di una label è pari alla somma dei consumi di gas di tutti gli opcode che compaiono fra questa e la successiva nel codice assembly. Tuttavia, questo approccio risulta insufficiente a causa di due problemi fondamentali: il primo è legato alla presenza delle istruzioni di salto nel bytecode della EVM, mentre il secondo si deve alla modalità in cui gli smart contracts vengono salvati sulla blockchain e a come le funzioni `public` ed `external` vengono successivamente invocate.

3.4.1 Il Controllo di Flusso

Così come tutti i linguaggi di basso livello, anche il bytecode di Ethereum contiene alcune istruzioni per permettere l'utilizzo dei salti. Queste sono `JUMPI`, per i salti condizionali, e `JUMP`, per saltare incondizionatamente e su di esse si basa il controllo di flusso del bytecode sul quale viene mappato quello

dei linguaggi di alto livello come Solidity. Purtroppo però la presenza dei salti rende impossibile determinare con precisione quali istruzioni si trovino realmente dopo una certa etichetta perché per farlo bisognerebbe conoscere il flusso dell'esecuzione. Questo non è chiaramente riportato nel file emesso dal compilatore e va quindi calcolato a parte tipicamente costruendo una struttura dati particolare denominata *Control Flow Graph*.

Definition 6. Dato un qualche programma P, si definisce *Control Flow Graph* il grafo orientanto $G = (N, E)$ dove ciascun nodo $n \in N$ è un blocco del programma P e ogni arco $(n_i, n_j) \in E$ rappresenta il passaggio del flusso di esecuzione dal nodo n_i al nodo n_j.

Nel caso dell'assembly, per il quale non è ovviamente possibile parlare di blocchi nel senso normalmente inteso per i linguaggi di alto livello, i nodi possono essere rappresentati dalla sequenza di instruzioni che precede un'operazione di salto condizionale (`JUMPI`) compresa quest'ultima. In Cerco2 l'implementazione del CFG è realizzata mediante un array di opcode il cui ultimo elemento è, per l'appunto, un'istruzione `JUMPI`. Ogni elemento dell'array è un oggetto JavaScript contente, oltre al nome dell'opcode, il numero della linea alla quale si trova l'istruzione nel sorgente assembly e, se previsti, gli argomenti dell'opcode. Per l'istruzione `PUSH`, ad esempio, troveremo l'elemento da inserire sullo stack della EVM. Gli oggetti che corrispondono all'istruzione `JUMPI` possiedono inoltre altre due proprietà, `true` e `false`, contenenti le operazioni che è possibile trovare seguendo l'uno o l'altro ramo della computazione e rappresentano quindi gli archi del CFG. L'algoritmo per la costruzione del grafo, che si trova all'interno del file `src/asm/controlGraph/create.js` in [15], opera quindi in maniera molto semplice: data in input una lista di istruzioni ottenute processando il file assembly generato dal compilatore, le esamina ad una ad una e la aggiunge ad un array locale fino a quando non si raggiunge un'istruzione che termina il flusso dell'esecuzione, come `RETURN` o `REVERT`, o un'istruzione di salto condizionato. Nel primo caso l'algoritmo termina restituendo l'array creato, che rappresenta così un ramo dell'esecuzione. Se incontra un `JUMPI`, al contrario, l'algoritmo procede ricorsivamente

sui due possibili rami alternativi della computazione. L'istruzione di salto funziona in modo analogo a quella di altri linguaggi di basso livello, perciò le istruzioni seguenti si trovano o subito dopo il salto (se la condizione è falsa) o di seguito ad un qualche *tag* (se la condizione è vera). Pertanto, mentre la chiamata ricorsiva sul ramo "falso" potrà proseguire direttamente dall'istruzione successiva al salto, non vale lo stesso per quella corrispondente al ramo "vero", in quanto è necessario conoscere la destinazione del salto. Per questo motivo l'algoritmo mantiene uno stack interno dei tag e ogni volta che incontra un'istruzione del tipo `PUSH [tag] n` aggiunge il numero `n` del tag alla pila. In seguito ad un salto, l'elemento in cima alla pila viene rimosso e si cerca nella lista delle istruzioni quella che definisce il tag. Una volta individuata l'esecuzione prosegue esaminando l'istruzione in questione e quelle a lei successive fino al salto seguente. Questo meccanismo non si applica solamente al ramo "true" dei salti condizionati ma risulta fondamentale anche per i salti non condizionati, per quanto in quel caso il percorso possibile sia solamente uno.

