Indice

Capitolo 1

Introduzione

Da sempre, un elemento portante della società è l'idea di scambio di risorse, che ha condizionato ogni epoca. Normalmente concludere uno scambio ha sempre previsto la figura di un garante che impone delle regole affinché lo scambio sia valido. Per la prima volta nella storia si hanno a disposizione i mezzi per eliminare questa figura, perché si ha un sistema in cui l'esistenza stessa di un contratto valida la transazione.

Questi tipi di contratti vengono definiti *smart contract*, e devono garantire le proprietà di fiducia, affidabilità e di sicurezza, che in precedenza erano delegate al garante, ma che adesso diventano possibili grazie alle blockchain. Queste ultime sono code nelle quali è consentita l'operazione di lettura, mentre l'unica operazione di scrittura è l'aggiunta (o transazione): non è possibile quindi modificare un elemento già presente nella blockchain; di conseguenza viene mantenuto un registro con la storia di tutte le transazioni eseguite. Inoltre questa struttura dati è distribuita, quindi è parte di una rete peer-to-peer in cui viene replicata per ogni nodo. Di conseguenza gli *smart contract*, se memorizzati nella blockchain, diventano programmi che possono essere eseguiti in modo distribuito e sicuro, e che controllano lo scambio di denaro tra diverse parti.

A livello di programmazione uno *smart contract* è definito da un identificativo, da uno stato – cioè i dati – e da del codice, inteso come insieme di metodi che modificano lo stato del contratto ed eseguono transazioni. Per implementare *smart contract* sono messi a disposizioni linguaggi di programmazione dedicati, a ognuno dei quali è associata la relativa blockchain, ad esempio troviamo il linguaggio Solidity per la blockchain Ethereum o Liquidity per Tezos.

Siccome possono gestire considerevoli somme di denaro, il codice dei contratti è codice critico: diventa quindi interessante fare analisi statica sul comportamento degli *smart contract*. In merito sono stati fatti diversi lavori; di questi ne è particolarmente interessante uno in cui emerge che, nel momento in cui un utente razionale agisce per minimizzare le perdite di denaro (o equivalentemente

per massimizzare il guadagno), il codice dello *smart contract* lo può forzare a determinate decisioni. In questo contesto per "forzare la decisione" si intende che da parte dell'utente l'aderenza al contratto avviene – perché se non ci fosse l'utente interessato perderebbe dei soldi. Questa analisi è stata fatta definendo un linguaggio, chiamato `scl` (smart calculus), con un ristretto insieme di funzioni sugli *smart contract*, che permette di modellare il comportamento di contratti e utenti (umani). `Scl` è un linguaggio ad attori minimale, che rende più semplice l'analisi di *smart contract* – ma allo stesso tempo espressivo abbastanza da essere Turing completo: permette l'invocazione di metodi, la modifica dei campi, il comportamento condizionale, la ricorsione, il sollevamento di eccezioni. Può essere scritto un programma in `scl` che corrisponde a un modello dove gli utenti e i contratti interagiscono tra loro. L'analisi prevede che il comportamento di umani e contratti nel sistema viene tradotto un'unica formula logica nell'aritmetica di Presburger e risolvendola si studia il comportamento per cui ciascun utente cerca di massimizzare il profitto.

Si è quindi realizzato che questa analisi non aveva a disposizione tutti gli strumenti necessari: il mio lavoro è stato costruire dei tool per permettere di completarla. Verrà quindi mostrata la loro implementazione che ha fornito mezzi di supporto, rendendo l'analisi più semplice.

In primo luogo, si può notare che la stesura del codice `scl` risulta complicata in quanto non presenta una sua sintassi: per scrivere il codice è necessario scrivere direttamente le strutture del linguaggio – che comunque sono ad alto livello, considerando che è implementato in `OCaml`. È quindi stato realizzato un parser verso `scl`: viene definita una sintassi più leggibile e conforme a quelle dei linguaggi di programmazione più conosciuti – sintassi che il parser riconosce, e da questa costruisce codice `scl`. Lo scopo del parser, dunque, è di fornire un mezzo per rendere più facile la stesura del codice. L'implementazione del parser avviene grazie alla tecnica dei parser combinator, dove ogni parser è una funzione che riconosce parte della sintassi e genera codice `scl`. I singoli parser a loro volta sono composti tra loro da delle funzioni di ordine superiore. Si riesce quindi a realizzare detto parser seguendo la sintassi costruita appositamente. Come conseguenza all'uso dei parser combinator, il parser in questione sarà LL(1): opera quindi in maniera ricorsiva costruendo l'albero di sintassi astratta da sinistra a destra, partendo dalla radice.

[...]

Capitolo 2

Intrinsically Typed Data Structures: uso per assicurare la type safety

Introduzione

Durante la compilazione verso un linguaggio qualsiasi, il compilatore si deve preoccupare del *type checking* cioè di quell'operazione che verifica che i vincoli di tipo nel programma vengano rispettati, condizione che viene chiamata *type safety*. Ad esempio un vincolo di tipo si ha in un assegnamento, dove è richiesto che il tipo del *lhs* sia compatibile con il tipo del *rhs*, altrimenti si può incorrere in diversi errori durante l'esecuzione.

In questa sezione vengono esaminate le tecniche per garantire la *type safety* nel parser e nel compilatore in questione, viene quindi definita una *Intrinsically Typed Data Structure* e vengono mostrate le tecniche presenti in `OCaml` per implementarla.

Approccio al type checking

Tradizionalmente il *type checking* avviene nella fase di analisi semantica nel processo di compilazione – quindi dopo che è avvenuto il parsing – una volta che è stato costruito l'albero di sintassi astratta (AST). Si ha così che il *type checking* avviene esternamente all'albero di derivazione: questo viene infatti sottoposto a un controllo, dove per ogni sottoalbero viene valutato il tipo e nel nodo viene verificato che i tipi dei relativi sottoalberi combacino.

In questo caso però, sia nel compilatore che nel parser, è stata usata una tecnica differente per cui l'albero di sintassi astratta è un *Intrinsically Typed Data Structure*. Questo significa che l'albero è una struttura dati con tipo dipendente – cioè il tipo non è fissato, ma è in relazione ai termini all'interno della struttura – che permette di esprimere tutti e i soli programmi ben tipati. È quindi la struttura stessa che definisce che cosa è ben tipato: non c'è bisogno quindi di implementare un *type checker* separato che analizzi l'albero, perché i vincoli di tipo sono già all'interno della definizione stessa dell'albero. Il *type checking*, quindi, avviene direttamente nel momento della costruzione dell'albero, garantendo sempre la *type safety*, ma in modo più immediato e con un'implementazione più semplice.

Per implementare parser e compilatore, si è interessati quindi a implementare un *Intrinsically Typed Data Structure* in `OCaml`: vediamo quindi gli strumenti che il linguaggio mette a disposizione.

