

# Meccanismi di condivisione ereditarietà, delega, concatenazione

Riccardo Solmi  
6 luglio 1999

## Abstract

Confronto tre meccanismi di condivisione: ereditarietà, delega e concatenazione. Definisco sintassi, semantica e tipi di tre nuclei di linguaggio scelti in modo da differire solo per il meccanismo di condivisione. Dimostro l'equivalenza dei tre linguaggi e quindi dei tre meccanismi.

## Introduzione

L'ereditarietà è il meccanismo di condivisione più diffuso ed è spesso indicata come la proprietà distintiva che differenzia la programmazione orientata agli oggetti da altri paradigmi. Il termine ereditarietà è spesso usato come sinonimo di meccanismo di condivisione.

Sono stati fatti alcuni tentativi di definire una tassonomia per i meccanismi di condivisione esistenti. L'intento dichiarato in questi scritti è stato di classificare tutte le possibili varianti; nessun tentativo è stato fatto per isolare i meccanismi da proprietà accidentali non esclusive e confrontarli con metodi formali in modo da individuare le reali differenze. Inoltre, in letteratura è rappresentata prevalentemente la contrapposizione fra meccanismi basati su classi e meccanismi basati su oggetti; questi ultimi identificati con il meccanismo di delega. Solo di recente è stata compresa l'originalità e la rilevanza della concatenazione.

Le idee alla base della programmazione class-based e object-based si possono fare risalire agli inizi degli anni sessanta; diciamo rispettivamente con Simula nel 1967 e Sketchpad nel 1963. I linguaggi object-based vengono usati in studi fondazionali e programmazione esplorativa ma anche in alcuni ambiti accessibili ad un pubblico più vasto come i palmari (NewtonScript) e i browser (JavaScript). I sostenitori della programmazione basata su oggetti attribuiscono al proprio modello maggiore concretezza, semplicità concettuale e flessibilità. Non di meno il paradigma dominante è quello basato su classi e ad oltre trent'anni di distanza nessun linguaggio object-based può essere considerato un serio antagonista di linguaggi come C++ o Java.

Sono state avanzate diverse ipotesi per spiegare le ragioni di questo insuccesso. Gran parte dei meriti e dei limiti descritti in letteratura e attribuiti all'uno o all'altro modello sono in realtà da attribuire a proprietà indipendenti e applicabili ad entrambi i modelli e che solo per ragioni storiche si trovano prevalentemente adottate nell'uno o nell'altro. Proprietà come le operazioni sugli attributi, l'uniformità di trattamento degli attributi e l'ereditarietà dinamica sono presenti nella maggior parte dei linguaggi object-based ma non in quelli class-based. Queste proprietà sono una naturale estensione dei linguaggi basati su oggetti ma non sono intrinsecamente legate ad essi come dimostrano alcuni linguaggi basati su classi (Hybrid).

Ogni meccanismo di condivisione definisce un modello concettuale con una propria terminologia ed una propria metafora di riferimento. Altre differenze sono osservabili nel modello di implementazione e nella rappresentazione; ancora una volta è importante chiedersi quali differenze siano inerenti ai modelli e quali invece siano motivate solo da ragioni storiche e possano essere superate.

Un meccanismo di condivisione comprende un meccanismo di conformità ed uno di allocazione. In questo scritto vengono presi in considerazione tre meccanismi di conformità: ereditarietà, delega e concatenazione, e due meccanismi di allocazione: istanziazione e clonazione. L'istanziazione e l'ereditarietà sono implementati nella maggior parte dei linguaggi basati su classi come ad esempio Java; la clonazione assieme a delega o a concatenazione sono implementati prevalentemente in linguaggi basati su oggetti come rispettivamente Self e Kevo.

Il confronto non può essere fatto direttamente sui linguaggi citati perché oltre al meccanismo che vogliamo analizzare possiedono altre proprietà indipendenti che condizionerebbero l'esito del confronto. Il confronto deve essere fatto alla pari isolando il meccanismo di condivisione: eliminando ovvero vincolando gli altri meccanismi in modo da ottenere tre nuclei di linguaggio sostanzialmente identici a meno del meccanismo che ci interessa.

Lo scritto è organizzato in sezioni come segue. Prima descrivo i tre meccanismi di condivisione. Poi definisco informalmente i linguaggi da confrontare individuando le proprietà indipendenti e motivando la scelta di scartarle. Poi definisco sintassi, semantica e tipi dei tre linguaggi e ne dimostro l'equivalenza. Infine considero le conseguenze della equivalenza nei modelli di implementazione e di rappresentazione dei linguaggi.

## Meccanismi di condivisione

Un meccanismo di condivisione ha due aspetti fondamentali: conformità e allocazione. Non è possibile definire un aspetto in termini dell'altro. Il meccanismo di conformità consente di avere una stessa informazione accessibile e modificabile in più contesti. Il meccanismo di allocazione consente di ottenere, data una informazione, due informazioni accessibili e modificabili separatamente.

I due aspetti di un meccanismo di condivisione possono essere meglio precisati rendendo le definizioni ortogonali. Un meccanismo dinamico e un meccanismo strutturale. Il primo rende disponibile una informazione in contesti diversi; il secondo permette di stabilire se l'informazione resa disponibile debba intendersi la stessa o distinta.

### **Meccanismi di conformità: ereditarietà, delega, concatenazione**

Un meccanismo di conformità consente di avere una stessa informazione accessibile e modificabile in più contesti. La definizione data vincola gli estremi dello spazio delle soluzioni. Da un lato possiamo mantenere una unica informazione, dall'altro possiamo replicarla per ogni accesso. Nel primo caso le richieste di accesso e modifica dovranno essere inoltrate dai contesti interessati all'informazione e la modifica verrà eseguita su quell'unica informazione (ereditarietà, delega). Nel secondo caso le richieste di accesso e modifica verranno eseguite localmente ma la modifica dovrà coinvolgere l'informazione in tutti i contesti in cui è replicata (concatenazione).

#### Ereditarietà

Una classe definisce l'insieme dei metodi e dei campi condivisi da tutte le istanze della classe e un insieme di campi specifici per ogni istanza. Le definizioni vengono ereditate da tutte le sottoclassi; in questo senso l'ereditarietà può essere definita un meccanismo di modificazione incrementale in presenza di un riferimento all'oggetto (*this*) legato in base al contesto (late-bound).

L'ereditarietà non può implementare la delega. La variabile riservata *this* viene automaticamente legata all'oggetto ricevente la richiesta di accesso e non cambia; in questo modo l'esecuzione di un metodo di una superclasse avviene come se fosse un metodo dell'oggetto originale. D'altra parte una invocazione ad un qualsiasi altro metodo comporta un cambiamento di *this*.

#### Delega

Un concetto viene rappresentato in due oggetti uno con i tratti comuni e l'altro prototipico contenente gli aspetti specifici delle istanze. Gli attributi condivisi risiedono in appositi oggetti (traits). Tali oggetti generalmente non sono concreti nel senso che contengono metodi che accedono a campi non presenti nell'oggetto. E' opportuno evitare l'accesso diretto a questi metodi aggiungendo una direttiva abstract che proibisca anche la clonazione.

Ogni oggetto mantiene dei riferimenti agli oggetti (traits) con gli attributi condivisi. Quando un oggetto riceve una richiesta di accesso ad un attributo che non possiede lo inoltra (delega). La modifica di un attributo condiviso è singola e si propaga automaticamente e inevitabilmente a tutti gli oggetti che condividono l'attributo. L'accesso ad un attributo condiviso avviene tramite delega all'oggetto che lo contiene. La relazione classe/sottoclasse è espressa dalla delega.

#### Concatenazione

Gli oggetti sono autosufficienti e concreti. Ogni oggetto concettualmente mantiene una propria copia di ogni attributo. Gli oggetti possono essere definiti senza la necessità di definire una gerarchia di classi. Ogni oggetto che condivide un attributo con altri oggetti ne riceve una copia; il sistema mantiene traccia degli attributi clonati da uno stesso prototipo (clone family). Gli oggetti possono essere modificati come singoli individui. L'accesso all'attributo è diretto, la modifica viene estesa a tutte le repliche. L'accesso ai metodi condivisi è uguale agli altri e non richiedono una diversa interpretazione di self.

### **Meccanismi di allocazione: istanziamento, clonazione**

Un meccanismo di allocazione consente di ottenere, data una informazione, due informazioni accessibili e modificabili separatamente.

#### Istanziamento

Definisco singole classi. L'operazione di istanziamento crea un nuovo oggetto a partire dalle definizioni della classe data e di tutte le classi da cui eredita. Gli attributi non condivisi provengono da tutta la gerarchia di ereditarietà. (copy down). E' un complemento obbligato per i meccanismi di conformità che non definiscono un prototipo delle istanze.

#### Clonazione

Definisco singoli oggetti. L'operazione di clonazione crea un nuovo oggetto duplicando un prototipo. Un oggetto può assumere il ruolo di prototipo se definisce tutti i campi che usa. I riferimenti al prototipo non vengono coinvolti dalla clonazione.