La procedura fin qui descritta presenta però un problema. Se nel codice sorgente sono presenti cicli con iterazione indeterminata o ricorsione allora questi verranno probabilmente tradotti con salti all'indietro. In questi casi l'algoritmo divergerà poiché non ha modo di determinare se e quando l'iterazione avrà termine. Tuttavia, ai fini della costruzione del CFG non è necessario ripercorrere più di una volta lo stesso salto in quanto sappiamo già come proseguirà l'esecuzione. Pertanto è sufficiente che l'algoritmo tenga traccia delle istruzioni già visitate evitando così di percorrere più volte il medesimo ramo dell'esecuzione. Infine, se allo stack delle destinazioni viene aggiunto un tag che è già stato esplorato, questo verrà semplicemente scartato in favore del primo non visitato o del termine della computazione se tutti i tag presenti risultassero già visitati. La struttura dati risultante corrisponde così più ad un albero che ad un grafo ma è comunque sufficiente per poter determinare quali istruzioni sono coperte da una certa label. Nel seguito continueremo comunque per semplicità a riferirci ad esso col nome di

Control Flow Graph.

3.4.2 Il Costo dell'Inizializzazione

Per comprendere cosa intendiamo con *costo dell'inizializzazione*, si consideri il codice riportato nell'esempio seguente che definisce un banale smart contract con due funzioni pubbliche.

```
contract SimpleStorage {
    uint256 storedData;

    function set(uint256 x) public {
        storedData = x;
    }

    function get() public view returns (uint256) {
        return storedData;
    }
}
```

Listing 3.7: Esempio di smart contract in Solidity

Il suo equivalente Yul, ottenuto tramite traduzione e compilazione via solc, è riportato di seguito, parzialmente modificato per ragioni di brevità.

```
object "SimpleStorage_24" {
    code {
        mstore(64, 128)
        if callvalue() { revert(0, 0) }
        let _1 := datasize("SimpleStorage_24_deployed")
        codecopy(128, dataoffset("SimpleStorage_24_deployed")
            , _1)
        return(128, _1)
    }
    object "SimpleStorage_24_deployed" {
        code {
            mstore(64, 128)
            if iszero(lt(calldatasize(), 4)) {
```

```
13                  switch shr(224, calldataload(0))
14                  case 0x60fe47b1 {
15                      if callvalue() { revert(0, 0) }
16                      if slt(add(calldatasize(), not(3)), 32) {
                            revert(0, 0) }
17                      let var_x := calldataload(4)
18                      emit __cost0
19                      sstore(0, var_x)
20                      return(128, 0)
21                  }
22                  case 0x6d4ce63c {
23                      if callvalue() { revert(0, 0) }
24                      if slt(add(calldatasize(), not(3)), 0) {
                            revert(0, 0) }
25                      emit __cost1
26                      return(128, add(abi_encode_uint256(sload
                            (0)), not(127)))
27                  }
28              }
29              revert(0, 0)
30          }
31      }
32  }
```

Listing 3.8: Versione Yul dello smart contract in 3.7

Grazie alle label `__cost0` e `__cost1` possiamo individuare le parti di codice che corrispondono direttamente al sorgente in Solidity. Come si può notare però, gran parte del codice presente nel frammento in Yul non ha un chiaro corrispettivo in Solidity ed è stato generato dal compilatore. Scopo di questo codice aggiuntivo è permettere il **deploy** del contratto (righe da 3 a 7), ovvero il suo caricamento all'interno della blockchain, e la corretta invocazione delle funzioni `set` (righe da 15 a 20) e `get` (righe da 23 a 26), per esempio controllando che gli argomenti passati siano in numero giusto e siano stati correttamente codificati. Queste righe costituiscono un problema non indifferente poiché essendo generate dal compilatore successivamente rispetto al

processo di labelling, come si nota, non sono coperte da alcuna etichetta ma vengono eseguite ad ogni chiamata e non considerarle significherebbe quindi produrre annotazioni di costo *unsound*.