Generic Algebraic Data Type

In prima battuta occorre capire cosa sia un *Algebraic Data Type* (ADT) che `OCaml` permette di definire. Un ADT è un tipo composto definito tramite intersezione o unione disgiunta di altri tipi. È mostrato un esempio di ADT:

```
type expr =
 | Int of int
 | Bool of bool
 | And of expr * expr
 | Plus of expr * expr
 | Eq of expr * expr
```

In questo esempio un tipo `expr` può essere `Int` di un intero, `Bool` di un booleano oppure `Plus`, `And` o `Eq` di due altre `expr`. Si ha quindi un'unione disgiunta di vari costruttori (che sono `Int`, `Bool`, `Plus`...), e all'interno di alcuni si ha un'intersezione – ad esempio all'interno di `Plus` dove si trova una coppia di `expr`. Si nota però, che, sempre nell'esempio, se si costruisse un'espressione `And((Int 9),(Bool true))` questa sarebbe un'espressione sintatticamente corretta, ma non semanticamente: non sarebbe ben tipata, siccome l'`And` si può fare solo tra tipi booleani. È necessario, quindi che quando viene valuta l'espressione ci sia un controllo di tipo: la funzione che si preoccupa della valutazione sarebbe:

```
let rec eval : expr -> int_or_bool =
function
 | Int n -> I n
 | Bool b -> B b
 | And(e1,e2) ->
 (match eval e1, eval e2 with
    B n, B m -> B (n && m)
    | _, _ -> raise TypeError)
 | ...
```

In questo modo scrivendo `And((Int 9),(Bool true))` non si riscontrerebbe nessun errore a tempo di compilazione, ma se si cercasse di calcolare `eval (And((Int 9),(Bool true)))` allora sarebbe sollevata un'eccezione a tempo di esecuzione.

Sempre con l'obiettivo di avere delle strutture *Intrinsically typed* si è interessati a un costrutto in cui la sintassi della struttura stessa possa permettere di scrivere espressioni solo ben tipate (il concetto stesso di ben tipato è in realtà definito dalla stessa struttura). `OCaml` infatti mette a disposizione anche *Generalized Algebraic Data Type* (GADT), che come suggerisce il nome è la generalizzazione degli ADT. Come d'altronde era già permesso negli ADT, i GADT permettono di definire dei tipi parametrici – ma in più si ha la possibilità di vincolare sintatticamente il parametro al momento della sua istanziazione con il tipo di ritorno dei costruttori del GADT (che dovrà essere specificato). Si ha quindi che ciascun costruttore è definito come una funzione tra tipi:

```
type 'a expr =
| Int : int -> int expr
| Bool : bool -> bool expr
| And : bool expr * bool expr -> bool expr
| Plus : int expr * int expr -> int expr
| Eq : 'a expr * 'a expr -> bool expr
```

In questo esempio, quindi, l'espressione `And((Int 9),(Bool true))` risulterebbe mal tipata, in quanto non conforme alla struttura appena definita, dal momento che il costruttore `And` richiederebbe una coppia di espressioni booleane, ma il primo elemento della coppia `Int 9` è definito come un'espressione intera. Scrivendo infatti `And((Int 9),(Bool true))` in codice `OCaml`, sarebbe sollevato un errore a tempo di compilazione. Definendo così il tipo `expr` anche la valutazione del tipo risulta più semplice rispetto all'analoga nel caso del ADT, infatti non ci si deve preoccupare di nessun controllo di tipo:

```
let rec eval: : type a. a expr -> a =
function
 | Int n -> n
 | Bool b -> b
 | And(e1,e2) -> (eval e1) && (eval e2)
 | ...
```

Si ha quindi che la struttura del tipo `expr` definita con l'uso degli GADT vincola sintatticamente a scrivere espressioni ben tipate: ciò significa che `expr` è intrinsecamente tipata. Quindi l'implementazione dell'albero di sintassi astratta, sarà del tutto analoga a quella di `expr`, dove i costruttori, invece di essere relativi alle espressioni saranno relativi ai costrutti di un programma nel linguaggio.

Conclusione

Si è quindi visto che i GADT permettono la definizione di *Intrinsically Typed Data Structure*: usando questa tecnica nell'implementazione del parser e del compilatore in questione, si può quindi evitare di dover essere ridondanti nell'eseguire il type checking, avendo la certezza che il codice sia *type safe*, ben tipato e con meno probabilità che si presentino bug.

Capitolo 3

`Scl`: presentazione del linguaggio e definizioni del problema del parsing

Introduzione

In questa sezione si inizia a descrivere il parser, senza analizzare, per ora, l'aspetto implementativo. Viene in un primo luogo descritto `scl`, il linguaggio verso il quale viene fatto il parsing, poi viene definita la grammatica che dà luogo a una sintassi per `scl`, e infine per tale grammatica si mostra il relativo albero di sintassi astratta generato dal parser, in maniera da definire chiaramente la struttura dell'input e dell'output del parser.

Descrizione del linguaggio `scl`

`Scl` è un linguaggio imperativo ad attori (come per esempio `Erlang`) che permette di descrivere il comportamento di contratti e umani – cioè gli utenti che interagiscono sui contratti. `Scl` è un linguaggio usato per l'analisi di *smart contract*, quindi, con l'obiettivo di fare un'analisi più mirata, semplice ed essenziale, è stato reso volutamente minimale. Nonostante ciò `scl` è un linguaggio Turing completo, in quanto permette l'assegnamento, l'istruzione condizionale, l'invocazione di funzioni, la ricorsione e il sollevamento di eccezioni.

Un programma `scl` consiste in una una configurazione, ovvero un insieme di attori (contratti o umani) che vengono definiti con campi e metodi, in maniera analoga ai linguaggi di programmazione a oggetti. Essendo un linguaggio ad attori si ha che ogni attore conosce tutti gli altri attori della configurazione,

rendendo possibile – per esempio – che un umano chiami un metodo di uno specifico contratto, senza avere bisogno di parametri o campi aggiuntivi.

Sebbene siano due entità distinte, il codice di contratti e degli umani è molto simile, se non per il fatto che questi ultimi possono fallire –possono sollevare dunque un'eccezione – e hanno a disposizione l'operazione di scelta, che consiste in un operatore non deterministico con il quale non si sa a priori quale codice l'umano andrà ad eseguire. L'idea è che nella realtà l'umano non ha un comportamento deterministico e quindi si devono prendere in considerazioni tutte le sue azioni possibili; questa diventa la parte interessante dell'analisi con `scl` , in quanto si cerca di capire che comportamento sarà più vantaggioso per l'umano. Inoltre per gli umani è necessario definire i comandi iniziali, che descrivono il loro comportamento e le loro interazioni con gli altri attori della configurazione.

