9.3 Featherweight Java

Molte delle caratteristiche delle classi e degli oggetti possono essere codificate in un calcolo con funzioni, record, riferimenti e subtyping. Questo approccio è utile perché permette di comprendere caratteristiche complesse di questi linguaggi proprio rappresentandole in termini di costrutti lower-level. D'altra parte, alcuni aspetti, come i metodi virtuali o il parametro `this`, si studiano in modo più soddisfacente se aggiunti al linguaggio come primitivi, assiomatizzando quindi in modo diretto la loro sintassi, semantica operazionale e regole di tipo.

Featherweight Java, proposto da Igarashi, Pierce e Walder nel 1999 è un minimal core calculus che modella il type system di Java. Un *modello formale* di un linguaggio si usa

- per **descrivere** in modo *preciso* qualche aspetto del design del linguaggio,
- per **enunciare** le sue proprietà e **dimostrarle**.
- Permette così di individuare problemi che non erano stati osservati ad un livello più informale.
- Richiede un tradeoff tra *completezza* e *compattezza*: più aspetti si modellano insieme, più il formalismo diventa intrattabile e poco chiaro.

Il linguaggio FJ è estremamente compatto: contiene solo 5 tipi di termini, creazione di oggetti, invocazione di metodi, selezione di campi, cast e variabili. Non ci sono nemmeno gli assegnamenti, ma c'è tutto il necessario per studiare le core features del type system di Java, compreso definizione di classi mutuamente ricorsive, method override, method recursion tramite parametro `this`, subtyping. Vedremo infatti che questo è già abbastanza per catturare alcune sottigliezze del casting. Non essendoci gli assegnamenti, i campi di un oggetto sono inizializzati dal costruttore e non cambiano mai. Quindi FJ corrisponde al "frammento funzionale" di Java. Ne vedremo più avanti la versione imperativa.

La compattezza di FJ ne fa il punto di partenza per studiare estensioni di Java: ad esempio GJ=FJ+classi generiche, oppure FJ+inner classes, oppure FJ+RMI, oppure FJ+Mixins, FJ+Unknown-type per recuperare un po' della flessibilità dei linguaggi non tipati, ...

Un *programma* in FJ è un insieme di definizioni di classi più un termine da valutare. Consideriamo ad esempio la seguente definizione di classi CT:

```
class A extends Object { A() { super(); } }
class B extends Object { B() { super(); } }
class Pair extends Object {
    Object fst; Object snd;
    Pair(Object fst, Object snd)
          { super(); this.fst=fst; this.snd=snd; }
    Pair setfst(Object newfst)
           { return new Pair(newfst, this.snd); }
}
```

Si noti che vengono sempre indicati in modo esplicito (i) la superclasse, compreso Object, (ii) la chiamata super() al costruttore della superclasse, (iii) l'indicazione this.fst, in quanto this è considerata una variabile come le altre, non una keyword. Si noti inoltre come la definizione del metodo setfst realizzi una semantica funzionale per gli oggetti. Allora esempi di programmi sono:

- Selezione di campo: (CT, new Pair(new A(), new B()).snd) che valuterà a new B()
- Invocazione di metodo: (CT, new Pair(new A(), new B()).setfst(new B())) che valuterà a new Pair(new B(), new B())}
- Cast: ((Pair)(new Pair(new Pair(new A(), new B()), new A()).fst)).snd che valuterà a new B(). Si osservi che il cast è necessario perchè la selezione del campo fst restituisce un valore di tipo Object, che va quindi reso di tipo Pair prima di selezionarne il campo snd.

Gli *errori-runtime*, cioè i termini stuck in cui la computazione non può proseguire sono i seguenti: (i) l'accesso ad un campo o ad un metodo non definito nella classe corrispondente, (ii) il passaggio di parametri di numero e/o tipo scorretto e (iii) il tentativo di fare un cast ad untipo che non è una superclasse del tipo run-time dell'oggetto. Dimostreremo che gli errori di tipo (i) e (ii) (detti anche *message not understood errors*) non compaiono in programmi ben tipati, mentre gli errori di tipo (iii) non compaiono in programmi ben tipati che non contengono down cast.