## Scelta dei tre nuclei di linguaggio

In questa sezione vengono analizzate alcune proprietà indipendenti dai meccanismi di condivisione. Vengono fatte delle scelte dettate dalla volontà di uniformare i tre linguaggi riducendoli al minimo comune denominatore. Là dove non sembrava evidente la neutralità di una proprietà esclusa è stata fornita una indicazione su come aggiungerla agli altri linguaggi. Sono state escluse anche proprietà che sono presenti nella quasi totalità dei linguaggi come il passaggio di parametri e il ritorno di valori nelle chiamate a metodi. Le scelte fatte riducono i tre linguaggi a nuclei ben al di sotto della soglia di usabilità ma permettono di apprezzare le differenze dovute ai diversi meccanismi di condivisione.

### **Operazioni sugli attributi temporalmente illimitate**

Nei linguaggi basati su oggetti le operazioni sugli attributi sono spesso esplicite e ammesse in qualsiasi momento (operazioni sui record). Nei linguaggi basati su classi in genere sono implicite e limitate alla definizione della classe (sintattiche). I tre nuclei di linguaggi definiti ammettono solo operazioni sintattiche di aggiunta di campi e di aggiunta e ridefinizione di metodi nell'ambito delle definizioni di nuove classi/oggetti. Sono quindi escluse l'eliminazione di attributi e la possibilità di modificare classi/oggetti già definiti.

### **Uniformità di trattamento degli attributi**

La maggior parte dei linguaggi basati su oggetti gestisce in modo uniforme campi e metodi. In questo modo, l'accesso ad un campo non è distinguibile dalla chiamata ad un metodo e il campo può essere sostituito da un metodo che lo calcola, senza modificare il codice degli oggetti cliente.

Nei linguaggi in cui l'accesso ad un attributo è interpretabile come invio di un messaggio basta variare l'interpretazione del messaggio a seconda o meno della presenza di un metodo di accesso al campo. In linguaggi come Java in cui è più appropriata una interpretazione di accesso tramite scostamento rispetto la base dell'oggetto si può fare in modo che il compilatore definisca implicitamente (al pari del costruttore di default) i metodi di lettura e scrittura degli attributi e interpreti gli accessi e gli assegnamenti come zucchero sintattico. In questo modo le definizioni di nuovi metodi di accesso sarebbero automaticamente utilizzati da tutte le classi cliente senza modifiche né ricompilazioni; inoltre, in presenza di un compilatore JIT non si avrebbe alcun calo di prestazioni.

Per semplicità i tre linguaggi considerati distinguono fra campi e metodi.

### **Attributi condivisi**

Un attributo condiviso è un attributo accessibile a più oggetti e le cui modifiche, se si tratta di un campo, sono osservabili da tutti gli oggetti che possono accedere al campo.

Il modello basato su delega supporta in modo naturale gli attributi condivisi in particolare garantisce la non escludibilità della condivisione. Per avere analoghe garanzie strutturali nei modelli basati su ereditarietà e su concatenazione è necessario aggiungere una parola chiave per distinguere definizioni di campi condivisi da campi di istanza.

Linguaggi come Java permettono di definire campi e metodi statici. I campi condivisi sono equivalenti ai campi statici di Java ad eccezione del fatto che non necessitano dell'indicazione della classe come risolutore di scope. I metodi statici di Java vengono eseguiti in uno scope limitato ai soli attributi statici e con *this* slegato; è evidentemente possibile aggiungerli a tutti e tre i linguaggi ma data la non rilevanza ai fini del confronto sono stati scartati.

### **Ereditarietà dinamica**

Questa proprietà consiste nella possibilità di definire e modificare a runtime la gerarchia di condivisione.

E' presente in Self e nella maggior parte dei linguaggi basati su delega dei messaggi. L'espressività aggiunta viene in buona parte catturata dai design pattern Strategy e State che sono utilizzabili anche in linguaggi basati su classi servendosi di ridirezione di messaggi anziché delega. La sua esclusione equivale alla determinazione preprogrammata e anticipata dei percorsi di condivisione per gruppo.

### **Dominio dell'allocazione dinamica**

I linguaggi basati su classi forniscono l'operatore *new*, mentre i linguaggi basati su oggetti usano *clone*. Le due operazioni sono diverse; in particolare il dominio di *new* è limitato alle entità definite staticamente cioè le classi; invece la *clone*, limitandosi a copiare, è applicabile a tutti gli oggetti compresi quelli allocati dinamicamente. Per uniformare le due operazioni di allocazione dinamica si possono fare due scelte: o affiancare una *clone* alla *new* nel linguaggio basato su classi (vedi Java) oppure si può limitare il dominio di applicabilità della clone degli altri due linguaggi. E' stata fatta questa seconda scelta. Di conseguenza, data la non applicabilità del concetto di copia profonda all'istanziamento di una classe, ho dovuto anche limitare alla superficie del prototipo la profondità della copia (shallow copy vs deep copy).

### **Costruttori**

I costruttori hanno il compito di inizializzare lo stato degli oggetti allocati dinamicamente. Il costruttore viene invocato dalla operazione di allocazione dinamica (*new/clone*) e dà garanzie di uniformità degli oggetti creati. Di solito solo i

linguaggi basati su classi supportano i costruttori ma è solo una scelta di comodo per quanto ragionevole. Un costruttore è opportuno per le classi e per i prototipi in modo da inizializzare i campi; nel caso di una *clone* di un oggetto allocato dinamicamente sono già molto meno utili. I tre linguaggi presentati non supportano i costruttori.

### ***Istanziabilità***

La maggior parte dei linguaggi orientati agli oggetti distingue sintatticamente le entità istanziabili (con *new/clone*) da quelle che partecipano unicamente alla gerarchia di condivisione. Le entità non istanziabili prendono solitamente i nomi di, rispettivamente, classi astratte, tratti, aspetti. Una classe astratta dichiara metodi che non definisce. Tratti ed aspetti accedono ad attributi non definiti.

### ***Classi vs prototipi***

Una sottoclasse/sottoprototipo può aggiungere o ridefinire (*override*) metodi. Non vi è necessariamente una relazione fra il comportamento di una classe/prototipo e quello delle sue sottoclassi/sottoprototipi. I metodi ridefiniti possono avere un comportamento completamente diverso. Fanno eccezione linguaggi come BETA che considerano la ridefinizione come una estensione della definizione precedente.

I modelli basati su classi hanno concettualmente il problema della regressione infinita delle metaclassi. Una classe è un oggetto di un tipo che contiene un template per oggetti di un altro tipo e nessun oggetto è autosufficiente.

# Sintassi

Siano  $T, T'$  identificatori di classi o di oggetti,  $m$  identificatore di metodi e  $a, b$  identificatori di campi. Le definizioni  $D$ , i corpi  $B$ , le istruzioni  $S$ , le espressioni  $E$  e gli identificatori qualificati  $Q$  sono definiti per ogni linguaggio dalle sintassi che seguono.

## Descrizione

Le tre sintassi sono molto simili tra loro. L'unica differenza sostanziale è la necessità di distinguere fra attributi locali e condivisi nei linguaggi  $\langle_e$  e  $\langle_c$  in modo che questi possano dare le stesse garanzie strutturali del linguaggio  $\langle_d$ . Le altre differenze riguardano le parole chiave che, per scelta, rispettano la terminologia corrente di ciascun modello. Descrivo brevemente le sintassi.

## Definizioni

Un programma  $D$  è un insieme di definizioni (**class/object**). Ognuna può essere definita autonomamente o in relazione ad un'altra (**extends/delegates/embeds**).

## Corpi

Il corpo  $B$  di una classe o oggetto consiste di dichiarazioni di campi  $T a$  e definizioni di metodi  $m() \{ S \}$ . I metodi non ammettono né parametri né valori di ritorno. I linguaggi  $\langle_e$  e  $\langle_c$  distinguono fra attributi locali e condivisi (**static/shared**). Nel primo caso ogni oggetto (istanziato o clonato) ha la propria copia dell'attributo, nel secondo caso un'unica copia dell'attributo viene condivisa da tutti gli oggetti. Lo stesso comportamento viene ottenuto nel linguaggio  $\langle_d$  definendo gli attributi al livello appropriato della gerarchia di delega.

## Istruzioni

Il corpo  $S$  di un metodo è una sequenza di istruzioni eventualmente vuota. Sono ammesse istruzioni di assegnamento e di invocazione. Un assegnamento è usato per cambiare il valore di un campo. L'invocazione di un metodo comporta l'esecuzione del corpo del metodo.

## Espressioni

Una espressione  $E$  consiste nella creazione di un nuovo oggetto (**new/clone**) oppure nell'accesso ad un campo tramite un identificatore qualificato.

## Identificatori qualificati

Gli identificatori qualificati permettono di accedere agli attributi e agli attributi dei campi dell'oggetto su cui è stato invocato il metodo. Le parole chiave **this** e **self** denotano un riferimento all'oggetto su cui è stato invocato il metodo. Le parole chiave **super**, **delegated** e **embedded** permettono di aggirare la ridefinizione di un metodo.

