# Type Assigment Systems for Lambda Calculi and for the Lambda Calculus of Objects

**Luigi Liquori**
Computer Science Department
Universiy of Turin
Ph.D. Thesis in Computer Science

December 20, 1996

# Contents

# Chapter 1

# Prefazione (Italiano)

Questa tesi si inserisce nell'ambito dello studio della Teoria dei Tipi per il $\lambda$-calcolo. È suddivisa in una parte "fondazionale" ed in una "applicativa":

*i)* La parte fondazionale studia vari sistemi di assegnazione di tipi per il $\lambda$-calcolo puro. In particolare sono state provate le proprietà fondamentali di tali sistemi e sono stati considerati i rapporti con i corrispondenti sistemi di tipi per il $\lambda$-calcolo tipato e con la logica intuizionista.

*ii)* La parte applicativa studia una possibile estensione del $\lambda$-calcolo ad un linguaggio che può essere visto come "paradigma" per studiare tecniche nuove di programmazione come l'*Object Oriented*. Un sistema di assegnazione di tipi per tale linguaggio è stato definito e le proprietà fondamentali di tale sistema sono state provate.

## 1.1 Sistemi di Assegnazione di Tipi per il $\lambda$-calcolo

Il $\lambda$-calcolo [Chu41] è un potente formalismo per esprimere funzioni. La sua regola di computazione $(\beta)$ offre un semplice meccanismo di calcolo.

Un *Sistema di Assegnazione di Tipi* (tas) per il $\lambda$-calcolo è un sistema formale, espresso in deduzione naturale (à la Prawitz [Pra65]), che assegna ad un $\lambda$-termine $M$ un tipo $\phi$. In un tas, i "tipi" assegnati ad $M$ sono visti come *proprietà* possedute dal termine $M$. Da un punto di vista *set-theoretic*, un tipo $\phi$ è interpretato come un sottoinsieme del

*Dominio Semantico di Interpretazione* $\mathcal{S}$, i cui elementi rappresentano l'interpretazione dei $\lambda$-termini a cui posso assegnare il tipo $\phi$. Per esempio, se si può derivare un tipo $\phi$ per $M$, denotato con $\vdash M : \phi$, allora l'interpretazione di $\phi$, denotata come $[\![\,\phi\,]\!]$, sarà un sottoinsieme di $\mathcal{S}$ e l'interpretazione di $M$, denotata come $[\![\,M\,]\!]$, sarà un elemento di $[\![\,\phi\,]\!]$.

Negli ultimi decenni, la Teoria dei Tipi è diventata importante nello studio dei linguaggi di programmazione. Consideriamo il $\lambda$-calcolo come paradigma di un linguaggio di programmazione: se posso assegnare ad un $\lambda$-termine $M$ un tipo $\phi$, allora $\phi$ rappresenta una proprietà *invariante* per computazione, ovvero se $M$ si riduce ad $N$, espresso con $M \rightarrow_\beta N$, allora anche $N$ avrà la proprietà $\phi$. Da un punto di vista set-theoretic, l'interpretazione $[\![\,N\,]\!]$ coincide con quella di $[\![\,M\,]\!]$. Quindi $[\![\,M\,]\!]$ ed $[\![\,N\,]\!]$ denotano lo stesso elemento del dominio semantico $\mathcal{S}$ di interpretazione dei $\lambda$-termini.

La proprietà sintattica che esprime l'invarianza computazionale dei tipi è chiamata "proprietà di riduzione del soggetto" (SR), cioè :

$$\vdash M : \phi \;\&\; M \rightarrow_\beta N \;\Rightarrow\; \vdash N : \phi.$$

Un'altra interessante proprietà computazionale, che un sistema di assegnazione di tipi può avere, è la cosiddetta "normalizzazione forte" (SN) dei $\lambda$-termini tipabili: tale proprietà caratterizza tutte le computazioni di un termine a cui abbiamo assegnato un tipo. In un tas che possiede SN, ogni termine tipabile $M$ raggiunge in un numero finito di passi la sua forma normale $N$, ovvero nessuna $\beta$-regola è applicabile su $N$. Questa proprietà garantisce la terminazione dei programmi che sono tipabili.

Quando guardiamo al $\lambda$-calcolo come un linguaggio di programmazione paradigmatico, possiamo immaginare i $\lambda$-termini come programmi scritti in stile ML [MTH90], dove l'utente può scrivere un programma $P$, ed un tipo (se esiste) per $P$ viene "inferito" nella fase di compilazione. L'assegnazione di un tipo per $P$ può essere visto come una "interpretazione astratta" del programma che può essere usata come criterio di correttezza.

I tas sono stati introdotti da Curry [Cur34] per la *Logica Combinatoria* ed in seguito modificati da Curry, Feys, Hindley e Seldin [CF58, CHS72] per il $\lambda$-calcolo. Nella sua forma più generale, un tas prova "giudizi" della forma

$$\Gamma \vdash M : \phi,$$

dove $M$ è un termine del $\lambda$-calcolo puro, $\phi$ è un tipo e $\Gamma$ è un "contesto" che assegna tipi alle variabili libere di $M$. Tale giudizio viene interpretato come l'assegnazione del tipo $\phi$ al termine $M$, quando assegnamo alle variabili libere di $M$ i tipi specificati in $\Gamma$.

In letteratura esistono diversi tas: l'insieme dei $\lambda$-termini a cui possiamo assegnare un tipo e l'insieme dei "costruttori di tipi" (e quindi di tipi) varia da un sistema all'altro. Ad esempio, il *Curry Type Assignment System* ($F1$) [Cur34], ha come unico costruttore di tipo la "freccia" o costruttore di funzioni. In questo sistema possiamo assegnare alla funzione identità $\lambda x.x$ tutti i tipi ottenibili per sostituzione dal tipo più generale $\alpha{\rightarrow}\alpha$, dove $\alpha$ è una variabile di tipo. Diremo che $\alpha{\rightarrow}\alpha$ è lo "schema di tipo principale" per $\lambda x.x$ ed indicheremo con

$$\{\phi{\rightarrow}\phi \mid \phi \in \mathcal{T}ype_{Curry}\},$$

l'insieme di tutti i tipi per l'identità .

Un esempio di $\lambda$-termine al quale non possiamo assegnare nessun tipo nel Curry's Type Assignment System, è $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. Intuitivamente, $\Omega$ non è tipabile poichè il sottotermine $xx$ non può essere tipato: se lo fosse, allora la variabile $x$ dovrebbe avere allo stesso tempo un tipo funzionale $\phi{\rightarrow}\psi$ ed un tipo $\phi$, e ciò contraddice l'unicità dell'informazione di tipo contenuta nel contesto che assegna un tipo ad $x$.

Altri tas importanti in letteratura sono il sistema di assegnazione di tipi del secondo ordine, meglio conosciuto come *Polymorphic Type Assignment System* ($F2$) [Lei83], il sistema di assegnazione di tipi di ordine superiore ($F\omega$) [GR88], il sistema di assegnazione di tipi ricorsivi ($\lambda\mu$) [CC90], ed il sistema di assegnazione di tipi intersezione, o *Intersection Type Assignment System* ($\lambda\cap$) [BCD83].

## 1.2 Sistemi Tipati per il $\lambda$-calcolo

I tipi possono essere usati direttamente per "decorare" i termini del $\lambda$-calcolo. Il linguaggio decorato viene detto *$\lambda$-calcolo tipato* [Chu41]. Nel $\lambda$-calcolo tipato, a differenza del $\lambda$-calcolo puro, ogni termine chiuso possiede un unico tipo (in alcuni sistemi a meno di $\beta$-riduzioni sui tipi). Ad esempio, la funzione identità tra interi $\lambda x{:}int.x$ ha tipo $int{\rightarrow}int$. In questo approccio, detto *à la Church*, un *Sistema di Tipi* (ts) è un insieme di regole, espresso in deduzione naturale, che prova giudizi della forma $\Gamma \vdash M : \phi$, dove $M$ è un

$\lambda$-termine tipato, $\phi$ è un tipo e $\Gamma$ è un contesto. Il significato del giudizio è che $M$ ha un (unico) tipo $\phi$, in un contesto $\Gamma$ che assegna i tipi alle variabili libere di $M$. Esempi di $\lambda$-calcoli tipati sono il *Church's Typed $\lambda$-calculus* ($\lambda\!\rightarrow$) [Chu41], il sistema tipato di ordine superiore ($\lambda\omega$), il *Girard's Polymorphic Type System* ($\lambda 2$ o *System F*) [Gir86], il *Logical Framework* (*LF* o $\lambda P$) di Harper, Honsell e Plotkin [HHP92] ed il *Calcolo delle Costruzioni* di Coquand e Huet ($\lambda CC$ o $\lambda P\omega$) [CH88].

Se guardiamo al $\lambda$-calcolo come paradigma di un linguaggio di programmazione, l'approccio à la Church corrisponde ad un linguaggio "esplicitamente tipato" come, ad esempio HASKELL.

### 1.2.1 "Cubi" di Sistemi Tipati per il $\lambda$-calcolo e Logica Intuizionista

H. Barendregt [Bar92] ha studiato alcuni sistemi tipati per il $\lambda$-calcolo, integrandoli in un unico formalismo, detto *cubo dei $\lambda$-calcoli*, o *$\lambda$-cubo* o $\mathcal{TS}$-cubo (vedi Figura *2.1*). Questo formalismo consiste in un numero ridotto di "schemi" di regole ed in una regola di "formazione" di tipo che permette di introdurre differenti costruttori di tipo per ogni sistema del $\lambda$-cubo. In particolare, 8 sistemi tipati per il $\lambda$-calcolo sono stati studiati e disposti (graficamente) sui vertici di un cubo. L'origine del cubo corrisponde al $\lambda$-calcolo tipato semplice di Church, mentre le tre dimensioni del cubo corrispondono all'introduzione di nuove regole di formazione di tipo, cioè i *Tipi Polimorfi*, i *Tipi di Ordine Superiore* ed i *Tipi Dipendenti da Termini.*

Lo studio di ts trae le sue motivazioni dalla relazione che intercorre tra alcuni sistemi tipati ed alcuni sistemi formali della logica intuizionista. Tale relazione è stata profondamente studiata da Curry ed Howard [How80], e viene riferita in letteratura come *Isomorfismo di Curry-Howard*, o principio "formule logiche come tipi" e "prove logiche come $\lambda$-termini tipati".

Per quanto riguarda la parte sinistra del cubo (cioè quella senza tipi dipendenti), la relazione tra sistemi tipati per il $\lambda$-calcolo e sistemi della logica intuizionista è la seguente:

*I tipi sono in corrispondenza biunivoca con le formule della logica,*
*ed i $\lambda$-termini tipati sono in corrispondenza biunivoca con*
*le dimostrazioni delle formule corrispondenti al loro tipo.*

Ad esempio il $\lambda$-termine tipato $\lambda x{:}int.x$ interpreta la codifica della prova logica per $int{\rightarrow}int$ cioè la deduzione naturale

$$\frac{int \vdash int}{\vdash int{\rightarrow}int}\ (I)$$

Le corrispondenze tra sistemi tipati e sistemi formali della logica intuizionista sono:

$\lambda{\rightarrow}$ corrisponde alla *Logica Proposizionale*,
$\lambda 2$ corrisponde alla *Logica Proposizionale del Secondo Ordine*,
$\lambda\omega$ corrisponde alla *Logica Proposizionale di Ordine Superiore*,
$\lambda CC$ corrisponde alla *Logica dei Predicati di Ordine Superiore*.

Se invece consideriamo la parte del cubo con tipi dipendenti, allora la connessione tra sistemi tipati e logiche intuizioniste è meno chiara [Ber88]. Infatti, nel sistema più potente del $\lambda$-cubo $\lambda CC$, ogni formula logica provabile $\varphi$ è tale che la sua interpretazione in $\lambda CC$ è "abitata", cioè esiste un $\lambda$-termine tipato chiuso $M$, tale che possiamo derivare per $M$ il tipo che corrisponde alla codifica di $\varphi$ ("soundness" dell'isomorfismo di Curry-Howard), ma esistono tipi abitati in $\lambda CC$ tali che la loro interpretazione nel sistema logico corrispondente non è derivabile ("incompleteness" dell'isomorfismo di Curry-Howard).

## 1.3 Relazione tra Tas e Ts per il $\lambda$-calcolo

In [Lei83, GHR93] si è osservato che alcuni tas ($F1$, $F2$, $F\omega$) possono essere ottenuti da ts applicando ai termini tipati ed agli schemi di regola una opportuna funzione, detta di "cancellazione di tipi" che elimina le decorazioni di tipo dai $\lambda$-termini tipati. In particolare:

$F1$ si ottiene da $\lambda{\rightarrow}$,
$F2$ si ottiene da $\lambda 2$,
$F\omega$ si ottiene da $\lambda\omega$.

In [GHR93], la funzione di cancellazione, chiamata $\mathcal{E}$, è stata estesa a tutti i sistemi tipati del $\lambda$-cubo di Barendregt. Questa funzione "induce" 8 tas che possono essere disposti (graficamente), come nel $\lambda$-cubo, in un nuovo cubo, detto *Cubo dei Sistemi di Assegnazione di Tipi* (o $\mathcal{TAS}$*-cubo*). La parte destra del $\mathcal{TAS}$-cubo rappresenta un primo tentativo di definire tas con tipi dipendenti da termini. In particolare, il tas corrispondente al sistema tipato $LF$ viene chiamato $DF1$: esso rappresenta il più piccolo tas con tipi dipendenti da termini, mentre il tas corrispondente al Calcolo delle Costruzioni $\lambda CC$, viene chiamato $DF\omega$.

Se consideriamo la parte sinista del $\mathcal{TAS}$-cubo (quella senza tipi dipendenti), la funzione di cancellazione $\mathcal{E}$ induce un isomorfismo tra ts e tas derivazioni: ovvero, se $\mathcal{D}$ è una derivazione corretta in un sistema tipato, allora applicando $\mathcal{E}$ ad ogni oggetto (contesti e $\lambda$-termini tipati) in $\mathcal{D}$ otteniamo una valida derivazione nel corrispondente tas ottenuto dal sistema tipato via $\mathcal{E}$.

L'isomorfismo di Curry-Howard tra i sistemi di assegnazione di tipi e sistemi della logica intuizionista dovrà essere riscritto in forma più debole:

*I tipi sono in corrispondenza biunivoca con le formule della logica, le derivazioni di tipo sono in corrispondenza biunivoca con le dimostrazioni delle formule, ed i $\lambda$-termini puri sono in corrispondenza biunivoca con l'insieme (infinito) delle dimostrazioni delle formule logiche corrispondenti ai tipi che possono essere loro assegnati.*

Ad esempio il $\lambda$-termine puro $\lambda x.x$ è in corrispondenza biunivoca con l'insieme di tutte le dimostrazioni delle formule logiche corrispondenti ai tipi che possono essergli assegnati. Se assegnamo a $\lambda x.x$ il tipo (polimorfo) $\forall\alpha.\alpha{\to}\alpha$ con la derivazione

$$\dfrac{\dfrac{x{:}\alpha \vdash x : \alpha}{\vdash \lambda x.x : \alpha{\to}\alpha}\ (\to Intro)}{\vdash \lambda x.x : \forall\alpha.\alpha{\to}\alpha}\ (\forall\ Intro)$$

allora l'insieme delle dimostrazioni delle formule logiche conterrà la dimostrazione

$$\dfrac{\dfrac{\alpha \vdash \alpha}{\vdash \alpha{\to}\alpha}\ (I)}{\vdash \forall\alpha.\alpha{\to}\alpha}\ (\forall\ I)$$

Le corrispondenze tra sistemi di assegnazione di tipi e sistemi formali della logica intuizionista sono:

$F1$ corrisponde alla *Logica Proposizionale*,
$F2$ corrisponde alla *Logica Proposizionale del Secondo Ordine*,
$F\omega$ corrisponde alla *Logica Proposizionale di Ordine Superiore.*

### 1.3.1 I Tipi Dipendenti dai Termini

I tipi dipendenti sono stati introdotti da de Bruijn [dB80] per codificare formule nella Logica dei Predicati. Nella sua forma essenziale, un tipo dipendente ha la forma

$$\Pi x{:}\phi.\psi,$$

dove $\Pi$ è un binder per la variabile di termine $x$, $\phi$ e $\psi$ sono tipi, e $x$ può occorrere in $\psi$ (ma non in $\phi$). I tipi dipendenti possono essere assegnati ai termini del $\lambda$-calcolo tipato. Dato un termine $M[x]$ in cui la variabile $x$ occorre libera, possiamo assegnare ad $M[x]$ il tipo dipendente $\psi[x]$ in cui la variabile $x$ occorre libera. Possiamo quindi costruire la funzione $\lambda x{:}\phi.M[x]$ ed assegnarle il tipo $\Pi x{:}\phi.\psi[x]$.

Se interpretiamo i tipi come insiemi e le $\lambda$-astrazioni come funzioni tra insiemi, otterremo una funzione dal dominio, rappresentato dall'interpretazione di $\phi$, denotata con $[\![\,\phi\,]\!]$, ad una "famiglia di insiemi", denotata con $[\![\,\psi[x]\,]\!]$, parametrizzata dal valore che associamo in input a tale funzione. Per il ts $LF$, l'isomorfismo di Curry-Howard è ancora valido e tale sistema corrisponde al sistema logico formale del *Calcolo dei Predicati (PRED).*

I tipi dipendenti sono interessanti anche dal punto di vista dell'espressività dei linguaggi di programmazione: infatti la possibilità di definire, ad esempio, strutture liste dati parametrizzate nel numero dei suoi elementi viene espressa, in forma coincisa, con il tipo dipendente $\Pi n{:}int.list(n)$, dove $list(\ )$ è un "costruttore" (ovvero una costante) di lista parametrica.

Lo studio di tas con tipi dipendenti dai termini non può essere univocamente determinato dalla ricerca delle connessioni tra la teoria dei tipi e i sistemi formali della logica. esso, invece, trae motivazioni dall'informatica, ed in particolare dallo studio di linguaggi di programmazione stile ML, dove il programmatore scrive codice non tipato ed il compilatore inferisce un tipo per il programma. Se un tipo è inferito, allora ciò corrisponde ad un criterio di correttezza del programma. La possibilità di introdurre una disciplina di tipi più raffinata come quella dei tipi dipendenti dai termini incrementa l'espressività del linguaggio.

## 1.4 Tas per Linguaggi Object-Oriented

I linguaggi di programmazione Object-Oriented possono essere classificati, in accordo al modello sottostante, come linguaggi basati sulle *Classi* (class-based) o basati sul concetto di *Delegati* (delegation-based). Nei linguaggi basati sulle classi, come ad esempio *SmallTalk* [BI82, GR83] e $C^{++}$ [ES90], l'implementazione di un oggetto è specificata da un "template", cioè la sua classe, ed ogni oggetto è creato per istanza dalla sua classe. I linguaggi basati sul concetto di delegati, come ad esempio *Self* [US87], sono costruiti sull'idea che gli oggetti sono creati "dinamicamente" modificando altri oggetti attraverso opportune operazioni. Più precisamente, l'interfaccia di un oggetto (cioè le operazioni, o messaggi, a cui esso risponde) può essere estesa aggiungendo un metodo oppure ridefinendo (override) il corpo di un metodo già presente nell'interfaccia. Gli oggetti modificati sono considerati come *Prototipi* per gli oggetti creati. Una "computazione" in un linguaggio object-oriented è una sequenza di "scambi di messaggi" tra oggetti; un messaggio è eseguito dall'oggetto ricevente se il messaggio è nell'interfaccia dell'oggetto stesso, oppure è delegato all'oggetto prototipo.

Gli oggetti sono entità del primo ordine (ovvero, possono essere passati come parametri oppure possono essere il risultato di una computazione), costituiti da insiemi di metodi (gli attributi, o variabili di istanza, possono essere considerati come metodi costanti), ed i metodi sono funzioni con un parametro, usualmente chiamato *self*, che denota l'oggetto ricevente del messaggio. Tra le proposte più interessanti di linguaggi ad oggetti ricordiamo quelle di Abadi e Cardelli [AC94] e Fisher, Honsell e Mitchell [FHM94]:

*i)* Abadi e Cardelli [AC94] hanno presentato un calcolo ad oggetti che supporta riscrittura di metodi ed eredità via "sussunzione di oggetti" (ovvero un oggetto con una interfaccia estesa può essere utilizzato in ogni contesto che attende un oggetto con una interfaccia più limitata). La sussunzione tra oggetti ha la sua controparte semantica nella possibilità di definire una relazione di sottotipo.

*ii)* Fisher, Honsell e Mitchell [FHM94] hanno presentato un calcolo funzionale, il *Lambda Calcolo ad Oggetti*, ovvero un $\lambda$-calcolo arricchito con delle primitive *ad-hoc* per costruire oggetti via estensione o riscrittura di metodi ed invio di messaggi ad oggetti. Questo calcolo offre:

*a)* Una grande espressività computazionale che permette una naturale codifica del funzionale di punto fisso e dei numeri naturali.

*b)* Un semplice meccanismo di eredità per "prototipizzazione".

*c)* Il *dinamic-lookup* dei metodi (ovvero, un oggetto risponde ad un messaggio eseguendo il metodo corrispondente al messaggio ricevuto definito più recentemente).

*d)* Un sistema statico di tipi che supporta una elegante forma di *mytype* specializzazione dei metodi (ovvero, un metodo ricorsivo "specializza" il suo tipo nell'oggetto che eredita), ed il riconoscimento dell'errore *"message-not-understood"*, ottenuto inviando un messaggio ad un oggetto che non possiede nel suo protocollo il messaggio stesso.

## 1.5 Il Lambda Calculus of Objects

Un oggetto costruito via estensione nel Lambda Calculus of Objects ($\lambda_{obj}$) ha la seguente forma:

$$\langle \mathtt{e}_1 \leftarrow\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle,$$

dove $\mathtt{e}_1$ rappresenta il prototipo (da cui si eredita il protocollo) ed $\mathtt{m}$ è il nome del metodo che stiamo aggiungendo, con relativo corpo $\mathtt{e}_2$. Analogamente, un oggetto costruito via override in $\lambda_{obj}$ ha la seguente forma:

$$\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle.$$

Spedire un messaggio $\mathtt{m}$ ad un oggetto $\mathtt{e}$ viene espresso con:

$$\langle \mathtt{e}_1 \leftarrow\!*\, \mathtt{m} = \mathtt{e}_2 \rangle \Leftarrow \mathtt{m},$$

dove $\leftarrow\!*$ è $\leftarrow\!+$, o $\leftarrow$, e viene interpretato nella semantica operazionale di $\lambda_{obj}$ con:

$$\mathtt{e}_2\langle \mathtt{e}_1 \leftarrow\!* \, \mathtt{m} = \mathtt{e}_2 \rangle.$$

Questa forma di applicazione, permette di trattare naturalmente il simbolo *self* (che denota l'oggetto ricevente) dei linguaggi object-oriented direttamente via $\lambda$-astrazione. Intuitivamente, il corpo del metodo `m`, cioè $\mathtt{e}_2$, è una funzione il cui primo parametro attuale sarà sempre l'oggetto ricevente stesso.

Se inviamo un messaggio `m` ad un oggetto `e` che non possiede il metodo `m` nella sua interfaccia (o in quella del suo prototipo) allora la valutazione di `e` produrrà il valore errore *message-not-understood*. Ad esempio la valutazione $\langle \mathtt{m} = \mathtt{e} \rangle \Leftarrow \mathtt{n}$ restituisce *message-not-understood*.

La creazione degli oggetti in $\lambda_{obj}$ è abbastanza rigida, poichè il sistema di tipi impone che i metodi aggiunti (o riscritti) devono riferire, nel loro corpo, a metodi che sono già stati inseriti nel prototipo a cui facciamo riferimento. Questa caratteristica induce una semantica operazionale che non è molto naturale: tale semantica usa una definizione di oggetto in *forma normale* (o *standard*) per cui ogni metodo viene definito (o riscritto) esattamente una volta; ciò significa considerare gli oggetti come *sequenze ordinate* di metodi invece di *insiemi*. Inoltre il sistema di tipi non prevede una relazione di sottotipo. Soluzioni parziali (ed ortogonali) a questo problema sono state presentate in [BL95, FM95].