9.4 Sintassi

Iniziamo la presentazione formale del linguaggio FJ descrivendone la sintassi. Assumiamo un insieme infinito di variabili $x, y, ...$, un insieme infinito di nomi di classi $A, B, C, ...$ e un insieme infinito di identificatori di campi dati e di metodi, che indicheremo rispettivamente con $f, g, ...$ e $m, n, ...$

$$\begin{array}{lrl} \textit{Dichiarazione di classe} & CL & ::= \ \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{A}\,f;\ K\ \tilde{M}\} \\ \textit{Dichiarazione costruttore} & K & ::= \ C(\tilde{A}\,g, \tilde{B}\,f)\ \{\mathsf{super}(\tilde{g});\ \mathsf{this}.\tilde{f} = \tilde{f}\} \\ \textit{Dichiarazione di metodo} & M & ::= \ C\ m(\tilde{A}\,x)\{\mathsf{return}\ t;\} \end{array}$$

$$\begin{array}{lrll} \textit{Termini} & t & ::= & x & \text{variabile} \\ & & & t.f & \text{selezione di campo} \\ & & & t.m(\tilde{t}) & \text{invocazione metodo} \\ & & & \mathsf{new}\,C(\tilde{t}) & \text{creazione oggetto} \\ & & & (C)\,t & \text{cast di tipo} \end{array} \qquad \begin{array}{lrl} \textit{Valori} & v, u & ::= \ \mathsf{new}\,C(\tilde{v}) \end{array}$$

Assumiamo che ci sia una variabile this che non si usa mai come parametro formale, ma che all'interno del corpo di un metodo è implicitamente legata all'oggetto di invocazione (vedi regola di semantica). Indichiamo con $\tilde{f}$ la tupla $f_1, .., f_n$, e analogamente $\tilde{C}\tilde{f}$ indica $C_1 f_1, ..., C_n f_n$. Assumiamo inoltre che non compaiano nomi duplicati nelle liste di nomi di campi dato, nomi di metodi, nomi di parameteri formali.

La dichiarazione di classe $\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{C}\tilde{f};\ K\ \tilde{M}\}$ dichiara una classe C sottoclasse di D, che quindi eredita i campi e i metodi di D, aggiungendone di nuovi. Assumiamo che i campi dati $\tilde{f}$ in C abbiano nomi distinti da quelli ereditati da D, evitando cioè il fenomeno di oscuramento presente in Java. È invece possibile che l'elenco dei metodi $\tilde{M}$ di C conenga alcuni metodi definiti in D, effettuando dunque un overriding di tali metodi.

La dichiarazione di costruttore $C(\tilde{A}\,g, \tilde{B}\,f)\,\{\mathsf{super}(\tilde{g});\ \mathsf{this}.\tilde{f} = \tilde{f}\}$ indica come inizializzare tutti i campi dati, prima quelli ereditati dalla superclasse, poi quelli nuovi definiti in C. L'uso corretto del costruttore sarà forzato dalla regola di tipo che tipa le classi.

Una Class Table CT è una funzione che associa ad ogni nome di classe C una dichiarazione CL. Assumiamo che una class table sia ben fatta, cioè che se una classe C appare da qualche parte nella class table ci sia una corrispondente dichiarazione di C. Supponiamo inoltre che nella class table non ci sia una entry per Object, che è una classe speciale senza superclasse e senza alcun metodo (diversamente da Java).

EXERCISE 9.1. Si noti che una class table può contenere definizioni di classi mutuamente ricorsive. Scrivere un esempio. □

La relazione di sottotipo è definita dalle seguenti regole:

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\{...\}}{C <: D} \qquad \frac{}{C <: C} \qquad \frac{C <: D \qquad D <: E}{C <: E}$$

Assumiamo che in una class table ben fatta la relazione di sottotipo indotta non contenga cicli. Infine, un Programma è una coppia (CT, t) contenente una class table e un termine da valutare.