### **Sintassi del nucleo basato su ereditarietà (< e)**

D ::= **class** T { B }  
| **class** T **extends** T' { B }  
| D ; D

B ::= **static** T a  
| T a  
| m() { S }  
| B ; B

S ::= ε  
| Q = E  
| Q()   
| S ; S

E ::= **new**T  
| Q

Q ::= **this**  
| a  
| m  
| **super.m**  
| a.b  
| a.m

### **Sintassi del nucleo basato su delega (< d)**

D ::= **object** T { B }  
| **object** T **delegates** T' { B }  
| D ; D

B ::= T a  
| m() { S }  
| B ; B

S ::= ε  
| Q = E  
| Q()   
| S ; S

E ::= **clone** T  
| Q

Q ::= **self**  
| a  
| m  
| **delegated** m  
| a.b  
| a.m

### **Sintassi del nucleo basato su concatenazione (< c)**

D ::= **object** T { B }  
| **object** T **embeds** T' { B }  
| D ; D

B ::= **shared** T a  
| T a  
| m() { S }  
| B ; B

S ::= ε  
| Q = E  
| Q()   
| S ; S

E ::= **clone** T  
| Q

Q ::= **self**  
| a  
| m  
| **embedded** m  
| a.b  
| a.m

## Semantica

Per specificare la semantica definiamo una relazione  $\mathcal{A}, \mathcal{M} \Vdash_{\mathbf{e}} \mathcal{M}$ , dove  $\mathcal{A}$  e  $\mathcal{M}$  implementano

una memoria dereferenziata;  $\mathbf{e}$  è un ambiente che associa una entità alla propria definizione.

In questa sezione il termine *entità* è usato per significare classe o prototipo in modo da uniformare la descrizione dei tre linguaggi.

### La memoria dereferenziata

$\mathcal{A}$  mappa identificatori di entità e di oggetti in triple  $\langle T_r, T_c, l \rangle$  dove  $T_r$  è il tipo reale,  $T_c$  è il tipo corrente usato nella semantica di **super/delegated/embedded** e  $l$  è la locazione dell'oggetto. Un identificatore è un nome di classe o di prototipo oppure la locazione di un oggetto creato dinamicamente oppure infine la variabile speciale **this/self**. Si assume l'esistenza e implicitamente l'uso di una funzione biunivoca che mappa locazioni in identificatori. Dato un identificatore di oggetto  $o$  e una tripla  $\langle T_r, T_c, l \rangle$ , l'aggiornamento  $\mathcal{A}[o \mapsto \langle T_r, T_c, l \rangle]$  è definito come la funzione  $\mathcal{A} \dot{\mathcal{C}} \text{con } \mathcal{A} \dot{\mathcal{C}}(x) = \mathcal{A}(x) \text{ " } x \neq o \text{ e } \mathcal{A} \dot{\mathcal{C}}(o) = \langle T_r, T_c, l \rangle$ .

$\mathcal{M}$  mappa locazioni in valori; i valori sono locazioni. Una locazione contiene un campo di un record;  $l.a$  denota l'accesso al campo  $a$  del record che inizia alla locazione  $l$ . Date le locazioni  $l$  e  $v$ , l'aggiornamento  $\mathcal{M}[l \mapsto v]$  è definito come la funzione  $\mathcal{M} \dot{\mathcal{C}} \text{con } \mathcal{M} \dot{\mathcal{C}}(x) = \mathcal{M}(x) \text{ " } x \neq l \text{ e } \mathcal{M} \dot{\mathcal{C}}(l) = v$ .

### L'ambiente $\mathbf{e}$

$\mathbf{e}$  mappa identificatori di entità  $T$  in triple  $\langle T \dot{\mathcal{C}} k, f \rangle$  dove  $T \dot{\mathcal{C}}$  la super entità diretta di  $T$  nella gerarchia di condivisione (superclasse, prototipo delegato, prototipo incorporato);  $k$  mappa nomi di campi in scostamenti rispetto alla locazione di base dell'oggetto;  $f$  mappa nomi di metodi in corpi di metodo.  $\mathbf{e}$  viene usato per indicare l'assenza di una super entità.

### Funzioni ausiliarie

Data la forte somiglianza fra i tre linguaggi e per facilitare il confronto fra le semantiche, sono state definite delle regole uguali per tutti a meno delle parole chiave proprie di ogni linguaggio, e le differenze sono state concentrate in apposite funzioni ausiliarie  $G$  e  $U$ .

$G$  permette di accedere ad un attributo di un oggetto: se l'attributo è un campo restituisce il valore, se è un metodo ne restituisce il corpo.

$U$  permette di aggiornare il valore di un campo e restituisce  $\mathcal{M}$  aggiornato.

### La Clone Family $F$

È usata solo nella semantica del linguaggio  $\langle \mathbf{e} \rangle$ .  $F$  mappa identificatori di entità  $T$  in insiemi di identificatori di oggetti; ogni insieme contiene tutti gli oggetti che incorporano il tipo associato.  $F$  viene usata nella funzione ausiliaria  $U$  in modo da propagare l'aggiornamento di un campo condiviso a tutti gli oggetti che lo contengono.  $F$  deve essere aggiornata ogni volta che viene eseguita una **clone**; la procedura che si occupa dell'aggiornamento è  $C$ .

### Inizializzazione

La semantica di un programma  $D;S$  è definita valutando la configurazione  $\mathcal{A}, \mathcal{M} \Vdash_{\mathbf{e}} S$  in un ambiente  $\mathbf{e}$  costruito con le definizioni  $D$  e con la memoria dereferenziata inizializzata con le entità definite in  $D$  e **this/self** legato ad un oggetto allocato dinamicamente in base alla prima entità definita in  $D$ .

**Semantica del nucleo basato su ereditarietà**

$$A = o \mapsto \langle T_r, T_c, \ell \rangle$$

$$\mathcal{M} = l.a \mapsto v \text{ where } l.a = \ell + \partial\ell \text{ and } k(a) = \partial\ell$$

$$\mathcal{E}(T) = \langle T', k, f \rangle \quad k(a) = \begin{cases} \partial\ell & \text{if } a \in \text{dom}(k) \\ \text{undef} & \text{else} \end{cases} \quad f(m) = \begin{cases} S & \text{if } m \in \text{dom}(f) \\ \text{undef} & \text{else} \end{cases}$$

$$G(o, a) = \begin{cases} v & \text{if } \exists k(a) \\ G(T', a) & \text{if } T' \neq \mathbf{e} \\ \text{undef} & \text{else} \end{cases} \text{ where } \begin{matrix} A(o) = \langle T_r, T_c, \ell \rangle, & \mathcal{E}(T_r) = \langle T', k, f \rangle, \\ k(a) = \partial\ell, & \mathcal{M}(\ell + \partial\ell) = v \end{matrix}$$

$$G(o, m) = \begin{cases} S & \text{if } \exists f(m) \\ G(T', m) & \text{if } T' \neq \mathbf{e} \\ \text{undef} & \text{else} \end{cases} \text{ where } \begin{matrix} A(o) = \langle T_r, T_c, \ell \rangle, & \mathcal{E}(T_r) = \langle T', k, f \rangle, & f(m) = S \end{matrix}$$

$$G(o, \text{super}.m) = \begin{cases} S & \text{if } \exists f'(m) \\ G(T', \text{super}.m) & \text{if } T' \neq \mathbf{e} \\ \text{undef} & \text{else} \end{cases} \text{ where } \begin{matrix} A(o) = \langle T_r, T_c, \ell \rangle, & \mathcal{E}(T_c) = \langle T', k, f \rangle, \\ \mathcal{E}(T') = \langle T'', k', f' \rangle, & f'(m) = S \end{matrix}$$

$$U(\mathcal{M}, o, a, v) = \mathcal{M}[\ell \mapsto v] \parallel G(o, a) = v$$

Regole per le istruzioni:

$$[ASS_1] \frac{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} e \quad \mathcal{A}', \mathcal{M}' \Vdash_{\mathcal{E}} v}{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} a = e} \quad \mathcal{A}', \mathcal{M}' \Vdash_{\mathcal{E}} v \quad (U(\mathcal{M}', \text{this}, a, v) = \mathcal{M}'')$$

$$[ASS_2] \frac{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} e \quad \mathcal{A}', \mathcal{M}' \Vdash_{\mathcal{E}} v'}{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} a.b = e} \quad \mathcal{A}', \mathcal{M}' \Vdash_{\mathcal{E}} v' \quad (G(\text{this}, a) = v, \quad U(\mathcal{M}', v, b, v') = \mathcal{M}'')$$

$$[CALL_1] \frac{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} S \quad \mathcal{A}', \mathcal{M}' \Vdash_{\mathcal{E}} e}{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} m() \quad \mathcal{A}', \mathcal{M}' \Vdash_{\mathcal{E}} e} \quad (G(\text{this}, m) = S)$$

$$[CALL_2] \frac{\mathcal{A}[\text{this} \mapsto \langle T_r, T_c, \ell \rangle], \mathcal{M} \Vdash_{\mathcal{E}} S \quad \mathcal{A}' \mathcal{M}' \Vdash_{\mathcal{E}} e}{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} \text{super}.m() \quad \mathcal{A}'[\text{this} \mapsto \langle T_r, T_c, \ell \rangle], \mathcal{M}' \Vdash_{\mathcal{E}} e} \quad \left( \begin{matrix} \mathcal{A}(\text{this}) = \langle T_r, T_c, \ell \rangle, \\ \mathcal{E}(T_c) = \langle T_c', k, f \rangle, \\ G(\text{this}, \text{super}.m) = S \end{matrix} \right)$$

$$[CALL_3] \frac{\mathcal{A}[\text{this} \mapsto \langle T_r, T_c, \ell \rangle], \mathcal{M} \Vdash_{\mathcal{E}} S \quad \mathcal{A}' \mathcal{M}' \Vdash_{\mathcal{E}} e}{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} a.m() \quad \mathcal{A}'[\text{this} \mapsto \mathbf{a}], \mathcal{M}' \Vdash_{\mathcal{E}} e} \quad \left( \begin{matrix} \mathcal{A}(\text{this}) = \mathbf{a}, & G(\text{this}, a) = v, \\ \mathcal{A}(v) = \langle T_r, T_c, \ell \rangle, & G(v, m) = S \end{matrix} \right)$$

$$[SEQ] \frac{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} S \quad \mathcal{A}', \mathcal{M}' \Vdash_{\mathcal{E}} e \quad \mathcal{A}', \mathcal{M}' \Vdash_{\mathcal{E}} S' \quad \mathcal{A}'', \mathcal{M}'' \Vdash_{\mathcal{E}} e}{\mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} S; S' \quad \mathcal{A}'', \mathcal{M}'' \Vdash_{\mathcal{E}} e}$$