`Scl` però non prevede una vera e propria sintassi: se si vuole implementare un attore è necessario scrivere direttamente le strutture dati del linguaggio. Queste strutture dati sono già in partenza *Intrinsically typed* siccome sono implementate in `OCaml` facendo uso dei GADT. Di conseguenza se si è interessati ad assegnare una sintassi al linguaggio, si può facilmente implementare il parser avendo come albero di sintassi astratta in output direttamente queste strutture, senza passare per una struttura intermedia. Si ha quindi un minor sforzo nella fase di implementazione, avendo comunque il vantaggio di non preoccuparsi di incorrere in errori di tipo durante il parsing, dal momento che la stessa struttura in output garantisce la *type safety*.

Grammatica e sintassi per `scl`

Il parser verso `scl` deve costruire l'albero di sintassi astratta del testo in input, basandosi sulla sua grammatica specifica, la quale necessita di essere definita. La definizione di questa sintassi è fondamentale in quanto costituisce l'utilità e lo scopo dello stesso parser: fornisce al programmatore uno strumento per scrivere in modo semplice e leggibile un testo che diventerà codice `scl` . È quindi necessario che la grammatica debba coprire tutti i costrutti del linguaggio `scl`, cercando di rimanere fedele e conforme alla sua struttura, senza perdere di espressività.

```
configuration ::= act*
act ::= (Human | Contract) ('(' Int ')')? '{' decl* meth* '}'
stm ::= decl | Var '=' rhs | 'if' bexpr 'then' stm 'else' stm | stm stm
        | stm '+' stm | '{' stm '}'
decl ::= t Var ('=' v)?
meth ::=  'function' Var '('(t Var (',' t Var)*)? '):' t '{' stm 'return' e '}'
rhs ::= e | (cexpr '.')? Var ('.' value '(' iexpr ')')? '(' (e (',' e)*)? ')'
v ::= Int | Bool | String
Bool ::= 'true' | 'false'
iexpr ::= Int | Var | 'fail' | iexpr? '-' iexpr | iexpr '+' iexpr
```

```
        | Int '*' iexpr | max '(' iexpr ',' iexpr ')' | symbol '(' String ')' | '(' iexp ')'
bexpr ::= Bool | Var | 'fail' | iexpr '>' iexpr | iexpr '>=' iexpr
        | iexpr '<' iexpr | iexpr '<=' iexpr | e '==' e | '!' bexpr
        | bexpr '&&' bexpr | bexpr '||' bexpr | '(' bexpr ')'
sexpr ::= String | Var | 'fail'
cexpr ::= this | Var | 'fail | contr_addr '(' String ')'
hexpr ::= Var | fail | hum_addr '(' String ')'
e ::= iexpr | bexpr | sexpr | cexpr | hexpr
t ::= int | bool | string | Contract | Human
```

In figura viene mostrata la grammatica su cui si basa l'analizzatore sintattico, nella quale sono presenti i simboli terminali che necessitano di essere riconosciuti durante l'analisi lessicale: che sono `Int` (ovvero l'insieme dei numeri interi), `String` (l'insieme delle stringhe) e `Var` (l'insieme di tutte le variabili), a cui si aggiungono tutte le parole chiave (ad esempio `return`, `if`, `+`, `>`, ...) che devono essere specificate al lexer. Invece la parte di analisi sintattica, una volta riconosciuti i simboli terminali, si preoccuperà di riconoscere i simboli non terminali e le opportune produzioni ricalcando la struttura del programma in input.

Questa sintassi permette di definire un insieme di attori, ognuno dei quali ha sia dei campi, che devono essere dichiarati, che una lista di metodi. Questi ultimi sono composti da una firma (con tipo in input e in output, dove quest'ultimo è sempre presente), uno statement (consistente in una lista di comandi di assegnamento, o istruzione condizionale o, nel caso di attori umani, scelta) e espressione di ritorno. I tipi nella grammatica sono gli stessi tipi di `scl`: interi, booleani, stringhe, contratti e umani.

Particolarmente interessante è il fatto che nella dichiarazione dell'attore si può specificare tra `()` il *balance* iniziale, ovvero i soldi che sono inizialmente assegnati al contratto o all'umano. Questo equivale a fare un assegnamento alla variabile intera denominata `balance`. È inoltre possibile inviare dei soldi al contratto nelle chiamate di funzioni semplicemente con la parola chiave `value` e specificando il valore.

Si può facilmente notare che la grammatica scelta è ambigua, come viene mostrato in questo breve esempio di un assegnamento: `x = y` che si sviluppa a partire dal non terminale `stm`

```
stm -> Var = rhs -> x = rhs -> x = e
```

uno statement, quindi diventa un assegnamento di un'espressione a una variabile. L'espressione ora ha può avere due produzioni:

Caso 1: l'espressione generica diventa un'espressione intera

```
x = iexpr -> x = y
```

Caso 2: l'espressione generica diventa un'espressione booleana

```
x = bexpr -> x = y
```

Si è visto che in entrambi i casi si è prodotta la stessa sequenza partendo dallo stesso non terminale, ma con derivazioni diverse. L'ambiguità è quindi dovuta al fatto che diversi non terminali (tipicamente relativi alle espressioni) possono accettare indistintamente delle variabili a prescindere dal loro tipo. È necessario quindi che il parser quando incorre nel riconoscimento di una variabile riconosca direttamente il suo tipo. Successivamente si vedrà come viene affrontato il problema.

Albero di sintassi astratta di ritorno

Dopo aver definito la grammatica è necessario mostrare l'albero di sintassi astratta generato come output dal parser, che rappresenterà la struttura logica del programma in input. Come si è detto precedentemente in questo caso l'albero di sintassi astratta corrisponde direttamente al codice `scl`. In figura viene mostrato il tipo di ritorno per la grammatica della figura precedente.

```
configuration ::= {contract: a_contract list, human: a_human list}
a_contract ::= (Contract string, meth list, decl list)
a_human ::= (Human string, meth list, decl list, stack)
stack ::= Return (t,e) | SComp(Stm stm, stack)
stm ::= Assign (var,e) | IfThenElse (e,stm,stm) | Comp (stm,stm) |
        Choice(stm,stm)
decl ::= Let((t,string),v)
program ::= meth,(vlist,stm,e)
meth ::= (t,tlist,string)
rhs ::= Expr e | Call (e,meth,elist) | CallWithValue (e,meth,elist,int)
var ::= (t,string)
v ::= int | bool | string | Contract string | Human string
e ::= Value v | Var var | Fail | This | Field var |
      Plus (e,e) | Mult (e,e) | Minus e | Max(e,e) | Geq(e,e)
      | Gt(e,e) | Eq(t,e,e) | And(e,e) | Or (e,e) | Not e |
      Symbol string
elist ::= ENil | ECons (e,elist)
tlist ::= TNil | TCons (t,tlist)
vlist ::= VNil | VCons (var,vlist)

t ::= Int | Bool | String | ContractAddress | HumanAddress
```

Come si può notare la grammatica definita precedentemente rimane molto conforme all'albero di sintassi astratta. Infatti ai simboli non terminali della grammatica sono fatti corrispondere i nodi dell'albero, e ai simboli terminali invece sono fatte corrispondere le foglie: si può quindi presupporre che ci sia una certa associazione tra albero di sintassi astratta e grammatica. Ciò che deve fare il parser è avendo un testo in input, in un primo luogo riconoscere passo per passo il testo nella grammatica, e nel mentre, sulla base di questa associazione costruire l'albero di sintassi astratta.