Il sistema di assegnazione di tipi per $\lambda_{obj}$ determina un criterio di correttezza (computazionale) sui programmi che sono tipabili; si dice, quindi, che il tas è "sound" rispetto alla semantica operazionale, cioè :

*Se* $\vdash \mathtt{e} : \phi$*, allora la valutazione di* `e` *non produrrà mai*
*l'errore message-not-understood.*

La parte centale del sistema di tipi usato in [FHM94] consiste nel tipo degli oggetti: un tipo di un oggetto ha la seguente forma:

$$\mathtt{class}\, t.\langle\!\langle \mathtt{m}_1{:}\phi_1, \ldots, \mathtt{m}_k{:}\phi_k \rangle\!\rangle.$$

La struttura $\langle\!\langle \mathtt{m}_1{:}\phi_1, \ldots, \mathtt{m}_k{:}\phi_k \rangle\!\rangle$ viene chiamata *riga*. La variabile $t$ può occorrere nei $\phi_i$, con il significato dell'oggetto stesso. Dunque il tipo di un oggetto è essenzialmente un *tipo record ricorsivo*. Il tipo oggetto descrive la proprietà di un oggetto di possedere come interfaccia $\mathtt{m}_1, \ldots, \mathtt{m}_k$ metodi e quindi di poter rispondere a $\mathtt{m}_i$ messaggi ($1 \leq i \leq k$), producendo come output un risultato di tipo $\phi_i$, dove ogni occorrenza di $t$ nei $\phi_i$ sta per il tipo $\mathtt{class}\, t.\langle\!\langle \mathtt{m}_1{:}\phi_1, \ldots, \mathtt{m}_k{:}\phi_k \rangle\!\rangle$.

## 1.6 Una Relazione di Sottotipo per $\lambda_{obj}$

In [BL95] si è arricchito il sistema di tipi di $\lambda_{obj}$ con una relazione di sottotipo. La definizione intuitiva si sottotipo è :

*Un tipo $\phi$ è un sottotipo di un tipo $\psi$, denotato con $\phi \preceq \psi$, se ogni oggetto di tipo $\phi$ può essere usato in un contesto che attende un oggetto di tipo $\psi$.*

Poichè nei linguaggi delegation-based i tipi rappresentano le "interfacce" degli oggetti, allora la relazione di sottotipo assicura compatibilità tra interfacce più estese verso interfacce più limitate (ma non, ovviamente, il viceversa). Questa relazione di sottotipo è anche detta di "sottotipo in larghezza".

Gli autori di [FM94] osservavano che nei linguaggi delegation-based non era possibile aggiungere una relazione di sottotipo. Gli autori di [BL95] hanno individuato una restrizione della relazione di sottotipo in larghezza compatibile con le caratteristiche peculiari dei linguaggi delegation-based. Questa restrizione permette di ottenere $\phi \preceq \psi$, dove $\phi$ e $\psi$ sono tipi oggetti, solo se i metodi di $\phi$, che non sono in $\psi$, non sono riferiti nei rimanenti metodi di $\psi$. Il sistema di tipi esteso con la relazione di sottotipo permette di tipare un maggior numero di espressioni senza perdere nessuna proprietà peculiare di $\lambda_{obj}$.

L'estensione del sistema di tipi con la relazione di sottotipo è stata possibile aggiungendo i *Labeled-Types*, che codificano nel tipo di un metodo `m` l'informazione relativa ai metodi usati da `m`. L'aggiunta dei labeled-types ha permesso una definizione di una semantica operazionale più intuitiva di quella originale di [FHM94]: infatti, le regole di valutazione cercano i metodi negli oggetti direttamente, senza ricorrere a nessuna "forma normale". Quindi il nuovo calcolo, che è stato chiamato $\lambda_{obj}^{\preceq}$, è una estensione conservativa del calcolo originale $\lambda_{obj}$.

## 1.7 Risultati della Tesi

In questa sezione riassumeremo, brevemente, i contributi originali della tesi:

*i)* Per i sistemi del $\mathcal{TAS}$-cubo sono state provate le proprietà sintattiche fondamentali come la proprietà di Church-Rosser, la riduzione del soggetto e la normalizzazione forte dei termini tipabili. La dimostrazione della proprietà di Church-Rosser utilizza la tecnica delle *riduzioni parallele* sviluppata da Tait e Martin-Löf [Tak89]. La riduzione del soggetto è particolarmente elaborata per la presenza nel $\mathcal{TAS}$-cubo di regole non dirette dalla sintassi (in particolare quelle che introducono i tipi del secondo ordine) e perchè i $\lambda$-termini possono occorrere nei tipi. La normalizzazione forte dei tas con tipi dipendenti è stata ottenuta mappando, attraverso una funzione [PM89, GHR93] che cancella le dipendenze, i tas con tipi dipendenti (parte destra del $\mathcal{TAS}$-cubo) nei corrispondenti tas senza tipi dipendenti (parte sinistra del $\mathcal{TAS}$-cubo), per i quali vale la proprietà di normalizzazione forte.

*ii)* In [GHR93] sono state studiate le relazioni che intercorrono tra il $\mathcal{TS}$-cubo ed il $\mathcal{TAS}$-cubo. In [GHR93] è stato provato che, in presenza di tipi dipendenti, non esiste una corrispondenza biunivoca tra ts e tas derivazioni. Inoltre è stata congetturata l'esistenza di una corrispondenza biunivoca tra ts e tas giudizi. In questa tesi, la congettura è stata smentita per i tas con tipi dipendenti e polimorfi, mostrando un tas giudizio che non è ottenibile per cancellazione da un giudizio derivabile nel ts corrispondente. Questo risultato implica che esistono tipi abitati in tas che non sono ottenibili per cancellazione dai tipi abitati nei corrispondenti ts.

*iii)* Per i tas con i tipi dipendenti ma senza tipi polimorfi è stata contestualmente provata una corrispondenza biunivoca tra ts a tas giudizi.

*iv)* Dopo questo risultato negativo, ci si è chiesto se era possibile modificare opportunamente la funzione di cancellazione $\mathcal{E}$ in modo tale da indurre un nuovo cubo di sistemi di assegnazione di tipi. Una nuova funzione, meno distruttiva della precedente, detta $\mathcal{E}'$, è stata definita ed un nuovo $\mathcal{TAS}'$-cubo è stato definito. Per questo cubo è stato provato l'isomorfismo tra derivazioni.

(Parte di questi risultati sono stati pubblicati in [vBLRU94, vBLRU95]).

*v)* Viene presentata una estensione del Lambda Calculus of Objects ($\lambda_{obj}^{\preceq}$), dove gli oggetti che possiedono una interfaccia estesa possono essere sussunti in un contesto che attende un oggetto con una interfaccia più limitata. Questo calcolo è una estensione conservativa del suo predecessore. Per introdurre la sussunzione tra metodi sono stati introdotti i Labeled-Types (all'interno di un tipo oggetto) che codificano l'informazione di "quali" messaggi sono stati usati all'interno del body di un metodo.

*vi)* Per il calcolo $\lambda_{obj}^{\preceq}$, sono state provate le proprietà di riduzione del soggetto e di soundness del sistema di tipi.

(Parte di questi risultati sono stati pubblicati in [BL95]).

## 1.8 Organizzazione della Tesi

Questa Tesi di Dottorato è organizzata nei seguenti capitoli:

- Nel Capitolo 2 è presentato il $\lambda$-cubo di Barendregt, insieme alle sue proprietà sintattiche fondamentali. Inoltre sono mostrate le differenze principali che intercorrono tra le versioni originali di alcuni sistemi tipati ed le corrispondenti (equivalenti) versioni nel $\lambda$-cubo. Infine è mostrata una nuova (equivalente) versione del $\lambda$-cubo ($\mathcal{TS}$-cubo), dove la sintassi viene divisa in tre differenti (e mutuamente ricorsive) categorie sintattiche; questa versione sarà utilizzata nei successivi capitoli per definire un corrispondente cubo di sistemi di assegnazione di tipi per il $\lambda$-calcolo.

- Nel Capitolo 3 è presentato un cubo di sistemi di assegnazione di tipi ($\mathcal{TAS}$-cubo), ottenuto dal $\mathcal{TS}$-cubo applicando una opportuna funzione di cancellazione che cancella le informazioni di tipo dalle $\lambda$-astrazioni. Le proprietà sintattiche di tale cubo (Church-Rosser, Subject Reduction, Strong Normalization ed altre proprietà fondamentali) sono state provate.

- Il Capitolo 4 è devoluto allo studio delle relazioni che intercorrono tra il $\mathcal{TS}$-cubo ed il $\mathcal{TAS}$-cubo. In particolare sono state definite delle relazioni di *consistenza, similarità* , ed *isomorfismo* tra ts e tas corrispondenti. Sono state provate la similarità dei ts e tas senza tipi dipendenti ed il fallimento della similarità per i sistemi con i tipi dipendenti. Inoltre è stata presentata una nuova definizione di una funzione di cancellazione che induce un nuovo cubo di sistemi di assegnazione di tipi ($\mathcal{TAS}'$-cubo). Per questo cubo è stato provato l'isomorfismo con il $\mathcal{TS}$-cubo. Poichè il $\lambda$-cubo ed il $\mathcal{TS}$-cubo sono equivalenti, l'isomorfismo del $\mathcal{TAS}'$-cubo con il $\lambda$-cubo segue "a fortiori". Si presentano infine lavori correlati e sviluppi futuri della ricerca.

- Il Capitolo 5 introduce i concetti fondamentali dei linguaggi Object-Oriented e della Teoria dei Tipi sottostante a tali linguaggi. Sono inoltre presentati i modelli class-based e delegation-based con particolare riferimento al "modello esistenziale" (class-based) di Pierce e Turner [PT94] ed al Calculus of Primitive Objects di Abadi e Cardelli [AC94].

- Il Capitolo 6 presenta il Lambda Calculus of Objects di Fisher Honsell e Mitchell [FHM94]. La sintassi, la semantica operazionale ed il sistema di tipi sono date e le proprietà di Subject Reduction e Soundness del sistema di assegnazione di tipi sono enunciate. Sono presentati un discreto numero di esempi di oggetti che mostrano l'espressività del calcolo; per alcuni oggetti è presentata la corrispondente derivazione di tipo.

- Il Capitolo 7 introduce una estensione conservativa di $\lambda_{obj}$, ottenuta aggiungendo una restrizione della relazione di sottotipo in larghezza al tas del Lambda Calculus of Objects. Viene inoltre riportata l'osservazione di [FM95] riguardo all'impossibilità di aggiungere una relazione di sottotipo ai linguaggi delegation-based. Il nuovo linguaggio $\lambda^{\preceq}_{obj}$ con relativa semantica operazionale viene definito ed il sistema di tipi, basato sui labeled-types, viene presentato. Un buon numero di esempi che mostrano l'espressività di $\lambda^{\preceq}_{obj}$ ed un esempio di un oggetto, tipabile in $\lambda^{\preceq}_{obj}$ ma non in $\lambda_{obj}$, è considerato. Le proprietà sintattiche fondamentali provate per $\lambda_{obj}$ sono state provate anche per $\lambda^{\preceq}_{obj}$. Si discutono infine lavori correlati e sviluppi futuri della ricerca.

# Chapter 2

# Typed Systems for the $\lambda$-Calculus

## Introduction

Types can be used as predicates for terms of $\lambda$-calculus. Terms can be directly *decorated* with types and then every closed term comes directly with a unique, intrinsic type. In this *fully typed* approach, a *typed system* is a set of rules for proving judgments of the shape $\Gamma \vdash_t M : \phi$, where $M$ is a typed term, $\phi$ is a type, and $\Gamma$ is a context. The meaning of such a judgment is: the term $M$ has type $\phi$ under the context $\Gamma$, and $\Gamma$ records the types of the free variables of $M$ and $\phi$.

The typed approach, called *à la Church* by Barendregt, gives rise to several typed $\lambda$-calculi, where terms are decorated with types in various ways. Examples of typed $\lambda$-calculi are the simply typed $\lambda$-calculus ($\lambda{\rightarrow}$) of Church [Chu41], the second order $\lambda$-calculus of Girard and Reynolds ($\lambda 2$) [Gir86, Rey74], and the Calculus of Constructions ($\lambda P\omega$) of Coquand and Huet [Coq91, CH88]. Barendregt gave in [Bar92] a compact and appealing presentation of a class of typed systems, arranging them in a *cube*. In this cube, every vertex represents a different typed system. One vertex is the *origin* and represents the simply typed $\lambda$-calculus; the three dimensions of the cube represent the introduction of some new rules of type formation, namely *Polymorphism*, *Higher-Order* and *Dependencies* (see Definition *2.4.1 (iv)*). This structure allows for a deep comparative analysis of different typed $\lambda$-calculi.

The relation with (intuitionistic) logic through the so-called Curry-Howard isomorphism [How80], or '*formulae-as-types*' principle, has been profoundly studied for Barendregt's cube, and has been clearly established for the plane of the cube without dependences. However, in the opposite plane, this relation is less clear, as demonstrated by Berardi in [Ber88]. The eight typed systems in Barendregt's cube correspond "roughly" to eight different intuitionistic logical systems through the following principle:

*Given a typed term $M$, if we can derive for $M$ a type $\phi$ in a system of the $\lambda$-cube, then the term $M$ can be seen as the coding of a logical proof, proving the formula $\varphi$ that can be interpreted as the type $\phi$ assigned to $M$.*

For example, $\lambda\!\rightarrow$ corresponds to minimal propositional logic, $\lambda 2$ to minimal second-order propositional logic, and $\lambda P\omega$ to minimal higher-order predicate logic.

When we look at $\lambda$-calculus as a paradigmatic programming language, the typed approach corresponds to explicitly typed languages, like for example HASKELL.

This chapter is organized as follows: Section *2.1* contains a short presentation of Barendregt's cube, as presented in his paper [Bar92]. Moreover, we present, without proof, the main syntactical results that Barendregt's cube enjoys, like the Church-Rosser Property, subject reduction, strong normalization, and a list of properties that will be useful in the next chapters. A through investigation of this properties can be found in [Bar92, GN91]. In Section *2.2*, we present the main differences between the original presentation of some well known systems, namely the simply typed Church's $\lambda$-calculus, and the second order polymorphic Girard's $\lambda$-calculus, and the corresponding presentations in the Barendregt's cube, whereas Section *2.3* gives an idea of what a dependent-type is. Section *2.4* present a stratified presentation of the same cube, where stratified means that we split the abstract syntax into three syntactic categories, called "$\lambda$-terms", "constructors" and "kinds": the reason to do this will become clear in the subsequent chapters of this thesis, where a corresponding cube of *type assignment systems* will be defined, and some suitable relations between the two cubes will studied.

## 2.1 The Cube of the Typed Systems for the $\lambda$-Calculus

Barendregt's cube is a uniform presentation of eight typed systems. It provides a fine structure of the Calculus of Constructions [CH88], which is the strongest system in the cube. All systems are given in a *constructive way*, in the sense that in the rules there are no statements written such as, e.g., $\sigma \in \mathcal{T}ype$. In fact, some well known systems, like $\lambda\!\rightarrow$ and $\lambda 2$, are given in a constructive way. Barendregt's cube is normally presented using a rather compact notation, using *rule schemes* rather than rules. The classification of these systems is given by controlling the way in which abstraction is allowed. We begin the presentation with the syntax and some useful definitions.

*Definition 2.1.1 Given the set of* sorts, *i.e.:*

$$s \in \{*, \Box\},$$

*the sets of* pseudo-terms *is inductively defined by the following grammar:*

$$A ::= s \mid a \mid \lambda a{:}A.B \mid AB \mid \lambda a{:}A.B \mid \Pi a{:}A.B.$$

*Definition 2.1.2 The set of* free variables of $A$, *denoted by* $\mathcal{FV}(A)$, *is inductively defined by:*

$$\begin{aligned}
\mathcal{FV}(*) &= \emptyset,\\
\mathcal{FV}(a) &= \{a\},\\
\mathcal{FV}(BC) &= \mathcal{FV}(B) \cup \mathcal{FV}(C),\\
\mathcal{FV}(\Pi a{:}B.C) &= \mathcal{FV}(B) \cup (\mathcal{FV}(C) \setminus \{a\}),\\
\mathcal{FV}(\lambda a{:}B.C) &= \mathcal{FV}(B) \cup (\mathcal{FV}(C) \setminus \{a\}).
\end{aligned}$$

*Definition 2.1.3 The result of a simultaneous substitution* $\vartheta = [A_1/a_1, \ldots, A_n/a_n]$, *applied to a term* $D$, *is denoted either by* $D[A_1/a_1, \ldots, A_n/a_n]$, *or by* $D^\vartheta$. *We normally assume that no variable bound in* $D$ *is free in any of the* $A_i$*'s and that the set* $\{a_1, \ldots, a_n\}$ *is disjoint from the set of bound variables of* $D$. *Formally, the substitution on typed terms is inductively defined by:*

$$\begin{aligned}
a_i^\vartheta &\equiv A_i, \text{ for } 1 \leq i \leq n,\\
b^\vartheta &\equiv b, \text{ for every } b \notin \{a_1, \ldots, a_n\},\\
(BC)^\vartheta &\equiv B^\vartheta C^\vartheta,\\
(\lambda b{:}B.C)^\vartheta &\equiv \lambda b{:}B^\vartheta.C^\vartheta,\\
(\Pi b{:}B.C)^\vartheta &\equiv \Pi b{:}B^\vartheta.C^\vartheta.
\end{aligned}$$

*Definition 2.1.4* (Typed reduction) *$\beta$-reduction on typed terms (denoted as $\twoheadrightarrow_\beta$ ) is defined as usual, i.e., as the contextual, reflexive and transitive closure of the following one-step reduction rule:*

$$(\lambda a{:}A.B)C \rightarrow_\beta B[C/a].$$

*The symbol $=_\beta$ denotes $\beta$-conversion, i.e., the least equivalence relation generated by $\twoheadrightarrow_\beta$ .*

*Definition 2.1.5* *The set of* subterms *of $A$, denoted by $\mathcal{ST}(A)$, is inductively defined by:*

$$\begin{array}{rcl}
\mathcal{ST}(*) & = & \{*\}, \\
\mathcal{ST}(a) & = & \{a\}, \\
\mathcal{ST}(BC) & = & \{BC\} \cup \mathcal{ST}(B) \cup \mathcal{ST}(C), \\
\mathcal{ST}(\Pi a{:}B.C) & = & \{\Pi a{:}B.C\} \cup \mathcal{ST}(B) \cup \mathcal{ST}(C), \\
\mathcal{ST}(\lambda a{:}B.C) & = & \{\lambda a{:}B.C\} \cup \mathcal{ST}(B) \cup \mathcal{ST}(C).
\end{array}$$

*Definition 2.1.6* *i) A* statement *is an expression of the form:*

$$A : B,$$

*where $A$ and $B$ are pseudo-terms. The left part of the statement is called the* subject*, the right part is called the* predicate*.*

*ii) A* declaration *is a statement whose subject is a variable.*

*Definition 2.1.7* *i) A* context *is a sequence of declarations, whose subjects are distinct. The empty context is denoted by $\varepsilon$.*

*ii) Equality on contexts is inductively defined by:*

*a) $\varepsilon = \varepsilon$;*

*b) $\Gamma, a{:}A = \Gamma', b{:}B$, if $\Gamma = \Gamma'$, $a \equiv b$, and $A \equiv B$.*

*iii) We write $a{:}A \in \Gamma$, if the declaration $a{:}A$ occurs in $\Gamma$.*

*iv) The* domain *of $\Gamma$, denoted by $\mathcal{D}om(\Gamma)$, is the set $\{a \mid \exists\ A\ [a{:}A \in \Gamma]\}$.*

*v)* *If* $\Gamma_1$ *and* $\Gamma_2$ *are contexts such that* $\mathcal{D}om\,(\Gamma_1) \cap \mathcal{D}om\,(\Gamma_2) = \emptyset$, *then* $\Gamma_1, \Gamma_2$ *is a context obtained by concatenating* $\Gamma_1$ *to* $\Gamma_2$.

*vi)* $\mathcal{FV}(\Gamma) = \bigcup\{\mathcal{FV}(A) \mid \exists a\ [a{:}A \in \Gamma]\}$.

*vii)* *We extend the notion of substitution to contexts by:* $\varepsilon^{\vartheta} = \varepsilon$, *and* $(\Gamma, b{:}B)^{\vartheta} = \Gamma^{\vartheta}, b{:}B^{\vartheta}$.

In the next definition, we give the presentation of Barendregt's typed systems, as can be found in [Bar92]

*Definition 2.1.8* (Barendregt's General Typed System) *i) The following rules are used to derive* judgments *of the form*

$$\Gamma \vdash_t A : B,$$

*where* $\Gamma$ *is a context and* $A : B$ *is a statement; in this case* $A$ *and* $B$ *are called (legal) terms, and* $\Gamma$ *is a (legal) context.*

*ii) The derivation rules are:*

$$(Axiom) \qquad \frac{}{\varepsilon \vdash_t * : \Box}$$

$$(Proj) \qquad \frac{\Gamma \vdash_t A : s \quad a \notin \mathcal{D}om\,(\Gamma)}{\Gamma, a{:}A \vdash_t a : A}$$

$$(Weak) \qquad \frac{\Gamma \vdash_t A : B \quad \Gamma \vdash_t C : s \quad c \notin \mathcal{D}om\,(\Gamma)}{\Gamma, c{:}C \vdash_t A : B}$$

$$(Conv) \qquad \frac{\Gamma \vdash_t A : B \quad \Gamma \vdash_t C : s \quad B =_\beta C}{\Gamma \vdash_t A : C}$$

$$(I) \qquad \frac{\Gamma, a{:}A \vdash_t B : C}{\Gamma \vdash_t \lambda a{:}A.B : \Pi a{:}A.C}$$

$$(E) \qquad \frac{\Gamma \vdash_t A : \Pi c{:}C.D \quad \Gamma \vdash_t B : C}{\Gamma \vdash_t AB : D[B/c]}$$

$$(F_{s_1,s_2}) \qquad \frac{\Gamma, a{:}A \vdash_t B : s_1}{\Gamma \vdash_t \Pi a{:}A.B : s_2}$$

*iii) The eight systems are defined by taking the* $(Axiom)$, $(Proj)$, $(Weak)$, $(Conv)$, $(I)$, $(E)$ *rules, plus a specific subset of the set of rules* $\{(F_{*,*}), (F_{\Box,*}), (F_{*,\Box}), (F_{\Box,\Box})\}$ *as follows:*

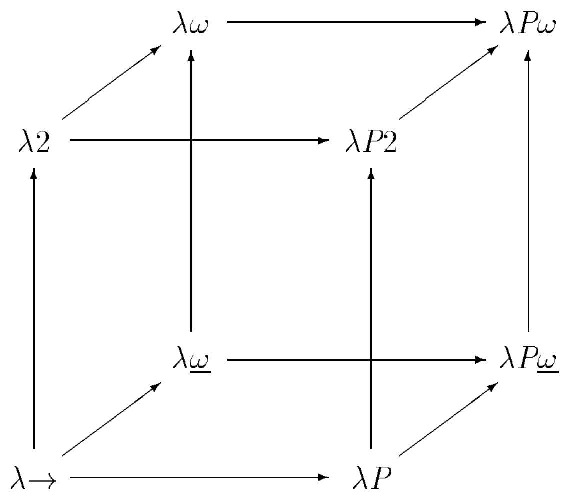



Figure 2.1: Barendregt's $\lambda$-Cube

| | | | | |
|---|---|---|---|---|
| $\lambda\!\to$ | $(F_{*,*})$ | | | |
| $\lambda 2$ | $(F_{*,*})$ | $(F_{\Box,*})$ | | |
| $\lambda\underline{\omega}$ | $(F_{*,*})$ | | $(F_{*,\Box})$ | |
| $\lambda P$ | $(F_{*,*})$ | $(F_{\Box,*})$ | $(F_{*,\Box})$ | |
| $\lambda P\underline{\omega}$ | $(F_{*,*})$ | | | $(F_{\Box,\Box})$ |
| $\lambda\omega$ | $(F_{*,*})$ | $(F_{\Box,*})$ | | $(F_{\Box,\Box})$ |
| $\lambda P2$ | $(F_{*,*})$ | | $(F_{*,\Box})$ | $(F_{\Box,\Box})$ |
| $\lambda P\omega$ | $(F_{*,*})$ | $(F_{\Box,*})$ | $(F_{*,\Box})$ | $(F_{\Box,\Box})$, |

*where the pair $(s_1, s_2)$, in the rule $(F_{s_1,s_2})$, denotes the sort of the premise and of the conclusion, respectively; as example, the rule $(F_{\Box,*})$ denotes the following rule:*

$$\frac{\Gamma, a{:}A \vdash_t B : \Box}{\Gamma \vdash_t \Pi a{:}A.B : *}\ (F_{\Box,*})$$

*which corresponds to the introduction of a polymorphic-type.*

Figure *2.1* shows the eight typed systems arranged as vertices of the *Barendregt's cube*. Observe that, in this cube, the edges are intended as an inclusion relation between systems.