Per risolvere questo problema la procedura per l'assegnazione dei costi alle label (che si trova all'interno del file `src/asm/costs.js` in [15]) definisce una variabile interna `startupCost` proprio con lo scopo di tenere traccia dei costi legati all'inizializzazione. Lo pseudocodice dell'algoritmo completo è riportato nel Listato 3.9.

```
1  fn assignCosts(cfg, labelStack = [], startupCost = 0, skip =
       false) {
2    for i in cfg {
3      if i.type == "LABEL" {
4        if !i.visited {
5          skip = false;
6          costs[i.label] = 0
7        } else {
8          skip = true;
9        }
10
11       if labelStack.empty() {
12         costs[i.label] += startupCost;
13         startupCost = nil;
14       }
15
16       labelStack.push(i.label);
17       i.visited = true
18     } else if !skip {
19       if labelStack.empty() {
20         startupCost += i.gas
21       } else {
22         costs[labelStack.top] += i.gas;
23       }
24     }
25   }
26
27   if (cfg.last.type === "JUMPI") {
```

```
28          jumpi = cfg.last;
29          assign_costs(jumpi.trueBranch, labelStack,
                startupCost);
30          assign_costs(jumpi.falseBranch, labelStack,
                startupCost);
31      }
32  }
```

Listing 3.9: Pseudocodice dell'algoritmo per l'assegnazione dei costi alle etichette

La procedura `assignCosts` conduce una visita in profondità (DFS) del CFG, chiamandosi ricorsivamente sui due rami dell'istruzione `JUMPI`. Durante la visita l'algoritmo tiene traccia delle label via via incontrate in uno stack, in modo che la label corrente si trovi sempre sulla cima. Inoltre, se l'etichetta corrente è già stata visitata (ad esempio perché parte di una funzione `private` che l'ottimizzatore ha sottoposto a *inlining*) allora il suo costo è già stato calcolato e pertanto è possibile ignorare tutte le istruzioni seguenti fino alla prossima label. Questa operazione avviene attraverso l'uso della variabile `skip`. Ciò nonostante, le label ignorate vengono comunque aggiunte allo stack. Questo è necessario per far funzionare il meccanismo dello `startupCost`. Fin quando la visita non raggiunge un'etichetta infatti `labelStack` è vuoto, poiché come detto le istruzioni per l'inizializzazione non sono coperte da alcuna label. Il loro consumo di gas viene quindi sommato alla variabile `startupCost` fino a quando non si incontra la prima label, alla quale il costo accumulato viene infine assegnato. Per garantire che ciò si verifichi solo una volta è quindi fondamentale che tutte le etichette visitate siano aggiunte alla pila ma non vengano mai rimosse.
Completata la visita, la variabile di tipo dizionario `costs`, che si suppone sia globale, contiene le associazioni *etichetta-costo* per ciascuna delle label raggiunte da una possibile computazione del contratto (ricordiamo infatti che le label vengono generate dal compilatore nella fase di parsing ma potrebbero essere eliminate dalla fase di ottimizzazione se appartenti a *dead code*).

3.5 Instrumentazione

L'ultima fase del metodo del labelling consiste, come riportato nel paragrafo §2.2, nell'operazione di *instrumentazione* che, dato in input un programma scritto nel linguaggio sorgente etichettato $L_{1,l}$, genera in output un programma funzionalmente equivalente a quello originale ma capace di monitorare il costo della sua esecuzione. L'instrumentazione infatti non altera in alcun modo il comportamento del programma originale poiché le modifiche effettuate si limitano a:

1. L'aggiunta di una porzione di codice che permetta di definire ed accedere ad una qualche variabile di costo

2. La sostituzione delle label contenute nel sorgente etichettato, con incrementi progressivi della variabile di costo sopracitata di valore pari al costo associato alla label rimpiazzata

Nella nostra implementazione il punto 1 è affidato al compilatore che è stato quindi opportunamente modificato per emettere una versione parzialmente instrumentata del codice sorgente. Nel Listato 3.10 è riportato l'output corrispondente al sorgente del Listato 3.7.

```
1  contract SimpleStorageInstrumented {
2      uint256 storedData;
3
4      function set(uint256 x) public {
5          /* __cost0 */
6          storedData = x;
7      }
8
9      // NOTE: was a view function
10     function get() public returns (uint256) {
11         /* __cost1 */
12         return storedData;
13     }
14
15     // Instrumentation Machinery
```

```
16      uint256 private __costAcc = 0;
17      function __SimpleStorage_getCost() public view returns(
            uint256) {
18          return __costAcc;
19      }
20      function __SimpleStorage_resetCost() public {
21          __costAcc = 0;
22      }
23  }
```