Regole per le espressioni:

$$[CLONE] \mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} \text{new } T \quad \mathcal{A}[\ell' \mapsto \langle T_r, T_c, \ell' \rangle], \mathcal{M}[\ell' \mapsto \text{New}(T)] \Vdash_{\mathcal{E}} \ell' \quad \left( \begin{matrix} \mathcal{A}(T) = \langle T_r, T_c, \ell \rangle, \\ \ell' \text{ new location} \end{matrix} \right)$$

$$[THIS] \mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} \text{this} \quad \mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} \ell \quad (\mathcal{A}(\text{this}) = \langle T_r, T_c, \ell \rangle)$$

$$[FIELD_1] \mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} a \quad \mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} \ell \quad (G(\text{this}, a) = \ell)$$

$$[FIELD_2] \mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} a.b \quad \mathcal{A}, \mathcal{M} \Vdash_{\mathcal{E}} \ell \quad (G(\text{this}, a) = v \quad G(v, b) = \ell)$$

**Semantica del nucleo basato su delega**

$$\mathcal{A} = o \mapsto \langle T_r, T_c, \ell \rangle$$

$$\mathcal{M} = \ell.a \mapsto v \quad \text{where } \ell.a = \ell + \partial\ell \quad \text{and } k(a) = \partial\ell$$

$$\mathcal{E}(T) = \langle T', k, f \rangle \quad k(a) = \begin{cases} \partial\ell & \text{if } a \in \text{dom}(k) \\ \text{undef} & \text{else} \end{cases} \quad f(m) = \begin{cases} S & \text{if } m \in \text{dom}(f) \\ \text{undef} & \text{else} \end{cases}$$

$$G(o, a) = \begin{cases} v & \text{if } \exists k(a) \\ G(T', a) & \text{if } T' \neq \mathbf{e} \text{ where } \mathcal{A}(o) = \langle T_r, T_c, \ell \rangle, \mathcal{E}(T_r) = \langle T', k, f \rangle, \\ \text{undef} & \text{else} \end{cases} \quad k(a) = \partial\ell, \quad \mathcal{M}(\ell + \partial\ell) = v$$

$$G(o, m) = \begin{cases} S & \text{if } \exists f(m) \\ G(T', m) & \text{if } T' \neq \mathbf{e} \text{ where } \mathcal{A}(o) = \langle T_r, T_c, \ell \rangle, \mathcal{E}(T_r) = \langle T', k, f \rangle, f(m) = S \\ \text{undef} & \text{else} \end{cases}$$

$$G(o, \text{delegated } m) = \{G(T', m) \text{ where } \mathcal{A}(o) = \langle T_r, T_c, \ell \rangle, \mathcal{E}(T_c) = \langle T', k, f \rangle\}$$

$$U(\mathcal{M}, o, a, v) = \mathcal{M}[\ell \mapsto v] \parallel G(o, a) = v$$

Regole per le istruzioni:

$$[ASS_1] \frac{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} e \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} v}{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} a = e} \quad (U(\mathcal{M}', \text{self}, a, v) = \mathcal{M}'')$$

$$[ASS_2] \frac{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} e \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} v'}{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} a.b = e} \quad (G(\text{self}, a) = v \quad U(\mathcal{M}', v, b, v') = \mathcal{M}'')$$

$$[CALL_1] \frac{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} S \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} \mathbf{e}}{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} m()} \quad (G(\text{self}, m) = S)$$

$$[CALL_2] \frac{\mathcal{A}[\text{self} \mapsto \langle T_r, T_c, v \rangle], \mathcal{M} \Vdash_{\varepsilon} S \quad \mathcal{A}' \mathcal{M}' \Vdash_{\varepsilon} \mathbf{e}}{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} \text{delegated } m()} \quad \left( \begin{array}{l} \mathcal{A}(\text{self}) = \langle T_r, T_c, v \rangle, \\ \mathcal{E}(T_c) = \langle T_c', k, f \rangle, \\ G(\text{self}, \text{delegated } m) = S \end{array} \right)$$

$$[CALL_3] \frac{\mathcal{A}[\text{self} \mapsto \langle T_r, T_c, \ell \rangle], \mathcal{M} \Vdash_{\varepsilon} S \quad \mathcal{A}' \mathcal{M}' \Vdash_{\varepsilon} \mathbf{e}}{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} a.m()} \quad \left( \begin{array}{l} \mathcal{A}(\text{self}) = \mathbf{a}, \quad G(\text{self}, a) = v, \\ \mathcal{A}(v) = \langle T_r, T_c, \ell \rangle, \quad G(v, m) = S \end{array} \right)$$

$$[SEQ] \frac{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} S \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} \mathbf{e} \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} S' \quad \mathcal{A}'', \mathcal{M}'' \Vdash_{\varepsilon} \mathbf{e}}{\mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} S; S'} \quad \mathcal{A}'', \mathcal{M}'' \Vdash_{\varepsilon} \mathbf{e}$$

Regole per le espressioni:

$$[CLONE] \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} \text{clone } T \quad \mathcal{A}[\ell' \mapsto \langle T_r, T_c, \ell' \rangle], \mathcal{M}[\ell' \mapsto \text{Clone}(T)] \Vdash_{\varepsilon} \ell' \quad \left( \begin{array}{l} \mathcal{A}(T) = \langle T_r, T_c, \ell \rangle \\ \ell' \text{ new location} \end{array} \right)$$

$$[SELF] \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} \text{self} \quad \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} \ell \quad (\mathcal{A}(\text{self}) = \langle T_r, T_c, \ell \rangle)$$

$$[FIELD_1] \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} a \quad \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} \ell \quad (G(\text{self}, a) = \ell)$$

$$[FIELD_2] \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} a.b \quad \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} \ell \quad (G(\text{self}, a) = v \quad G(v, b) = \ell)$$

**Semantica del nucleo basato su concatenazione**

$$A = o \mapsto \langle T_r, T_c, \ell \rangle$$

$$\mathcal{M} = \ell.a \mapsto v \quad \text{where } \ell.a = \ell + \partial\ell \quad \text{and } k(a) = \partial\ell$$

$$\mathcal{E}(T) = \langle T', k, f \rangle \quad k(a) = \begin{cases} \langle \text{isShared}, \partial\ell \rangle & \text{if } a \in d \text{ om}(k) \\ \text{undef} & \text{else} \end{cases} \quad f(m) = \begin{cases} S & \text{if } m \in d \text{ om}(f) \\ \text{undef} & \text{else} \end{cases}$$

$$\Phi(T) = \{o_1, \dots, o_n\}$$

$$C(T, \ell) \triangleq \Phi(T_i) = \Phi(T_i) \cup \{\ell\} \quad \forall T_i \in \{T, T', T'', \dots\} \text{ where } \mathcal{E}(T) = \langle T', k, f \rangle, \mathcal{E}(T') = \langle T'', k, f \rangle, \dots$$

$$G(o, a) = \begin{cases} v & \text{if } \exists k(a) \text{ where } A(o) = \langle T_r, T_c, \ell \rangle, \quad \mathcal{E}(T_r) = \langle T', k, f \rangle, \\ \text{undef} & \text{else} \quad k(a) = \langle \text{isShared}, \partial\ell \rangle, \quad \mathcal{M}(\ell + \partial\ell) = v \end{cases}$$

$$G(o, m) = \begin{cases} S & \text{if } \exists f(m) \\ \text{undef} & \text{else} \end{cases} \quad \text{where } A(o) = \langle T_r, T_c, \ell \rangle, \quad \mathcal{E}(T_r) = \langle T', k, f \rangle, \quad f(m) = S$$

$$G(o, \text{embedded } m) = \begin{cases} S & \text{if } \exists f'(m) \\ \text{undef} & \text{else} \end{cases} \quad \text{where } A(o) = \langle T_r, T_c, \ell \rangle, \quad \mathcal{E}(T_c) = \langle T', k, f \rangle, \\ \mathcal{E}(T') = \langle T'', k', f' \rangle, \quad f'(m) = S$$

$$U(\mathcal{M}, o, a, v) = \begin{cases} \sum_{\forall o_i \in \Phi(T_r)} \mathcal{M}[\ell \mapsto v] \parallel G(o_i, a) = v & \text{if } \text{isShared} \quad A(o) = \langle T_r, T_c, \ell' \rangle \\ \mathcal{M}[\ell \mapsto v] \parallel G(o, a) = v & \text{else} \quad \text{where } \mathcal{E}(T_r) = \langle T', k, f \rangle \\ & k(a) = \langle \text{isShared}, \partial\ell \rangle \end{cases}$$