*Definition 2.1.9* *i) If $\Gamma \vdash_t A : *$, then $A$ is a "type" in the context $\Gamma$.*

*ii) If $\Gamma \vdash_t A : \Box$, then $A$ is a "kind" in the context $\Gamma$.*

*iii) If $\Gamma \vdash_t A : B$ and $\Gamma \vdash_t B : *$, then $A$ is a (typed) $\lambda$-term in the context $\Gamma$.*

*iv) If $\Gamma \vdash_t A : B$ and $\Gamma \vdash_t B : \Box$, then $A$ is a "constructor". Since $\vdash_t * : \Box$, it follows that a type is also a constructor.*

In what follows, we adopt the notational convention on the names of pseudo-terms: $\lambda$-term-variables will be denoted by $x$, $y$, $z$, ..., and $\lambda$-terms by $M$, $N$, ..., whereas type- and constructors-variables will be denoted by $\alpha$, $\beta$, $\gamma$, ..., and types and constructors by $\phi$, $\psi$, $\sigma$, $\tau$, ... This choice will be useful in the next chapter, when those conventions will be taken as part of the syntax of a new "stratified" presentation of the $\lambda$-cube.

*Remark 2.1.10* Most of the systems in the $\lambda$-cube appear elsewhere in the literature. In particular: the system $\lambda\!\rightarrow$ corresponds to Church's typed system [Chu41], whereas $\lambda 2$ is the second order polymorphic $\lambda$-calculus, which is essentially Girard's system $F$ [Gir86].

The step to go from system $\lambda\!\rightarrow$ to system $\lambda 2$ is to allow for *polymorphic-types*, i.e., introducing the mechanism of binding (free) constructor-variables, as for

$$\Gamma \vdash_t \lambda\alpha{:}{*}.M : \Pi\alpha{:}{*}.\phi,$$

and replacing them by constructors, as in

$$\Gamma \vdash_t M\psi : \phi[\psi/\alpha].$$

The step to go from system $\lambda\!\rightarrow$ to system $\lambda\underline{\omega}$ is to allow for *higher-order types*, i.e., *types dependent on types*, that makes it possible to derive

$$\Gamma \vdash_t (\lambda\alpha{:}{*}.\psi)\phi : *.$$

The step to go from system $\lambda\!\rightarrow$ to system $\lambda P$, is to allow for the construction of *dependent-types*, i.e., *types dependent on $\lambda$-terms*, that makes it possible to derive

$$\Gamma \vdash_t (\lambda x{:}\phi.\psi)N : *.$$

So, in the previous table, the rule $(F_{\Box,*})$ introduces polymorphic-types, the rules $(F_{*,\Box})$, and $(F_{\Box,\Box})$ introduces the higher-order types, and dependent-types respectively. The well known systems of the simply typed $\lambda$-calculus, and the system $F$ of Girard are given in an equivalent constructive version. The system $\lambda P$ is often called *Logical Framework* [HHP92]. The system $\lambda P\omega$ (also called $\lambda CC$) is the Calculus of Construction of Coquand and Huet [CH88].

### 2.1.1 Properties of Barendregt $\lambda$-Cube

The properties of this cube are proved in [Bar92, GN91]. We list just a few of them, those that are used in the next chapters.

*Proposition 2.1.11* *If* $\Gamma \vdash_t A : s_1$ *and* $\Gamma \vdash_t A : s_2$, *then* $s_1 \equiv s_2$. $\blacksquare$

*Property 2.1.12* $A[B/b][C/c] \equiv A[C/c][B[C/c]/b]$, *provided* $b \notin \mathcal{FV}(C)$. $\blacksquare$

*Property 2.1.13* (Church-Rosser Property for Typed Systems) $A =_\beta B$ *&* $\Gamma \vdash_t A : C$ *&* $\Gamma \vdash_t B : C \;\Rightarrow\; \exists\, D\, [A \twoheadrightarrow_\beta D$ *&* $B \twoheadrightarrow_\beta D]$. $\blacksquare$

*Property 2.1.14* *The General Typed System derives judgments of the following shapes:*

$$\Gamma \vdash_t M : \phi, \quad \Gamma \vdash_t \phi : K, \quad or \quad \Gamma \vdash_t K : \Box. \qquad \blacksquare$$

*Definition 2.1.15* *i) A typed* legal context *is inductively defined as follows:*

*a)* $\varepsilon$ *is legal;*

*b)* $\Gamma, a{:}A$ *is legal* $\;\Leftrightarrow\;$ $\Gamma$ *is legal* *&* $\Gamma \vdash_t A : s$, *&* $a \notin \mathcal{D}om\,(\Gamma)$.

*ii) The relation* $\sqsubseteq$ *is inductively defined on legal contexts as follows:*

*a)* $\varepsilon \sqsubseteq \Gamma$*;*

*b)* $\Gamma \sqsubseteq \Gamma' \;\Rightarrow\; \Gamma, a{:}A \sqsubseteq \Gamma', a{:}A$*;*

*c)* $\Gamma \sqsubseteq \Gamma' \;\Rightarrow\; \Gamma \sqsubseteq \Gamma', a{:}A$.

*Property 2.1.16* *i)* $\Gamma \sqsubseteq \Gamma'$ *&* $\Gamma \vdash A : B \;\Rightarrow\; \Gamma' \vdash A : B$.

*ii)* $\mathcal{D}\colon \Gamma, c{:}C \vdash_t A : B \;\Rightarrow\; \exists\, \mathcal{D}' \subseteq \mathcal{D}\, [\mathcal{D}'\colon \Gamma \vdash_t C : s]$.

*Property 2.1.17* (Typed Generation Lemma) *i)* $\Gamma \vdash_t a : A \;\Rightarrow\; \exists\, s, B\, [\Gamma \vdash_t B : s$ *&* $a{:}B \in \Gamma$ *&* $A =_\beta B]$.

$$\text{Syntax of } \mathcal{T}ype^{\rightarrow}\text{:} \quad \phi ::= \alpha \mid \phi{\rightarrow}\phi$$

$$(Axiom) \quad \frac{x{:}\phi \in \Gamma \qquad \phi \in \mathcal{T}ype^{\rightarrow}}{\Gamma \vdash_t x{:}\phi}$$

$$(\rightarrow Elim) \quad \frac{\Gamma \vdash_t M : \phi{\rightarrow}\psi \qquad \Gamma \vdash_t N : \phi}{\Gamma \vdash_t MN : \psi} \qquad (\rightarrow Intro) \quad \frac{\Gamma, x{:}\phi \vdash_t M : \psi}{\Gamma \vdash_t \lambda x{:}\phi.M : \phi{\rightarrow}\psi}$$

Figure 2.2: Church's Typed $\lambda$-Calculus

*ii)* $\Gamma \vdash_t \Pi a{:}A.B : C \;\Rightarrow\; \exists\, s_1, s_2, s_3\ [\Gamma \vdash_t A : s_1 \;\&\; \Gamma, a{:}A \vdash_t B : s_2 \;\&\; C =_\beta s_3]$.
*iii)* $\Gamma \vdash_t \lambda a{:}A.B : C \;\Rightarrow\; \exists\, s, D\ [\Gamma \vdash_t \Pi a{:}A.D : s \;\&\; \Gamma, a{:}A \vdash_t B : D \;\&\; C =_\beta \Pi a{:}A.D]$.
*iv)* $\Gamma \vdash_t AB : C \;\Rightarrow\; \exists\, D, E\ [\Gamma \vdash_t A : \Pi d{:}D.E \;\&\; \Gamma \vdash_t B : D \;\&\; C =_\beta E[B/d]]$. ■

*Property 2.1.18* $\Gamma \vdash_t A : B \;\Rightarrow\; B \equiv \Box \vee \Gamma_t \vdash_t B : s$. *In particular, if* $A \equiv M$, *then* $B \equiv *$, *and* $\phi$ *is a* type *with respect to the context* $\Gamma$. ■

*Property 2.1.19* (Termination for typed terms) *If* $\Gamma \vdash_t A : B$, *then* $A$ *and* $B$ *are both strongly normalizing.* ■

The next section will present the main differences between the systems in the cube and their original versions. Moreover, Section *2.3* will present the main ideas underlying dependent-types of $\lambda P$.

## 2.2 Constructive Type Systems

In this section, we show the differences between the original presentation of the simply and polymorphic typed $\lambda$-calculi, presented in Figures *2.1.1* and *2.1.1*, and their presentations in the $\lambda$-cube. In this cube, the systems are given in a *constructive* way, in the sense that types are formally generated by the system itself and not in the informal metalanguage. There is a constant $*$ such that $\phi : *$ corresponds to $\phi \in \mathcal{T}ype$. So, the original notion of *context as set* containing at least all the declarations for the free variables of the subject of the judgment, becomes the notion of *context as list*, containing

Syntax of $\mathcal{T}\!ype^{\forall}$: $\phi ::= \alpha \mid \phi \to \phi \mid \forall \alpha.\phi$

$$(\mathit{Axiom}) \quad \frac{x{:}\phi \in \Gamma \qquad \phi \in \mathcal{T}\!ype^{\forall}}{\Gamma \vdash_t x{:}\phi}$$

$$(\to \mathit{Elim}) \quad \frac{\Gamma \vdash_t M : \phi \to \psi \qquad \Gamma \vdash_t N : \phi}{\Gamma \vdash_t MN : \psi} \qquad (\to \mathit{Intro}) \quad \frac{\Gamma, x{:}\phi \vdash_t M : \psi}{\Gamma \vdash_t \lambda x{:}\phi.M : \phi \to \psi}$$

$$(\forall \mathit{Elim}) \quad \frac{\Gamma \vdash_t M : \forall \alpha.\phi \qquad \psi \in \mathcal{T}\!ype^{\forall}}{\Gamma \vdash_t M\psi : \phi[\psi/\alpha]} \qquad (\forall \mathit{Intro}) \quad \frac{\Gamma \vdash_t M : \phi \qquad \alpha \notin \mathcal{FV}(\Gamma)}{\Gamma \vdash_t \Lambda \alpha.M : \forall \alpha.\phi}$$

Figure 2.3: The Girard's Polymorphic λ-Calculus

at least all the declarations for the free variables of both the subject and the predicate of the judgment.

We will now show, with help of an example, how some informal statements can be formally written as derivations in the λ-cube.

*Example 2.2.1* *i)* Consider the following definition of arrow-type formation:

$$\alpha, \beta \in \mathcal{T}\!ype \Rightarrow \alpha \to \beta \in \mathcal{T}\!ype.$$

This corresponds to the following derivation in the λ-cube:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varepsilon \vdash_t * : \Box}\,(\mathit{Axiom})\quad (1)}{\alpha{:}* \vdash_t \alpha : *}\,(\mathit{Proj}) \qquad \dfrac{\vdots}{\alpha{:}* \vdash_t * : \Box}\quad (2)}{\alpha{:}*, \beta{:}* \vdash_t \alpha : *}\,(\mathit{Weak}) \qquad \dfrac{\vdots}{\alpha{:}*, \beta{:}* \vdash_t \beta : *}\quad (3)}{\alpha{:}*, \beta{:}*, x{:}\alpha \vdash_t \beta : *}\,(\mathit{Weak})}{\alpha{:}*, \beta{:}* \vdash_t \Pi x{:}\alpha.\beta : *}\,(I)$$

where (1), (2), and (3) stand for $\alpha \notin \mathcal{D}om\,(\varepsilon)$, $\beta \notin \mathcal{D}om\,(\alpha{:}*)$, and $y \notin \mathcal{D}om\,(\alpha{:}*, \beta{:}*)$ respectively, and we will see in this section that $\Pi x{:}\alpha.\beta \equiv \alpha \to \beta$, if $x \notin \mathcal{FV}(\alpha) \cup \mathcal{FV}(\beta)$.

*ii)* Consider the following derivation in the original presentation of the simply typed λ-calculus of Church:

$$(Axiom)\ \frac{\alpha{\to}\beta \in \mathcal{T}ype}{y{:}\alpha{\to}\beta \vdash_{Ch} y : \alpha{\to}\beta}$$

This derivation will be denoted in its constructive version $\lambda\!\to$, by the following derivation:

$$(Proj)\ \frac{\mathcal{D} \quad y \notin \mathcal{D}om\,(\alpha{:}*, \beta{:}*)}{\alpha{:}*, \beta{:}*, y{:}\Pi x{:}\alpha.\beta \vdash_t y : \Pi x{:}\alpha.\beta}$$

where $\mathcal{D}$ is the derivation of part $(i)$, and $x$ does not occurs in $\alpha$, and $\beta$.

*Remark 2.2.2* Observe that the order of the type-variables in the context of the first example is not important, although in the second example it is. In fact, we can derive (using different derivations)

$$\beta{:}*, \alpha{:}* \vdash_t \Pi x{:}\alpha.\beta : *,$$
$$\beta{:}*, \alpha{:}*, y{:}\Pi x{:}\alpha.\beta \vdash_t y : \Pi x{:}\alpha.\beta,$$

but we cannot derive

$$\alpha{:}*, y{:}\Pi x{:}\alpha.\beta, \beta{:}* \vdash_t y : \Pi x{:}\alpha.\beta,$$
$$\beta{:}*, y{:}\Pi x{:}\alpha.\beta, \alpha{:}* \vdash_t y : \Pi x{:}\alpha.\beta,$$
$$y{:}\Pi x{:}\alpha.\beta, \alpha{:}*, \beta{:}* \vdash_t y : \Pi x{:}\alpha.\beta,$$

as one easily checks by looking at the rule schemes of the cube. So, modifying the order of a (legal) context affects derivability.

## 2.3 Type Dependencies

In the previous section, we showed the main differences between the original presentation of the simply typed $\lambda$-calculus of Church, system $F$, and their equivalent variants in the Barendregt's cube. In this section, we present the main ideas that motivate the use of dependent-types. Dependent-types have been fruitfully used for fully formalizing mathematical objects (see AUTOMATH [dB80], LF [HHP92], and NUPRL [Con86], and the Type Theory of Martin-Löf [Mar84]). What follows is mainly taken from [Bar92].

Dependent-types where invented by N.G. de Bruijn in the seventies; the main idea is the introduction of *product-types*. Product-types have the following form:

$$\Pi x{:}\phi.\psi,$$

where $x$ is a $\lambda$-term variable, $\phi$ and $\psi$ are types, and $x$ may occur free in $\psi$. The underlying idea is as follows: for every $x$ of type $\phi$, there exists a term $M[x]$ of type $\psi[x]$, where $M[x]$ and $\psi[x]$ denote that $M$ and $\psi$ contain free occurrences of $x$. Then we can build the function:

$$\lambda x{:}\phi.M[x] \text{ of type } \Pi x{:}\phi.\psi[x].$$

So, by interpreting types as set and $\lambda$-abstractions as functions over sets, we get a function from the domain, represented by the interpretation of $\phi$, to the codomain, represented by the interpretation of a family of types, each one depending on the input argument of the above function. If the bound variable $x$ does not occur in $\psi$, i.e. (with a little abuse of notation), $\psi[\ ] \equiv \psi$, then the family of sets obviously collapses in a single set, representing the codomain of the function. In this case we recover the usual interpretation of the arrow-type.

The typed systems with dependences where invented to prove the *validity* of a logic formulas in the predicate calculus PRED. In fact, the main idea is as follows:

[dB80] *Given a logic formula $\varphi$ in PRED, there exist a translation map $||\ ||$ from formulas in PRED into types in $\lambda P$ such that $\varphi$ is a valid formula if and only $||\varphi||$ is inhabited in $\lambda P$, i.e. there exists $\Gamma$, and $M$, such that $\Gamma \vdash_t M : ||\varphi||$ is derivable in system $\lambda P$.*

In fact, system $\lambda P$ is given that name because of PRED. The first project realizing this correspondence was AUTOMATH [dB80], which uses a slight modification of $\lambda P$. The main difference between $\lambda P$ and the system used by AUTOMATH, is that in $\lambda P$ the kind of a logic formulas is $*$, whereas AUTOMATH introduces a special constant *Prop* of kind $*$, which represents the kind of well formed formulas, and a logic formula $\varphi$ is valid if and only if its translation, $T|\varphi|$ is inhabited, where $T$ is a special variable of kind $Prop{\rightarrow}*$.

A recent system using dependent-types for formalizing logics is the system LF [HHP92], developed at the Edimburgh University by Harper, Honsell and Plotkin. It is essentially the system $\lambda P$.

## 2.4 The Stratified Presentation of the $\mathcal{TS}$ Cube

In the previous section, we showed the original presentation of the Barendregt's cube of typed systems, as in [Bar92]. In this section, we present a 'stratified' version of the systems in the cube, i.e., we will split the terms considered by Barendregt in three different classes, being those of $\lambda$-terms, constructors, and kinds, such that each class comes with its own derivations rules. This should enable the appreciation of the presentation of our cube of type assignment systems in the next chapter.

*Notational Conventions* In this section (and in Chapters 2, and 3), a *term* will be either an *(un)typed $\lambda$-term*, a *constructor*, a *kind*, or a *sort*. The symbols $M$, $N$, $P$, $Q$, ... range over (un)typed $\lambda$-terms; $\phi$, $\psi$, $\xi$, $\mu$, ... range over constructors; $K$ ranges over kinds; $s$ ranges over sorts: $A$, $B$, $C$, $D$, ... range over arbitrary terms; $x$, $y$, $z$, ... range over $\lambda$-term-variables; $\alpha$, $\beta$, $\gamma$, ... range over constructor-variables; $a$, $b$, $c$, ... range over $\lambda$-term-variables and constructor-variables. The symbol $\Gamma$ will range over contexts. All symbols can appear indexed. The symbol $\equiv$ denotes the syntactic identity of terms, and we will consider terms modulo $\alpha$-conversion. The notation $\Pi_{i=1}^{n} a_i{:}A_i.B$ is an abbreviation of $\Pi a_1{:}A_1.\cdots\Pi a_n{:}A_n.B$.

*Definition 2.4.1* (Abstract Stratified Typed Syntax) *Given the set of* sorts*, i.e.:*

$$s \in \{*, \square\},$$

*the sets of* typed $\lambda$-terms *($\Lambda_t$),* typed constructors *($Cons_t$), and* typed kinds *($Kind_t$) are mutually defined by the following grammar, where $M, \phi$, and $K$ are metavariables for $\lambda$-terms, constructors and kinds respectively:*

$$M ::= x \mid \lambda x{:}\phi.M \mid MM \mid \lambda\alpha{:}K.M \mid M\phi$$

$$\phi ::= \alpha \mid \Pi x{:}\phi.\phi \mid \Pi\alpha{:}K.\phi \mid \lambda x{:}\phi.\phi \mid \lambda\alpha{:}K.\phi \mid \phi\phi \mid \phi M$$

$$K ::= * \mid \Pi x{:}\phi.K \mid \Pi\alpha{:}K.K$$

*The set $T_t$ of* typed terms *is the union of the sets $\Lambda_t$, $Cons_t$ and $Kind_t$.*

*Remark 2.4.2* The introduction of three classes of 'terms' in Definition *2.4.1* induces a stratified version of the set derivation rules; each class comes with its own derivations rules. The names of the rules are, to save space, restricted to a few characters. We have tried to use an orthogonal approach in baptizing the rules: in general, a name for a rule is composed like $(X{-}Y_Z)$, meaning that:

*i)* it is a rule that follows the syntax of objects in class $X$, where $X$ is omitted for $\lambda$-terms, is $C$ for constructors, and $K$ for kinds,

*ii)* $Y$ is either

  *a)* $I$ for an introduction rule, that are used to deal with the various $\lambda$-abstractions,

  *b)* $E$ for an elimination rule, that deal with applications,

  *c)* $F$ for a formation rule, that deal with the $\Pi$-abstraction,

*iii)* and $Z$ is used (as $X$ above) to indicate the class either of the bound variable (in case of an introduction or formation rule), or of the right-hand side term in an application (in case of a formation rule).

*Definition 2.4.3* (General Typed System) *i) The General Typed System proves* judgments *of the form*

$$\Gamma \vdash_t A : B,$$

*where $\Gamma$ is a context and $A : B$ is a statement.*

*ii) The General Typed System's set of rules, can be divided in four groups, depending of the subjects of the statements:*

*Common Rules*

$$(Proj)\quad \frac{\Gamma \vdash_t A : s \quad a \notin \mathcal{D}om\,(\Gamma)}{\Gamma, a{:}A \vdash_t a : A} \qquad (Weak)\quad \frac{\Gamma \vdash_t A : B \quad \Gamma \vdash_t C : s \quad c \notin \mathcal{D}om\,(\Gamma)}{\Gamma, c{:}C \vdash_t A : B}$$

$$(Conv)\quad \frac{\Gamma \vdash_t A : B \quad \Gamma \vdash_t C : s \quad B =_\beta C}{\Gamma \vdash_t A : C}$$

*Typed $\lambda$-Term Rules*

$$(I)\quad \frac{\Gamma, x{:}\phi \vdash_t M : \psi}{\Gamma \vdash_t \lambda x{:}\phi.M : \Pi x{:}\phi.\psi} \qquad (E)\quad \frac{\Gamma \vdash_t M : \Pi x{:}\phi.\psi \quad \Gamma \vdash_t N : \phi}{\Gamma \vdash_t MN : \psi[N/x]}$$

$$(I_K)\quad \frac{\Gamma, \alpha{:}K \vdash_t M : \phi}{\Gamma \vdash_t \lambda \alpha{:}K.M : \Pi \alpha{:}K.\phi} \qquad (E_K)\quad \frac{\Gamma \vdash_t M : \Pi \alpha{:}K.\phi \quad \Gamma \vdash_t \psi : K}{\Gamma \vdash_t M\psi : \phi[\psi/\alpha]}$$

*Typed Constructor Rules*

$$(C\text{-}I_C)\quad \frac{\Gamma, x{:}\phi \vdash_t \psi : K}{\Gamma \vdash_t \lambda x{:}\phi.\psi : \Pi x{:}\phi.K} \qquad (C\text{-}E_C)\quad \frac{\Gamma \vdash_t \psi : \Pi x{:}\phi.K \quad \Gamma \vdash_t M : \phi}{\Gamma \vdash_t \psi M : K[M/x]}$$

$$(C\text{-}I_K)\quad \frac{\Gamma, \alpha{:}K_1 \vdash_t \psi : K_2}{\Gamma \vdash_t \lambda \alpha{:}K_1.\psi : \Pi \alpha{:}K_1.K_2} \qquad (C\text{-}E_K)\quad \frac{\Gamma \vdash_t \phi : \Pi \alpha{:}K_1.K_2 \quad \Gamma \vdash_t \psi : K_1}{\Gamma \vdash_t \phi\psi : K_2[\psi/\alpha]}$$

$$(C\text{-}F_C)\quad \frac{\Gamma, x{:}\phi \vdash_t \psi : *}{\Gamma \vdash_t \Pi x{:}\phi.\psi : *} \qquad (C\text{-}F_K)\quad \frac{\Gamma, \alpha{:}K \vdash_t \phi : *}{\Gamma \vdash_t \Pi \alpha{:}K.\phi : *}$$

*Typed Kind Rules*

$$(Axiom)\quad \frac{}{\varepsilon \vdash_t * : \Box} \qquad (K\text{-}F_C)\quad \frac{\Gamma, x{:}\phi \vdash_t K : \Box}{\Gamma \vdash_t \Pi x{:}\phi.K : \Box}$$

$$(K\text{-}F_K)\quad \frac{\Gamma, \alpha{:}K_1 \vdash_t K_2 : \Box}{\Gamma \vdash_t \Pi \alpha{:}K_1.K_2 : \Box}$$

*iii) Let the following sets of rules be defined by:*

$$\begin{aligned}
\textit{Base-Rules} &= \{(Axiom), (Proj), (Weak), (I), (E), (C\text{-}F_C)\},\\
\textit{Polymorphism} &= \{(I_K), (E_K), (C\text{-}F_K)\},\\
\textit{Dependencies} &= \{(C\text{-}I_C), (C\text{-}E_C), (K\text{-}F_C), (Conv)\},\\
\textit{Higher-Order} &= \{(C\text{-}I_K), (C\text{-}E_K), (K\text{-}F_K), (Conv)\}.
\end{aligned}$$

*When allowing abstraction to constructor-variables, redexes in constructors can be created. The rule (Conv) is present in both the sets Higher-Order and Dependencies because we want to identify $\beta$-equivalent constructors.*

*iv) The eight typed systems in the stratified version of Barendregt's cube can be represented by the set of derivation rules used in each system.*

$$\begin{aligned}
\lambda{\to} &= \textit{Base}-\textit{Rules},\\
\lambda\underline{\omega} &= \lambda{\to} \cup \textit{Higher}-\textit{Order},\\
\lambda 2 &= \lambda{\to} \cup \textit{Polymorphism},\\
\lambda\omega &= \lambda{\to} \cup \textit{Higher}-\textit{Order} \cup \textit{Polymorphism},\\
\lambda P &= \lambda{\to} \cup \textit{Dependencies},\\
\lambda P\underline{\omega} &= \lambda{\to} \cup \textit{Dependencies} \cup \textit{Higher}-\textit{Order},\\
\lambda P2 &= \lambda{\to} \cup \textit{Dependencies} \cup \textit{Polymorphism},\\
\lambda P\omega &= \lambda{\to} \cup \textit{Dependencies} \cup \textit{Higher}-\textit{Order} \cup \textit{Polymorphism}.
\end{aligned}$$

For each set of rules $\mathcal{S}$, we write $\Gamma \vdash_{\mathcal{S}} A : B$ to indicate that $\Gamma \vdash_t A : B$ can be derived using only the rules in $\mathcal{S}$. The expression 'system $\mathcal{S}$' refers to the typed system obtained by restricting the full system to allow only the rules in $\mathcal{S}$.