Listing 3.10: Instrumentazione parziale di 3.7

Come si nota viene definita una variabile di stato del contratto denominata `__costAcc`. La variabile è dichiarata `private` per evitare che si generino dei conflitti nel caso in cui all'interno dello stesso file sorgente vengano definiti due contratti e uno di questi erediti dall'altro. Solidity infatti non permette la ridichiarazione di una variabile di stato in un contratto derivato se questa è già presente nel contratto base, a meno che entrambe le variabili non siano dichiarate come `private`. Allo stesso modo le funzioni `getCost` e `resetCost`, necessarie per poter accedere a `__costAcc` e resettarla fra un'esecuzione e la successiva, contengono nel nome anche quello del contratto originale (non instrumentato) per evitare che siano sovrascritte negli eventuali contratti derivati. Infine la procedura di instrumentazione rimuove i modificatori `pure` e `view` dalle definizioni delle funzioni poiché questi risultano chiaramente incompatibili con gli aggiornamenti della variabile di stato `__costAcc` con cui le label verranno sostituite. Questa modifica non altera in alcun modo la bontà dell'instrumentazione in quanto non influisce sul comportamento del programma ma solo sull'insieme di ottimizzazioni che il compilatore può applicare e questo, nel caso del codice instrumentato, non ha nessuna importanza.

Questo approccio per quanto possa apparire complesso permette di semplificare la realizzazione del punto 2, la cui implementazione è parte di Cerco2. Il compilatore non conosce infatti i costi che sono stati assegnati a ciascuna label e pertanto non può realizzare un'instrumentazione completa. Cerco2

prende quindi in input il file parzialmente instrumentato generato dal compilatore e sostituisce i commenti contenenti le label con opportuni incrementi della variabile di costo `__costAcc`. Tuttavia, come sappiamo dal paragrafo §3.3.2, il costo di ciascuna etichetta potrebbe contenere anche elementi simbolici che vengono quindi riportati nell'output finale (Listato 3.11) sotto forma di commenti in aggiunta agli incrementi della variabile `__costAcc`.

```
1  contract SimpleStorageInstrumented {
2      uint256 storedData;
3
4      function set(uint256 x) public {
5          /* __cost0 */
6          __costAcc = __costAcc + 153; /* + sstore(0, x) */
7          storedData = x;
8      }
9
10     // NOTE: was a view function
11     function get() public returns (uint256) {
12         /* __cost1 */
13         __costAcc = __costAcc + 1029; /* + memaccess(128, add
               (not(127),abi_encode_uint256(sload(0)))) */
14         return storedData;
15     }
16
17     uint256 private __costAcc = 0;
18     function __SimpleStorage_getCost() public view returns(
           uint256) {
19         return __costAcc;
20     }
21     function __SimpleStorage_resetCost() public {
22         __costAcc = 0;
23     }
24 }
```

Listing 3.11: Instrumentazione di 3.7

Le funzioni top level in questi commenti, ovvero quelle che non appaiono mai come argomenti di altre funzioni, sono:

- `memaccess(offset,size)`: rappresenta un'operazione di accesso alla memoria di dimensione pari a `size` a partire dalla posizione `offset`.
- `sizeof(var)`: corrisponde all'operazione di calcolo della dimensione in *parole di memoria* di `var`.
- `sstore`: analoga dell'omonimo opcode del bytecode.
- `exists(account)`: corrisponde all'operazione di verifica dell'esistenza di `account` all'interno della blockchain.

Il loro scopo è fornire delle indicazioni allo sviluppatore sull'origine dei consumi di gas di una certa porzione di codice quando non risulta possibile determinarli completamente. Per questo motivo, se le loro chiamate appaiono nei commenti a fianco degli aggiornamenti della variabile `__costAcc`, la procedura di instrumentazione aggiunge al file generato anche le corrispettive definizioni in pseudocodice di queste funzioni. La definizione della funzione `memaccess(offset,size)`, per esempio, è riportata di seguito e corrisponde pressoché alla sottrazione fra $C_{mem}(\boldsymbol{\mu}'_i)$ e $C_{mem}(\boldsymbol{\mu}_i)$ nell'espressione della funzione di costo del gas (5).

```
1  def memaccess(offset, length):
2      # access with zero length will not extend the memory
3      if length == 0:
4        return
5
6      newMemorySize = words(offset + length)
7      if newMemorySize < currentMemorySize:
8        return; # do nothing if we have enough memory
9
10     cost = newMemorySize * 3 +
11       newMemorySize * newMemorySize / 512
12     if cost > currentHighestMemCost:
13       # consume an amount of gas equal to cost -
             currentHighestMemCost
14       currentHighestMemCost = cost
15
```

```
16      currentMemorySize = newMemorySize
```