Conclusione

È stata presentata la grammatica del parser, anche se è ancora necessario risolvere il problema dell'ambiguità. Questa verrà però risolta a livello d'implementazione del parser, poiché il parsing si concluderà correttamente solo se ci sarà concordanza di tipo. Siccome è stata definita la struttura dell'input (ovvero la grammatica del linguaggio) e la struttura dell'output (ovvero la struttura dell'AST) si è pronti per la descrizione dell'implementazione.

Capitolo 4

Implementazione del parser

Introduzione

In questa sezione ci si concentra sull'implementazione del parser verso `scl`. Vengono quindi descritte le tecniche e le scelte implementative che portano il parser a prendere in input una lista di caratteri e determinarne la correttezza basandosi sulla grammatica, generando quindi il relativo albero di sintassi astratta per tale lista.

Il parser in considerazione è implementato in `OCaml`, questa scelta risulta abbastanza ovvia dal momento che lo stesso `scl` è implementato in questo linguaggio.

Analisi lessicale

Il parser non può fare l'analisi sintattica direttamente sul testo in input: per iniziare deve avere in ingresso una sequenza di token. Ogni token è una coppia nome-valore, dove il nome rappresenta una determinata categoria sintattica, mentre il valore è una stringa del testo. Quindi per generare i token si inizia facendo l'analisi lessicale, dividendo le stringhe in input in diverse categorie (le categorie sintattiche, appunto).

Con questo obiettivo si è usato il modulo `Genlex` di `OCaml`, che permette di generare un analizzatore lessicale che in `OCaml` consiste in una funzione che prende in input uno stream di caratteri e restituisce in output una lista di token. Inoltre questo strumento è particolarmente vantaggioso rispetto a un'analisi lessicale senza uso di moduli aggiuntivi, perché toglie la preoccupazione di fornire un'implementazione per l'aggiunta di commenti nel testo, così come diventa automatica la rimozione degli spazi bianchi tra le varie stringhe. I token quindi sono riconosciuti e ad ognuno di essi è associato una categoria (o nome). Le

possibili categorie sono: interi, stringhe, identificativi e parole chiave. Mentre interi, stringhe e identificativi sono già noti al lexer, perché sono caratteristici di tutti i linguaggi, le parole chiave richiedono di essere esplicitate, perché sono specifiche del lessico del linguaggio di cui fare il parsing.

Concettualmente l'analizzatore sintattico richiederebbe uno stream di token, quindi ci si aspetterebbe che il parser implementato richieda un input del tipo `token Stream.t`. Infatti il tipo `Stream` di `OCaml` offrirebbe un vantaggio in termini di memorizzazione: non è necessario che tutta la sequenza di token sia in memoria, e sebbene l'analizzatore lessicale generato da `make_lexer` –la funzione di `GenLex` che lo genera– restituisca un tipo `token Stream.t`, è stato scelto di usare una lista di token. Questa è stata una scelta di natura implementativa: come sarà più chiaro successivamente, il parser ha bisogno di fare backtracking, e con gli `Stream` di `OCaml` l'operazione diventerebbe ardua siccome quando si esamina un elemento in uno stream, l'elemento precedente viene perso. Con la lista invece l'iterazione è molto più semplice e si può procedere in entrambe le direzioni senza preoccupazione.

L'analisi lessicale, quindi, relativamente alla grammatica consiste nel riconoscere i simboli terminali come tali, in modo che successivamente ci si concentri separatamente nel riconoscimento delle produzioni della grammatica.

Implementazione con parser combinator

Una volta definita la grammatica e riconosciuti i simboli terminali di questa si hanno tutte le basi necessarie per l'implementazione `OCaml` del parser. Il parser può essere implementato con diverse tecniche: similmente a quanto fatto per l'analisi lessicale si potrebbero usare librerie di parsing che generano analizzatori sintattici, oppure si può usare la tecnica del parser combinator. In questo caso è stata preferita quest'ultima tecnica, che consiste nel considerare un parser come una funzione di ordine superiore che avendo uno o più parser in input restituisce un nuovo parser (da qui il nome). A livello di codice per parser si intende una funzione che prende una lista di token e restituisce l'albero di derivazione e la lista dei token rimanenti.

```
type 'ast parser =
token t -> (vartable * bool) -> token t * 'ast * (vartable * bool)
```

Per l'implementazione è necessario definire un'eccezione che ogni funzione `parser` solleverà qualora non dovesse essere in grado di riconoscere l'input, questa eccezione è stata denominata `Fail`.

Con l'ausilio dei parser combinator si possono tradurre facilmente le produzioni della grammatica definita in precedenza: innanzitutto è necessario descrivere gli operatori sintattici necessari, ovvero la concatenazione, la stella di Kleene, l'unione, la costante e l'operatore possibilità. L'implementazione di ogni operatore risulta semplice: ad ognuno di questi è fatta corrispondere una funzione, che nella

logica dei parser combinator genera un nuovo parser specifico per l'operatore in questione. Questi parser verranno poi composti tra loro, secondo le regole della grammatica, per riuscire a riconoscere una qualsiasi produzione.

Per l'operatore costante –che riconosce i simboli terminali– viene generato un parser che verifica se il simbolo richiesto corrisponde al simbolo in input: nel caso in cui questo succeda l'input può essere consumato:

```
let const : token -> (token -> 'ast) -> 'ast parser =
 fun t1 f t2 tbl ->
  if (List.length t2 > 0) && (t1 = (List.hd t2)) then
   (junk t2), f t1, tbl
  else
   raise Fail
```

Per l'operatore di unione sono necessari due parser in input. Per implementare l'idea di scelta non deterministica viene eseguito il primo parser: nel caso questo sollevi `Fail` facendo backtracking viene eseguito il parser rimanente:

```
let choice : 'ast parser -> 'ast parser -> 'ast parser
= fun p1 p2 s tbl ->
 try p1 s tbl with Fail -> p2 s tbl
```

La concatenazione invece consiste nel prendere in input due parser e eseguirli sequenzialmente unendo successivamente i due output:

```
let concat :
'ast1 parser -> 'ast2 parser -> ('ast1 -> 'ast2 -> 'ast3) -> 'ast3 parser =
fun p1 p2 f s tbl ->
  let rest1,ast1,tbl1 = p1 s tbl in
  let rest2,ast2,tbl2 = p2 rest1 tbl1 in
  rest2,f ast1 ast2,tbl2
```