If $\Gamma \vdash_t M : \phi$ for a typed $\lambda$-term $M$, then $\Gamma \vdash_t \phi : *$ (see [Bar92]). In this case we say that $\phi$ is a *type* or, to be more precise, a type with respect to the context $\Gamma$.

*Definition 2.4.4* *i) We write $\mathcal{D}$: $\Gamma \vdash_t A : B$ to express that $\mathcal{D}$ is a derivation for the judgment $\Gamma \vdash_t A : B$.*

*ii) We write $\mathcal{D}' \subseteq \mathcal{D}$ when $\mathcal{D}'$ is a subderivation of $\mathcal{D}$.*

*iii) In what follows we will use the notation:*

$$\mathcal{D}\!: \frac{\mathcal{C}_1 \qquad \cdots \qquad \mathcal{C}_n}{\mathcal{C}}\ (R)$$

*to denote the derivation $\mathcal{D}$, proving the judgment $\mathcal{C}$, that is obtained by applying the rule $(R)$ to the premises $\mathcal{C}_1, \ldots, \mathcal{C}_n$, which are conclusions of some derivations.*

# Chapter 3

# The Cube of Type Assignments Systems $\mathcal{TAS}$

## Introduction

In the previous chapter, we saw that terms of the $\lambda$-calculus can be directly decorated with types. In this *fully typed* approach, every closed term comes directly with a unique, intrinsic type. In this chapter, we discuss another way of giving types to terms of the $\lambda$-calculus: the *type assignment* approach. It was introduced by Curry [Cur34] for the Theory of Combinators, and then modified by Curry [CF58, CHS72] for the $\lambda$-calculus; it, essentially, proves judgments of the shape:

$$\Gamma \vdash M : \phi,$$

where $M$ is a term of the (untyped) $\lambda$-calculus, $\phi$ is a type (i.e. an element of a given set $\mathcal{T}ype$ of types), and $\Gamma$ is a context, assigning types to the free variables of $M$ and $\phi$.

Such a judgment means that we can assign the type $\phi$ to the $\lambda$-term $M$, when types are assigned to the free variables of $M$ and $\phi$ as specified in the context $\Gamma$. In this approach, types are viewed as *predicates*, or *properties*, of terms, and each closed term can be assigned either none or infinitely many types. This approach were called *à la Curry* by Barendregt, and these systems are sometimes called *type assignment systems*. When we look at $\lambda$-calculus as a paradigmatic programming language, this approach corresponds to ML-like languages, where the user can write programs in a completely untyped language, and types are automatically inferred at compile time. The approach can be also intended as the construction of an abstract interpretation of the program, that can be used as a correctness criterion.

In [Cur34, Lei83, GR88], it was observed that some of the type assignment systems already known in the literature can also be obtained from a typed system through an *erasing function* that erases type information from terms in a typed system. In particular, the Curry type assignment system [Cur34] ($F1$) can, in this way, be obtained from $\lambda\!\rightarrow$, the polymorphic type assignment system ($F2$) [Lei83] from $\lambda 2$, and the higher-order type assignment system ($F\omega$) [GR88] from the higher-order $\lambda$-calculus $\lambda\omega$.

For those systems, if $\mathcal{D}$ is a typed derivation, and $\mathcal{E}$ is the above meant erasing function, then by applying $\mathcal{E}$ to the "subject" of every judgment in $\mathcal{D}$, we obtain a valid type assignment derivation with the same structure of the typed one. Vice versa, every type assignment derivation can be viewed as the result of an application of $\mathcal{E}$ to a typed one. In particular, the erasing function $\mathcal{E}$ induces an *isomorphism* between every typed system on the dependency-free side of Barendregt's cube and the corresponding type assignment system.

In [GHR93], the erasing function was extended in a natural way to all typed systems in Barendregt's cube, including the systems with dependent-types, as studied in [Ber88, HHP92]. The essential difference is that the domain of $\mathcal{E}$ was extended to includes types too, since now terms can occur in types. To be precise, if a typed system consists of a set $\mathcal{S}$ of derivation rules, the rules of the corresponding type assignment system can be obtained by applying $\mathcal{E}$ to every object occurring in the rules of $\mathcal{S}$.

This erasing function $\mathcal{E}$ induces a *cube of type assignment systems*. Namely, for every typed system $\mathcal{S}_t$ in Barendregt's cube, there is a corresponding type assignment system $\mathcal{S}_u$, of which the rules are obtained from those of $\mathcal{S}_t$ via the extended erasing function $\mathcal{E}$. Note that, in this setting, if $\Gamma \vdash M : \phi$ is a typed judgment, then the corresponding type assignment judgment is $\mathcal{E}(\Gamma) \vdash \mathcal{E}(M) : \mathcal{E}(\phi)$, where now $\mathcal{E}(\phi)$ can be different from $\phi$ ($\mathcal{E}(\Gamma)$ from $\Gamma$), in case $\phi$ is a dependent-type ($\Gamma$ contains dependent-types).

The fact that in [GHR93] also systems that contain dependences were considered, was a first attempt to study dependent-types in a type assignment approach. In that paper, and also in [PM89], was proved that the introduction of dependences does not increase the expressiveness of a system, i.e., the terms typable in a type assignment system with dependences are all nothing but those typable in the similar system, obtained from the first by erasing the dependences.

The above mentioned erasing function $\mathcal{E}$, at least for the dependency-free plane of $\mathcal{TS}$ and $\mathcal{TAS}$, induces an isomorphism between derivations in corresponding systems. More precisely, if $\mathcal{D}$ is a derivation in a typed system, by applying $\mathcal{E}$ to every object (i.e. term, constructor, or kind) in $\mathcal{D}$, a valid derivation in the corresponding type assignment system is obtained. Vice-versa, again only for dependency-free systems, every type assignment derivation can be obtained by applying $\mathcal{E}$ to a typed one.

The 'formulae-as-types' principle [How80] can be extended to the above type assignment systems as follows:

*Given an untyped term $M$, if we can assign a type $\phi$ in the type assignment systems $F1$(respectively $F2$, and $F\omega$), with a derivation $\mathcal{D}$, then:*

*i) $\mathcal{D}$ can be interpreted as the coding of a proof for the logic formulas $\varphi$ which corresponds to the interpretation of the type $\phi$ assigned to $M$.*

*ii) $M$ can be interpreted as the coding of a "logical proof schema", whose instances (of the schema) prove, respectively, all the logic formulas $\varphi_i$'s that can be interpreted as the types $\phi_i$'s that can be assigned to $M$.*

Clearly, the fact that the classes of derivations for typed and a type assignment systems are isomorphic means that they have the same underlying logical system.

In this chapter, we show an example of a inhabited type in $\mathcal{TAS}$, that cannot be obtained through erasure of an inhabited type in $\mathcal{TS}$. This negative result of course implies that the logical sides of these two cubes are different; however, this difference only shows up in the plane of the cube with dependences, where already $\mathcal{TS}$ has lost a clear connection with logic [Ber88].

Furthermore, it is also our opinion that there is more to types than just logic: studying types is not solely justifiable through the connection between types and logic, as is clearly shown by, for example, the type system developed for ML that models type-constants and recursion [Mil78], and the intersection-type discipline [BCD83]. In our view, the main motivation for $\mathcal{TAS}$ comes from the ML-style of approaching types: to have type-free code with type assignment seen as a correctness criterion, or safety means, but always outside of programs rather than built in. Certainly, in order to be correctly applied in this way, a type assignment system must enjoy some fundamental properties, like the Church-Rosser property, the subject-reduction property and normalization. We prove these properties for all systems in $\mathcal{TAS}$. So, $\mathcal{TAS}$ can make sense even if it does not fit the corresponding $\mathcal{TS}$: it is just another way to select legitimate code. Studying type systems with dependences can be of value from the point of view of abstract interpretation; such type assignment system could introduce a more refined notion of types in a programming language setting. For example, since the version of $F1$ with dependences is decidable, and the core of the type system for ML is based on $F1$, designing a version of ML with dependent-types seems feasible.

This chapter is organized as follows: Section *3.1* contains the stratified presentation of the cube of type assignment systems. Starting from the typed stratified cube, we will define an erasing function $\mathcal{E}$ and, using this function, obtain the related cube of type assignment systems. The same approach can be found in [GHR93]. In Section *3.2*, the properties of the type assignment systems belonging to this cube are studied; Sections *3.3*, and *3.4* contain the proofs of the Church-Rosser property, and of the subject reduction, and finally Section *3.5* contains the proof of strong normalization.

## 3.1 The Cube of Type Assignment Systems

In this section, we will present the cube of type assignment system as was first presented in [GHR93]. The definition of the type assignment cube is based on the definition of the type-erasing function $\mathcal{E}$, to be defined below, that erases all type information in typed $\lambda$-terms. In fact, both the syntax of terms, and the rules of the type assignment systems in the cube are obtained directly from the corresponding syntax and rules of the typed systems in Barendregt's cube, by applying $\mathcal{E}$. Note that, since both constructors and kinds can depend on $\lambda$-terms, $\mathcal{E}$ can modify all objects. Since $\lambda$-terms do not contain type information, we cannot create $\lambda$-terms dependent on types.

From now on, we will reserve the name *typed systems* ($\mathcal{TS}$) for the systems of Barendregt's cube, and we reserve the expression *type assignment systems* ($\mathcal{TAS}$) for the systems to be defined below.

As already mentioned in the introduction, for the plane of the $\mathcal{TS}$-cube without dependences there exists a function that, erasing type information from typed $\lambda$-terms, allows to switch from a typed system to a corresponding type assignment system. To be precise, it erases type information from $\lambda$-bindings occurring in $\lambda$-terms, while leaving all type information that decorates bindings in constructors and kinds intact. In [GHR93], a more general function $\mathcal{E}$ was defined, by extending the domain of the above function to terms with dependences in a natural way, as shown in the Definition *3.1.1*.

*Definition 3.1.1* *i)* $\{*, \Box\}$ *is the set of* sorts*.*

*ii)* *The sets of* $\lambda$-terms *(*$\Lambda$*),* constructors *(Cons), and* kinds *(Kind) are mutually defined by the following grammar, where* $M, \phi$ *and* $K$*, are metavariables for* $\lambda$*-terms, constructors and kinds respectively:*

$$M ::= x \mid \lambda x.M \mid MM$$

$$\phi ::= \alpha \mid \Pi x{:}\phi.\phi \mid \Pi\alpha{:}K.\phi \mid \lambda x{:}\phi.\phi \mid \lambda\alpha{:}K.\phi \mid \phi\phi \mid \phi M$$

$$K ::= * \mid \Pi x{:}\phi.K \mid \Pi\alpha{:}K.K$$

*The set* $T_u$ *of* terms *is the union of the sets* $\Lambda$*, Cons and Kind.*

The elimination of type information from a typed $\lambda$-term yields an untyped $\lambda$-term. Both the syntax and the rules of the $\mathcal{TAS}$ cube can be obtained directly from the corresponding rules of the $\mathcal{TS}$ cube by applying the following type-erasing function $\mathcal{E}$.

*Definition 3.1.2* *The* erasing function $\mathcal{E} : T_t \to T_u$ *is inductively defined as follows:*

*i) On* $\Lambda_t$*.*

$$\begin{array}{rcl}
\mathcal{E}(x) &=& x, \\
\mathcal{E}(MN) &=& \mathcal{E}(M)\mathcal{E}(N), \\
\mathcal{E}(M\phi) &=& \mathcal{E}(M), \\
\mathcal{E}(\lambda x{:}\phi.M) &=& \lambda x.\mathcal{E}(M), \\
\mathcal{E}(\lambda\alpha{:}K.M) &=& \mathcal{E}(M).
\end{array}$$

*ii) On $Cons_t$.*

$$\begin{array}{rcl}
\mathcal{E}(\alpha) &=& \alpha, \\
\mathcal{E}(\Pi x{:}\phi.\psi) &=& \Pi x{:}\mathcal{E}(\phi).\mathcal{E}(\psi), \\
\mathcal{E}(\Pi\alpha{:}K.\psi) &=& \Pi\alpha{:}\mathcal{E}(K).\mathcal{E}(\psi), \\
\mathcal{E}(\lambda x{:}\phi.\psi) &=& \lambda x{:}\mathcal{E}(\phi).\mathcal{E}(\psi), \\
\mathcal{E}(\lambda\alpha{:}K.\psi) &=& \lambda\alpha{:}\mathcal{E}(K).\mathcal{E}(\psi), \\
\mathcal{E}(\phi\psi) &=& \mathcal{E}(\phi)\mathcal{E}(\psi), \\
\mathcal{E}(\phi M) &=& \mathcal{E}(\phi)\mathcal{E}(M).
\end{array}$$

*iii) On $Kind_t$.*

$$\begin{array}{rcl}
\mathcal{E}(*) &=& *, \\
\mathcal{E}(\Pi x{:}\phi.K) &=& \Pi x{:}\mathcal{E}(\phi).\mathcal{E}(K), \\
\mathcal{E}(\Pi\alpha{:}K_1.K_2) &=& \Pi\alpha{:}\mathcal{E}(K_1).\mathcal{E}(K_2).
\end{array}$$

The erasing function is extended to contexts in the obvious way and we use the notation $\mathcal{E}(\Gamma)$. Note that the behaviour of $\mathcal{E}$ is such that, in the image of $\mathcal{E}$, $\lambda$-terms are completely untyped, while constructors and kinds are 'partially' typed. The notions of free variables, subterms and $\beta$-reduction are similar to their 'fully typed' counterparts, but slightly modified, according to the untyped term syntax.

*Definition 3.1.3* *i) The set of* free variables of $A$ *is defined as in Definition 2.1.2, extended with:*

$$\mathcal{FV}(\lambda x.M) = \mathcal{FV}(M) \setminus \{x\}.$$

*ii) Substitution for $\lambda$-terms is defined as in Definition 2.1.3, extended with:*

$$(\lambda x.M)^{\vartheta} \equiv \lambda x.M^{\vartheta}.$$

*iii) The set of* subterms *of $A$ is defined as in Definition 2.1.5, extended with:*

$$\mathcal{ST}(\lambda x.M) = \{\lambda x.M\} \cup \mathcal{ST}(M).$$

The 'untyped variant' of Property *2.1.12* also holds.

*Lemma 3.1.4* $A[B/b][C/c] \equiv A[C/c][B[C/c]/b]$, *provided* $b \notin \mathcal{FV}(C)$.

*Proof:* By induction on the definition of substitution.

$[A \equiv a$ and $a \not\equiv b$ and $a \not\equiv c]$: then $a[B/b][C/c] \equiv a \equiv a[C/c][B[C/c]/b]$.

$[A \equiv b]$: then $b[B/b][C/c] \equiv B[C/c] \equiv b[B[C/c]/b] \equiv b[C/c][B[C/c]/b]$.

$[A \equiv c]$: then $c[B/b][C/c] \equiv C \equiv c[C/c][B[C/c]/b]$.

$[A \equiv A_1A_2]$: then $(A_1A_2)[B/b][C/c] \equiv A_1[B/b][C/c]A_2[B/b][C/c] \equiv (IH)$ $A_1[C/c][B[C/c]/b]A_2[C/c][B[C/c]/b] \equiv (A_1A_2)[C/c][B[C/c]/b]$.

$[A \equiv \lambda a{:}A_1.A_2]$: then $(\lambda a{:}A_1.A_2)[B/b][C/c] \equiv \lambda a{:}A_1[B/b][C/c].A_2[B/b][C/c] \equiv (IH)$ $\lambda a{:}A_1[C/c][B[C/c]/b].A_2[C/c][B[C/c]/b] \equiv (\lambda a{:}A_1.A_2)[C/c][B[C/c]/b]$.

$[A \equiv \lambda x.M]$: then
$(\lambda x.M)[B/b][C/c] \equiv \lambda x.M[B/b][C/c] \equiv (IH)\lambda x.M[C/c][B[C/c]/b] \equiv$ $(\lambda a.A_1)[C/c][B[C/c]/b]$.

$[A \equiv \Pi a{:}A_1.A_2]$: then $(\Pi a{:}A_1.A_2)[B/b][C/c] \equiv \Pi a{:}A_1[B/b][C/c].A_2[B/b][C/c] \equiv (IH)$ $\Pi a{:}A_1[C/c]\ [B[C/c]/b].A_2[C/c][B[C/c]/b] \equiv (\Pi a{:}A_1.A_2)[C/c][B[C/c]/b]$. ∎

*Definition 3.1.5* ($\beta$-reduction, $\beta$-conversion) *i) Let:*

$$\begin{aligned}(\lambda x{:}\phi.\psi)M &\to_\beta \psi[M/x],\\ (\lambda\alpha{:}K.\phi)\psi &\to_\beta \phi[\psi/\alpha],\\ (\lambda x.M)N &\to_\beta M[N/x].\end{aligned}$$

*The* $\beta$*-reduction* $\to_\beta$ *is defined as the contextual closure of these rules, i.e.:*

$$\begin{aligned}A\to_\beta A' &\Rightarrow AB\to_\beta A'B\\ A\to_\beta A' &\Rightarrow BA\to_\beta BA'\\ M\to_\beta M' &\Rightarrow \lambda x.M\to_\beta\lambda x.M'\\ A\to_\beta A' &\Rightarrow \lambda a{:}A.B\to_\beta\lambda a{:}A'.B\\ A\to_\beta A' &\Rightarrow \lambda b{:}B.A\to_\beta\lambda b{:}B.A'\\ A\to_\beta A' &\Rightarrow \Pi a{:}A.B\to_\beta\Pi a{:}A'.B\\ A\to_\beta A' &\Rightarrow \Pi b{:}B.A\to_\beta\Pi b{:}B.A'.\end{aligned}$$

*ii)* $\beta$*-reduction* $\twoheadrightarrow_\beta$ *is the reflexive, transitive closure of* $\to_\beta$*, i.e.:*

$$
\begin{array}{rcl}
 & & A \twoheadrightarrow_\beta A \\
A \rightarrow_\beta A' & \Rightarrow & A \twoheadrightarrow_\beta A' \\
A \twoheadrightarrow_\beta A' \ \& \ A' \twoheadrightarrow_\beta A'' & \Rightarrow & A \twoheadrightarrow_\beta A''
\end{array}
$$

*iii) $\beta$-conversion $=_\beta$ is the minimal equivalence relation generated by $\twoheadrightarrow_\beta$, i.e.:*

$$
\begin{array}{rcl}
A \twoheadrightarrow_\beta A' & \Rightarrow & A =_\beta A' \\
A =_\beta A' & \Rightarrow & A' =_\beta A \\
A =_\beta A' \ \& \ A' =_\beta A'' & \Rightarrow & A =_\beta A''
\end{array}
$$

In the next lemma, we show that $\beta$-conversion and substitution behave well together.

*Lemma 3.1.6 If $A =_\beta B$, then $A[C/c] =_\beta B[C/c]$.*

*Proof:* By induction on the definition of $=_\beta$. We just consider the case when $A$ is a redex, by induction on $\twoheadrightarrow_\beta$; the complete proof follows easily by induction.

$[A \equiv \lambda x.M$ and $B \equiv M[N/x]]$: then $((\lambda x.M)N)[C/c] \equiv (\lambda x.M)[C/c]N[C/c] \equiv (\lambda x.M[C/c])N[C/c] \rightarrow_\beta M[C/c][N[C/c]/x]$ which is equal, by Lemma *3.1.4*, to $M[N/x][C/c]$.

$[A \equiv (\lambda x{:}\phi.\psi)M$ and $B \equiv \psi[M/x]]$: then $((\lambda x{:}\phi.\psi)M)[C/c] \equiv (\lambda x{:}\phi.\psi)[C/c]M[C/c] \equiv (\lambda x{:}\phi[C/c].\psi[C/c])M[C/c] \rightarrow_\beta \psi[C/c][M[C/c]/x]$ which is equal, by Lemma *3.1.4*, to $\psi[M/x][C/c]$.

$[A \equiv (\lambda\alpha{:}K.\phi)\psi$ and $B \equiv \phi[\psi/\alpha]]$: then $((\lambda\alpha{:}K.\phi)\psi)[C/c] \equiv (\lambda\alpha{:}K.\phi)[C/c]\psi[C/c] \equiv (\lambda\alpha{:}K[C/c].\phi[C/c])\psi[C/c] \rightarrow_\beta \phi[C/c][\psi[C/c]/\alpha]$ which is equal, by Lemma *3.1.4*, to $\phi[\psi/\alpha][C/c]$. ∎

The notion of statement and context are defined as in Definition *2.1.6*, and *2.1.7*, taking into account the differences in syntax. Given the difference in syntax, the type assignment judgments and rules as presented in Definition *3.1.7* are only in appearance similar to those of Definition *2.4.3*. Note that the denotation of a rule is only different for the rules $(I)$, $(I_K)$ and $(E_K)$. We will, therefore, take the liberty of using the same notation and names for rules.

*Definition 3.1.7* (General Type Assignment System ) *i) The following rules are used to derive* judgments *of the form*

$$\Gamma \vdash A : B,$$

*where* $\Gamma$ *is a context and* $A : B$ *is a statement.*

*ii) The General Type Assignment System set of rules, can be divided in four groups, depending of the subjects of the statements:*

*Common Rules*

$$(\mathit{Proj})\quad \frac{\Gamma \vdash A : s \quad a \notin \mathcal{D}om\,(\Gamma)}{\Gamma, a{:}A \vdash a : A} \qquad (\mathit{Conv})\quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad B =_\beta C}{\Gamma \vdash A : C}$$

$$(\mathit{Weak})\quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad c \notin \mathcal{D}om\,(\Gamma)}{\Gamma, c{:}C \vdash A : B}$$

*$\lambda$-Term Rules*

$$(I)\quad \frac{\Gamma, x{:}\phi \vdash M : \psi}{\Gamma \vdash \lambda x.M : \Pi x{:}\phi.\psi} \qquad (E)\quad \frac{\Gamma \vdash M : \Pi x{:}\phi.\psi \quad \Gamma \vdash N : \phi}{\Gamma \vdash MN : \psi[N/x]}$$

$$(I_K)\quad \frac{\Gamma, \alpha{:}K \vdash M : \phi}{\Gamma \vdash M : \Pi\alpha{:}K.\phi} \qquad (E_K)\quad \frac{\Gamma \vdash M : \Pi\alpha{:}K.\phi \quad \Gamma \vdash \psi : K}{\Gamma \vdash M : \phi[\psi/\alpha]}$$

*Constructor Rules*

$$(C\text{-}I_C)\quad \frac{\Gamma, x{:}\phi \vdash \psi : K}{\Gamma \vdash \lambda x{:}\phi.\psi : \Pi x{:}\phi.K} \qquad (C\text{-}E_C)\quad \frac{\Gamma \vdash \psi : \Pi x{:}\phi.K \quad \Gamma \vdash M : \phi}{\Gamma \vdash \psi M : K[M/x]}$$

$$(C\text{-}I_K)\quad \frac{\Gamma, \alpha{:}K_1 \vdash \psi : K_2}{\Gamma \vdash \lambda\alpha{:}K_1.\psi : \Pi\alpha{:}K_1.K_2} \qquad (C\text{-}E_K)\quad \frac{\Gamma \vdash \phi : \Pi\alpha{:}K_1.K_2 \quad \Gamma \vdash \psi : K_1}{\Gamma \vdash \phi\psi : K_2[\psi/\alpha]}$$

$$(C\text{-}F_C)\quad \frac{\Gamma, x{:}\phi \vdash \psi : *}{\Gamma \vdash \Pi x{:}\phi.\psi : *} \qquad (C\text{-}F_K)\quad \frac{\Gamma, \alpha{:}K \vdash \phi : *}{\Gamma \vdash \Pi\alpha{:}K.\phi : *}$$

*Kind Rules*

$$(\mathit{Axiom})\quad \frac{}{\varepsilon \vdash * : \Box} \qquad (K\text{-}F_C)\quad \frac{\Gamma, x{:}\phi \vdash K : \Box}{\Gamma \vdash \Pi x{:}\phi.K : \Box}$$

$$(K\text{-}F_K)\quad \frac{\Gamma, \alpha{:}K_1 \vdash K_2 : \Box}{\Gamma \vdash \Pi\alpha{:}K_1.K_2 : \Box}$$

*iii) Let the following sets of rules be defined as in Definition 2.4.3 (iii), i.e.:*

$$\begin{aligned}
\mathit{Base\text{-}Rules} &= \{(\mathit{Axiom}), (\mathit{Proj}), (\mathit{Weak}), (I), (E), (C\text{-}F_C)\},\\
\mathit{Polymorphism} &= \{(I_K), (E_K), (C\text{-}F_K)\},\\
\mathit{Dependencies} &= \{(C\text{-}I_C), (C\text{-}E_C), (K\text{-}F_C), (\mathit{Conv})\},\\
\mathit{Higher\text{-}Order} &= \{(C\text{-}I_K), (C\text{-}E_K), (K\text{-}F_K), (\mathit{Conv})\}.
\end{aligned}$$

*Notice that, unlike for the derivation rules of Definition 2.4.3 (ii), the subject does not change in the type assignment rules ($I_K$) and ($E_K$). These two, together with the rules (Weak) and (Conv), are called the* not syntax-directed rules.

iv) *The eight type assignment systems can be distinguished, as in Definition 2.4.3 (iv), by the set of derivation rules used in each system.*

$$
\begin{aligned}
F1 &= \text{Base}-\text{Rules}, \\
F\underline{\omega} &= F1 \cup \mathit{Higher}-\mathit{Order}, \\
F2 &= F1 \cup \mathit{Polymorphism}, \\
F\omega &= F1 \cup \mathit{Higher}-\mathit{Order} \cup \mathit{Polymorphism}, \\
DF1 &= F1 \cup \mathit{Dependencies}, \\
DF\underline{\omega} &= F1 \cup \mathit{Dependencies} \cup \mathit{Higher}-\mathit{Order}, \\
DF2 &= F1 \cup \mathit{Dependencies} \cup \mathit{Polymorphism}, \\
DF\omega &= F1 \cup \mathit{Dependencies} \cup \mathit{Higher}-\mathit{Order} \cup \mathit{Polymorphism}.
\end{aligned}
$$

Like for $\mathcal{TS}$ we will write, for each set of rules $\mathcal{S}$, $\Gamma \vdash_{\mathcal{S}} A : B$ to indicate that $\Gamma \vdash A : B$ can be derived using only the rules in $\mathcal{S}$. Then, Figure *3.1* shows the eight type assignment systems arranged as vertices of a cube (the $\mathcal{TAS}$ cube). Also in this cube the edges are intended as an inclusion relation between systems.