Regole per le istruzioni:

$$[\text{ASS}_1] \frac{A, \mathcal{M} \Vdash_{\varepsilon} e \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} v}{A, \mathcal{M} \Vdash_{\varepsilon} a = e \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} e} \quad (U(\mathcal{M}', \text{self}, a, v) = \mathcal{M}'')$$

$$[\text{ASS}_2] \frac{A, \mathcal{M} \Vdash_{\varepsilon} e \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} v'}{A, \mathcal{M} \Vdash_{\varepsilon} a.b = e \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} e} \quad (G(\text{self}, a) = v, \quad U(\mathcal{M}', v, b, v') = \mathcal{M}'')$$

$$[\text{CALL}_1] \frac{A, \mathcal{M} \Vdash_{\varepsilon} S \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} e}{A, \mathcal{M} \Vdash_{\varepsilon} m() \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} e} \quad (G(\text{self}, m) = S)$$

$$[\text{CALL}_2] \frac{A[\text{self} \mapsto \langle T_r, T_c, v \rangle], \mathcal{M} \Vdash_{\varepsilon} S \quad \mathcal{A}' \mathcal{M}' \Vdash_{\varepsilon} e}{A, \mathcal{M} \Vdash_{\varepsilon} \text{embedded } m() \quad \mathcal{A}'[\text{self} \mapsto \langle T_r, T_c, v \rangle], \mathcal{M}' \Vdash_{\varepsilon} e} \quad \left( \begin{array}{l} A(\text{self}) = \langle T_r, T_c, \ell \rangle, \\ \mathcal{E}(T_c) = \langle T_c', k, f \rangle, \\ G(\text{self}, \text{embedded } m) = S \end{array} \right)$$

$$[\text{CALL}_3] \frac{A[\text{self} \mapsto \langle T_r, T_c, \ell \rangle], \mathcal{M} \Vdash_{\varepsilon} S \quad \mathcal{A}' \mathcal{M}' \Vdash_{\varepsilon} e}{A, \mathcal{M} \Vdash_{\varepsilon} a.m() \quad \mathcal{A}'[\text{self} \mapsto \mathbf{a}], \mathcal{M}' \Vdash_{\varepsilon} e} \quad \left( \begin{array}{l} A(\text{self}) = \mathbf{a}, \quad G(\text{self}, a) = v, \\ A(v) = \langle T_r, T_c, \ell \rangle, \quad G(v, m) = S \end{array} \right)$$

$$[\text{SEQ}] \frac{A, \mathcal{M} \Vdash_{\varepsilon} S \quad \mathcal{A}', \mathcal{M}' \Vdash_{\varepsilon} e \quad \mathcal{A}'', \mathcal{M}'' \Vdash_{\varepsilon} S' \quad \mathcal{A}''', \mathcal{M}''' \Vdash_{\varepsilon} e}{A, \mathcal{M} \Vdash_{\varepsilon} S; S' \quad \mathcal{A}''', \mathcal{M}''' \Vdash_{\varepsilon} e}$$

Regole per le espressioni:

$$[\text{CLONE}] \quad A, \mathcal{M} \Vdash_{\varepsilon} \text{clone } T \quad \mathcal{A}[\ell' \mapsto \langle T_r, T_c, \ell' \rangle], \mathcal{M}[\ell' \mapsto \text{Clone}(T)] \Vdash_{\varepsilon} \ell' \quad \left( \begin{array}{l} A(T) = \langle T_r, T_c, \ell \rangle, \\ \ell' \text{ new location} \\ C(T, \ell') \end{array} \right)$$

$$[\text{SELF}] \quad A, \mathcal{M} \Vdash_{\varepsilon} \text{self} \quad \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} \ell \quad (A(\text{self}) = \langle T_r, T_c, \ell \rangle)$$

$$[\text{FIELD}_1] \quad A, \mathcal{M} \Vdash_{\varepsilon} a \quad \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} \ell \quad (G(\text{self}, a) = \ell)$$

$$[\text{FIELD}_2] \quad A, \mathcal{M} \Vdash_{\varepsilon} a.b \quad \mathcal{A}, \mathcal{M} \Vdash_{\varepsilon} \ell \quad (G(\text{self}, a) = v \quad G(v, b) = \ell)$$

## Tipi

Assumiamo che una classe definisca un tipo e che le sottoclassi siano dei sottotipi. Analogamente nel modello basato su delega un prototipo definisce un tipo ed è sottotipo dei tipi dei prototipi a cui delega. Infine, nel modello basato su concatenazione, un prototipo autosufficiente definisce un tipo ed è sottotipo dei tipi che incorpora.

I tipi  $\tau$  e i tipi base  $\rho$  sono induttivamente definiti su un insieme contabile di attributi tipati:

$$t ::= [r] \quad r ::= \begin{cases} a : t \\ m : void \rightarrow void \\ r; r \end{cases}$$

Assumiamo che i nomi che fanno parte di un tipo  $\tau$  siano due a due differenti. Inoltre, due tipi che differiscono unicamente per l'ordine dei tipi base sono considerati uguali.

Si assume che sia definita sui tipi una *relazione di sottotipo* minima e chiusa rispetto le seguenti regole:

$$t <: t \quad [r; r'] <: [r] \quad \frac{t <: t' \quad t' <: t''}{t <: t''}$$

Un ambiente  $\Gamma$  associa identificatori a tipi. Un ambiente è una sequenza di coppie  $x : \tau$ ; l'operatore di sequenza usato è "+", che è associativo e parzialmente commutativo. Precisamente:

$$x : T + x' : T' = x' : T' + x : T \quad \text{if } x \neq x'$$

Definisco  $\Gamma$  come segue:

$$\Gamma(x) = \begin{cases} t & \text{if } \Gamma = \Gamma' + x : t \\ undef & \text{else} \end{cases}$$

I sistemi dei tipi definiti di seguito per i tre linguaggi usano due forme di asserzione:  $\Gamma \mid M$  e  $\Gamma \mid E : \tau$ . La prima afferma che un segmento del programma è ben tipato; la seconda esplicita il tipo delle espressioni. Per ciascun linguaggio, la relazione  $\mid$  è la minima chiusa rispetto al corrispondente sistema di inferenza dei tipi. I tre sistemi sono identici a meno delle parole chiave usate.

**Sistema di inferenza dei tipi per  $\langle e \rangle$**

Regole per le definizioni:

$$\frac{\Gamma + T : \mathbf{t} + \text{this} : \mathbf{t} \vdash B}{\Gamma + T : \mathbf{t} \vdash \text{class } T \{B\}} \quad \frac{\Gamma + T : [\mathbf{r}; \mathbf{r}'] + T' : [\mathbf{r}'] + \text{this} : [\mathbf{r}; \mathbf{r}'] + \text{super} : [\mathbf{r}'] \vdash B}{\Gamma + T : [\mathbf{r}; \mathbf{r}'] + T' : [\mathbf{r}'] \vdash \text{class } T \text{ extends } T' \{B\}}$$

$$\frac{\Gamma \vdash D \quad \Gamma \vdash D'}{\Gamma \vdash D; D'}$$

Regole per i corpi:

$$\frac{\Gamma + \text{this} : [a : \mathbf{t}; \mathbf{r}] \vdash a : \mathbf{t} \quad \Gamma \vdash T : \mathbf{t}}{\Gamma \vdash \text{static } T a} \quad \frac{\Gamma + \text{this} : [a : \mathbf{t}; \mathbf{r}] \vdash a : \mathbf{t} \quad \Gamma \vdash T : \mathbf{t}}{\Gamma \vdash T a}$$

$$\frac{\Gamma + \text{this} : [m : \text{void} \rightarrow \text{void}; \mathbf{r}] \vdash m : \text{void} \rightarrow \text{void} \quad \Gamma \vdash S}{\Gamma \vdash m() \{S\}} \quad \frac{\Gamma \vdash B \quad \Gamma \vdash B'}{\Gamma \vdash B; B'}$$

Regole per le istruzioni:

$$\Gamma \vdash \mathbf{e} \quad \frac{\Gamma \vdash Q : \mathbf{t} \quad \Gamma \vdash E : \mathbf{t}}{\Gamma \vdash Q = E} \quad \frac{\Gamma \vdash E : \mathbf{t} \quad \mathbf{t} <: \mathbf{t}'}{\Gamma \vdash E : \mathbf{t}'}$$

$$\frac{\Gamma \vdash Q : \text{void} \rightarrow \text{void}}{\Gamma \vdash Q ()} \quad \frac{\Gamma \vdash S \quad \Gamma \vdash S'}{\Gamma \vdash S; S'}$$