La stella di Kleene vede l'esecuzione dello stesso parser finché questo non solleva `Fail` –caso per cui l'esecuzione dell'operatore termina:

```
let kleenestar :
'ast2 parser -> 'ast1 -> ('ast1 -> 'ast2 -> 'ast1) -> 'ast1 parser =
 fun p empty_ast f s t ->
  let rec aux p1 s1 acc tbl=
  try
   let (rest1, ast1, ntbl) = p1 s1 tbl in
   aux p1 rest1 (f acc ast1) ntbl
  with Fail -> (s1, acc, tbl)
  in aux p s empty_ast t
```

Per concludere, l'operatore di possibilità corrisponde un parser che semplicemente prevede che possa fallire:

```
let option : 'ast parser -> 'ast option parser =
 fun p s tbl -> try
```

```
  let next,res,ntbl = p s tbl in next,Some res,ntbl
 with Fail -> s,None,tbl
```

Dopo aver definito questi operatori diventa banale scrivere il parser. Ad ogni non terminale della grammatica viene costruita una nuova funzione di tipo `parser`, costituita dalla corretta composizione delle funzioni di parsing appena descritte, coerentemente con quanto definito dalle regole della grammatica. Degli esempi li si avranno successivamente.

Si è quindi spiegato come il parser riesce a riconoscere la grammatica e si è visto che le funzioni per gli operatori sintattici implementati con la logica dei parser combinator prendono in input funzioni per la costruzione dell'albero di sintassi astratta.

Costruzione dell'albero di sintassi astratta

Come spiegato nel capitolo precedente (3) tra la grammatica e l'albero di sintassi astratta per tale grammatica si ha un'associazione abbastanza diretta, quindi mentre si riconduce il testo alle produzioni della grammatica è abbastanza immediato costruire l'albero relativo. A titolo di esempio vediamo come nel caso atomico, una volta riconosciuto il token sia abbastanza immediato costruire la foglia:

```
let value : type a. a tag -> token -> a expr = fun tag tok ->
 match tag,tok with
 | String,Genlex.String x -> Value x
 | Int,Int x -> Value x
 | Bool,Kwd "true" -> Value true
 | Bool,Kwd "false" -> Value false
 | _ -> raise Fail

let value_pars tag s = const (List.hd s) (value tag) s
```

Nell'esempio `value_pars` è il parser per il non terminale `v` della grammatica. Si tratta infatti semplicemente di passare dalla lista di token a un'espressione in `scl`. Si noti che `Value` è un costrutto dell'AST di `scl`.

Nel caso non atomico, ovvero durante la costruzione di un nodo dell'albero, dal momento che due `parser` sono combinati è necessario fare un controllo sul tipo. È vero che essendo l'albero di sintassi astratta una *Intrinsically Typed Data Structure* non si può incorre in problemi di tipo, tuttavia al momento della costruzione della struttura è necessario fare il *type checking* in modo che i costruttori del GADT abbiano la certezza che i tipi combacino. In `scl`, ad esempio, il costruttore per l'assegnamento è definito in questo modo:

```
type stm =
 | Assign : 'a field * 'a rhs -> stm
 | ...
```

Quindi una volta che si ha un `field` e un `rhs`, per costruire il relativo `Assign` è necessario controllare che `field` e `rhs` abbiano lo stesso tipo. Sempre secondo la logica dei GADT è stato definito il tipo delle prove:

```
type (_,_) eq = Refl : ('a,'a) eq
```

Usando questo particolare tipo è poi possibile implementare una funzione che controlli e assicuri che due tipi definiti in `scl` combacino:

```
let eq_tag : type a b. a tag -> b tag -> (a,b) eq option = fun t1 t2 ->
 match t1,t2 with
 | Int, Int -> Some Refl
 | Bool, Bool -> Some Refl
 | String, String -> Some Refl
 | ContractAddress, ContractAddress -> Some Refl
 | HumanAddress, HumanAddress -> Some Refl
 | _,_ -> None
```

Una volta che si ha a disposizione la funzione `eq_tag` diventa possibile implementare un parser per l'assegnamento, quindi che generi un `Assign` combinando il risultato dei parser che riconoscono `field` e `rhs`, e controllando che il tipo di ritorno dei due parser sia il medesimo:

```
let assign_pars =
 concat (concat field_pars (kwd "=") fst) rhs_pars
 (fun (AnyField(tfield,name)) (AnyRhs(trhs,r))->
 match eq_tag tfield trhs with
 | Some Refl -> Assign((tfield,name),r)
 | None -> raise Fail)
```

Nell'esempio si ha una funzione `parser` in cui è esplicitata la forma sintattica che deve avere un assegnamento (si veda l'analogia nell'assegnamento nella grammatica) ed è specificato come viene costruito l'albero di derivazione. In questo caso `AnyField` e `AnyRhs` sono costrutti che contengono rispettivamente il valore del campo e del *rhs*, ovviamente con i rispettivi tipi. Questi infatti sono il risultato dei parser `field_pars` e `rhs_pars`, che in `assign_pars` vengono combinati generando un nuovo nodo `Assign` nell'albero sintattico.

Analogamente a quanto è avvenuto in questi due casi può essere costruito ogni nodo o foglia dell'albero.

Si è quindi visto che con l'ausilio dei parser combinator il parser generato è top-down, cioè che inizia a costruire l'albero dalla radice. Per iniziare definisce una *configuration* vuota (il simbolo iniziale) che verrà riempita dall'attività dell'analizzatore sintattico. In questo tipo di parsing l'input viene consumato da sinistra a destra con solo un simbolo di lookahead – che se non viene riconosciuto implica il backtracking. Sebbene il parsing abbia i requisiti per essere di tipo LL(1) la grammatica in realtà non lo è. Questo perché, come visto in precedenza, presenta delle ambiguità; in aggiunta la grammatica è ricorsiva sinistra, infatti se

il parser cercasse di risolvere una regola con la ricorsione sinistra l'esecuzione andrebbe in loop infinito. È quindi necessario risolvere questi problemi, eliminando la ricorsione sinistra e le ambiguità.