9.5 Semantica Operazionale

Definizioni ausiliarie utili:

Lookup di campi: $fields(C) = \tilde{C}\,\tilde{f}$

$$fields(\mathsf{Object}) = \emptyset$$

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{C}\,\tilde{f};\ K\ \tilde{M}\} \quad fields(D) = \tilde{D}\,\tilde{g}}{fields(C) = \tilde{D}\,\tilde{g},\ \tilde{C}\,\tilde{f}}$$

Lookup di definizioni di metodi $mbody(m, C) = (\tilde{x}, t)$

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{C}\,\tilde{f};\ K\ \tilde{M}\} \quad B\ m\ (\tilde{B}\,\tilde{x})\ \{\mathsf{return}\ t;\} \in \tilde{M}}{mbody(m, C) = (\tilde{x}, t)}$$

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{C}\,\tilde{f};\ K\ \tilde{M}\} \quad m \text{ non definito in } \tilde{M}}{mbody(m, C) = mbody(m, D)}$$

Definiamo ora la semantica operazionale tramite un sistema di regole di transizione tra termini del linguaggio FJ: $t \longrightarrow t'$. Si osservi che i valori che risultano da una terminazione corretta sono oggetti costruiti passando al costruttore dei paramentri completamente valutati, i.e. $\mathsf{new}\,C(\tilde{v})$.

(PROJNEW)
$$\frac{fields(C) = \tilde{C}\,\tilde{f} \quad f_i \in \tilde{f}}{(\mathsf{new}\,C(\tilde{v})).f_i \longrightarrow v_i}$$

(INVKNEW)
$$\frac{mbody(m, C) = (\tilde{x}, t) \quad |\tilde{x}| = |\tilde{v}'|}{(\mathsf{new}\,C(\tilde{v})).m(\tilde{v}') \longrightarrow t\{\tilde{x} := \tilde{v}',\ \mathsf{this} := \mathsf{new}\,C(\tilde{v})\}}$$

(FIELD)
$$\frac{t \longrightarrow t'}{t.f \longrightarrow t'.f}$$

(INVKRECV)
$$\frac{t \longrightarrow t'}{t.m(\tilde{t}) \longrightarrow t'.m(\tilde{t})}$$

(NEWARG)
$$\frac{t_i \longrightarrow t_i'}{\mathsf{new}\,C(\tilde{v}, t_i, \tilde{t}) \longrightarrow \mathsf{new}\,C(\tilde{v}, t_i', \tilde{t})}$$

(INVKARG)
$$\frac{t_i \longrightarrow t_i'}{v.m(\tilde{v}, t_i, \tilde{t}) \longrightarrow v.m(\tilde{v}, t_i', \tilde{t})}$$

(CASTNEW)
$$\frac{C <: D}{(D)(\mathsf{new}\,C(\tilde{v})) \longrightarrow \mathsf{new}\,C(\tilde{v})}$$

(CAST)
$$\frac{t \longrightarrow t'}{(C)t \longrightarrow (C)t'}$$

La regola (INVKNEW) illustra che la semantica scelta per il passaggio dei parametri è del tipo call-by-value. La semantica del cast, in accordo con quanto succede in Java, controlla che il tipo dell'oggetto sia un sottotipo del tipo target del cast, se questo è il caso l'operazione di cast viene rimossa e l'oggetto non viene modificato in alcun modo. A differenza di FJ in Java se il cast non ha successo viene sollevata un'eccezione, in FJ semplicemente il termine è stuck.

EXERCISE 9.2. Descrivere la semantica operazionale dei seguenti termini:

- new Pair(new A(), new B()).snd
- (Pair)new Pair(new A(), new B())
- new Pair(new A(), new B()).setfst(new B())
- ((Pair) (new Pair(new Pair(new A(), new B()), new A()).fst)).snd
- (B) ((A)new C())

□

EXERCISE 9.3. Scrivere un programma con override di un metodo e descriverne la valutazione, evidenziando il binding dinamico per la chiamata del metodo riscritto. □

9.6 Sistema di tipi

Diamo ora le regole per tipare i termini del linguaggio FJ. Sia Γ un insieme di assunzioni di tipo per variabili, della forma $x : C$. Il type system darà un tipo ai programmi, quindi utilizza giudizi di tre forme:

- $\Gamma \vdash t : C$ per indicare che il termine t ha tipo C in Γ.
- $D\ m\ (\tilde{C\ x})\ \{\mathsf{return}\ t;\}\ OK\ in\ C$ per indicare che il metodo m è ben formato nella classe C
- $\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{C\ f};\ K\ \tilde{M}\}\ OK$ per indicare che la definizione della classe C è ben formata.

Un programma (CT, t) è dunque ben tipato se sono derivabili i giudizi $CT\ OK$ e $\emptyset \vdash t : C$ per qualche tipo C.

Il sistema di regole è in forma **algoritmica**, cioè invece di avere una regola di subsumption, il subtyping è inserito direttamente nei punti in cui può essere usato. In questo modo per ogni termine (ad eccezione del cast) c'è una sola regola di tipo che si applica.