Regole per le espressioni:

$$\frac{\Gamma \vdash T : \mathbf{t} \quad \text{concrete } \mathbf{t}}{\Gamma \vdash \text{new } T : \mathbf{t}} \quad \frac{\Gamma \vdash Q : \mathbf{t}}{\Gamma \vdash E : \mathbf{t}}$$

$$\text{concrete } \mathbf{t} \triangleq \text{expthis} \subseteq \text{this} \wedge \text{expsuper} \subseteq \text{super} \wedge \forall [a : \mathbf{t}'] \in \text{this} \text{ exp} \mathbf{t}' \subseteq \mathbf{t}'$$

Regole per gli identificatori qualificati:

$$\Gamma + \text{this} : \mathbf{t} \vdash \text{this} : \mathbf{t}$$

$$\Gamma + \text{expthis} : [a : \mathbf{t}; \mathbf{r}] \vdash a : \mathbf{t}$$

$$\Gamma + \text{expthis} : [m : \text{void} \rightarrow \text{void}; \mathbf{r}] \vdash m : \text{void} \rightarrow \text{void}$$

$$\Gamma + \text{expsuper} : [m : \text{void} \rightarrow \text{void}; \mathbf{r}] \vdash \text{super}.m : \text{void} \rightarrow \text{void}$$

$$\Gamma + \text{expthis} : [a : \mathbf{t}; \mathbf{r}] + \text{exp} \mathbf{t} : [b : \mathbf{t}'; \mathbf{r}] \vdash a.b : \mathbf{t}'$$

$$\Gamma + \text{expthis} : [a : \mathbf{t}; \mathbf{r}] + \text{exp} \mathbf{t} : [m : \text{void} \rightarrow \text{void}; \mathbf{r}] \vdash a.m : \text{void} \rightarrow \text{void}$$

**Sistema di inferenza dei tipi per  $\langle_d$**

Regole per le definizioni:

$$\frac{\Gamma + T : \mathbf{t} + \mathit{self} : \mathbf{t} \vdash B}{\Gamma + T : \mathbf{t} \vdash \mathit{object} T \{B\}} \quad \frac{\Gamma + T : [\mathbf{r}; \mathbf{r}'] + T' : [\mathbf{r}'] + \mathit{self} : [\mathbf{r}; \mathbf{r}'] + \mathit{delegated} : [\mathbf{r}'] \vdash B}{\Gamma + T : [\mathbf{r}; \mathbf{r}'] + T' : [\mathbf{r}'] \vdash \mathit{object} T \mathit{delegates} T' \{B\}}$$

$$\frac{\Gamma \vdash D \quad \Gamma \vdash D'}{\Gamma \vdash D; D'}$$

Regole per i corpi:

$$\frac{\Gamma + \mathit{self} : [a : \mathbf{t}; \mathbf{r}] \vdash a : \mathbf{t} \quad \Gamma \vdash T : \mathbf{t}}{\Gamma \vdash T a} \quad \frac{\Gamma + \mathit{self} : [m : \mathit{void} \rightarrow \mathit{void}; \mathbf{r}] \vdash m : \mathit{void} \rightarrow \mathit{void} \quad \Gamma \vdash S}{\Gamma \vdash m() \{S\}}$$

$$\frac{\Gamma \vdash B \quad \Gamma \vdash B'}{\Gamma \vdash B; B'}$$

Regole per le istruzioni:

$$\Gamma \vdash \mathbf{e} \quad \frac{\Gamma \vdash Q : \mathbf{t} \quad \Gamma \vdash E : \mathbf{t}}{\Gamma \vdash Q = E} \quad \frac{\Gamma \vdash E : \mathbf{t} \quad \mathbf{t} <: \mathbf{t}'}{\Gamma \vdash E : \mathbf{t}'}$$

$$\frac{\Gamma \vdash Q : \mathit{void} \rightarrow \mathit{void}}{\Gamma \vdash Q() } \quad \frac{\Gamma \vdash S \quad \Gamma \vdash S'}{\Gamma \vdash S; S'}$$

Regole per le espressioni:

$$\frac{\Gamma \vdash T : \mathbf{t} \quad \mathit{concrete} \mathbf{t}}{\Gamma \vdash \mathit{clone} T : \mathbf{t}} \quad \frac{\Gamma \vdash Q : \mathbf{t}}{\Gamma \vdash E : \mathbf{t}}$$

$$\mathit{concrete} \mathbf{t} \triangleq \mathit{exp} \mathit{self} \subseteq \mathit{self} \wedge \mathit{exp} \mathit{delegated} \subseteq \mathit{delegated} \wedge \forall [a : \mathbf{t}'] \in \mathit{self} \quad \mathit{exp} \mathbf{t}' \subseteq \mathbf{t}'$$

Regole per gli identificatori qualificati:

$$\Gamma + \mathit{self} : \mathbf{t} \vdash \mathit{self} : \mathbf{t}$$

$$\Gamma + \mathit{exp} \mathit{self} : [a : \mathbf{t}; \mathbf{r}] \vdash a : \mathbf{t}$$

$$\Gamma + \mathit{exp} \mathit{self} : [m : \mathit{void} \rightarrow \mathit{void}; \mathbf{r}] \vdash m : \mathit{void} \rightarrow \mathit{void}$$

$$\Gamma + \mathit{exp} \mathit{delegated} : [m : \mathit{void} \rightarrow \mathit{void}; \mathbf{r}] \vdash \mathit{delegated} m : \mathit{void} \rightarrow \mathit{void}$$

$$\Gamma + \mathit{exp} \mathit{self} : [a : \mathbf{t}; \mathbf{r}] + \mathit{exp} \mathbf{t} : [b : \mathbf{t}'; \mathbf{r}] \vdash a.b : \mathbf{t}'$$

$$\Gamma + \mathit{exp} \mathit{self} : [a : \mathbf{t}; \mathbf{r}] + \mathit{exp} \mathbf{t} : [m : \mathit{void} \rightarrow \mathit{void}; \mathbf{r}] \vdash a.m : \mathit{void} \rightarrow \mathit{void}$$

**Sistema di inferenza dei tipi per  $<_c$**

Regole per le definizioni:

$$\frac{\Gamma + T : \mathbf{t} + \mathit{self} : \mathbf{t} \vdash B}{\Gamma + T : \mathbf{t} \vdash \mathit{object} T \{B\}} \quad \frac{\Gamma + T : [\mathbf{r}; \mathbf{r}'] + T' : [\mathbf{r}'] + \mathit{self} : [\mathbf{r}; \mathbf{r}'] + \mathit{embedded} : [\mathbf{r}'] \vdash B}{\Gamma + T : [\mathbf{r}; \mathbf{r}'] + T' : [\mathbf{r}'] \vdash \mathit{object} T \mathit{embeds} T' \{B\}}$$

$$\frac{\Gamma \vdash D \quad \Gamma \vdash D'}{\Gamma \vdash D; D'}$$

Regole per i corpi:

$$\frac{\Gamma + \mathit{self} : [a : \mathbf{t}; \mathbf{r}] \vdash a : \mathbf{t} \quad \Gamma \vdash T : \mathbf{t}}{\Gamma \vdash \mathit{shared} T a} \quad \frac{\Gamma + \mathit{self} : [a : \mathbf{t}; \mathbf{r}] \vdash a : \mathbf{t} \quad \Gamma \vdash T : \mathbf{t}}{\Gamma \vdash T a}$$

$$\frac{\Gamma + \mathit{self} : [m : \mathit{void} \rightarrow \mathit{void}; \mathbf{r}] \vdash m : \mathit{void} \rightarrow \mathit{void} \quad \Gamma \vdash S}{\Gamma \vdash m() \{S\}} \quad \frac{\Gamma \vdash B \quad \Gamma \vdash B'}{\Gamma \vdash B; B'}$$

Regole per le istruzioni:

$$\Gamma \vdash \mathbf{e} \quad \frac{\Gamma \vdash Q : \mathbf{t} \quad \Gamma \vdash E : \mathbf{t}}{\Gamma \vdash Q = E} \quad \frac{\Gamma \vdash E : \mathbf{t} \quad \mathbf{t} <: \mathbf{t}'}{\Gamma \vdash E : \mathbf{t}'}$$

$$\frac{\Gamma \vdash Q : \mathit{void} \rightarrow \mathit{void}}{\Gamma \vdash Q()} \quad \frac{\Gamma \vdash S \quad \Gamma \vdash S'}{\Gamma \vdash S; S'}$$

Regole per le espressioni:

$$\frac{\Gamma \vdash T : \mathbf{t} \quad \mathit{concrete} \mathbf{t}}{\Gamma \vdash \mathit{clone} T : \mathbf{t}} \quad \frac{\Gamma \vdash Q : \mathbf{t}}{\Gamma \vdash E : \mathbf{t}}$$

$$\mathit{concrete} \mathbf{t} \triangleq \mathit{exp} \mathit{self} \subseteq \mathit{self} \wedge \mathit{exp} \mathit{embedded} \subseteq \mathit{embedded} \wedge \forall [a : \mathbf{t}'] \in \mathit{self} \mathit{exp} \mathbf{t}' \subseteq \mathbf{t}'$$