Eliminazione ricorsione sinistra

Per eliminare la ricorsione sinistra bisogna prima agire sulla grammatica senza andarne a cambiare il linguaggio generato e poi andare a modificare appropriatamente il relativo codice della funzione `parser`. Verrà mostrato soltanto il caso dell'eliminazione nel non terminale `iexpr` (connotazione per le espressioni intere), per gli altri non terminali la risoluzione è analoga. L'idea usata è quella di sostituire al non terminale due non terminali: il primo con tutte e sole regole senza ricorsione sinistra il secondo con le regole in cui questa è presente. I due non terminali vengono rispettivamente `atomic_iexpr` e `cont_iexpr`, e vengono definiti in questo modo:

```
atomic_iexpr ::= Int | Var | fail | '-' iexpr | max '(' iexpr ',' iexpr ')'
                | symbol string | '(' iexpr ')'
cont_iexpr ::= '+' iexpr | '-' iexpr | '*' iexpr
```

In questo esempio `atomic_iexpr` rappresenta un non terminale per alcune espressioni intere (appunto quelle atomiche), mentre `cont_iexpr` considerato singolarmente non ha alcuna valenza: deve essere concatenato a un'espressione intera, che sarà quindi `atomic_iexpr`. Il non terminale `iexpr` verrà costruito concatenando i due non terminali appena prodotti. Avrà quindi la seguente definizione:

```
iexpr ::= atomic_iexpr (cont_iexpr)?
```

Una volta eliminata la ricorsione sinistra a livello di sintassi, l'implementazione consiste banalmente nel trascrivere quanto definito, ancora una volta come composizione di parser.

```
let rec atomic_int_expr s =
 choice_list [
   comb_parser (base Int) (fun expr -> AnyExpr(Int,expr));
   concat (kwd "-") atomic_int_expr (fun _ -> minus) ;
   concat (concat (kwd "(") int_expr scd) (kwd ")") fst ;
   concat (concat (kwd "max") int_expr scd) int_expr max;
   concat (kwd "symbol") symbol_pars scd;
 ] s
and int_expr s =
 concat atomic_int_expr (option cont_int_expr)
 (fun x f -> match f with Some funct -> funct x | _ -> x) s
and binop s =
 choice_list [
  const (Kwd "+") (fun _ -> plus) ;
```

```
  const (Kwd "*") (fun _ -> mult) ;
  const (Kwd "-") (fun _ -> subtract)
 ] s
and cont_int_expr s = concat binop int_expr (fun f x -> f x) s
```

Controllo dei tipi e tabella dei simboli

Come mostrato in precedenza le ambiguità della grammatica, stando alla sua definizione, sono dovute al fatto che in non terminali che connotano espressioni di tipo diverso può comparire la stessa variabile. Per eliminare l'ambiguità è necessario, quindi, che in ogni espressione se è presente una variabile già si conosca il tipo di quest'ultima. Questo obiettivo può essere conseguito in due modi: o ad ogni occorrenza della variabile si specifica sia il tipo che il nome, oppure si specifica solo il nome della variabile ma si fa in modo che il parser possa leggere il tipo di dato in una tabella apposita (in cui è associata ad ogni nome di variabile il suo tipo). Siccome si preferisce che solo il nome sia identificativo per la variabile (come avviene in quasi tutti i linguaggi di programmazione) si sceglie la seconda opzione. Di conseguenza all'interno del programma su cui effettuare il parsing non potranno mai esistere due variabili diverse ma con lo stesso tipo, cosa che in realtà è possibile nella struttura degli attori in `scl`. Così facendo si ha una perdita di espressività minima, ma si ha il vantaggio di favorire una programmazione più consapevole.

La tabella è quindi relativa a un singolo attore, e contiene la lista delle coppie tipo-nome di tutte le sue variabili. Essendo `OCaml` un linguaggio funzionale, non può essere che la tabella sia una variabile globale, altrimenti non potrebbe essere modificabile. È necessario quindi che la stessa funzione di tipo `parser` richieda in input e restituisca in output la tabella, in modo tale che gli sia consentito aggiungere o rimuovere elementi. In particolare una nuova variabile deve essere aggiunta alla tabella quando viene dichiarata, e quando si ha un assegnamento di una variabile non ancora presente in tabella. Ad ogni aggiunta viene controllata la tabella e nel caso fosse già presente una variabile con lo stesso nome, il parser solleva `Fail` e termina la sua esecuzione. Invece ad ogni occorrenza della variabile viene fatta sempre una ricerca nella tabella per conoscerne il tipo.

Le variabili oltre a poter essere aggiunte alla tabella devono anche poter essere rimosse. Si prenda in considerazione una variabile locale all'interno di una funzione: essa non deve essere visibile in nessun'altra funzione esterna. Quando verrà quindi fatto successivamente il parsing di un'altra funzione, se questa seconda funzione presenta una variabile con lo stesso nome questa non dovrà essere presente nella tabella. Diventa allora necessario definire lo scope per le variabili, in modo che determini la presenza o meno della variabile nella tabella. Per come è definito il linguaggio questa è un'operazione banale, infatti in `scl` non ci sono funzioni annidate, quindi gli unici blocchi di cui può far parte una variabile sono il blocco globale dell'attore e il blocco locale della funzione. Quindi, durante il parsing, o la variabile è una variabile globale o è una variabile locale

della funzione presa in esame: non è possibile avere una variabile attiva che sia allo stesso tempo variabile locale di un'altra funzione. Quindi basta associare ad ogni variabile nella tabella un valore booleano che denota se la variabile è locale o meno, per far sì che quando venga finito il parsing di una funzione tutte le variabili locali vengano rimosse.

Analogamente alle variabili anche le funzioni possono essere aggiunte nella stessa tabella: quando viene dichiarato un metodo, la firma di questo –nome, lista dei tipi dei parametri e tipo di ritorno– viene salvata nella tabella dell'attore. Così, ogni volta che ci sarà una chiamata di funzione, dal nome di questa viene cercata la firma del metodo nella tabella, e si verifica che la lista delle espressioni nella chiamata sia coerente con la lista dei parametri presa dalla firma appena ritornata. In `scl` però si possono chiamare i metodi di uno specifico contratto, in questo caso non verrà fatto alcun controllo perché il contratto potrebbe essere esterno al testo su cui fare il parsing e, in questo caso, non si avrebbe nessuna informazione.

Un'altra informazione che il parser deve conoscere durante la sua esecuzione è se l'attore che sta analizzando sia umano o contratto: come si è visto prima all'umano sono permessi più comandi e quindi il suo codice può presentare diversi comandi. Questa informazione, che si traduce in un booleano, è affiancato alla tabella come input e output di ogni funzione di tipo `parser`.

Conclusione

Si è quindi visto che per il parser LL(1) implementato è stato necessario definire in primo luogo una sintassi per `scl`. Con la tecnica dei parser combinator poi è bastato trascrivere le stesse regole della grammatica come combinazione di più parser, facendo però alcuni accorgimenti: si è tolta la ricorsione sinistra per evitare che il parser divergesse. Grazie all'uso dei GADT a livello di implementazione del parser, si è poi fatto in modo che il parsing si concluda con successo solamente se c'è concordanza tra i tipi effettivamente usati nelle espressioni con la grammatica che definisce le espressioni stesse.

Il problema dell'ambiguità della grammatica, invece, è stato aggirato mediante la creazione di una tabella che permettesse di ricordare l'associazione tra nome di variabile e tipo per tutta la durata del parsing.