The notion of derivation and subderivation for a judgment are the same as in Definition *2.4.4*, and an analogue of Property *2.1.14* also holds:

*Lemma 3.1.8 The General Type Assignment System derives judgments of the following shapes:*

$$\Gamma \vdash M : \phi, \qquad \Gamma \vdash \phi : K, \qquad \mathit{or} \qquad \Gamma \vdash K : \Box.$$

*Proof:* By looking at the rules and by observing that the sets $\mathit{Cons}$, and $\mathit{Kind}$ are closed for the substitution of $\lambda$-term-variables by terms, and constructor-variables by constructors. ∎

As before, a *type* is a constructor of kind $*$ (and this is again a context-dependent property). A $\lambda$-term $M$ is *typable* if there are a context $\Gamma$ and a constructor $\phi$, such that $\Gamma \vdash M : \phi$ (we will prove in Section *3.2* that $\phi$ must be a type).

$$\begin{tikzcd}
 & F\omega \arrow[rr] & & DF\omega \\
F2 \arrow[ur] \arrow[rr] & & DF2 \arrow[ur] & \\
 & F\underline{\omega} \arrow[uu] \arrow[rr] & & DF\underline{\omega} \arrow[uu] \\
F1 \arrow[uu] \arrow[ur] \arrow[rr] & & DF1 \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 3.1: The Cube of the Type Assignment Systems

*Remark 3.1.9* Notice that, in the left-hand side of the cube, the constructors coincide with the typed ones, because there they cannot depend on $\lambda$-terms. This no longer holds in the right-hand side: here we can build constructors like

$$(\lambda x{:}\phi.\psi)N,$$

where $N$ is a pure, untyped $\lambda$-term.

The system $F1$ corresponds to the well-known Curry type assignment system, whereas $F2$ is the type assignment version of the second order $\lambda$-calculus. The three dimensions in this cube of type assignment systems correspond, as for Barendregt's cube, to the introduction of *polymorphic-types*, *higher-order types* and *dependent-types*. The systems $DF1$, $DF\underline{\omega}$, $DF2$, and $DF\omega$ represent the first attempt to define type assignment systems with *dependent-types*.

## 3.2 Basic Properties of $\mathcal{TAS}$

In this section, we will prove that all systems in the $\mathcal{TAS}$ cube satisfy good computational properties, like the Church-Rosser property, the subject reduction property, and strong normalization of typable terms. To prove these results, we need more definitions and technical lemmas, stating properties of the systems that are also of independent interest.

The following proposition states that every term, typable by $*$ or $\Box$, cannot be typable by both, and guarantees consistency of the system. This property holds for the, so called, *functional pure type systems* [Bar91].

*Proposition 3.2.1 If* $\Gamma \vdash A : s_1$ *and* $\Gamma \vdash A : s_2$, *then* $s_1 \equiv s_2$.

*Proof:* This is an obvious consequence of Lemma *3.1.8*. ■

*Definition 3.2.2 A* legal context *is inductively defined as follows:*

*i) The empty context* $\varepsilon$ *is legal;*

*ii)* $\Gamma, a{:}A$ *is legal if and only if* $\Gamma$ *is legal,* $\Gamma \vdash A : s$, *and* $a \notin \mathcal{D}om\,(\Gamma)$.

In the next lemma, we show that every context in a derivable judgment is also legal.

*Lemma 3.2.3 If* $\Gamma \vdash A : B$, *then* $\Gamma$ *is legal.*

*Proof:* By induction on derivations.

[(*Proj*), (*Weak*)]: let $\Gamma = \Gamma', c{:}C$, for some $\Gamma', c$, and $C$. Then the conclusion is $\Gamma', c{:}C \vdash A : B$. From the assumptions we get

$$\Gamma' \vdash A : B, \quad \text{and} \quad \Gamma' \vdash C : s, \quad \text{and} \quad c \notin \mathcal{D}om\,(\Gamma').$$

By induction $\Gamma'$ is legal, and hence, by Definition *3.2.2*, $\Gamma', c{:}C$ is legal.

[(*Conv*), (Elimination rules)]: by induction.

[(Introduction, Product rules)]: then the conclusion is either

$$\Gamma \vdash \lambda c{:}C.D : \Pi c{:}C.E, \text{ or } \Gamma \vdash \lambda x.M : \Pi c{:}C.E, \text{ or } \Gamma \vdash \Pi c{:}C.D : s,$$

for some $x, c, C, D, E, M$, and $s$. Take, as example, the first case: from the assumption we get

$$\Gamma, c{:}C \vdash D : E.$$

Then, by induction $\Gamma, c{:}C$ is legal, and, by Definition *3.2.2*, $\Gamma$ is legal. ■

We define the following relations on legal contexts:

*Definition 3.2.4* *i)* $\Gamma \sqsubseteq \Gamma'$ *if and only if* $\Gamma$ *is a prefix of* $\Gamma'$.

*ii) The relation* $\dot{\sqsubseteq}$ *is inductively defined as*

*a)* $\varepsilon \mathrel{\dot{\sqsubseteq}} \Gamma$;

*b) If* $\Gamma \mathrel{\dot{\sqsubseteq}} \Gamma'$, *then* $\Gamma, a{:}A \mathrel{\dot{\sqsubseteq}} \Gamma', a{:}A$;

*c) If* $\Gamma \mathrel{\dot{\sqsubseteq}} \Gamma'$, *then* $\Gamma \mathrel{\dot{\sqsubseteq}} \Gamma', a{:}A$.

For these relations, the following lemma holds.

*Lemma 3.2.5* *i) If* $\Gamma_1 \sqsubseteq \Gamma_2$, *then* $\Gamma_1 \mathrel{\dot{\sqsubseteq}} \Gamma_2$.

*ii) If* $\Gamma_1 \mathrel{\dot{\sqsubseteq}} \Gamma_2$ *then* $\mathcal{D}om\,(\Gamma_1) \subseteq \mathcal{D}om\,(\Gamma_2)$.

*iii) If* $\Gamma_1, a{:}A, \Gamma_2 \mathrel{\dot{\sqsubseteq}} \Gamma_3, a{:}A$ *then* $\Gamma_2 = \varepsilon$.

*iv) If* $\Gamma_1, a{:}A, \Gamma_2 \mathrel{\dot{\sqsubseteq}} \Gamma_3, a{:}A, \Gamma_4$ *then* $\Gamma_1 \mathrel{\dot{\sqsubseteq}} \Gamma_3$, *and* $\mathcal{D}om\,(\Gamma_2) \subseteq \mathcal{D}om\,(\Gamma_4)$.

*Proof:* Easy, using Definition *3.2.4 (ii)*. ■

The next lemma states that all statements of a context occurring in a judgment are obtained by proper subderivations.

*Lemma 3.2.6* *If* $\mathcal{D} : \Gamma, c{:}C \vdash A : B$, *then* $\mathcal{D}' : \Gamma \vdash C : s$, *and* $\mathcal{D}' \subseteq \mathcal{D}$.

*Proof:* By easy induction on derivations. ■

The next lemma states that the free variables a derivable the statement are declared in the context of the judgment, and that if a statement is derivable, then it is also derivable in a bigger context.

*Lemma 3.2.7* *If* $\Gamma \mathrel{\dot{\sqsubseteq}} \Gamma'$ *and* $\Gamma \vdash A : B$, *then*

*i)* (Free Variable) $\mathcal{FV}(A) \cup \mathcal{FV}(B) \subseteq \mathcal{D}om\,(\Gamma)$.

*ii)* (Thinning) $\Gamma' \vdash A : B$.

*Proof:* *i)* By induction on derivations.

[(*Weak*), (*Proj*)]: by induction.

[(Introduction, Product rules)]: take for instance the $(C\text{-}I_K)$ rule:

$$\frac{\mathcal{D} : \Gamma, \alpha{:}K_1 \vdash \psi : K_2}{\Gamma \vdash \lambda\alpha{:}K_1.\psi : \Pi\alpha{:}K_1.K_2.}\ (C\text{-}I_K)$$

By induction, $\mathcal{FV}(\psi) \cup \mathcal{FV}(K_2) \subseteq \mathcal{D}om\,(\Gamma, \alpha{:}K_1) = \mathcal{D}om\,(\Gamma) \cup \{\alpha\}$. By Lemma *3.2.6*, we find $\mathcal{D}' \subseteq \mathcal{D}$ such that $\mathcal{D}' : \Gamma \vdash K_1 : s$, so, by induction, $\mathcal{FV}(K_1) \subseteq \mathcal{D}om\,(\Gamma)$. Since:

$$\mathcal{FV}(\lambda\alpha{:}K_1.\psi) \cup \mathcal{FV}(\Pi\alpha{:}K_1.K_2) = \mathcal{FV}(K_1) \cup (\mathcal{FV}(\psi) \cup \mathcal{FV}(K_2)) \setminus \{\alpha\},$$

the thesis follows.

[(Elimination rules), (*Conv*)]: take for instance the $(E)$ rule:

$$\frac{\Gamma \vdash M : \Pi x{:}\phi.\psi \quad \Gamma \vdash N : \phi}{\Gamma \vdash MN : \psi[N/x].}\ (E)$$

By induction, if $c \in \mathcal{FV}(M) \cup \mathcal{FV}(\Pi x{:}\phi.\psi)$, then $c \in \mathcal{D}om\,(\Gamma)$, and if $c \in \mathcal{FV}(N) \cup \mathcal{FV}(\phi)$, then $c \in \mathcal{D}om\,(\Gamma)$. Observe that:

$$\mathcal{FV}(\psi[N/x]) = \mathcal{FV}(\psi) \setminus \{x\} \cup \mathcal{FV}(N),$$

and the thesis follows.

*ii)* By induction on derivations.

[(*Proj*),(*Weak*)]: take for instance the (*Proj*) rule:

$$\frac{\Gamma_1 \vdash B : s \quad b \notin \mathcal{D}om\,(\Gamma_1)}{\Gamma_1, b{:}B \vdash b : B,}\ (Proj)$$

for some $\Gamma_1$. Then, by Lemma *3.2.5 (ii)*, $\mathcal{D}om\,(\Gamma_1, b{:}B) \subseteq \mathcal{D}om\,(\Gamma')$, i.e., $\Gamma' = \Gamma_1', b{:}B, \Gamma_2'$, for suitable $\Gamma_1'$, $\Gamma_2'$. By Lemma *3.2.5 (iv)*, $\Gamma_1 \sqsubseteq\!\!\!\cdot\ \Gamma_1'$, and, by induction, $\Gamma_1' \vdash B : s$. Since $\Gamma'$ is legal, also $b \notin \mathcal{D}om\,(\Gamma_1')$. Apply a (*Proj*) rule to obtain $\Gamma_1', b{:}B \vdash b : B$. Again by the legality of $\Gamma'$, it follows that for all $c{:}C \in \Gamma_2'$, there exists $\Gamma_2'' \sqsubseteq \Gamma_2'$, such that $\Gamma_1', b{:}B, \Gamma_2'' \vdash C : s$. So, we can apply a (*Weak*) rule a suitable number of times, to get $\Gamma_1, b{:}B, \Gamma_2' \vdash b : B$, as desired.

[(Introduction, Product rules)]: take for instance the $(C\text{-}I_C)$ rule:

$$\frac{\Gamma, x{:}\phi \vdash \psi : K}{\Gamma \vdash \lambda x{:}\phi.\psi : \Pi x{:}\phi.K.}\ (C\text{-}I_C)$$

By induction, for every $\Gamma'$, such that $\Gamma, x{:}\phi \sqsubseteq \Gamma'$, we have $\Gamma' \vdash \psi : K$. By Lemma *3.2.5 (ii)*, we have that $\Gamma' = \Gamma'_1, x{:}\phi, \Gamma'_2$, for suitable $\Gamma'_1$, and $\Gamma'_2$. By Lemma *3.2.5 (iv)*, $\Gamma \sqsubseteq \Gamma'_1$, so $\Gamma \sqsubseteq \Gamma'_1, x{:}\phi$. By induction, $\Gamma'_1, x{:}\phi \vdash \psi : K$. Apply a $(C\text{-}I_C)$ rule to obtain $\Gamma'_1 \vdash \lambda x{:}\phi.\psi : \Pi x{:}\phi.K$. By the legality of $\Gamma'$ (using the same reasoning as in the (*Proj*) case), apply a (*Weak*) rule a suitable number of times.

[(Elimination rules), (*Conv*)]: by induction. ∎

The following relation is introduced to abbreviate a sequence of derivation rules $(I_K)$ and $(E_K)$, that together correspond to polymorphism, and also takes the presence of rule (*Conv*) into account. It will be of use in Lemma *3.2.9* and in Theorem *3.4.4*.

*Definition 3.2.8 We define the relation $\preceq$ on constructors inductively by:*

$$\begin{aligned}
\phi =_\beta \psi \ &\Rightarrow\ \phi \preceq \psi, \\
\phi \ &\preceq\ \Pi\alpha{:}K.\phi, \\
\Pi\alpha{:}K.\phi \ &\preceq\ \phi[\xi/\alpha], \\
\phi \preceq \psi \preceq \xi \ &\Rightarrow\ \phi \preceq \xi.
\end{aligned}$$

The four cases in Definition *3.2.8* reflect, respectively, an application of rule (*Conv*), $(I_K)$, or $(E_K)$, and a sequence of not syntax-directed rules.

Worth noticing is the second case of Definition *3.2.8*, because it illustrates an important difference between the original presentation of the polymorphic type assignment system [Lei83], and our presentation as a system in the topology of the $\mathcal{TAS}$ cube. The equivalent rule for $(I_K)$ in the polymorphic type assignment system is:

$$\frac{\Gamma \vdash M : \phi \qquad \alpha \notin \mathcal{FV}(\Gamma)}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\phi}\ (\forall{-}Intro)$$

The type $\forall\alpha.\sigma$ is essentially the constructor $\Pi\alpha{:}{*}.\sigma$, and $\alpha \notin \mathcal{FV}(\Gamma)$ is a side-condition, indicating that binding of the type-variable $\alpha$ is only allowed when $\alpha$ does not occur free in any predicate belonging to the context. The polymorphic type assignment system needs this side-condition to avoid to assign, for example,

$$x{:}\alpha \vdash x : \forall\alpha.\alpha.$$

The $\mathcal{TAS}$ presentation of this system does not require this extra condition on the derivation rule $(I_K)$: in fact, types are generated by the system itself, using only legal contexts, which are essentially linear ordered sets of declarations, in the derivations. In these systems, it is impossible to apply a $(I_K)$ rule to the derivation for

$$\alpha{:}*, x{:}\alpha \ \vdash x : \alpha,$$

because $\alpha{:}*$ is not the right-most declaration in the context; when $\Gamma, \alpha{:}* \vdash M : \phi$, then, by legality of the context, $\alpha$ does not occur in $\Gamma$, so especially does not occur free in $\Gamma$. We can say that the extra condition on $(I_K)$ is *hidden* in the definition of legal context.

*Lemma 3.2.9* (Generation for $\lambda$-terms) *i) If $\Gamma \vdash x : \xi$, then there is $\xi'$, such that $x{:}\xi' \in \Gamma$ and $\xi' \preceq \xi$.*

*ii) If $\mathcal{D} : \Gamma \vdash \lambda x.M : \xi$, then there are $\Gamma'$, $\phi$, $\psi$, and $\mathcal{D}' \subseteq \mathcal{D}$, such that $\Pi x{:}\phi.\psi \preceq \xi$ and*

$$\mathcal{D}' : \frac{\Gamma', x{:}\phi \vdash M : \psi}{\Gamma' \vdash \lambda x.M : \Pi x{:}\phi.\psi.}\ (I)$$

*iii) If $\mathcal{D} : \Gamma \vdash MN : \xi$, then there are $\Gamma'$, $\phi$, $\psi$, and $\mathcal{D}' \subseteq \mathcal{D}$, such that $\psi[N/x] \preceq \xi$ and*

$$\mathcal{D}' : \frac{\Gamma' \vdash M : \Pi x{:}\phi.\psi \quad \Gamma' \vdash N : \phi}{\Gamma' \vdash MN : \psi[N/x].}\ (E)$$

*Proof:* By induction on derivations.

*i)* [(*Proj*)]: then the conclusion is $\Gamma',x{:}\xi \ \vdash x : \xi$, for some $\Gamma'$. Take $\xi' \equiv \xi$.

[(*Weak*)]: let $\Gamma = \Gamma',a{:}A$, for some $\Gamma'$, $a$, and $A$. Then the conclusion is $\Gamma',a{:}A \vdash x : \xi$. By induction, there exists $\xi'$, such that $x{:}\xi' \in \Gamma'$, and $\xi' \preceq \xi$. Since $\Gamma' \sqsubseteq \Gamma$, by Definition *3.2.5 (ii)* also $x{:}\xi' \in \Gamma$.

[(*Conv*)]: then one premise is $\Gamma \vdash x : \xi''$, for some $\xi''$, and, by Definition *3.2.8*, $\xi'' \preceq \xi$. By induction, there exists $\xi'$ such that $x{:}\xi' \in \Gamma$, and $\xi' \preceq \xi''$, and, again by Definition *3.2.8*, $\xi' \preceq \xi$.

[$(I_K)$]: let $\xi \equiv \Pi\alpha{:}K.\xi''$, for some $\alpha, K$, and $\xi''$. Then the premise is $\Gamma, \alpha{:}K \vdash x : \xi''$. By induction, there exists $\xi'$ such that $x{:}\xi' \in \Gamma, \alpha{:}K$, and $\xi' \preceq \xi''$. Since $x \not\equiv \alpha$, also $x{:}\xi' \in \Gamma$, and, by Definition *3.2.8*, $\xi' \preceq \Pi\alpha{:}K.\xi''$.

[$(E_K)$]: let $\xi \equiv \xi''[\phi/\alpha]$, for some $\xi'', \phi$, and $\alpha$. Then one premise is $\Gamma \vdash x : \Pi\alpha{:}K.\xi''$, for some $K$. By induction, there exists $\xi'$ such that $x{:}\xi' \in \Gamma$, and $\xi' \preceq \Pi\alpha{:}K.\xi''$, and, by Definition *3.2.8*, $\xi' \preceq \xi''[\phi/\alpha]$.

*ii)* [$(\mathit{Weak})$]: by induction.

[$(\mathit{Conv})$,$(E_K)$]: by induction, and Definition *3.2.8*.

[$(I_K)$]: let $\xi \equiv \Pi\alpha{:}K.\xi''$, for some $\alpha, K$, and $\xi''$. Then the premise is $\Gamma, \alpha{:}K \vdash \lambda x.M : \xi''$. By induction, there exists $\Gamma', \phi, \psi$, and $\mathcal{D}' \subseteq \mathcal{D}$ such that its last rule is $(I)$, and the conclusion is the judgment $\Gamma' \vdash \lambda x.M : \Pi x{:}\phi.\psi$, with $\Pi x{:}\phi.\psi \preceq \xi''$. By Definition *3.2.8*, $\Pi x{:}\phi.\psi \preceq \Pi\alpha{:}K.\xi''$.

[$(I)$]: then the conclusion is $\Gamma \vdash \lambda x.M : \Pi x{:}\phi.\psi$, for some $\phi$, and $\psi$. Immediate.

*iii)* [$(\mathit{Weak})$]: by induction.

[$(\mathit{Conv})$,$(I_K)$]: by induction, and Definition *3.2.8*.

[$(E_K)$]: let $\xi \equiv \xi'[\psi/\alpha]$, for some $\xi', \alpha$, and $\psi$. Then one premise is $\Gamma \vdash MN : \Pi\alpha{:}K.\xi'$. By induction, there exists $\Gamma', \phi, \psi'$, and $\mathcal{D}' \subseteq \mathcal{D}$ such that the last rule is $(E)$, and the conclusion is the judgment $\Gamma' \vdash MN : \psi'[N/x]$, and $\psi'[N/x] \preceq \Pi\alpha{:}K.\xi'$. By Definition *3.2.8*, $\psi'[N/x] \preceq \xi'[\psi/\alpha]$.

[$(E)$]: then the conclusion is $\Gamma \vdash MN : \psi[N/x]$, for some $x$, and $\psi$. Immediate.

■

Notice that this lemma states more than, for example, Property *2.1.17*, since it explicitly states the existence of a subderivation. This will be convenient in the proof of Theorem *3.4.4*. Also the following properties hold.

*Lemma 3.2.10* (Generation for constructors and kinds) *i) If $\Gamma \vdash \alpha : K$, then there is $K'$, such that $\alpha{:}K' \in \Gamma$ and $K' =_\beta K$.*

*ii) If $\Gamma \vdash \lambda a{:}A.B : C$, then there are $\Gamma' \sqsubseteq \Gamma$, and $D$, such that $\Gamma', a{:}A \vdash B : D$, and $\Pi a{:}A.D =_\beta C$.*

*iii) If $\Gamma \vdash AB : C$, then there are $\Gamma' \sqsubseteq \Gamma$, $d, D$, and $E$, such that $\Gamma' \vdash A : \Pi d{:}D.E$, and $\Gamma' \vdash B : D$, and $E[B/d] =_\beta C$.*

*iv) If $\Gamma \vdash \Pi a{:}A.B : s$, then there is $\Gamma' \sqsubseteq \Gamma$, such that $\Gamma', a{:}A \vdash B : s$.*

*Proof:* By induction on derivations.

*i)* [(*Proj*)]: let $\Gamma{=}\Gamma'{,}\alpha{:}K$, for some $\Gamma'$. Then the conclusion is $\Gamma'{,}\alpha{:}K \ \vdash \alpha : K$. Take $K' =_\beta K$.

[(*Weak*)]: let $\Gamma{=}\Gamma'{,}a{:}A$, for some $\Gamma'$, $a$, and $A$. Then the conclusion is $\Gamma'{,}a{:}A \vdash \alpha : K$. By induction, there exists $K'$, such that $\alpha{:}K' \in \Gamma'$, and $K' =_\beta K$. Since $\Gamma' \sqsubseteq \Gamma$, it follows, by Definition *3.2.5 (ii)*, that $\alpha{:}K' \in \Gamma$.

[(*Conv*)]: then one premise is $\Gamma \vdash \alpha : K''$, for some $K''$, with $K'' =_\beta K$. By induction, there exists $K'$ such that $\alpha{:}K' \in \Gamma$ and $K' =_\beta K''$, and, by Definition *3.1.5 (iii)*, $K'' =_\beta K$.

Parts *(ii)*, *(iii)*, and *(iv)* come immediately, or by applying the induction hypothesis.
■

The following lemma shows that all subterms of typable terms are typable.

*Lemma 3.2.11 Let $B \in \mathcal{ST}(A)$. If $\Gamma \vdash A : C$, then there exist $\Gamma'$, and $D$, such that $\Gamma' \vdash B : D$.*

*Proof:* By induction on derivations.

[(*Proj*)]: let $\Gamma{=}\ \Gamma''{,}\, c{:}C$, for some $\Gamma''$, and $c$. Then the conclusion is $\Gamma''{,}\, c{:}C \vdash c : C$. Take $\Gamma'{=}\Gamma$, and $D \equiv C$.