Regole per gli identificatori qualificati:

$$\Gamma + \mathit{self} : \mathbf{t} \vdash \mathit{self} : \mathbf{t}$$

$$\Gamma + \mathit{exp} \mathit{self} : [a : \mathbf{t}; \mathbf{r}] \vdash a : \mathbf{t}$$

$$\Gamma + \mathit{exp} \mathit{self} : [m : \mathit{void} \rightarrow \mathit{void}; \mathbf{r}] \vdash m : \mathit{void} \rightarrow \mathit{void}$$

$$\Gamma + \mathit{exp} \mathit{embedded} : [m : \mathit{void} \rightarrow \mathit{void}; \mathbf{r}] \vdash \mathit{embedded} m : \mathit{void} \rightarrow \mathit{void}$$

$$\Gamma + \mathit{exp} \mathit{self} : [a : \mathbf{t}; \mathbf{r}] + \mathit{exp} \mathbf{t} : [b : \mathbf{t}'; \mathbf{r}] \vdash a.b : \mathbf{t}'$$

$$\Gamma + \mathit{exp} \mathit{self} : [a : \mathbf{t}; \mathbf{r}] + \mathit{exp} \mathbf{t} : [m : \mathit{void} \rightarrow \mathit{void}; \mathbf{r}] \vdash a.m : \mathit{void} \rightarrow \mathit{void}$$

## Equivalenza semantica

Per dimostrare l'equivalenza dei tre linguaggi e quindi dei tre meccanismi di condivisione definisco tre trasformazioni da un linguaggio ad un altro in modo da realizzare una catena chiusa; poi mostro che per un qualsiasi programma si possono eseguire le stesse regole di transizione definite nella semantica. Una trasformazione mappa un albero di entità di un linguaggio in un albero di entità equivalente di un altro linguaggio. Poiché le regole semantiche sono esattamente le stesse per tutti i linguaggi a meno di alcune funzioni ausiliarie, l'equivalenza semantica viene dimostrata facendo vedere che queste funzioni ausiliarie danno gli stessi risultati quando vengono applicate agli alberi relativi alla trasformazione.

Per ogni coppia di linguaggi vi sono infinite (banali) trasformazioni che preservano la semantica dei programmi; per individuare una trasformazione si può uguagliare a due a due le funzioni ausiliarie e ricavare da queste equazioni i vincoli che devono essere soddisfatti dalla trasformazione. La trasformazione esiste se i vincoli sono compatibili. Le trasformazioni definite di seguito vengono ricavate in questo modo pertanto non sono completamente specificate.

Prima di procedere alla definizione delle trasformazioni riepiloghiamo in una tabella le differenze semantiche fra i tre linguaggi.

	<b>Ereditarietà</b>	<b>Delega</b>	<b>Concatenazione</b>
<b>G(o,a)</b>	Si propaga solo per accedere ai campi condivisi	Si propaga solo per accedere ai campi condivisi	Sempre locale
<b>G(o,m)</b>	Si propaga per accedere ai metodi ereditati	Si propaga per accedere ai metodi non locali (sempre)	Sempre locale (self legato)
<b>U(M,o,a,v)</b>	Singolo aggiornamento; locale per i campi di istanza e remoto per quelli condivisi	Singolo aggiornamento; locale per i campi di istanza e remoto per quelli condivisi	Per i campi di istanza locale e singolo; per i campi condivisi locale più clone family e replicato.
<b>Definizione campi</b>	Sparsi lungo tutta la catena di ereditarietà. I campi condivisi sono marcati	Sparsi lungo la catena di delega: i campi condivisi si trovano solo nei tratti comuni mentre i campi d'istanza si trovano solo nei prototipi	Locali ad ogni prototipo che li contiene. I campi condivisi sono marcati
<b>Definizione metodi</b>	Sparsi lungo tutta la catena di ereditarietà. Possono accedere solo a campi già definiti	Sparsi lungo la catena di delega limitatamente ai tratti. Possono accedere anche ai campi definiti nell'albero di delega sottostante	Locali. Accedono sempre e solo ad attributi locali
<b>New/clone</b>	Crea un oggetto contenente tutti i campi di istanza definiti lungo la catena di ereditarietà.	Copia il prototipo; i delegati non vengono copiati	Copia il prototipo; gli oggetti incorporati vengono implicitamente copiati
<b>oggetto</b>	Oggetto limitato ai campi di istanza; i metodi e i campi condivisi sono sparsi nelle classi della catena di ereditarietà	Oggetto limitato ai campi di istanza; i metodi e i campi condivisi sono sparsi negli oggetti della catena di delega	Oggetto completamente autosufficiente; la clone family permette di risalire a tutte le repliche dei campi condivisi

### **Trasformazione da $\langle_e a \rangle_d$**

Per ogni classe E definisco due oggetti  $D_t$  e  $D_p$  con ruoli rispettivamente di tratto e prototipo. In  $D_t$  definisco gli eventuali campi statici di E e tutti i metodi; in  $D_p$  definisco tutti i campi di istanza definiti in E e nelle classi che compongono la catena di ereditarietà. Per ogni classe E e superclasse E':  $D_p$  delega a  $D_t$  e  $D_t$  delega al  $D_t$  definito per la superclasse.  $D_p$  viene clonato dove E viene istanziato. Se una classe E non viene istanziata non è necessario definire  $D_p$ .  $D_p$  è clonabile quando E è istanziabile.

$$\text{Vincolo } G_e(e,a) = G_d(d,a)$$

Il dominio di entrambe le funzioni è dato dall'insieme dei campi definiti rispettivamente nella catena di ereditarietà di e e nella catena di delega di d. Poiché la trasformazione mappa tutti i campi definiti nella catena di ereditarietà sugli oggetti della catena di delega limitandosi a fare degli spostamenti i due domini sono uguali.

$$\text{Vincolo } G_e(e,m) = G_d(d,m)$$

Il dominio di entrambe le funzioni è dato dall'insieme dei metodi definiti rispettivamente nella catena di ereditarietà di e e nella catena di delega di d. Poiché la trasformazione mappa tutti i metodi definiti nella catena di ereditarietà sugli oggetti della catena di delega limitandosi a fare degli spostamenti i due domini sono uguali.

$$\text{Vincolo } U_e(M,e,a,v) = U_d(M,d,a,v)$$

Il dominio di entrambe le funzioni è dato dall'insieme dei campi definiti rispettivamente nella catena di ereditarietà di e e nella catena di delega di d. Poiché la trasformazione mappa tutti i campi definiti nella catena di ereditarietà sugli oggetti della catena di delega limitandosi a fare degli spostamenti i due domini sono uguali. In entrambi i casi l'aggiornamento di un campo condiviso avviene in una entità della catena di condivisione comune a tutti gli oggetti che condividono il campo.

### **Trasformazione da $\langle_d a \rangle_c$**

Per ogni prototipo D definisco un oggetto autosufficiente C come segue. Definisco in C tutti gli attributi (campi e metodi) definiti negli oggetti che compongono la catena di delega. I campi definiti nei tratti vengono marcati come campi condivisi in C. Per i metodi ridefiniti includo solo l'ultima definizione. La gerarchia di contenimento riproduce la gerarchia di delega. D e C sono entrambe clonabili oppure non lo sono nessuno dei due.

$$\text{Vincolo } G_d(d,a) = G_c(c,a)$$

Il dominio di  $G_d$  su cui la funzione è definita è l'insieme dei campi definiti lungo la catena di delega di d; i campi di istanza sono tutti in d, i campi condivisi sono sparsi lungo la catena. La trasformazione mappa la definizione di tutti i campi della catena di delega di d sull'oggetto autosufficiente c limitandosi a marcare i campi condivisi. La funzione  $G_c$  cerca i campi localmente pertanto ha lo stesso dominio di  $G_d$ .

$$\text{Vincolo } G_d(d,m) = G_c(c,m)$$

Il dominio di  $G_d$  su cui la funzione è definita è l'insieme dei metodi definiti lungo la catena di delega di d. La trasformazione mappa i metodi definiti lungo la catena di delega di d sull'oggetto autosufficiente c. La funzione  $G_c$  cerca i metodi localmente pertanto ha lo stesso dominio di  $G_d$ .

$$\text{Vincolo } U_d(M,d,a,v) = U_c(M,c,a,v)$$

Il dominio di  $U_d$  su cui la funzione è definita è l'insieme dei campi definiti lungo la catena di delega di d; i campi di istanza sono tutti in d, i campi condivisi sono sparsi lungo la catena. La trasformazione mappa la definizione di tutti i campi della catena di delega di d sull'oggetto autosufficiente c limitandosi a marcare i campi condivisi. La funzione  $U_c$  cerca i campi localmente pertanto ha lo stesso dominio di  $U_d$ . L'aggiornamento di un campo definito lungo la catena di d è osservabile da tutti gli oggetti che condividono quel tratto della catena; analogamente, l'aggiornamento di un campo condiviso in c viene replicato in tutti gli oggetti che lo condividono.

### **Trasformazione da $\langle_c a \rangle_e$**

Per ogni clone family di oggetti F(C) definisco una classe E con i soli attributi non incorporati più gli attributi da incorporare. L'attributo a di un oggetto C è non incorporato se è definito e usato in C e in nessun altro oggetto C' della catena di contenimento. Gli attributi da incorporare sono quegli attributi usati ma definiti nel sotto albero di contenimento; se la definizione di un metodo da incorporare è la stessa in tutti i rami del sotto albero di contenimento viene incorporata altrimenti viene incorporato un metodo vuoto; la definizione di un campo da incorporare è la stessa in tutti i rami del sotto albero di contenimento. La gerarchia di contenimento diventa una gerarchia di ereditarietà. La clone di un oggetto C viene sostituita dalla new della corrispondente classe E; se C è clonabile E è istanziabile.