Capitolo 5

Compilatore da `scl` a Solidity

Introduzione

Una volta che si hanno a disposizione tutti gli strumenti per programmare in `scl` ci si accorge che ciò che viene a mancare è un'esecuzione vera e propria del codice su una blockchain: `scl` fornisce tutti i mezzi per fare analisi, ma solo con `scl` non è possibile fare deploy di ciò che si scrive. È necessario quindi un compilatore da `scl` verso un linguaggio già conosciuto su cui è possibile fare deploy. Per la traduzione dei contratti è necessario quindi un linguaggio di *smart contract*. Si è scelto così Solidity, principalmente per il fatto che è il più conosciuto e a differenza di altri (come ad esempio Liquidity) allo stato attuale è difficile che venga abbandonato il progetto. Con Solidity diventa possibile fare il deploy dei contratti sulla sua blockchain Ethereum.

Solidity e Ethereum

Ethereum è una piattaforma decentralizzata che permette lo sviluppo di *smart contract* da parte di programmatori. Ethereum mette quindi a disposizione Solidity, un linguaggio che in una logica di programmazione ad oggetti (Class-based) permette di programmare *smart contract*. Il compilatore di Solidity traduce il programma in bytecode, e poi il programma tradotto viene eseguito nella EVM, la macchina virtuale di Ethereum che costituisce l'ambiente di esecuzione degli *smart contract*.

Solidity è un linguaggio che permette di definire le classi di contratti, che possono essere ereditati da altre classi o interfacce specificando metodi e campi. Ogni contratto nella blockchain ha un indirizzo a cui altri contratti si possono riferire

per interagire. Inoltre tutti i contratti hanno un balance che consiste nella quantità di Ether (criptovaluta di Ethereum) che può essere modificato con transazioni (chiamate di funzioni) da parte di utenti o contratti. Le funzioni possono essere marcate come `payable`, in modo tale da poter ricevere Ether quando chiamate. Ci sono poi indirizzi speciali come il noto `this` e `msg.sender`, dove quest'ultimo si riferisce all'indirizzo al chiamato. Ogni contratto inoltre ha un Abstract Binary Interface (`abi`) che rappresenta l'interfaccia di un contratto, diventa interessante nella fase di deploy, in quanto insieme all'indirizzo del contratto costituiscono tutte le informazioni necessarie per la creazione del contratto . Solidity permette di riferirsi a un indirizzo di un contratto la parola chiave `address`, che costituisce il tipo generico di un oggetto (analogamente a `Object` in Java), e che diventa utile in fase di compilazione quando ci si riferisce a un contratto non conosciuto.

Differenze tecniche tra `scl` e Solidity

È stata appena introdotta una differenza importante tra `scl` e Solidity: il primo linguaggio definisce un oggetto contratto, mentre il secondo definisce la classe del contratto. È necessario marcare questa differenza perché in `scl`, a differenza di Solidity, all'interno di un contratto ci si può riferire specificamente ad altri contratti nella configurazione. Si potrebbe pensare, invece, che questa cosa possa avvenire anche in Solidity, riferendosi a questi contratti tramite la clausola `new`, ma sarebbe una cosa diversa perché così verrebbe creato un nuovo contratto, invece che un riferimento a contratto specifico. Sarebbe invece più opportuno riferirsi a ogni contratto attraverso il loro indirizzo. Il problema è che al momento della scrittura del codice Solidity non si sa quali siano questi indirizzi, mentre in `scl` non si ha questo problema perché ogni contratto conosce implicitamente tutti gli altri della configurazione. Diventa quindi necessario separare la parte di compilazione in codice Solidity dalla generazione del codice, dove Solidity crea la classe per ciascun oggetto contratto della configurazione, dalla parte di deploy, dove sulla base del codice Solidity vengono generati gli effettivi contratti nella blockchain con i relativi indirizzi.

Albero di sintassi astratta per Solidity

Essendo `scl` un codice minimale, nella fase di compilazione si genera un sottoinsieme del codice di Solidity: non saranno presenti tutti i suoi costrutti, ma solo quelli di cui ne fa relativo uso `scl`.

Il processo di compilazione verso Solidity si divide in due parti: la prima in cui dall'albero di sintassi astratta `scl` si passa al relativo albero del sottoinsieme di Solidity, la seconda in cui dall'albero appena generato si traduce in effettivo codice Solidity. L'idea è che l'albero di sintassi astratta intermedio venga generato ricalcando i costrutti del linguaggio Solidity, cosicché la seconda fase risulti immediata. Ancora una volta è quindi necessario un AST, e analogamente al

caso precedente si preferisce che sia *Intrinsically Typed*, quindi viene sempre implementato con i GADT di OCaml. In questo modo si ha un AST ben tipato conforme ai costrutti Solidity, la traduzione quindi avviene facendo corrispondere ad ogni tipo di `scl` un tipo di Solidity. L'albero di cui si parla è il seguente:

```
configuration ::= a_contract list
a_contract ::= Contract(string,decl list,program list)
stm ::= Assignment (var,e) | IfElse (e,stm,stm) | Sequence (stm,stm)
decl ::= Declaration((t,string),v)
program ::= meth,(stm,e)
meth ::= (string,paraml,(t,storage?),view*,visibility*)
storage ::= Storage | Memory | Calldata
paraml ::= PNil | PCons((t,storage?),paraml)
view ::= View | Pure | Payable
visibility ::= External | Public | Internal | Private
intf_id ::= InterfaceId string
var ::= (t,string)
v ::= int | bool | string | interface_id | AddrInt intf_id
e ::= Value v | Var var | This | MsgValue | Balance
      | CastInterf (intf_id, e) | Addr e | Plus (e,e)
      | Mult (e,e) | Minus (e,e) | Max(e,e) | Geq(e,e)
      | Gt(e,e) | Eq(t,e,e) | And(e,e) | Or (e,e) | Not e
      | Symbol string | CondExpr(e,e,e) | Call(e, meth, elist)
elist ::= ExprNil | ExprCons (e,elist)

t ::= Int | Bool | String | Interf | Address
```

Come si può notare la definizione dell'AST per Solidity, presenta qualche differenza rispetto a quello per `scl`: vengono rimosse, ovviamente, le regole caratteristiche per gli umani, ma vengono aggiunti nuovi nodi. Si nota che nella definizione di funzione (`meth`) viene specificato oltre al nome, i parametri e il tipo di ritorno, la visibilità e la view che sono caratteristiche di Solidity. La visibilità, come in altri linguaggi di programmazione ad oggetti, denota chi può chiamare la funzione, mentre la view (più specifica per Solidity) è relativa alla possibilità che ha la funzione di cambiare lo stato interno. Ma la maggior modifica è relativa ai tipi: non si ha più un unico `ContractAddress`, ma si ha sia il tipo `Interf` sia il tipo `Address`, e due nuove espressioni `CastInterf` e `Addr` che permettono il cast tra i due tipi. Questa distinzione di tipi è dovuta al fatto che, come si diceva prima, in Solidity sono realizzate le classi relative ai contratti `scl`, ma un contratto con gli altri contratti della configurazione interagisce mediante l'indirizzo.