[(*Weak*),(*Conv*),(Elimination rules)]: we consider the second case, the other being similar. Then one premise is $\Gamma \vdash A : E$, for some $E$, where $E =_\beta C$. If $B \equiv A \in \mathcal{ST}(A)$, then take $\Gamma'{=}\Gamma$, and $D \equiv E$, else the thesis follows by induction.

[(Introduction, Product rules)]: take for instance the rule $(C\text{-}I_C)$:

$$\frac{\mathcal{D} : \Gamma, x{:}\phi \vdash \psi : K}{\Gamma \vdash \lambda x{:}\phi.\psi : \Pi x{:}\phi.K.}\ (C\text{-}I_C)$$

Recall that $\mathcal{ST}(\lambda x{:}\phi.\psi) = \{\lambda x{:}\phi.\psi\} \cup \mathcal{ST}(\phi) \cup \mathcal{ST}(\psi)$. The result follows directly for $B \equiv \lambda x{:}\phi.\psi$, and by induction for $B \in \mathcal{ST}(\psi)$. If $B \in \mathcal{ST}(\phi)$, then, by Lemma *3.2.6* we find a $\mathcal{D}' \subseteq \mathcal{D}$, such that $\mathcal{D}' : \Gamma \vdash \phi : s$, so the thesis follows by induction. ■

The next lemma shows that substitution of variables declared in a context for terms of the same type does not affect derivability.

*Lemma 3.2.12* (Substitution) *If* $\Gamma_1, c{:}C, \Gamma_2 \vdash A : B$ *and* $\Gamma_1 \vdash D : C$*, then* $\Gamma_1, \Gamma_2[D/c] \vdash A[D/c] : B[D/c]$.

*Proof:* By induction on derivations.

[(*Proj*)]: if this rule is applied to the variable $c$, then we have:

$$\frac{\Gamma_1 \vdash C : s \quad c \notin \mathcal{D}om\,(\Gamma_1)}{\Gamma_1, c{:}C \vdash c : C,}\ (Proj)$$

and the result follows immediately from the assumption $\Gamma_1 \vdash D : C$. Otherwise, if the (*Proj*) rule is applied to a variable different from $c$, i.e., $\Gamma_2 = \Gamma_2', b{:}B$, for some $\Gamma_2'$, and $b$, and

$$\frac{\Gamma_1, c{:}C, \Gamma_2' \vdash B : s \quad b \notin \mathcal{D}om\,(\Gamma_1, c{:}C, \Gamma_2')}{\Gamma_1, c{:}C, \Gamma_2', b{:}B \vdash b : B.}\ (Proj)$$

Then, by induction, we obtain $\Gamma_1, \Gamma_2'[D/c] \vdash B[D/c] : s$. Notice that, from the assumption $\Gamma_1 \vdash D : C$ and Lemma *3.2.7 (i)*, we know that $\mathcal{FV}(D) \subseteq \mathcal{D}om\,(\Gamma_1)$. Since $b \notin \mathcal{D}om\,(\Gamma_1, c{:}C, \Gamma_2')$, also $b \notin \mathcal{D}om\,(\Gamma_1, \Gamma_2'[D/c])$. Then, by applying a (*Proj*) rule, we obtain $\Gamma_1, \Gamma_2'[D/c], b{:}B[D/c] \vdash b : B[D/c]$, as desired.

[(*Weak*)]: like for rule (*Proj*), we have to consider two subcases: the first is of the form:

$$\frac{\Gamma_1 \vdash A : B \quad \Gamma_1 \vdash C : s \quad c \notin \mathcal{D}om\,(\Gamma_1)}{\Gamma_1, c{:}C \vdash A : B.}\ (Weak)$$

By applying Lemma *3.2.7 (i)*, we get $\mathcal{FV}(A) \cup \mathcal{FV}(B) \subseteq \mathcal{D}om\,(\Gamma_1)$, and by the assumption $c \notin \mathcal{D}om\,(\Gamma_1)$, also $A[D/c] \equiv A$ and $B[D/c] \equiv B$. So the result follows directly from the assumption $\Gamma_1 \vdash A : B$. The second case is of the form:

$$\frac{\Gamma_1, c{:}C, \Gamma_2' \vdash A : B \quad \Gamma_1, c{:}C, \Gamma_2' \vdash E : s \quad e \notin \mathcal{D}om\,(\Gamma_1, c{:}C, \Gamma_2')}{\Gamma_1, c{:}C, \Gamma_2', e{:}E \vdash A : B,}\ (Weak)$$

i.e., $\Gamma_2 = \Gamma_2', e{:}E$, for some $\Gamma_2'$, $e$, and $E$. By induction, we get

$$\Gamma_1,\Gamma_2'[\mathrm{D/c}] \vdash A[D/c] : B[D/c], \quad \text{and} \quad \Gamma_1,\Gamma_2'[\mathrm{D/c}] \vdash E[D/c] : s.$$

Since $e \notin \mathcal{D}om\,(\Gamma_1, c{:}C, \Gamma_2')$, also $e \notin \mathcal{D}om\,(\Gamma_1, \Gamma_2'[D/c])$. Apply a (*Weak*) rule and obtain $\Gamma_1$,$\Gamma'_2$[D/c],e:E[D/c] $\vdash A[D/c] : B[D/c]$, as desired.

[(*Conv*),(Elimination, Introduction, Product rules)]: all these cases follow by induction. The case of the (*Conv*) rule also requires Lemma *3.1.6*. ∎

Now we are able to prove the property that all predicates in derivable statements are typable.

*Lemma 3.2.13* (Predicate Typability) *If* $\Gamma \vdash A : B$*, then* $B \equiv \Box$*, or* $\Gamma \vdash B : s$*.*

*Proof:* By induction on derivations.

[(*Axiom*),(*Conv*), (*Proj*)]: immediate.

[(*Weak*)]: by induction hypothesis.

[(Elimination Rules)]: then the assumptions are $\Gamma \vdash C : \Pi e{:}E.F$, and $\Gamma \vdash D : E$, for some $C, D$, $e, E$, and $F$, such that $A \equiv CD$, and $B \equiv F[D/e]$. By induction, $\Pi e{:}E.F \equiv \Box$, or $\Gamma \vdash \Pi e{:}E.F : s$, for some $s$. The first case is impossible, whereas, by applying the the Generation Lemma *3.2.10 (iv)* to the second case, there is $\Gamma'$, such that $\Gamma', e{:}E \vdash F : s$, and $\Gamma' \sqsubseteq \Gamma$. From Lemma *3.2.5 (i)*, and the Thinning Lemma *3.2.7 (ii)*, also $\Gamma, e{:}E \vdash F : s$. Apply the Substitution Lemma *3.2.12*, and obtain $\Gamma \vdash F[D/e] : s$, as desired.

[(Introduction Rules)]: then the assumption is $\Gamma, c{:}C \vdash E : D$, for some $c, C, D$, and $E$, such that $B \equiv \Pi c{:}C.D$. By induction, $D \equiv \Box$, or $\Gamma, c{:}C \vdash D : s$, for some $s$. The first case is impossible, whereas, for the second case, apply a product rule to obtain $\Gamma \vdash \Pi c{:}C.D : s$.

[(Product Rules)]: then the assumption is $\Gamma, c{:}C \vdash D : s$, for some $c, C, D$, and $s$, such that $B \equiv s$. Clearly $B \equiv \Box$, or $B \equiv *$. The first case is immediate, whereas, for the second case, observe that $\Gamma' \vdash * : \Box$ is derivable for all legal contexts $\Gamma'$. ∎

The following lemma says that we cannot assign to a $\lambda$-terms anything but an element belonging to the set $\mathcal{T}ype$.

*Lemma 3.2.14* *If* $\Gamma \vdash M : \phi$*, then* $\Gamma \vdash \phi : *$*, i.e.,* $\phi$ *is a type with respect to* $\Gamma$*.*

*Proof:* By Lemma *3.2.13*, either $\phi \equiv \Box$, or $\Gamma \vdash \phi : s$. But, by Lemma *3.1.8*, $\phi \not\equiv \Box$, since $M \in \Lambda$. We finish the proof by showing

$$\Gamma \vdash \phi : s \;\Rightarrow\; s \equiv *,$$

by induction on derivations.

[(*Proj*)]: then $\phi$ is a constructor variable, say $\alpha$. Then $\Gamma = \Gamma', \alpha{:}s$, for some $\Gamma'$, and $s$, and, by Lemma *3.2.6*, also $\Gamma' \vdash s : s'$, for some $s'$. But this is possible only if $s \equiv *$, and $s' \equiv \Box$.

[(*Weak*)]: by induction.

[(Elimination rules)]: take for example the rule $(C\text{-}E_C)$: then the conclusion is $\Gamma \vdash \phi : K[M/x]$, for some $K, M$, and $x$, such that $K[M/x] \equiv s$. By looking at the syntax of kinds, we have that $K[M/x] \equiv *$.

[(Product rules)]: immediate.

No other cases are possible ∎

## 3.3 The Church-Rosser Theorem

The notion of reduction, as presented in Definition *3.1.5*, satisfies the Church-Rosser property.

*Theorem 3.3.1* (Church-Rosser Property for type assignment systems) *If $A \twoheadrightarrow_\beta A'$ and $B \twoheadrightarrow_\beta B'$, then there exists $C$, such that $A' \twoheadrightarrow_\beta C$ and $B' \twoheadrightarrow_\beta C$.*

*Proof:* In the terminology of Klop [Klo80], our $\beta$-reduction is a regular combinatory reduction system, and thus the Church-Rosser Property follows from Theorem 3.11 in [Klo80]. ∎

As an illustration, and to give some details about how the Church-Rosser property is obtained, we now present another proof, that uses the technique of *parallel reduction* developed by Tait and Martin-Löf.

Intuitively, parallel reduction reduces a number of $\beta$-redexes simultaneously.

*Definition 3.3.2 The parallel $\beta$-reduction, denoted by $\Rightarrow_\beta$, is inductively defined as follows:*

$$
\begin{array}{lll}
(\beta_1) & a \Rightarrow_\beta a & \\
(\beta_2) & \lambda x.M \Rightarrow_\beta \lambda x.M' & \textit{if } M \Rightarrow_\beta M' \\
(\beta_3) & \lambda a{:}A.B \Rightarrow_\beta \lambda a{:}A'.B' & \textit{if } A \Rightarrow_\beta A', \textit{ and } B \Rightarrow_\beta B' \\
(\beta_4) & \Pi a{:}A.B \Rightarrow_\beta \Pi a{:}A'.B' & \textit{if } A \Rightarrow_\beta A', \textit{ and } B \Rightarrow_\beta B' \\
(\beta_5) & AB \Rightarrow_\beta A'B' & \textit{if } A \Rightarrow_\beta A', \textit{ and } B \Rightarrow_\beta B' \\
(\beta_6) & (\lambda x.M)N \Rightarrow_\beta M'[N'/x] & \textit{if } M \Rightarrow_\beta M', \textit{ and } N \Rightarrow_\beta N' \\
(\beta_7) & (\lambda a{:}A.B)C \Rightarrow_\beta B'[C'/a] & \textit{if } B \Rightarrow_\beta B', \textit{ and } C \Rightarrow_\beta C'.
\end{array}
$$

The rules $(\beta_1), \ldots, (\beta_5)$ mean that the relation $\Rightarrow_\beta$ includes the identity on $\lambda$-terms, i.e., $A \Rightarrow_\beta A$ holds for each $A$. Worth noticing is the rule $(\beta_7)$, that does not reduce the type (kind) of the bound variable $a$: this is because the type (kind) $A$ of $a$ does not play any role in the redex, and, moreover, it disappear in the contractum. However, given a term with redexes inside it, we may apply the rules $(\beta_6)$, and $(\beta_7)$ rather than $(\beta_5)$. Thus, $A \Rightarrow_\beta A'$ means intuitively that $A'$ is obtained from $A$, by simultaneous contraction of some $\beta$-redexes possibly overlapping each other. For the parallel reduction, the following lemma holds:

*Lemma 3.3.3* *i) If $A \rightarrow_\beta A'$ then $A \Rightarrow_\beta A'$.*
*ii) If $A \Rightarrow_\beta A'$ then $A \twoheadrightarrow_\beta A'$.*
*iii) If $A \Rightarrow_\beta A'$, and $B \Rightarrow_\beta B'$ then $A[B/b] \Rightarrow_\beta A'[B'/b]$.*

*Proof:* Part $(i)$ can be proved by easy induction on $\rightarrow_\beta$, whereas parts *(ii)*, and *(iii)* can be proved by induction on $A$. ■

Now, to prove the Church-Rosser Theorem, it is sufficient to show the *diamond property* for $\Rightarrow_\beta$, i.e.

*Property 3.3.4* (Diamond Property) *If $A \Rightarrow_\beta B_1$, and $A \Rightarrow_\beta B_2$, then there exists $A'$, such that $B_1 \Rightarrow_\beta A'$, and $B_2 \Rightarrow_\beta A'$.*

Indeed, we can adapt the stronger statement from [Tak89] to our $\mathcal{TAS}$ cube, as follows:

*Theorem 3.3.5* (Strong Church-Rosser) *If $A \Rightarrow_\beta B$, then there exists a term $A^*$, depending only on $A$, such that $B \Rightarrow_\beta A^*$.*

*Proof:* Define the mapping $*$ by induction on terms as follows:

$$
\begin{array}{rcl}
a^* & \equiv & a \\
(\lambda x.M)^* & \equiv & \lambda x.M^* \\
(\lambda a{:}A.B)^* & \equiv & \lambda a{:}A^*.B^* \\
(\Pi a{:}A.B)^* & \equiv & \Pi a{:}A^*.B^* \\
(AB)^* & \equiv & A^*B^* \quad \text{if } AB \text{ is not a } \beta\text{-redex} \\
((\lambda x.M)N)^* & \equiv & M^*[N^*/x] \\
((\lambda a{:}A.B)C)^* & \equiv & B^*[C^*/a].
\end{array}
$$

Now the proof proceeds by induction on $A$:

[$A \equiv a$]: if $a \Rightarrow_\beta B$, then $B \equiv a \Rightarrow_\beta a \equiv a^*$.

[$A \equiv \lambda x.M$]: if $\lambda x.M \Rightarrow_\beta B$, then $B \equiv \lambda x.N$ for some $N$, with $M \Rightarrow_\beta N$. Since $M \in \mathcal{ST}(A)$, by induction, we get $N \Rightarrow_\beta M^*$. This implies $\lambda x.N \Rightarrow_\beta \lambda x.M^* \equiv (\lambda x.M)^*$.

[$A \equiv \lambda c{:}C.D$, and $A \equiv \Pi c{:}C.D$]: we consider the first case, the second being simpler. If $\lambda c{:}C.D \Rightarrow_\beta B$, then $B \equiv \lambda c{:}C'.D'$ for some $C'$, and $D'$, with $C \Rightarrow_\beta C'$, and $D \Rightarrow_\beta D'$. Since $C \in \mathcal{ST}(A)$, and $D \in \mathcal{ST}(A)$, by induction, we get $C' \Rightarrow_\beta C^*$, and $D' \Rightarrow_\beta D^*$. This implies $\lambda c{:}C'.D' \Rightarrow_\beta \lambda c{:}C^*.D^* \equiv (\lambda c{:}C.D)^*$.

[$A \equiv CD$, and $A$ is not a $\beta$-redex]: if $CD \Rightarrow_\beta B$, then $B \equiv C'D'$, for some $C'$, and $D'$, with $C \Rightarrow_\beta C'$, and $D \Rightarrow_\beta D'$. Then, by induction, we get $C' \Rightarrow_\beta C^*$, and $D' \Rightarrow_\beta D^*$. This implies $C'D' \Rightarrow_\beta \; C^*D^* \equiv (CD)^*$.

[$A \equiv (\lambda x.M)N$, and $A \equiv (\lambda c{:}C.D)E$]: we consider the second case, the first being similar. If $((\lambda c{:}C.D)E) \Rightarrow_\beta B$, then either $B \equiv ((\lambda c{:}C'.D')E')$, or $B \equiv D'[E'/c]$, for some $c, C', D'$, and $E'$, with $C \Rightarrow_\beta C'$, $D \Rightarrow_\beta D'$, and $E \Rightarrow_\beta E'$. Then, by induction, $C' \Rightarrow_\beta C^*$, $D' \Rightarrow_\beta D^*$, and $E' \Rightarrow_\beta E^*$: there are two cases to consider:

(subcase 1): if $B \equiv (\lambda c{:}C'.D')E'$, then $B \Rightarrow_\beta D'[E'/c] \Rightarrow_\beta D^*[E^*/c]$ which is equal, by Lemma *3.3.3 (iii)*. to $(D[E/c])^*$.

(subcase 2): if $B \equiv D'[E'/c]$, then $B \Rightarrow_\beta D^*[E^*/c] \equiv (D[E/c])^*$, by Lemma *3.3.3 (iii)*. ∎

## 3.4 The Subject Reduction Theorem

We now come to the proof that the here defined notion of type assignment is closed for subject reduction on typable $\lambda$-terms, i.e., if $\Gamma \vdash M : \psi$ and $M \twoheadrightarrow_\beta N$, then also $\Gamma \vdash N : \psi$. The proof of this result is not immediate, because of the presence of the derivations rules that are not syntax-directed. It requires a sequence of lemmas; to start with, the next lemma states that contexts can be considered modulo $\beta$-conversion of predicates.

*Lemma 3.4.1* (*Conv* in Context) *If* $\Gamma_1, a{:}A, \Gamma_2 \vdash B : C$*, then* $\Gamma_1, a{:}A', \Gamma_2 \vdash B : C$*, for all* $A'$ *such that* $\Gamma_1, a{:}A'$ *is legal and* $A' =_\beta A$*.*

*Proof:* By induction on derivations.

[(*Proj*)]: if this rule is applied to the variable $a$, then we have:

$$\frac{\Gamma_1 \vdash C : s \quad a \notin \mathcal{D}om\,(\Gamma_1)}{\Gamma_1, a{:}A \vdash a : A.}\ (Proj)$$

Then, by Definition *3.2.2*, also $\Gamma_1 \vdash A' : s$, for some $s$, and we can build the following derivation:

$$\frac{\dfrac{\Gamma_1 \vdash A' : s \quad a \notin \mathcal{D}om\,(\Gamma_1)}{\Gamma_1, a{:}A' \vdash a : A'}\ (Proj) \qquad \dfrac{\Gamma_1 \vdash A : s \quad \Gamma_1 \vdash A' : s \quad a \notin \mathcal{D}om\,(\Gamma_1)}{\Gamma_1, a{:}A' \vdash A : s}\ (Weak) \quad A =_\beta A'}{\Gamma_1, a{:}A' \vdash a : A.}\ (Conv)$$

Otherwise, if the (*Proj*) rule is applied to a variable $c$ different from $a$, so $\Gamma_2{=}\Gamma_2', c{:}C$, for some $\Gamma_2', c$, and $B \equiv c$, then, by induction, we get $\Gamma_1, a{:}A',\Gamma_2' \vdash C : s$. Since c $\notin$ $\mathcal{D}om\,(\Gamma_1, a{:}A', \Gamma_2')$, apply a (*Proj*) rule to obtain $\Gamma_1, a{:}A',\Gamma_2', c{:}C \vdash c : C$, as desired.

[(*Weak*)]: if this rule is applied to the variable $a$, then $\Gamma_1 \vdash B : C$ and $a \notin \mathcal{D}om\,(\Gamma_1)$. Then, again by Definition *3.2.2*, also $\Gamma_1 \vdash A' : s$, for some $s$, and we can also derive

$$\frac{\Gamma_1 \vdash B : C \quad \Gamma_1 \vdash A' : s \quad a \notin \mathcal{D}om\,(\Gamma_1)}{\Gamma_1, a{:}A' \vdash B : C.}\ (Weak)$$

As above, if this rule is applied to a variable $c$ different from $a$, then the result follows by induction, and an application of a (*Weak*) rule.

[(*Conv*)]: by induction, we get $\Gamma_1, a{:}A', \Gamma_2 \vdash B : D$, and $\Gamma_1, a{:}A', \Gamma_2 \vdash C : s$, for some $D$ such that $D =_\beta C$. Then, apply a (*Conv*) rule, and obtain $\Gamma_1, a{:}A', \Gamma_2 \vdash B : C$, as desired.

[(Introduction, Elimination, Product rules)]: by induction. ■

The following lemma gives a stronger formulation for the Generation Lemma for $\lambda$-term abstraction, without the $\preceq$ relation on types.

*Lemma 3.4.2* (Abstraction Types) *If* $\Gamma \vdash \lambda x.M : \xi$*, then there are* $K_1, \ldots, K_k$*,* $k \geq 0$*,* $\phi$ *and* $\psi$*, such that* $\xi =_\beta \Pi_{i=1}^k \alpha_i{:}K_i.\Pi x{:}\phi.\psi$ *and* $\Gamma, \alpha_1{:}K_1, \ldots, \alpha_k{:}K_k, x{:}\phi \vdash M : \psi$*.*

*Proof:* By induction on derivations.

[($I$)]: immediate, with $k = 0$.

[(*Weak*)]: let $\Gamma{=}\Gamma', c{:}C$, for some $\Gamma', c$, and $C$. Then the assumption are $\Gamma' \vdash \lambda x.M : \xi$, and $\Gamma' \vdash C : s$. By induction,

$$\xi =_\beta \Pi_{i=1}^k \alpha_i{:}K_i.\Pi x{:}\phi.\psi, \quad \text{and} \quad \Gamma', \alpha_1{:}K_1, \ldots, \alpha_k{:}K_k, x{:}\phi \vdash M : \psi.$$

Since $\Gamma', \alpha_1{:}K_1, \ldots, \alpha_k{:}K_k, x{:}\phi \sqsubseteq \Gamma', c{:}C, \alpha_1{:}K_1, \ldots, \alpha_k{:}K_k, x{:}\phi$, we can apply the Thinning Lemma *3.2.7 (ii)*, to obtain $\Gamma', c{:}C, \alpha_1{:}K_1, \ldots, \alpha_k{:}K_k, x{:}\phi \vdash M : \psi$.

[(*Conv*)]: then the assumption are $\Gamma \vdash \lambda x.M : \xi'$, and $\xi' =_\beta \xi$, for some $\xi'$. By induction,

$$\xi' =_\beta \Pi_{i=1}^k \alpha_i{:}K_i.\Pi x{:}\phi.\psi, \quad \text{and} \quad \Gamma, \alpha_1{:}K_1, \ldots, \alpha_k{:}K_k, x{:}\phi \vdash M : \psi,$$

so, $\xi =_\beta \Pi_{i=1}^k \alpha_i{:}K_i.\Pi x{:}\phi.\psi$.

[($I_K$)]: let $\xi \equiv \Pi\beta{:}K.\xi'$, for some $\beta, K$, and $\xi'$. Then the assumptions are $\Gamma, \beta{:}K \vdash M : \xi'$. By induction,

$$\xi' =_\beta \Pi_{i=1}^k \alpha_i{:}K_i.\Pi x{:}\phi.\psi, \quad \text{and} \quad \Gamma, \beta{:}K, \alpha_1{:}K_1, \ldots, \alpha_k{:}K_k, x{:}\phi \vdash M : \psi,$$

and also, by the Church-Rosser Theorem *3.3.1*,
$\Pi\beta{:}K.\xi' =_\beta \Pi\beta{:}K.\Pi_{i=1}^k \alpha_i{:}K_i.\Pi x{:}\phi.\psi$.

[($E_K$)]: let $\xi \equiv \xi'[\mu/\alpha]$, for some $\alpha, \xi'$, and $\mu$. Then the assumptions are $\Gamma \vdash \lambda x.M : \Pi\alpha{:}K.\xi'$, and $\Gamma \vdash \mu : K$, for some $K$. By induction,

$$\Pi\alpha{:}K.\xi' =_\beta \Pi_{i=1}^k \alpha_i{:}K_i.\Pi x{:}\phi.\psi, \quad \text{and} \quad \Gamma, \alpha_1{:}K_1, \alpha_2{:}K_2, \ldots, \alpha_k{:}K_k, x{:}\phi \ \vdash M : \psi.$$

By the Church-Rosser Theorem *3.3.1*, $\beta$-convertible terms have a common reduct, so it must be that $k \geq 1$, $K =_\beta K_1$ (assuming by $\alpha$-conversion that $\alpha$ is $\alpha_1$), and

$$\xi' =_\beta \Pi_{i=2}^k \alpha_i{:}K_i.\Pi x{:}\phi.\psi.$$

By Lemma *3.1.6*, we have $\xi'^\vartheta =_\beta \Pi_{i=2}^k \alpha_i{:}K_i^\vartheta.\Pi x{:}\phi^\vartheta.\psi^\vartheta$, where $\vartheta$ stands for the substitution $[\mu/\alpha]$. By Lemma *3.2.13*, $\Gamma \vdash K : s$, for some $s$, therefore $\Gamma, \alpha{:}K$ is legal. So we can apply the $(Conv)$ in Context Lemma *3.4.1*, to get:

$$\Gamma, \alpha{:}K, \alpha_2{:}K_2, \ldots, \alpha_k{:}K_k, x{:}\phi \ \vdash M : \psi,$$

and, finally, by the Substitution Lemma *3.2.12*, we obtain

$$\Gamma, \alpha_2{:}K_2^\vartheta, \ldots, \alpha_k{:}K_k^\vartheta, x{:}\phi^\vartheta \ \vdash M : \psi^\vartheta,$$

as desired.