Tipi dei termini

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C}\ (\textsc{Var}) \qquad \frac{|\tilde{f}| = |\tilde{t}| \quad fields(C) = \tilde{D\ f} \quad \Gamma \vdash \tilde{t} : \tilde{A} \quad \tilde{A} <: \tilde{D}}{\Gamma \vdash \mathsf{new}\ C(\tilde{t}) : C}\ (\textsc{New})$$

$$\frac{\Gamma \vdash t' : C \quad mtype(m, C) = \tilde{A} \to B \quad |\tilde{t}| = |\tilde{A}| \quad \Gamma \vdash \tilde{t} : \tilde{D} \quad \tilde{D} <: \tilde{A}}{\Gamma \vdash t'.m(\tilde{t}) : B}\ (\textsc{Invk})$$

$$\frac{\Gamma \vdash t : C \quad fields(C) = \tilde{D\ f} \quad f_i \in \tilde{f}}{\Gamma \vdash t.f_i : D_i}\ (\textsc{Field}) \qquad \frac{\Gamma \vdash t : D \quad D <: C}{\Gamma \vdash (C)t : C}\ (\textsc{Ucast})$$

$$\frac{\Gamma \vdash t : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)t : C}\ (\textsc{Dcast}) \qquad \frac{\Gamma \vdash t : D \quad C \not<: D \quad D \not<: C \quad warning}{\Gamma \vdash (C)t : C}\ (\textsc{Stupid cast})$$

Dove il predicato ausiliario *mytype* è definito come segue:

$$\frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{C\ f};\ K\ \tilde{M}\} \quad B\ m\ (\tilde{A\ x})\ \{\mathsf{return}\ t;\} \in M}{mtype(m, C) = \tilde{A} \to B} \qquad \frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\tilde{C\ f};\ K\ \tilde{M}\} \quad m \text{ non definito in } M}{mtype(m, C) = mtype(m, D)}$$

Si osservi che il nostro sistema di tipi considera ben tipato anche un cast in cui il tipo di partenza e il tipo target non sono in relazione di subtyping, mentre questo caso è rifiutato dal compilatore Java. Questa regola è necessaria in FJ per provare il teorema di preservazione dei tipi rispetto alla semantica small step, cioè per dare un tipo a tutti i termini *intermedi* a cui valuta un programma ben tipato. Può succedere infatti che un termine che non contiene stupid cast evolva ad un termine che li contiene. Consideriamo ad esempio il seguente programma, dove A e B sono classi non legate da alcuna relazione:

$$(A)((\mathsf{Object})\mathsf{new}\ B()) \longrightarrow (A)\mathsf{new}\ B()$$

Il termine iniziale contiene un upcast e un downcast, mentre il termine finale contiene uno stupid cast. La regola (STUPID CAST) contiene uno warning proprio per indicare la natura speciale/strana di questa

regola. Di conseguenza una derivazione di tipo in FJ corrisponde ad una derivazione di tipo legale in Java solo se non contiene questa regola.

EXERCISE 9.4. Perché c'e' una regola di tipo sia per l'upcast che per il downcast, mentre c'è la sola regola di valutazione per Up-cast, nella semantica operazionale? □

EXERCISE 9.5. Ha senso aggiungere a FJ la regola di subtyping per i tipi freccia $A \to B$? □

Tipi dei metodi e delle classi La definizione della classe C è ben formata se il suo costruttore K chiama il costruttore della superclasse con i giusti parametri e inizializza correttamente tutti i campi dati della classe C, e se tutti i metodi di C sono ben formati in C. A sua volta, un metodo m è ben formato nella classe C se il suo corpo rispetta la seguantura del metodo. In più il predicato $override(m, D, \tilde{C} \to A)$ indica che, se un metodo m di tipo $\tilde{C} \to A$ ridefisce un metodo della (super)classe D, allora la segnatura di m in C è uguale alla segnatura di m in D[12].