Vincolo  $G_c(c,a) = G_e(e,a)$

Il dominio di  $G_c$  su cui la funzione è definita è l'insieme dei campi definiti in  $c$ . La trasformazione mappa la definizione dei campi di  $c$  sulla gerarchia di ereditarietà di  $e$ ; ogni campo viene definito in una ed una sola classe. La funzione *new* concentra i campi di istanza in  $e$ . La funzione  $G_c$  cerca i campi di istanza in  $e$  ed estende la ricerca dei campi condivisi alla gerarchia di ereditarietà pertanto ha lo stesso dominio di  $G_c$ . Poiché la gerarchia di ereditarietà è comune a tutte le istanze di una stessa classe i campi definiti lungo la gerarchia sono effettivamente condivisi.

Vincolo  $G_c(c,m) = G_e(e,m)$

Il dominio di  $G_c$  su cui la funzione è definita è l'insieme dei metodi definiti in  $c$ . La trasformazione mappa la definizione dei metodi di  $c$  sulla gerarchia di ereditarietà di  $e$ ; ogni metodo viene definito in una ed una sola classe. La funzione  $G_c$  estende la ricerca dei metodi alla gerarchia di ereditarietà pertanto ha lo stesso dominio di  $G_c$ .

Vincolo  $U_c(M,c,a,v) = U_e(M,e,a,v)$

Il dominio di  $U_c$  su cui la funzione è definita è l'insieme dei campi definiti in  $c$ . La trasformazione mappa la definizione dei campi di  $c$  sulla gerarchia di ereditarietà di  $e$ ; ogni campo viene definito in una ed una sola classe. La funzione *new* concentra i campi di istanza in  $e$ . La funzione  $U_c$  cerca i campi di istanza in  $e$  ed estende la ricerca dei campi condivisi alla gerarchia di ereditarietà pertanto ha lo stesso dominio di  $U_c$ . L'aggiornamento di un campo condiviso in  $c$  viene replicato in tutti gli oggetti che lo condividono (clone family); analogamente, poiché la gerarchia di ereditarietà è comune a tutte le sotto classi di una stessa classe, l'aggiornamento di un campo definito in una classe  $E$  lungo la gerarchia di  $e$  è osservabile da tutte le istanze delle sotto classi di  $E$ ; e poiché la clone family viene mappata in  $E$  il campo è osservabile esattamente da tutte le istanze che lo condividono.

## Conseguenze

L'equivalenza dimostrata nella sezione precedente in termini di trasformabilità da un linguaggio all'altro rende intercambiabili in particolare i modelli di implementazione e di rappresentazione.

### **Modello di implementazione**

Il modello di condivisione non impone delle restrizioni al modello di implementazione. Ereditarietà, delega e concatenazione sono tre distinte strategie per implementare un meccanismo di condivisione e possono essere la base per i modelli a classi o a prototipi. Il compilatore può tipicamente scegliere la rappresentazione più appropriata. Ad esempio nei linguaggi basati su ereditarietà o su delega è opportuno definire a livello di prototipo una tabella dei metodi disponibili in modo da non dovere propagare le richieste alla gerarchia di condivisione. Analogamente in presenza di concatenazione è necessario ricorrere a copia virtuale pena l'impossibilità di avere molti oggetti.

### **Modello di rappresentazione**

Il modello di condivisione non impone vincoli al modello di rappresentazione. Ogni modello ha una rappresentazione che si mappa direttamente sul modello concettuale. Il modello basato su concatenazione richiede un ambiente di programmazione per mostrare e aggiornare gli attributi condivisi perché vengono replicati. Assumiamo come generale la disponibilità di un ambiente di programmazione capace di elaborare il modello concettuale prima di mostrarlo. Definisco le tre viste che possono essere adottate da ciascun modello mediante le trasformazioni definite nella sezione precedente.

#### Viste

- Vista di ereditarietà. E' una vista incrementale; mostra gli attributi delle entità introdotti dalle entità stesse (possono essere di istanza o condivisi).
- Vista di delega. Mostra gli attributi di istanza di una entità e gli attributi condivisi in base alla gerarchia di condivisione.
- Vista di concatenazione. Mostra tutti gli attributi di una entità sia quelli specifici sia quelli condivisi con altri, distinguendo visivamente gli attributi condivisi.

#### Osservazioni

L'esistenza di una gerarchia di ereditarietà o di delega pone il problema di sapere quali attributi possieda un oggetto ovvero a quali messaggi sia in grado di rispondere, in modo ad esempio di vedere se gli attributi interagiscono correttamente o è necessario apportare modifiche. Una vista basata su concatenazione mostra tutti gli attributi ed è pertanto la più adatta per questo scopo.

Ereditarietà e delega sono concettualmente asimmetrici. La modifica di una superclasse o di un tratto inevitabilmente modifica i discendenti. A livello di rappresentazione è possibile nascondere questa asimmetria.

### **Tipabilità**

Il modello di condivisione non limita la tipabilità. Abbiamo già visto nella sezione sui tipi che tutti e tre i linguaggi sono tipabili. E' facile osservare guardando la definizione delle trasformazioni che un programma correttamente tipato in un linguaggio viene trasformato in un programma correttamente tipato in un altro.

## Conclusioni

Ora possiamo chiederci quali siano le differenze intrinseche fra un linguaggio basato su classi e uno basato su oggetti. L'unica differenza è una scelta diversa riguardo a quando determinare le caratteristiche dei prototipi: all'ultimo momento oppure in modo anticipato.

In un linguaggio basato su classi la determinazione degli attributi di un oggetto allocato dinamicamente viene ritardata fino al momento della istanziazione mentre nei linguaggi basati su oggetti questo lavoro viene anticipato alla definizione. I primi, almeno concettualmente, richiedono un operatore di allocazione – *new* – che ha bisogno di analizzare la gerarchia di ereditarietà come se fosse una ricetta; mentre per i secondi è sufficiente un operatore – *clone* – che duplica un prototipo precedentemente definito. Nei linguaggi basati su delega il lavoro di definizione del prototipo viene effettuato dal programmatore che deve progettare la gerarchia di delega in modo da separare i prototipi dai tratti comuni. Nei linguaggi basati su concatenazione, è l'ambiente di programmazione che si occupa di eseguire automaticamente il lavoro.

I linguaggi basati su delega mostrano che la distinzione fra sottoclasse e istanziare non è necessaria. Un solo tipo di relazione: delega viene contrapposto ad istanziazione (*is a*) e sottoclassamento (*kind of*).

I linguaggi basati su oggetti autosufficienti mostrano che non è necessario definire manualmente le entità non concrete (classi astratte e traits) né preoccuparsi di riorganizzare la gerarchia di condivisione. Questi linguaggi consentono di creare prima concetti individuali e poi di generalizzarli. I sistemi basati su classi o su oggetti con delega richiedono di fornire prima una definizione dell'insieme astratto. Il problema è che è difficile stabilire in anticipo quali siano le caratteristiche essenziali di un concetto.

La differenza fra avere gerarchie di classi o gerarchie di oggetti deleganti da una parte e oggetti autosufficienti dall'altra è ben più rilevante che non la divisione fra basati su classi e basati su oggetti. Nel primo caso il programmatore si deve preoccupare di definire esplicitamente le classi comuni da estendere e gli oggetti condivisi (traits), nel secondo caso il sistema provvede automaticamente a mantenerla e a modificarla.

L'equivalenza dei tre linguaggi ci dice d'altra parte che l'implementazione, la rappresentazione e la tipabilità sono fondamentalmente indipendenti dai tre modelli. Questo ci permette di aggiungere ad un linguaggio esistente le caratteristiche positive degli altri modelli riducendo la scelta del modello ad una questione di gusto e di tradizione. Ad esempio abbiamo osservato che la vista di concatenazione con le relative operazioni risulta desiderabile anche per gli attuali linguaggi a classi.

In prospettiva i linguaggi basati su concatenazione attaccano la centralità attribuita alla gerarchia di condivisione come elemento organizzante; e i linguaggi basati su delega (dinamica) attaccano la rigidità del meccanismo di dispatching. Entrambi gli attacchi riguardano caratteristiche fondamentali della programmazione orientata agli oggetti pertanto anche in prospettiva più che orientare la scelta verso un modello anziché un altro possono fare emergere ragioni per superare il paradigma nel suo complesso.

## Bibliografia

- [Lie86] Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, 1986.
- [SLU88] Lynn Andrea Stein, Henry Lieberman, David Ungar, *A Shared View of Sharing: The Treaty of Orlando*. In *Object-oriented concepts, applications, and databases*, Kim W., Lochowsky F., eds, Addison-Wesley, 1988, 31-48.
- [Tai92] Antero Taivalsaari, *Kevo – a prototype-based object-oriented language based on concatenation and module operations*, TR DCS-197-1R, University of Victoria, 1992.
- [UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, Urs Holzle, *Organizing Programs Without Classes*, Lisp and Symbolic Computation, 4(3), 1991, 223-242.