The other cases are impossible. ■

The following lemma says that the introduction rule for $\lambda$-term abstraction can always be the last applied rule in the derivation.

*Lemma 3.4.3* (Term Abstraction) *If* $\Gamma \vdash \lambda x.M : \Pi x{:}\phi.\psi$, *then* $\Gamma, x{:}\phi \vdash M : \psi$.

*Proof:* By the Abstraction Types Lemma *3.4.2*, we find $K_1, \ldots, K_k, \phi'$, and $\psi'$, such that $\Pi x{:}\phi.\psi \ =_\beta \Pi_{i=1}^k \alpha_i{:}K_i.\Pi x{:}\phi'.\psi'$, and since these two expressions have a common reduct, it must be that $k = 0$, $\phi =_\beta \phi'$, and $\psi =_\beta \psi'$. Again by Lemma *3.4.2*, we know that $\Gamma, x{:}\phi' \vdash M : \psi'$. Since $\Gamma \vdash \lambda x.M : \Pi x{:}\phi.\psi$, by Lemma *3.2.14*, also $\Gamma \vdash \Pi x{:}\phi.\psi : *$. By the Generation Lemma *3.2.10 (iv)*, there exists $\Gamma' \sqsubseteq \Gamma$, such that $\Gamma', x{:}\phi \vdash \psi : *$, so by the Thinning Lemma *3.2.7 (ii)*, also $\Gamma, x{:}\phi \ \vdash \psi : *$, and, by Lemma *3.2.3*, $\Gamma, x{:}\phi$ is legal. Since $\phi =_\beta \phi'$, by the $(Conv)$ in Context Lemma *3.4.1*, also $\Gamma, x{:}\phi \vdash M : \psi'$. Moreover, since $\Gamma, x{:}\phi \vdash \psi : *$ and $\psi =_\beta \psi'$, we can apply $(Conv)$ rule to this last derivation and obtain $\Gamma, x{:}\phi \vdash M : \psi$, as desired. ■

Using this last lemma, it becomes easy to prove that the notion of type assignment we consider is closed for subject reduction on $\lambda$-terms.

*Theorem 3.4.4* (Subject Reduction for Terms) *If* $\Gamma \vdash M : \psi$ *and* $M \twoheadrightarrow_\beta N$, *then* $\Gamma \vdash N : \psi$.

*Proof:* By induction on the number of $\beta$-reduction steps in $M \twoheadrightarrow_\beta N$. We just consider the base case, the inductive step is straightforward. The base case is proved by structural induction on the context in which the redex occurs: we only consider the case $M \equiv (\lambda x.P)Q$ and $N \equiv P[Q/x]$. Let $\mathcal{D}$ be a derivation for $\Gamma \vdash (\lambda x.P)Q : \psi$. By the Generation Lemma *3.2.9 (iii)*, $\mathcal{D}$ has the following structure:

$$
\begin{array}{c}
\mathcal{D}_1\!: \dfrac{\begin{array}{cc}\vdots & \vdots \\ \Gamma' \vdash (\lambda x.P) : \Pi x{:}\phi'.\psi' & \Gamma' \vdash Q : \phi'\end{array}}{\Gamma' \vdash (\lambda x.P)Q : \psi'[Q/x]}\,(E) \\
\vdots \\
\mathcal{D}\!: \dfrac{}{\Gamma \vdash (\lambda x.P)Q : \psi,}
\end{array}
$$

with $\psi'[Q/x] \preceq \psi$. That is, there is a subderivation $\mathcal{D}_1$, ending with an application of rule $(E)$, which is followed by a (possibly empty) sequence of applications of the not syntax-directed rules (*Weak*), (*Conv*), $(I_K)$ and $(E_K)$. By the Term Abstraction Lemma *3.4.3*, we obtain

$$\Gamma', x{:}\phi' \vdash P : \psi'.$$

Since $\Gamma' \vdash Q : \phi'$, by the Substitution Lemma *3.2.12*, we obtain

$$\Gamma' \vdash P[Q/x] : \psi'[Q/x].$$

Apply the same rules as used to go from $\mathcal{D}_1$ to $\mathcal{D}$ to obtain

$$\Gamma \vdash P[Q/x] : \psi,$$

as desired. ∎

*Theorem 3.4.5* (Subject Reduction for Constructors and Kinds) *i) If* $\Gamma \vdash \phi : K$ *and* $\phi \twoheadrightarrow_\beta \psi$, *then* $\Gamma \vdash \psi : K$.
*ii) If* $\Gamma \vdash K : s$ *and* $K \twoheadrightarrow_\beta K'$, *then* $\Gamma \vdash K' : s$.

*Proof:* By induction on the number of $\beta$-reduction steps in $\phi \twoheadrightarrow_\beta \psi$, or $K \twoheadrightarrow_\beta K'$. We just consider the base case, the inductive step is straightforward. The base case is proved by structural induction on the context in which the redex occurs:

*i)* $\phi \twoheadrightarrow_\beta \psi$. There are two kind of redexes to consider, namely the one used to reduce types dependent on $\lambda$-terms, and the one used to reduce higher-order types. We consider the first case, the second being similar.

Let $\phi \equiv (\lambda x{:}\xi.\mu)M$, and $\psi \equiv \mu[M/x]$, for some $\xi, \mu$, and $M$. By Generation Lemma *3.2.10 (iii)*, there exists $\Gamma'$, and $K'$, with $\Gamma' \sqsubseteq \Gamma$, such that

$$\Gamma' \vdash \lambda x{:}\xi.\mu : \Pi x{:}\xi.K', \quad \text{and} \quad \Gamma' \vdash M : \xi, \quad \text{and} \quad K'[M/x] =_\beta K.$$

Again by Generation Lemma *3.2.10 (ii)*, and *(iv)*, there exists $\Gamma''$, and $K''$, with $\Gamma'' \sqsubseteq \Gamma'$, such that

$$\Gamma'', x{:}\xi \vdash \mu : K'', \quad \text{and} \quad \Pi x{:}\xi.K'' =_\beta \Pi x{:}\xi.K'.$$

By the Thinning Lemma *3.2.7 (ii)*, we get

$$\Gamma, x{:}\xi \vdash \mu : K'', \quad \text{and} \quad \Gamma, x{:}\xi \vdash M : \xi.$$

By the Substitution Lemma *3.2.12*, we obtain

$$\Gamma \vdash \mu[M/x] : K''[M/x].$$

By the Predicate Lemma *3.2.13*, we find $s$ such that we can derive

$$\Gamma \vdash K : s.$$

Since $K'' =_\beta K'$, by Lemma *3.1.6*, we get

$$K''[M/x] =_\beta K'[M/x].$$

Finally apply a $(Conv)$ rule to get

$$\Gamma \vdash \mu[M/x] : K,$$

as desired.

*ii)* $K \twoheadrightarrow_\beta K'$. Then there exists a subterm $A$ of $K$ such that we can apply one of the three $\beta$-reduction rules of Definition *3.1.5*. The thesis follows by the Subject Reduction Theorem *3.4.4* for $\lambda$-terms, and by part $(i)$, using the contextual closure of the $\rightarrow_\beta$ reduction rules. ■

*Theorem 3.4.6* (Subject Reduction for Contexts and Predicates) *Let* $\Gamma \twoheadrightarrow_\beta \Gamma'$ *be the (point-wise) extension of the* $\rightarrow_\beta$ *reduction rule on contexts.*

*i) If* $\Gamma \vdash A : B$, *and* $\Gamma \twoheadrightarrow_\beta \Gamma'$, *then* $\Gamma' \vdash A : B$

*ii) If* $\Gamma \vdash A : B$, *and* $B \twoheadrightarrow_\beta B'$, *then* $\Gamma \vdash A : B'$

*Proof:* *i)* All cases can be handled easily by induction. In case the last rule is (*Proj*) or (*Weak*), also use Subject Reduction Theorems *3.4.4* and *3.4.5*.

*ii)* We distinguish three cases, depending whether $A \in \Lambda$, $A \in Cons$, or $A \in Kind$. The last case is immediate, using Proposition *3.2.1*, whereas the first two are similar. We treat the first case. Let $A \equiv M$, for some $M$; then by Lemma *3.1.8*, $B \equiv \phi$, for some $\phi$. By Lemma *3.2.14*, $\Gamma \vdash \phi : *$. Apply the Subject Reduction Theorem *3.4.5*, to get $\Gamma \vdash \phi' : *$, and finally a (*Conv*) rule to get

$$\Gamma \vdash M : \phi',$$

as desired. ■

## 3.5 Normalization

An important property of type assignment systems is the strong normalization of typable terms; this is already know to hold for the systems $F\omega$, $F1$, $F2$ and $F\underline{\omega}$. Using this result, we will show that it also holds for the other four systems of the cube of type assignment systems.

For this, we use the function $\mathcal{E}d$ that 'erases dependencies', i.e., removes the $\lambda$-term information in dependent-types, as defined in [GHR93], that is based on a similar definition given in [PM89]. A similar function, erasing term-dependences in the Theory of Generalized Functionality of [Sel79], can also be found in [BY81].

*Definition 3.5.1 The function $\mathcal{E}d : T_u \to T_u$ is defined as follows:*

*i) On $\Lambda$.*

$$\mathcal{E}d(M) \;=\; M.$$

*ii) On Cons.*

$$\begin{aligned}
\mathcal{E}d(\alpha) &= \alpha, \\
\mathcal{E}d(\Pi x{:}\phi.\psi) &= \Pi x{:}\mathcal{E}d(\phi).\mathcal{E}d(\psi), \\
\mathcal{E}d(\Pi \alpha{:}K.\psi) &= \Pi \alpha{:}\mathcal{E}d(K).\mathcal{E}d(\psi), \\
\mathcal{E}d(\lambda x{:}\phi.\psi) &= \mathcal{E}d(\psi), \\
\mathcal{E}d(\lambda \alpha{:}K.\psi) &= \lambda \alpha{:}\mathcal{E}d(K).\mathcal{E}d(\psi), \\
\mathcal{E}d(\phi\psi) &= \mathcal{E}d(\phi)\mathcal{E}d(\psi), \\
\mathcal{E}d(\phi M) &= \mathcal{E}d(\phi).
\end{aligned}$$

*iii) On Kind.*

$$\begin{aligned}
\mathcal{E}d(*) &= *, \\
\mathcal{E}d(\Pi x{:}\phi.K) &= \mathcal{E}d(K), \\
\mathcal{E}d(\Pi \alpha{:}K_1.K_2) &= \Pi \alpha{:}\mathcal{E}d(K_1).\mathcal{E}d(K_2).
\end{aligned}$$

*iv) The function $\mathcal{E}d$ is (point-wise) extended to contexts, in the following way:*

$$\begin{aligned}
\mathcal{E}d(\varepsilon) &= \varepsilon, \\
\mathcal{E}d(\Gamma, a{:}A) &= \mathcal{E}d(\Gamma), a{:}\mathcal{E}d(A).
\end{aligned}$$

*Lemma 3.5.2 For $\mathcal{E}d$, the following properties hold:*

*i)*

$$\begin{aligned}
\mathcal{E}d(M[N/x]) &\equiv \mathcal{E}d(M)[\mathcal{E}d(N)/x], \\
\mathcal{E}d(\phi[\psi/\alpha]) &\equiv \mathcal{E}d(\phi)[\mathcal{E}d(\psi)/\alpha], \\
\mathcal{E}d(\psi[M/x]) &\equiv \mathcal{E}d(\psi), \\
\mathcal{E}d(K[M/x]) &\equiv \mathcal{E}d(K).
\end{aligned}$$

*ii) If $A =_\beta B$, then either $\mathcal{E}d(A) \equiv \mathcal{E}d(B)$, or $\mathcal{E}d(A) =_\beta \mathcal{E}d(B)$.*

*Proof:* *i)* Direct, using Definition *3.5.1*.

*ii)* By induction on the definition of $=_\beta$. We just consider the case when $A$ is a redex, by induction on $\twoheadrightarrow_\beta$ ; the complete proof follows easily by induction. There are three cases to consider, depending on which reduction rule is used:

$[(\lambda x.M)N \rightarrow_\beta M[N/x]]$: then
$\mathcal{E}d((\lambda x.M)N) \stackrel{def}{\equiv} (\lambda x.M)N \rightarrow_\beta M[N/x] \stackrel{def}{\equiv} \mathcal{E}d(M[N/x])$.

$[(\lambda\alpha{:}K.\phi)\psi \rightarrow_\beta \phi[\psi/\alpha]]$: then $\mathcal{E}d((\lambda\alpha{:}K.\phi)\psi) \stackrel{def}{\equiv} \mathcal{E}d(\lambda\alpha{:}K.\phi)\mathcal{E}d(\psi) \stackrel{def}{\equiv}$ $(\lambda\alpha{:}\mathcal{E}d(K).\mathcal{E}d(\phi))\ \mathcal{E}d(\psi) \ \rightarrow_\beta \mathcal{E}d(\phi)[\mathcal{E}d(\psi)/\alpha]$, which is equal, by part *(i)*, to $\mathcal{E}d(\phi[\psi/\alpha])$.

$[(\lambda x{:}\phi.\psi)M \rightarrow_\beta \psi[M/x]]$: then $\mathcal{E}d((\lambda x{:}\phi.\psi)M) \stackrel{def}{\equiv} \mathcal{E}d(\lambda x{:}\phi.\psi) \stackrel{def}{\equiv} \mathcal{E}d(\psi)$, which is equal, by part *(i)*, to $\mathcal{E}d(\psi[M/x])$. The proof for $\rightarrow_\beta$ follows easily by induction on the context in which the redex occurs. ∎

An important property of the cube of the type assignment systems (also proper to the cube of typed systems) is the fact that the following rule

$$\frac{\Gamma_1, c{:}C, \Gamma_2 \vdash A : B \quad c \notin \mathcal{FV}(\Gamma_2) \cup \mathcal{FV}(A) \cup \mathcal{FV}(B)}{\Gamma_1, \Gamma_2 \vdash A : B,}\ (Strength)$$

is an *admissible* rule. The original proof, for Pure Type Systems, is due to van Benthem [vBJ93], and requires a key lemma, first appeared in [Luo90], that was used for the system ECC, i.e. the so called *extended calculus of constructions.* Luo, in [GN91], proved the same lemma for functional pure type systems, i.e., for systems enjoying the Uniqueness of Sort Property *3.2.1.* This lemma will be useful to prove the Strong Normalization Theorem.

*Lemma 3.5.3 If* $\Gamma_1, c{:}C,\Gamma_2 \vdash A : B$ *and* $c \notin \mathcal{FV}(\Gamma_2) \cup \mathcal{FV}(A)$*, then there exists* $D$*, such that* $B \twoheadrightarrow_\beta D$*, and* $\Gamma_1,\Gamma_2 \vdash A : D$*.*

*Proof:* By induction on derivations.

[(Introduction rules)]: take for example the rule ($C$-$I_C$):

$$\frac{\Gamma_1, c{:}C, \Gamma_2, x{:}\phi \vdash \psi : K}{\Gamma_1, c{:}C, \Gamma_2 \vdash \lambda x{:}\phi.\psi : \Pi x{:}\psi.K.}\ (C\text{-}I_C)$$

Then, by induction, $\Gamma_1,\Gamma_2, x{:}\phi \vdash \psi : K'$, for some $K'$, with $K \twoheadrightarrow_\beta K'$. Apply a ($C$-$I_C$) rule to get $\Gamma_1,\Gamma_2 \vdash \lambda x{:}\phi.\psi : \Pi x{:}\phi.K'$, and observe that $\Pi x{:}\psi.K \twoheadrightarrow_\beta \Pi x{:}\psi.K'$.

[(Elimination rules)]: take for example the $(C\text{-}E_K)$ rule:

$$\frac{\Gamma_1, c{:}C, \Gamma_2 \vdash \phi : \Pi\alpha{:}K_1.K_2 \quad \Gamma_1, c{:}C, \Gamma_2 \vdash \psi : K_1}{\Gamma_1, c{:}C, \Gamma_2 \vdash \phi\psi : K_2[\psi/\alpha].} \; (C\text{-}E_K)$$

Then, by induction, there exists $K^*$, and $K_1''$ such that:

$$\Gamma_1,\Gamma_2 \vdash \phi : K^*, \quad \text{and} \quad \Gamma_1,\Gamma_2 \vdash \psi : K_1'',$$

with $\Pi\alpha{:}K_1.K_2 \twoheadrightarrow_\beta K^*$, and $K_1 \twoheadrightarrow_\beta K_1''$. Then, by looking at our reduction rules, it follows that there exists $K_1', K_2'$, such that $K^* \equiv \Pi\alpha{:}K_1'.K_2'$, with $K_1 \twoheadrightarrow_\beta K_1'$, and $K_1 \twoheadrightarrow_\beta K_1''$, and $K_2 \twoheadrightarrow_\beta K_2'$. By the Church-Rosser Theorem *3.3.1*, we find $K_1'''$, such that $K_1' \twoheadrightarrow_\beta K_1'''$, and $K_1'' \twoheadrightarrow_\beta K_1'''$. By the Predicate Lemma *3.2.13*, we get

$$\Gamma_1,\Gamma_2 \vdash \Pi\alpha{:}K_1'.K_2' : \Box, \quad \text{and} \quad \Gamma_1,\Gamma_2 \vdash K_1'' : \Box.$$

By the Subject Reduction Theorem for constructors and kinds *3.4.5*, we get

$$\Gamma_1,\Gamma_2 \vdash \Pi\alpha{:}K_1'''.K_2' : \Box, \quad \text{and} \quad \Gamma_1,\Gamma_2 \vdash K_1''' : \Box.$$

Now apply twice a $(Conv)$ rule and obtain

$$\Gamma_1,\Gamma_2 \vdash \phi : \Pi\alpha{:}K_1'''.K_2', \quad \text{and} \quad \Gamma_1,\Gamma_2 \vdash \psi : K_1''',$$

Finally, we can apply an $(C\text{-}E_K)$ rule and obtain

$$\Gamma_1,\Gamma_2 \vdash \phi\psi : K_2'[\psi/\alpha],$$

as desired.

[$(Conv)$]: then the derivation has the following shape:

$$\frac{\Gamma_1, c{:}C, \Gamma_2 \vdash A : D \quad \Gamma_1, c{:}C, \Gamma_2 \vdash B : s \quad B =_\beta D}{\Gamma_1, c{:}C, \Gamma_2 \vdash A : B.} \; (Conv)$$

Then, by induction,

$$\Gamma_1,\Gamma_2 \vdash A : D', \quad \text{and} \quad \Gamma_1,\Gamma_2 \vdash B : s,$$

for some $D'$, and $s$, such that $D \twoheadrightarrow_\beta D'$. By the Church-Rosser Theorem *3.3.1*, there is $D''$ such that $D', B \twoheadrightarrow_\beta D''$. By the Subject Reduction Theorem for constructors and kinds *3.4.5*, we get

$$\Gamma_1,\Gamma_2 \vdash D'' : s.$$

Finally, apply a $(Conv)$ rule to get

$$\Gamma_1, \Gamma_2 \vdash A : D'',$$

as desired.

[(Other rules)]: by induction and the Church-Rosser Theorem *3.3.1*. ∎

Now we can prove the following lemma that, informally, says that we can strengthen contexts in legal judgments.

*Lemma 3.5.4* (Strengthening) *If* $\Gamma_1, c{:}C, \Gamma_2 \vdash A : B$ *and* $c \notin \mathcal{FV}(\Gamma_2) \cup \mathcal{FV}(A) \cup \mathcal{FV}(B)$, *then* $\Gamma_1, \Gamma_2 \vdash A : B$

*Proof:* By Lemma *3.5.3*, there exists $D$, such that $B \twoheadrightarrow_\beta D$, and $\Gamma_1, \Gamma_2 \vdash A : D$. By the Predicate Typability Lemma *3.2.13*, there are two possibilities, namely $\Gamma_1, c{:}C, \Gamma_2 \vdash B : s$, or $B \in \{*, \Box\}$. In the second case we are immediately done, because $B \equiv D$, whereas in the first case we can once again apply Lemma *3.5.3*, to $\Gamma_1, c{:}C, \Gamma_2 \vdash B : s$, to prove $\Gamma_1, \Gamma_2 \vdash B : s$. Since $B \twoheadrightarrow_\beta D$, we can finish the proof by applying a $(Conv)$ rule to get $\Gamma_1, \Gamma_2 \vdash A : B$, as desired. ∎

The next theorem was first stated in [GHR93]. Below a complete proof is given; in [GHR93] the proof was only sketched.

*Theorem 3.5.5* (Erasing Theorem) *If* $\Gamma \vdash A : B$, *then* $\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(A) : \mathcal{E}d(B)$.

*Proof:* By induction on derivations.

[$(Conv)$]: then the derivation has the following shape:

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad B =_\beta C}{\Gamma \vdash A : C}\ (Conv).$$

By induction hypothesis we get

$$\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(A) : \mathcal{E}d(B), \quad \text{and} \quad \mathcal{E}d(\Gamma) \vdash \mathcal{E}d(C) : s.$$

By Lemma *3.5.2 (ii)*, either $\mathcal{E}d(B) =_\beta \mathcal{E}d(C)$, or $\mathcal{E}d(B) \equiv \mathcal{E}d(C)$: the second case is immediate, whereas, in the first case, apply a $(Conv)$ rule to get $\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(A) : \mathcal{E}d(C)$.

[$(C\text{-}I_C)$, $(K\text{-}F_C)$]: take as example the rule $(C\text{-}I_C)$. Then the derivation has the following shape:

$$\frac{\Gamma, x{:}\phi \vdash \psi : K}{\Gamma \vdash \lambda x{:}\phi.\psi : \Pi x{:}\phi.K}\ (C\text{-}I_C).$$

By induction, we get $\mathcal{E}d(\Gamma), x{:}\mathcal{E}d(\phi) \vdash \mathcal{E}d(\psi) : \mathcal{E}d(K)$. Looking at Definition *3.5.1*, it follows that $x \notin \mathcal{FV}(\mathcal{E}d(\psi)) \cup \mathcal{FV}(\mathcal{E}d(K))$. So, we can apply the Strengthening Lemma *3.5.4* to get $\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(\psi) : \mathcal{E}d(K)$, that is $\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(\lambda x{:}\phi.\psi) : \mathcal{E}d(K)$, as required.

[$(C\text{-}E_C)$]: then the derivation has the following shape:

$$\frac{\Gamma \vdash \psi : \Pi x{:}\phi.K \quad \Gamma \vdash M : \phi}{\Gamma \vdash \psi M : K[M/x]}\ (C\text{-}E_C).$$

By induction, we get $\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(\psi) : \mathcal{E}d(\Pi x{:}\phi.K)$. Since, by Definition *3.5.1*, $\mathcal{E}d(\psi M) \equiv\ \mathcal{E}d(\psi)$ and $\mathcal{E}d(\Pi x{:}\phi.K) \equiv \mathcal{E}d(K)$, and, by Lemma *3.5.2*, $\mathcal{E}d(K[M/x]) \equiv \mathcal{E}d(K)$, we get $\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(\psi M) : \mathcal{E}d(K[M/x])$, as desired.

[(*Proj*), (*Weak*)]: take for example the (*Weak*) rule:

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad c \notin \mathcal{D}om\,(\Gamma)}{\Gamma, c{:}C \vdash A : B}\ (Weak).$$

By induction, $\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(A) : \mathcal{E}d(B)$, and $\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(C) : s$, and $c \notin \mathcal{D}om\,(\mathcal{E}d(\Gamma))$. Since $\mathcal{D}om\,(\mathcal{E}d(\Gamma))$=$\mathcal{D}om\,(\Gamma)$, apply again a (*Weak*) rule to obtain $\mathcal{E}d(\Gamma), c{:}\mathcal{E}d(C) \vdash A : B$, which is the same as $\mathcal{E}d(\Gamma, c{:}C) \vdash A : B$, as desired.

[(Other Introduction, Elimination, Product rules)]: then $\mathcal{E}d$ is compositional with respect to the immediate subderivations of that rules, and the conclusion is the $\mathcal{E}d$ of the original one. In all these cases, the thesis follows by induction. ■

The following definition introduces the $\beta$-normal-form of a term.