Listing 3.12: Pseudocodice della funzione `memaccess(offset,size)`

3.5.1 Funzioni `public` e Costo dell'Inizializzazione

Il codice instrumentato riportato nel Listato 3.11 per quanto sia corretto in relazione al sorgente originale non lo è però in generale. Se infatti modificassimo il sorgente originale aggiungendo una chiamata alla funzione `get` all'interno della funzione `set`, come evidenziato nel codice del Listato 3.13, allora l'instrumentazione precedente non sarebbe più corretta.

```
1  contract SimpleStorage {
2      uint256 storedData;
3
4      function set(uint256 x) public {
5          get(); // aggiunta della chiamata a get()
6          storedData = x;
7      }
8      // -- get() --
9  }
```

Listing 3.13: Parziale modifica di 3.7

Quanto detto non dipende direttamente dalla modifica effettuata. Se la funzione `get` fosse stata dichiarata `private` infatti allora l'instrumentazione sarebbe ancora valida. Questo perché le funzioni `private`, insieme con quelle `internal`, non possono essere chiamate esternamente (ossia attraverso una transazione) ma solamente da altri metodi dello stesso contratto o al più da metodi appartenenti a contratti derivati (nel caso delle funzioni `internal`). Pertanto il costo dell'inizializzazione può essere pagato solo dalle funzioni `public` o `external`. Tuttavia, anche questi metodi possono essere invocati internamente ma mentre le chiamate alle funzioni `external` avvengono attraverso una transazione, quelle ai metodi `public` attraverso un semplice salto, esattamente come per le funzioni `private` e `internal`. Chiaramente in quest'ultimo caso non è necessario sostenere alcun costo per il controllo

dei paramentri della transazione o la decodifica degli argomenti, ma nell'instrumentazione nel Listato 3.11 questi sono inclusi comunque nel costo del metodo `get`. Per risolvere questo problema abbiamo modificato la procedura di instrumentazione parziale del compilatore in modo da generare, per ciascun metodo `public`, una coppia di metodi. Il primo esattamente identico all'originale ma con visibilità ristretta a `internal`, così da garantirne comunque l'accesso agli eventuali contratti derivati. Il secondo invece con un nome del tipo `<metodo-originale>_external`, visibilità `external` e corpo contente esclusivamente una chiamata al metodo originale e la prima label che appare nel corpo di quest'ultimo ma modificata con il nome seguente `<label-originale>.startup`. In seguito, abbiamo chiaramente corretto la procedura di sostituzione delle etichette di Cerco2 per far sì che le label con associato, oltre al costo "proprio", anche quello dell'inizializzazione venissero gestite correttamente. Per queste etichette infatti l'instrumentazione corretta assegna il costo dell'inizializzazione alla label con suffisso `startup` mentre quello "proprio" viene assegnato alla label originale. Nel Listato 3.14 è possibile osservare il risultato finale di questo processo, che riporta quindi l'instrumentazione corretta.

```
1  contract SimpleStorageInstrumented {
2      uint256 storedData;
3
4      function set(uint256 x) internal {
5          /* __cost0 */
6          __costAcc = __costAcc + 14; /* + sstore(0, x) */
7          storedData = x;
8      }
9
10     function set_external(uint256 x) external {
11         /* __cost0.startup */
12         __costAcc = __costAcc + 139;
13         set(x);
14     }
15
16     // NOTE: was a view function
```

```
17      function get() internal returns (uint256) {
18          /* __cost1 */
19          __costAcc = __costAcc + 874; /* + memaccess(128, add(
                not(127),abi_encode_uint256(sload(0)))) */
20          return storedData;
21      }
22
23      function get_external() external returns (uint256) {
24          /* __cost1.startup */
25          __costAcc = __costAcc + 155;
26          return get();
27      }
28
29      // -- definizioni funzioni getCost e resetCost
30  }
```