(C OK))

$$\frac{K = C(\tilde{D}\, g, \tilde{C}\, f)\,\{\mathsf{super}(\tilde{g});\ \mathsf{this}.\tilde{f} = \tilde{f};\}\quad fields(D) = \tilde{D}\, g\quad \tilde{M}\ OK\ in\ C}{\mathsf{class}\ C\ \mathsf{extends}\ D\,\{\tilde{C}\, f;\ K\ \ \tilde{M}\}\ OK}$$

(M OK IN C)

$$\frac{\tilde{x} : \tilde{C}, \mathsf{this} : C \vdash t : B\quad B <: A\quad CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\,\{...\}\quad override(m, D, \tilde{C} \to A)}{A\ m\ (\tilde{C}\, x)\ \{\mathsf{return}\, t;\}\ OK\ in\ C}$$

$$\frac{\text{Se}\ \ mytype(m, D) = \tilde{D} \to B\ \ \text{allora}\ \ \tilde{C} = \tilde{D}\ \text{e}\ A = B}{override(m, D, \tilde{C} \to A)}$$

9.7 Proprietà del sistema di tipi

Avendo dato il sistema di tipi in versione algoritmica, cioè senza regola di subsumption, il teorema di preservazione dei tipi dice che durante la valutazione il tipo di un termine può *decrescere*.

Theorem 9.6 (Subject Reduction)**.** *Se* $\Gamma \vdash t : C$ *e* $t \longrightarrow t'$ *allora* $\Gamma \vdash t' : C'$ *per qualche* $C' <: C$ □

Il teorema di progressione dice che un termine chiuso, ben tipato, che non riduce ulteriormente, è un valore oppure contiene un downcast oppure uno stupid cast. In altri termini, se un programma ben tipato è stuck, è perché contiene un downcast o uno stupid cast (ma non perché si cerca di selezionare un campo/metodo inesistente o perché il passaggio dei parametri ad un metodo è scorretto).

Theorem 9.7 (Progress)**.** *Sia* t *un termine chiuso, ben tipato, i.e.,* $\varnothing \vdash t : C$*. Allora o* $t \longrightarrow t'$ *per qualche* t' *oppure* t *è un valore, oppure* $\exists E$*, evaluation context, such that* $t = E[(C)(\mathsf{new}\, D(\tilde{v}))]$ *con* $D \not<: C$*.* □

Dove gli evaluation contexts sono definiti come segue:

$$E\ ::=\ [\,] \mid E.f \mid E.m(\tilde{t}) \mid v.m(\tilde{v}, E, \tilde{t}) \mid \mathsf{new}\, C(\tilde{v}, E, \tilde{t}) \mid (C)E$$

Ad esempio, sia $E[\,] = \mathsf{new}\, C().m(v, [\,], t')$, e dato un termine t si ha $E[t] = \mathsf{new}\, C().m(v, t, t')$.

Theorem 9.8 (Safety)**.** *Sia* t *un programma ben tipato, allora* t *non evolve runtime ad un termine che contiene un errore del tipo message-not-understood.* □

Per quanto riguarda i cast, osserviamo che un termine FJ che contiene un down-cast può ridursi a un termine con stupid cast, mentre un termine FJ che non contiene down-cast si riduce sempre a termini che non contengono né down-cast né stupid cast. Possiamo quindi dare un teorema più stringente per questi termini.

[12] Notare che $override(m, D, \tilde{C} \to A)$ è vero anche se m è un metodo nuovo aggiunto nella classe C.

Definition 9.9. Diciamo che un termine t ben tipato è *cast safe* se contiene solo up-cast, i.e. se è ben tipato senza usare le regole di tipo (DCAST) e (STUPID CAST). □

Proposition 9.10. *Se t è cast safe e $t \longrightarrow t'$ allora anche t' è cast safe.* □

Theorem 9.11 (Safety dei termini Cast Safe)**.** *Sia t un programma ben tipato cast safe, allora t non evolve ad un termine stuck, i.e. se $t \longrightarrow^* t' \not\longrightarrow$ allora t' è un valore.* □

Il teorema di safety dice che solo i programmi senza downcast terminano correttamente, ma i programmi Java in genere contengono down-cast; e in effetti questi programmi possono "fallire" a runtime. Il meccanismo del sollevamento/gestione delle eccezioni `ClassCastException` serve proprio per gestire un tale fallimento senza far abortire il programma.

EXERCISE 9.12. Aggiungere a FJ il termine ClassCastException: come cambia la semantica operazionale? Le regole di tipo? Il teorema di Safety e i teoremi di Preservazione e Progressione? Aggiungere in seguito anche la possibilit di gestire le eccezioni. □