Interfacce per le variabili `Contract`

Come spiegato precedentemente, si deve riprodurre in Solidity una configurazione in cui tutti i contratti si conoscono: per cui questi contratti devono poter essere espressi come campi all'interno di un contratto. In generale si è visto che in

`scl` esiste il tipo `Contract`, di conseguenza è possibile avere i campi di tipo contratto. Ci si chiede così come possono essere espressi questi campi in Solidity, e in particolare quale possa essere il loro tipo; logicamente se si volesse rimanere aderenti al codice `scl` questi dovrebbero essere di tipo `address`. Il problema però è che se ci si riferisce solo all'indirizzo del contratto allora anche le chiamate di funzione devono essere fatte sull'indirizzo –sarebbe come fare in Java una chiamata di metodo su un oggetto di tipo `Object`. Solidity permette questo tipo di chiamate a basso livello usando la funzione `call`, ma il loro tipo di ritorno, però, è espresso in byte e per fare la conversione si sarebbe obbligati a scrivere codice Assembly. Si preferisce quindi un'altra tecnica, dove le chiamate possano essere fatte su degli oggetti meno generici e dove si conosca il tipo di ritorno delle funzioni.

Ad ogni campo di tipo contratto allora è fatta corrispondere un'interfaccia che contiene la firma dei suoi metodi. Solidity permette appunto di fare il cast da interfaccia a indirizzo e viceversa. In questo modo è necessario lavorare sia con gli indirizzi dei contratti, riuscendo a rimanere fedeli al codice `scl`, e allo stesso tempo usare le interfacce, necessarie per le chiamate di funzione. In particolare si è scelto che i campi (sempre di tipo contratto) sono oggetti che hanno come tipo la loro interfaccia, mentre i parametri sono indirizzi a cui nella loro dichiarazione viene associata la loro relativa interfaccia. Quindi se si ha una chiamata di funzione su un parametro, allora viene fatto il cast alla sua interfaccia, mentre se viene passato un oggetto di tipo interfaccia come parametro attuale di una funzione viene fatto il cast verso l'indirizzo. Affinché le interfacce facciano il loro lavoro, è necessario che ad ogni chiamata di un metodo di un contratto, venga aggiunta la firma del metodo nell'interfaccia.

```
Contract sample {
    Contract interf
    int x
    function m (Contract addr): int{
        x = addr.f(interf)
        x = interf.g(addr)
        return x
    }

}
```

Nell'esempio di codice `scl` vede un campo di tipo contratto (`interf`) e un parametro di tipo contratto (`addr`), la compilazione verso solidity è:

```
interface Interf0{
    function g(address) external returns (int);
}
interface Interf1{
    function f(address) external returns (int);
}
contract sample {
```

```
    Interf0 interf;
    int x;

    function m(address addr) public returns (int){
        x = Interf1(addr).f(address(interf));
        x = interf.g(addr);
        return x;
    }
}
```

Si nota quindi che `interf` è tradotto come un oggetto con la sua interfaccia `Interf0`, mentre `addr` che nel codice `scl` è sempre dello stesso tipo, è un oggetto generico di tipo `address`, ma nonostante ciò ad `addr` viene associata l'interfaccia `Interf1`. Quando viene chiamata una funzione su `addr`, viene prima fatto il cast in `Interf1`, mentre quando viene chiamata una funzione su `interf` non viene fatto nessun cast. Invece nel passaggio di parametri si ha, il cast verso `address` quando viene passata `interf`, mentre non si ha il cast di `addr` siccome è già del tipo `address`. Si nota anche che le due funzioni chiamata sono aggiunte nelle relative interfacce: la funzione `g`, chiamata da `interf` è aggiunta in `Interf0`, e la funzione `f` chiamata da `addr` è aggiunta in `Interf1`.

Rimane da capire cosa fare a livello di interfacce quando si ha un assegnamento di contratti con interfacce diverse. In pratica si ha che deve avvenire un cast verso l'interfaccia del *lhs*, però, dal momento che i tipi in solidity sono nominali, sarebbe necessario anche che i metodi dell'interfaccia del *rhs* vengano aggiunti nel *lhs*. Si potrebbe quindi aggiungere manualmente i metodi all'interfaccia del *lhs*, o comunque far ereditare il *lhs* al *rhs*. Si è però aggirato il problema, in modo che Solidity se ne preoccupasse a tempo di esecuzione invece che durante la compilazione. Infatti è stato fatto prima un cast dal *rhs* all'indirizzo e poi dall'indirizzo all'interfaccia, e in questo modo è possibile qualsiasi cast tra due diversi oggetti senza che il compilatore solidity non segnali niente.

```
Contract p
Contract q
p = q
```

L'assegnamento diventa:

```
Interf0 p;
Interf1 q;

p = Interf0(address(q));
```

Inizializzazione degli indirizzi

Si sa quindi come avviene la traduzione per i tipi `Contract` di `scl`, non è ancora chiaro però come la compilazione possa far conoscere ad ogni contratto tutti gli

altri contratti della configurazione. Ovviamente in Solidity sarebbe necessario che un contratto debba avere gli indirizzi di degli ultimi. Questi indirizzi non saranno noti a tempo di compilazione poiché sono indirizzi relativi al posizionamento del contratto nella blockchain e quindi sono assegnati quando viene fatto deploy. Ma per fare in modo che a tempo di deploy si possano comunicare al contratto tutti gli indirizzi richiesti è necessario che nel codice solidity ci siano i mezzi per fare in modo che questa comunicazione avvenga. In particolare ogni contratto deve avere un campo per ogni altro contratto della configurazione, viene poi creata una funzione `init` che assegna gli indirizzi passati come parametro ai rispettivi campi. Questa funzione verrà poi chiamata in fase di deploy quando si conoscono effettivamente gli indirizzi.

```
Contract a{}
Contract b{}
Contract c{}
```

La configurazione base mostrata nell'esempio viene compilata in:

```
contract a {
    Interf1 c;
    Interf0 b;
    bool initialize = false;

    function init(address _b, address _c) public {
        if (!initialize){
            b = Interf0(_b);
            c = Interf1(_c);
            initialize = true;
        }
    }
}
contract b {
    Interf3 c;
    Interf2 a;
    bool initialize = false;

    function init(address _a, address _c) public {
        if (!initialize){
            a = Interf2(_a);
            c = Interf3(_c);
            initialize = true;
        }
    }
}
contract c {
    Interf5 b;
    Interf4 a;
```

```
    bool initialize = false;

    function init(address _a, address _b) public {
        if (!initialize){
            a = Interf4(_a);
            b = Interf5(_b);
            initialize = true;
        }
    }
}
```

dove la variabile booleana `initialize` assicura che la funzione `init` non venga chiamata più volte. Così ogni contratto si può riferire agli altri contratti della configurazione.

Considerazioni implementative

Problemi con il sistema di tipi in solidity

Conclusione

Capitolo 6

Fase di deploy con script Python

Capitolo 7

Conclusione