Listing 3.14: Instrumentazione corretta di 3.7

Capitolo 4

Testing

In questo capitolo presenteremo e discuteremo il risultato dell'applicazione della nostra implementazione del metodo del labelling ad alcuni esempi di smart contracts in Solidity. Gli esempi che tratteremo non rappresentano smart contract reali ma piuttosto casi costruiti ad-hoc col solo scopo di mostrare l'output generato ed evidenziare come questo sia in linea con le aspettative. Chiaramente testare l'implementazione significa molto di più che limitarsi ad osservare che gli output prodotti sembrino in qualche modo sensati. Allo stato attuale però, la nostra implementazione presenta alcune limitazioni importanti e tali da rendere vano e improprio qualunque tentativo di test diverso da quello proposto. Di queste la principale, soprattutto nell'ottica di un testing più approfondito, è senza dubbio quella relativa all'impossibilità di calcolare correttamente il consumo di gas di ciascuna delle istruzioni del bytecode e che abbiamo già parzialmente illustrato nel paragrafo §3.3. In futuro, una volta che questa e le altre limitazioni dell'attuale implementazione saranno state risolte, sarà senza dubbio possibile impiegare un qualche framework per il testing automatico di smart contracts, come Truffle[12], in maniera da porre a confronto i consumi di gas dei contratti originali, riportati dalla test suite, e quelli ottenuti tramite l'impiego del codice instrumentato in modo da verificare la correttezza (anche se non a livello formale) dell'implementazione e la bontà delle annotazioni di costo.

4.1 Costanti e Variabili immutable

Nell'esempio seguente sono presenti alcune variabili costanti, che vengono inizializzate dal codice che effettua il deploy del contratto sulla blockchain, e altre `immutable`, che vanno invece inizializzate dall'autore del contratto nel costruttore.

```
1  uint256 constant X = 32**22 + 8;
2
3  contract ConstAndImmutable {
4      string constant TEXT = "abc";
5      bytes32 constant MY_HASH = keccak256("abc");
6      uint256 immutable decimals;
7      uint256 immutable maxBalance;
8      address immutable owner = msg.sender;
9
10     constructor(uint256 _decimals, address _reference) {
11         decimals = _decimals;
12         maxBalance = _reference.balance;
13     }
14
15     function isBalanceTooHigh(address _other) public view
           returns (bool) {
16         return _other.balance > maxBalance;
17     }
18 }
```

Listing 4.1: Esempio Solidity con variabili `const` e `immutable`

La rispettiva instrumentazione è riportata nel Listato 4.2. Come ci aspettavamo, il costo legato all'assegnamento dei valori alle variabili `immutable` è stato associato alla label `__cost0` all'interno del costruttore, come testimoniato dalle operazioni `memaccess(128, sizeof(decimals))` e `memaccess(160, sizeof(balance(var_reference)))`. Inoltre, grazie alla suddivisione del metodo `isBalanceTooHigh` nelle due versioni `internal` e `external`, possiamo osservare come il costo della label `__cost1` sia probabilmente molto vicino a quello reale. Il consumo di gas dell'operazione `BALANCE`, nella quale

l'espressione `_other.balance` si traduce, è infatti pari a 700. In aggiunta ad essa a determinare significativamente il costo della label `__cost1` vi è poi solamente un accesso alla memoria ma ad una locazione precedentemente inizializzata dal costruttore e che pertanto, non necessitando di espandere la memoria, avrà un costo pari a 0.

```
1  uint256 constant X = 32**22 + 8;
2
3  contract ConstAndImmutableInstrumented {
4      string constant TEXT = "abc";
5      bytes32 constant MY_HASH = keccak256("abc");
6      uint256 immutable decimals;
7      uint256 immutable maxBalance;
8      address immutable owner = msg.sender;
9
10     constructor(uint256 _decimals, address _reference) {
11         /* __cost0 */
12         __costAcc = __costAcc + 257; /* + memaccess(128,
               sizeof(decimals)) + memaccess(160, sizeof(balance(
               var_reference))) + memaccess(_1, _2) + 3 * sizeof(
               _2) + memaccess(_1, _2) + memaccess(224, argSize)
               + 3 * sizeof(argSize) + memaccess(192, sizeof(
               caller())) */
13         decimals = _decimals;
14         maxBalance = _reference.balance;
15     }
16
17     // NOTE: was a view function
18     function isBalanceTooHigh(address _other) internal
           returns (bool) {
19         /* __cost1 */
20         __costAcc = __costAcc + 731; /* + memaccess(128,
               sizeof(gt(loadimmutable(),expr))) */
21         return _other.balance > maxBalance;
22     }
23
24     function isBalanceTooHigh_external(address _other)
           external returns (bool) {
```

```
25          /* __cost1.startup */
26          __costAcc = __costAcc + 193;
27          return isBalanceTooHigh(_other);
28      }
29
30      // -- __costAcc, getCost(), resetCost() omessi --
31  }
```

Listing 4.2: Instrumentazione di 4.1

4.2 Blocchi checked e unchecked

In questo secondo esempio è presente la stessa funzione ma in due versione leggermente differenti. Dalla versione 0.8.0 di Solidity, infatti, tutte le operazioni aritmetiche causano un revert della transazione in caso di overflow o underflow. Ciò accade di default, a meno che l'operazione in questione non venga inserita all'interno di un blocco `unchecked`. L'uso di questo blocco evita che il compilatore generi il codice necessario a gestire l'eventuale underflow o overflow e ad effettuare il revert della transazione.

```
1  contract CheckedUnchecked {
2      function f(uint256 a, uint256 b) public view returns (
           uint256) {
3          return a * (b + 42) + block.timestamp;
4      }
5
6      function f_unchecked(uint256 a, uint256 b) public view
           returns (uint256) {
7          unchecked {
8              return a * (b + 42) + block.timestamp;
9          }
10     }
11 }
```

Listing 4.3: Esempio Solidity con blocchi `unchecked` e non

Per questo motivo nel corrispondente codice instrumentato, il costo delle label __cost0 e __cost2 è sensibilmente diverso, esattamente come ci saremmo aspettati.

```
1  contract CheckedUncheckedInstrumented {
2      // NOTE: was a view function
3      function f(uint256 a, uint256 b) internal returns (
           uint256) {
4          /* __cost0 */
5          __costAcc = __costAcc + 389; /* + memaccess(memPos,
               sub(memPos,abi_encode_uint256(ret,memPos))) */
6          return a * (b + 42) + block.timestamp;
7      }
8
9      function f_external(uint256 a, uint256 b) external
           returns (uint256) {
10         /* __cost0.startup */
11         __costAcc = __costAcc + 163;
12         return f(a, b);
13     }
14
15     // NOTE: was a view function
16     function f_unchecked(uint256 a, uint256 b) internal
           returns (uint256) {
17         /* __cost1 */
18         __costAcc = __costAcc + 0;
19         unchecked {
20             /* __cost2 */
21             __costAcc = __costAcc + 142; /* + memaccess(
                   memPos_1, sub(memPos_1,abi_encode_uint256(
                   ret_1,memPos_1))) */
22             return a * (b + 42) + block.timestamp;
23         }
24     }
25
26     function f_unchecked_external(uint256 a, uint256 b)
27         external
28         returns (uint256)
```

```
29      {
30          /* __cost1.startup */
31          __costAcc = __costAcc + 243;
32          return f_unchecked(a, b);
33      }
34
35      // -- __costAcc, getCost(), resetCost() omessi --
36  }
```

Listing 4.4: Instrumentazione di 4.3

Conclusioni

Abbiamo presentato e descritto in dettaglio il metodo del *labelling*, proposto dal progetto CerCo [17], per realizzare compilatori capaci di riportare informazioni sul costo dell'esecuzione del codice oggetto all'interno del codice sorgente e lo abbiamo applicato con successo a *solc*, il compilatore del linguaggio per smart contracts Solidity. Purtroppo però, la nostra implementazione presenta attualmente due importanti limitazioni. La prima riguarda l'operazione di instrumentazione che, allo stato attuale, non gestisce correttamente le librerie in quanto al loro interno non è possibile dichiarare variabili di stato. La seconda invece, come osservato nel paragrafo §3.3.2, riguarda il calcolo dei consumi di gas delle singole istruzioni del codice oggetto. Abbiamo infatti appurato come la definizione della funzione di costo del gas non permetta di ottenere dei valori esatti per tutte le istruzioni del bytecode. Tuttavia, ottenere delle stime più accurate, almeno per quanto concerne l'uso della memoria, permetterebbe di rimuovere le attuali rappresentazioni simboliche in favore di valori numerici o funzioni di costo sugli input. In letteratura, sono stati recentemente presentati alcuni lavori volti ad affrontare questo problema, come [16] e [1]. In particolare, il lavoro svolto in [1], ha permesso la realizzazione di uno strumento simile a quello presentato in questo elaborato. Sviluppi futuri di questo lavoro potrebbero quindi estendere l'odierna implementazione attraverso uno o più degli approcci citati e migliorando la fase di instrumentazione, così da ottenere un *cost annotating compiler* completo per il linguaggio Solidity.

Bibliografia

[1] Elvira Albert et al. "GASTAP: A Gas Analyzer for Smart Contracts". In: *CoRR* abs/1811.10403 (2018). arXiv: 1811.10403. URL: http://arxiv.org/abs/1811.10403.

[2] Roberto M. Amadio et al. *Certifying cost annotations in compilers.* 2010. arXiv: 1010.1697 [cs.PL].

[3] *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, ago. 2006. ISBN: 0321486811. URL: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%5C&path=ASIN/0321486811.

[4] Silvia Crafa, Matteo Di Pirro e Elena Zucca. "Is Solidity Solid Enough?" In: *Financial Cryptography and Data Security.* A cura di Andrea Bracciali et al. Cham: Springer International Publishing, 2020, pp. 138–153. ISBN: 978-3-030-43725-1.

[5] Silvia Crafa e Matteo Di Pirro. "Solidity 0.5: when typed does not mean type safe". In: *CoRR* abs/1907.02952 (2019). arXiv: 1907.02952. URL: http://arxiv.org/abs/1907.02952.

[6] Ethereum Developers. *Ethereum Website.* URL: https://ethereum.org/.

[7] Ethereum Developers. *Gas Costs from Yellow Paper – EIP-150 Revision.* URL: https://docs.google.com/spreadsheets/d/1n6mRqkBz3iWcOlRem_mO09GtSKEKrAsfO7Frgx18pNU/edit#gid=0.

[8] Remix Developers. *Remix IDE.* URL: https://remix.ethereum.org/.

[9] Solidity Developers. *Solidity Documentation*. URL: https://docs.soliditylang.org/.

[10] Solidity Developers. *Solidity Source Code*. URL: https://github.com/ethereum/solidity/.

[11] Solidity Developers. *Yul Documentation*. URL: https://docs.soliditylang.org/en/latest/yul.html.

[12] Truffle Developers. *Truffle Suite Website*. URL: https://www.trufflesuite.com/.

[13] Inc. Free Software Foundation. *GCC Official Website*. URL: https://gcc.gnu.org/.

[14] M. Gabbrielli e S. Martini. *Linguaggi di programmazione. Principi e paradigmi*. Collana di istruzione scientifica. McGraw-Hill Companies, 2011. ISBN: 9788838665738. URL: https://books.google.it/books?id=w5rJYgEACAAJ.

[15] Mattia Guazzaloca. *Codice sorgente del tool Cerco2*. URL: https://github.com/mettz/thesis.

[16] Matteo Marescotti et al. "Computing Exact Worst-Case Gas Consumption for Smart Contracts". In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. A cura di Tiziana Margaria e Bernhard Steffen. Cham: Springer International Publishing, 2018, pp. 450–465. ISBN: 978-3-030-03427-6.

[17] CerCo Members. *CerCo Website*. URL: http://cerco.cs.unibo.it/.

[18] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: *Cryptography Mailing list at https://metzdowd.com* (mar. 2009).

[19] LLVM Project. *LLVM IR*. URL: https://llvm.org/docs/LangRef.html.

[20] Linus Torvalds. *Linus Torvalds talk "Nothing better than C"*. URL: https://www.youtube.com/watch?v=CYvJPra7Ebk.

[21] Daniel Davis Wood. “Ethereum: a Secure Decentralised Generalised Transaction Ledger”. In: 2014.

Ringraziamenti

Ringrazio il professor Sacerdoti Coen per la disponibilità e il costante supporto ricevuto durante lo sviluppo di questo elaborato di tesi. Ringrazio la mia famiglia, in modo particolare mia sorella, per avermi sopportato in queste settimane di lavoro intenso necessario a completarla. Ringrazio infine tutti gli amici, colleghi e non, che in questi anni mi hanno aiutato, più o meno direttamente, a portare a termine questo percorso.