*Definition 3.5.6* *i) The $\beta$-normal-form of a typed (untyped) term is inductively defined as follows:*

*a) If $B_1, \ldots, B_n$ are in $\beta$-normal-form $(n \geq 0)$, and $a$ is a variable, then $aB_1, \ldots, B_n$ is in $\beta$-normal-form.*

*b) If $M$ is in $\beta$-normal-form, and $x$ is a $\lambda$-term variable, then $\lambda x.M$ is in $\beta$-normal-form.*

*c) If $A, B$ are in $\beta$-normal-form, and $a$ is a variable, then $\lambda a{:}A.B$, and $\Pi a{:}A.B$ are in $\beta$-normal-form.*

*ii) The judgment $\Gamma \vdash A : B$ is in $\beta$-normal-form if $A, B$ are in $\beta$-normal-form, and all predicates in $\Gamma$ are in $\beta$-normal-form.*

Using this dependency-erasing function, we can relate the strong normalization problem for the full $\mathcal{TAS}$ cube to that of the plane without dependences, as done in the following theorem.

*Theorem 3.5.7* (Termination for terms) *If $\Gamma \vdash A : B$, then $A$ is strongly normalizing.*

*Proof:* In [GHR93], Theorem *2.2.1* states that if $\Gamma \vdash A : B$ is a derived judgment in $DF\omega$ ($DF1$, $DF2$, $DF\underline{\omega}$), then $\mathcal{E}d(\Gamma) \vdash \mathcal{E}d(A) : \mathcal{E}d(B)$ is derivable in $F\omega$ ($F1$, $F2$, $F\underline{\omega}$). Suppose now that

$$A \equiv A_0 \rightarrow_\beta A_1 \rightarrow_\beta A_2 \rightarrow_\beta \dots$$

is a sequence of $\beta$-reductions. By Lemma *3.5.2*, for every $i \geq 1$, either

$$\mathcal{E}d(A_i) \rightarrow_\beta \mathcal{E}d(A_{i+1}), \quad \text{or} \quad \mathcal{E}d(A_i) \equiv \mathcal{E}d(A_{i+1}).$$

Suppose the sequence $A_0 \rightarrow_\beta A_1 \rightarrow_\beta A_2 \rightarrow_\beta \dots$ is infinite. Since $\beta$-reduction in $F\omega$ ($F1$, $F2$, $F\underline{\omega}$) is strongly normalizing, there is an $n$, such that $\mathcal{E}d(A_j) \equiv \mathcal{E}d(A_{j+1})$, for every $j \geq n$. So from step $n$, every step in the infinite sequence $A_0 \rightarrow_\beta A_1 \rightarrow_\beta A_2 \rightarrow_\beta \dots$ corresponds to a reduction of a redex of the form $(\lambda x{:}\phi.\psi)M$. However, since $M$ is a $\lambda$-term, such a reduction cannot create new abstractions of the form $\lambda x{:}\phi.\psi$. Therefore, the number of such abstractions must decrease after every step, and our reduction cannot be infinite. ■

*Corollary 3.5.8* *If $\Gamma \vdash A : B$, then:*

*i) $B$ is strongly normalizing.*

*ii) All predicates in $\Gamma$ are strongly normalizing.*

*iii) The judgment $\Gamma \vdash A : B$ is strongly normalizing.*

*Proof:* *i)* By Lemma *3.2.13*, also $\Gamma \vdash B : s$ is a derivable statement, for some $s$. Hence we can apply the Theorem *3.5.7*, to get the strong normalization of $B$.

*ii)* By Lemma *3.2.3*, also $\Gamma$ is a legal context. We prove the thesis by induction on the length of the context $\Gamma$. The only case to consider is if $\Gamma = \Gamma', c{:}C$, for some $\Gamma', c$, and $C$. Then, by induction, all predicates in $\Gamma'$ are strongly normalizing, and, moreover, by Definition *3.2.2*, also $\Gamma' \vdash C : s$ is a derivable statement, for some $s$. Apply again Theorem *3.5.7*, to get the strong normalization of $C$.

*iii)* By Theorem *3.5.7*, and parts *(i)*,*(ii)*. ∎

# Chapter 4

# Relations between the $\mathcal{TS}$ and the $\mathcal{TAS}$ Cubes

## Introduction

In the previous chapter, we presented the $\mathcal{TAS}$ cube. In this cube, some type assignment systems can be obtained by simply applying a type erasing function to the corresponding typed system in the Barendregt's cube.

From the point of view of logic, in the presence of dependent-types, the existence of an isomorphism between a typed and a type assignment derivation means that the underlined logics of the cube $\mathcal{TAS}'$ are those of the typed cube of Barendregt's.

In [GHR93], it was observed that, perhaps surprisingly, in presence of dependences there no longer exists an isomorphism between corresponding systems of typed and type assignment cubes, in the sense that not every derivation in $\mathcal{TAS}$ is the image under erasure of a derivation in $\mathcal{TS}$.

However, in that paper was conjectured that at least there exists an isomorphism between judgments rather than between derivations, i.e.: a judgment $\Gamma \vdash M : \phi$ is true in one of the type assignment systems if and only if, in the corresponding typed system, a judgment $\Gamma_t \vdash_t M_t : \phi_t$ can be proved such that $\mathcal{E}(\Gamma_t) = \Gamma$, $\mathcal{E}(M_t) \equiv M$, and $\mathcal{E}(\phi_t) \equiv \phi$, where $\mathcal{E}$ is the erasing function of Definition *3.1.2*.

In this chapter where we will focus closely on the differences and similarities between $\mathcal{TS}$ and $\mathcal{TAS}$, and we disprove this conjecture, showing that it is true only for systems without polymorphism. The type assignment systems with polymorphism and dependences ($DF2$ and $DF\omega$, that correspond respectively to $\lambda P2$ and $\lambda P\omega$) are in some sense more powerful than their typed versions. In fact, we prove that there are judgments, provable in one of these systems, that cannot be obtained as erasure of typed judgments. This implies that there are types, inhabited in these systems, that are not erasure of inhabited types in the corresponding typed systems, and, moreover, that a term $M$ can be assigned more types than just those that can be obtained, through erasure, from types belonging to any typed version of $M$.

This result gives then rise to a new question, namely if it is possible to build a cube of type assignment systems that is isomorphic to Barendregt's cube, in the sense that typed and type assignment systems in the corresponding vertices are isomorphic. We solve this problem by defining a cube of type assignment systems $\mathcal{TAS}'$ that enjoy this property. This cube is based on the definition of a new erasing function $\mathcal{E}'$, that coincides with $\mathcal{E}$ when dependences are not present. The main difference between $\mathcal{E}$ and $\mathcal{E}'$ is that, while $\mathcal{E}$ always erases type information in terms, $\mathcal{E}'$ is context dependent and erases type information from a term only if that term does not occur in a type; otherwise it leaves the term unchanged. This cube has the the (somewhat inelegant) property that some type assignment rules explicitly use typed rules of the corresponding typed system in Barendregt's cube. But this seems to be the price to pay for obtaining isomorphism.

This chapter is organized as follows. Section *4.1* is devoted to the study of the relation between the two cubes, where we will disprove the conjecture cited above. In Section *4.2*, we will prove the similarity for systems without polymorphic-types. In Section *4.3* a new erasing function, together with the induced new cube of type assignment systems ($\mathcal{TAS}'$) will be presented. In that section, we will show that the type assignment systems of the $\mathcal{TAS}'$ cube are isomorphic to the systems in Barendregt's cube. The last section is devoted to open problems and future work.

## 4.1 Relations between Systems

In this section, we will focus on the relation between Barendregt's cube and the cube of type assignment systems. To do this, we introduce the notions of *consistency, similarity,* and *isomorphism* between typed systems and type assignment systems. Note that these notions depend on the choice of a particular erasing function.

*Definition 4.1.1 Let $\mathcal{S}_t$ and $\mathcal{S}_u$ be, respectively, a typed and type assignment system, and let $\mathcal{J} : T_t \rightarrow T_u$ be an erasing function.*

*i) We say that $\mathcal{S}_t$ and $\mathcal{S}_u$ are* consistent*, via $\mathcal{J}$, if $\mathcal{J}$ is "sound" with respect to provable judgments, i.e.:*

$$\Gamma_t \vdash_{\mathcal{S}_t} A_t : B_t \textit{ implies } \mathcal{J}(\Gamma_t) \vdash_{\mathcal{S}_u} \mathcal{J}(A_t) : \mathcal{J}(B_t),$$

*where $\mathcal{J}$ is (point-wise) extended to the context $\Gamma$.*

*ii) Systems $\mathcal{S}_t$ and $\mathcal{S}_u$ are* similar*, via $\mathcal{J}$, if they are consistent and, moreover, $\mathcal{J}$ is "complete" with respect to provable judgments, i.e.:*

$$\Gamma \vdash_{\mathcal{S}_u} A : B \textit{ implies that there exists } \Gamma_t,\ A_t \textit{ and } B_t, \textit{ such that}$$
$$\Gamma \vdash_{\mathcal{S}_t} A_t : B_t \textit{ and } \mathcal{J}(\Gamma_t) = \Gamma,\ \mathcal{J}(A_t) \equiv A \textit{ and } \mathcal{J}(B_t) \equiv B.$$

*iii) Let $\mathcal{D}er_t$ and $\mathcal{D}er_u$ be the sets of all derivations in $\mathcal{S}_t$ and $\mathcal{S}_u$. Systems $\mathcal{S}_t$ and $\mathcal{S}_u$ are* isomorphic*, via $\mathcal{J}$, if and only if there are $\mathcal{F}$: $\mathcal{D}er_t \rightarrow \mathcal{D}er_u$ and $\mathcal{G}$: $\mathcal{D}er_u \rightarrow \mathcal{D}er_t$, such that:*

*a) If $\mathcal{D}$: $\Gamma \vdash_{\mathcal{S}_t} A : B$, then $\mathcal{F}(\mathcal{D}) : \mathcal{J}(\Gamma) \vdash_{\mathcal{S}_u} \mathcal{J}(A_t) : \mathcal{J}(B_t)$.*

*b) $\mathcal{F} \circ \mathcal{G}$ and $\mathcal{G} \circ \mathcal{F}$ are the identity on $\mathcal{D}er_u$ and $\mathcal{D}er_t$, respectively.*

*c) Both $\mathcal{F}$ and $\mathcal{G}$ preserve the structure of derivations, (i.e., the tree obtained from a derivation by erasing all judgments, but not the names of the rules).*

Notice that the definition of isomorphism expresses more than just soundness and completeness of $\mathcal{J}$. Notice, moreover, that $\mathcal{F}$ is not defined by induction on derivations, a detail that will be of importance in Section *4.3*. Finally, notice that, in the previous definition, $\mathcal{S}_u$ is not assumed to be obtained from $\mathcal{S}_t$ through the application of $\mathcal{J}$ to the rules of $\mathcal{S}_t$.

Figure *4.1* shows the various functions between typed and untyped systems of $\lambda$-calculi that realize the above relations between typed and untyped judgments and derivations.

The definition of isomorphism between two systems was already given in [GHR93], but in a less general way. We have defined isomorphism with respect to an erasing function $\mathcal{J}$; the definition of isomorphism in [GHR93] used a *fixed* function. To be more precise, two systems are isomorphic according to the definition in [GHR93], if they are isomorphic in the sense of Definition *4.1.1* with respect to the function $\mathcal{F}$ that is defined as follows: $\mathcal{F}(\mathcal{D})$ is obtained from $\mathcal{D}$ by applying the erasing function $\mathcal{E}$ of

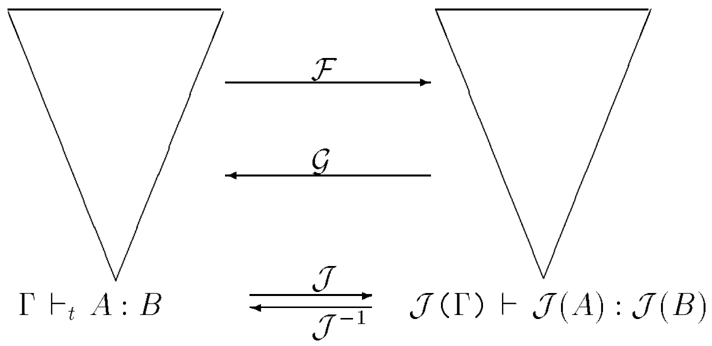



Figure 4.1: Functions between Typed and Untyped Judgments and Derivations

Definition *3.1.2* to all terms in $\mathcal{D}$; by abuse of notation, we denote $\mathcal{F}(\mathcal{D})$ by $\mathcal{E}(\mathcal{D})$.

The following proposition states that, for the $\mathcal{TS}$ and $\mathcal{TAS}$ cubes, the two notions of isomorphism coincide.

*Lemma 4.1.2 Let $\mathcal{S}_t$ and $\mathcal{S}_u$ be systems in corresponding vertices of $\mathcal{TS}$ and $\mathcal{TAS}$ cube respectively, and suppose they are isomorphic through the functions $\mathcal{F}$ and $\mathcal{G}$. Then $\mathcal{F}(\mathcal{D}) = \mathcal{E}(\mathcal{D})$, for every typed derivation $\mathcal{D}$.*

*Proof:* By easy induction. ∎

To show consistency of our systems, we need the following lemma that shows that type erasure and substitution behave well together, and that type erasure does not affect $\beta$-reduction.

*Lemma 4.1.3 i) $\mathcal{E}(A[B/b]) \equiv \mathcal{E}(A)[\mathcal{E}(B)/b]$.*
*ii) If $A =_\beta B$, then $\mathcal{E}(A) =_\beta \mathcal{E}(B)$.*

*Proof:* *i)* By induction on the Definition *2.1.3* of substitution, using Definition *3.1.2* of the erasing function.

$[A \equiv a \ \& \ a \not\equiv b]$: then $\mathcal{E}(a[B/b]) \equiv a \equiv \mathcal{E}(a)[\mathcal{E}(B)/b]$.

$[A \equiv b]$: then $\mathcal{E}(b[B/b]) \equiv \mathcal{E}(B) \equiv \mathcal{E}(b)[\mathcal{E}(B)/b]$.

$[A \equiv A_1A_2]$: then $\mathcal{E}((A_1A_2)[B/b]) \equiv \mathcal{E}(A_1[B/b])\mathcal{E}(A_2[B/b]) \equiv (IH)$
$\mathcal{E}(A_1)[\mathcal{E}(B)/b]\mathcal{E}(A_2)[\mathcal{E}(B)/b] \equiv \mathcal{E}(A_1A_2)[\mathcal{E}(B)/b]$.

$[A \equiv \lambda a{:}A_1.A_2]$: then $\mathcal{E}((\lambda a{:}A_1.A_2)[B/b]) \equiv \lambda a{:}\mathcal{E}(A_1[B/b]).\mathcal{E}(A_2[B/b]) \equiv (IH)$ $\lambda a{:}\mathcal{E}(A_1)[\mathcal{E}(B)/b].\mathcal{E}(A_2)[\mathcal{E}(B)/b] \equiv \mathcal{E}(\lambda a{:}A_1.A_2)[\mathcal{E}(B)/b]$.

$[A \equiv \lambda x.M]$: then $\mathcal{E}((\lambda x.M)[B/b]) \equiv \lambda x.\mathcal{E}(M[B/b]) \equiv (IH) \lambda x.\mathcal{E}(M)[\mathcal{E}(B)/b] \equiv$ $\mathcal{E}(\lambda x.M)[\mathcal{E}(B)/b]$.

$[A \equiv \Pi a{:}A_1.A_2]$: then $\mathcal{E}((\Pi a{:}A_1.A_2)[B/b]) \equiv \Pi a{:}\mathcal{E}(A_1[B/b]).\mathcal{E}(A_2[B/b]) \equiv (IH)$ $\Pi a{:}\mathcal{E}(A_1)[\mathcal{E}(B)/b].\mathcal{E}(A_2)[\mathcal{E}(B)/b] \equiv \mathcal{E}(\Pi a{:}A_1.A_2)[\mathcal{E}(B)/b]$.

*ii)* By induction on the definition of $=_\beta$ . We just consider the case when $A$ is a redex, by induction on $\twoheadrightarrow_\beta$ ; the complete proof follows easily by induction.

$[A \equiv (\lambda x.M)$ and $B \equiv M[N/x]]$: then $\mathcal{E}((\lambda x.M)N) \equiv \mathcal{E}(\lambda x.M)\mathcal{E}(N) \equiv$ $(\lambda x.\mathcal{E}(M))\mathcal{E}(N) \to_\beta \mathcal{E}(M)[\mathcal{E}(N)/x]$ which is equal, by part *(i)*, to $\mathcal{E}(M[N/x])$.

$[A \equiv (\lambda x{:}\phi.\psi)M$ and $B \equiv \psi[M/x]]$: then $\mathcal{E}((\lambda x{:}\phi.\psi)M) \equiv \mathcal{E}(\lambda x{:}\phi.\psi)\mathcal{E}(M) \equiv$ $(\lambda x{:}\mathcal{E}(\phi).\mathcal{E}(\psi))\mathcal{E}(M) \to_\beta \mathcal{E}(\psi)[\mathcal{E}(M)/x]$ which is equal, by part *(i)*, to $\mathcal{E}(\psi[M/x])$.

$[A \equiv (\lambda\alpha{:}K.\phi)\psi$ and $B \equiv \phi[\psi/\alpha]]$: then $\mathcal{E}((\lambda\alpha{:}K.\phi)\psi) \equiv \mathcal{E}(\lambda\alpha{:}K.\phi)\mathcal{E}(\psi) \equiv$ $(\lambda\alpha{:}\mathcal{E}(K).\mathcal{E}(\phi))\mathcal{E}(\psi) \to_\beta \mathcal{E}(\phi)[\mathcal{E}(\psi)/\alpha]$ which is equal, by part *(i)*, to $\mathcal{E}(\phi[\psi/\alpha])$. ■

The fundamental relationship between the judgments of the $\mathcal{TS}$ and the $\mathcal{TAS}$ are summarized as follows:

*Theorem 4.1.4 Let $\mathcal{S}_t$ and $\mathcal{S}_u$ be systems in corresponding vertices of $\mathcal{TS}$ and $\mathcal{TAS}$ cube respectively.*

*i) Systems $\mathcal{S}_t$ and $\mathcal{S}_u$ are consistent.*

*ii) If $\mathcal{S}_t$ and $\mathcal{S}_u$ do not contain* Dependencies *as subset rules, then $\mathcal{S}_t$ and $\mathcal{S}_u$ are isomorphic.*

*iii) If the assumption of part (ii) is not satisfied, then $\mathcal{S}_t$ and $\mathcal{S}_u$ are not isomorphic.*

*Proof:* See [GHR93]. The proof of parts *(i)* and *(ii)* uses Lemma *4.1.3*. ■

So all typed systems $\mathcal{S}_t$ are consistent with respect to the corresponding untyped systems $\mathcal{S}_u$. In addition, applying the erasing function to all judgments used in a derivation in $\mathcal{S}_t$ yields a correct derivation in $\mathcal{S}_u$. This implies that $\mathcal{S}_t$ and $\mathcal{S}_u$ are similar. Unfortunately, systems with dependences need not to be isomorphic, as we show below.

Although a counterexample for proving Theorem *4.1.4 (iii)* can be found in [GHR93], we will give here another, both for the convenience of the reader, and because it is an easier example (it does not make use of the (*Conv*) rule).

*Example 4.1.5* Consider the following derivation in *DF*1 (for reasons of readability, we use the notation $A{\rightarrow}B$ for $\Pi a{:}A.B$, when $a$ does not occur in $B$).

Let $\mathbf{O}$ stand for the $\lambda$-term $(\lambda xy.y)$ and let $\mathbf{I}$ denote the identity $(\lambda x.x)$. Let $\Gamma$ be a context consisting of the following declarations:

$$\alpha{:}*, \beta{:}*, \gamma{:}*, a{:}(\gamma{\rightarrow}\gamma){\rightarrow}*\,.$$

Clearly, we can derive both $\Gamma \vdash \mathbf{I} : \alpha{\rightarrow}\alpha$ and $\Gamma \vdash \mathbf{O} : (\alpha{\rightarrow}\alpha){\rightarrow}\gamma{\rightarrow}\gamma$; combining these gives $\Gamma \vdash \mathbf{OI} : \gamma{\rightarrow}\gamma$, with which we can derive:

$$\mathcal{D}_1\!:\ \Gamma \vdash a(\mathbf{OI}) : *.$$

By applying rules (*Weak*) and (*C-F$_C$*), we get $\Gamma \vdash a(\mathbf{OI}){\rightarrow}\gamma : *$. Applying rule (*Proj*) gives $\Gamma, u{:}a(\mathbf{OI}){\rightarrow}\gamma \vdash u : a(\mathbf{OI}){\rightarrow}\gamma$, and by applying a (*Weak*) (using again the derivation $\mathcal{D}_1$), we get:

$$\mathcal{D}_2\!:\ \Gamma, u{:}a(\mathbf{OI}){\rightarrow}\gamma, v{:}a(\mathbf{OI}) \vdash u : a(\mathbf{OI}){\rightarrow}\gamma.$$

On the other hand, we can also derive $\Gamma \vdash \mathbf{I} : \beta{\rightarrow}\beta$ and $\Gamma \vdash \mathbf{O} : (\beta{\rightarrow}\beta){\rightarrow}\gamma{\rightarrow}\gamma$, which can be used, as above, to obtain $\Gamma \vdash a(\mathbf{OI}){\rightarrow}\gamma : *$. Then we can easily build the following derivation $\mathcal{D}_3$:

$$\cfrac{\cfrac{\Gamma \vdash a(\mathbf{OI}) : * \quad \Gamma \vdash a(\mathbf{OI}){\rightarrow}\gamma : * \quad u \notin \mathcal{D}om\,(\Gamma)}{\Gamma, u{:}a(\mathbf{OI}){\rightarrow}\gamma \vdash a(\mathbf{OI}) : * \quad v \notin \mathcal{D}om\,(\Gamma, u{:}a(\mathbf{OI}){\rightarrow}\gamma)}(\mathit{Weak})}{\Gamma, u{:}a(\mathbf{OI}){\rightarrow}\gamma, v{:}a(\mathbf{OI}) \vdash v : a(\mathbf{OI})}(\mathit{Proj})$$

Thus, using derivations $\mathcal{D}_2$ and $\mathcal{D}_3$, and applying rule ($E$), we conclude with:

$$\mathcal{D}_4 : \frac{\Gamma, u{:}a(\mathbf{OI}){\rightarrow}\gamma, v{:}a(\mathbf{OI}) \vdash u : a(\mathbf{OI}){\rightarrow}\gamma \quad \Gamma, u{:}a(\mathbf{OI}){\rightarrow}\gamma, v{:}a(\mathbf{OI}) \vdash v : a(\mathbf{OI})}{\Gamma, u{:}a(\mathbf{OI}){\rightarrow}\gamma, v{:}a(\mathbf{OI}) \vdash uv : \gamma}$$

The above described derivation, although legal in *DF*1, is not an erasure of any derivation in the fully typed system $\lambda P$. To see this, note that we used two different types for different occurrences of the $\lambda$-term $\mathbf{I}$ (and thus also for different occurrences of $\mathbf{O}$) to obtain the two derivations $\mathcal{D}_2$ and $\mathcal{D}_3$. The expression $\mathbf{OI}$, however, is free of types, so the final typing for the application $uv$ is correct. But if we want to obtain a correct derivation in $\lambda P$ of which $\mathcal{D}_4$ is the erasure, we have to assign *the same type* to the occurrences of $\mathbf{I}$ in the types of $u$ and $v$.

More precisely, assume that $\mathcal{D}_4$ is obtained by erasure of a typed derivation $\mathcal{D}'_4$, such that:

$$\mathcal{D}'_4\text{: } \Gamma_t, u{:}aM_t{\to}\gamma, v{:}aM_t \vdash_t uv : \gamma,$$

for suitable $\Gamma_t$ and $M_t$, such that $M_t$ is a typed $\lambda$-term of type $\gamma{\to}\gamma$, and $\mathcal{E}(M_t) \equiv \mathbf{OI}$. The latter implies that:

$$M_t \equiv (\lambda\alpha_1{:}K_1 \ldots \lambda\alpha_n{:}K_n.\mathbf{O}_t\mathbf{I}_t)\phi_1 \cdots \phi_m,$$

for some $K_1, \ldots, K_n, \phi_1, \ldots, \phi_n$, and $n, m \geq 0$, and some $\mathbf{O}_t$,$\mathbf{I}_t$, such that $\mathcal{E}(\mathbf{O}_t) \equiv \mathbf{O}$ and $\mathcal{E}(\mathbf{I}_t) \equiv \mathbf{I}$. But the fact that $M_t$ must have type $\gamma{\to}\gamma$ implies that $n = m = 0$, and so $M_t \equiv \mathbf{O}_t\mathbf{I}_t$. Since $\mathcal{D}_4$ is obtained from $\mathcal{D}'_4$ by erasure, there must be two subderivations of $\mathcal{D}'_4$ proving, respectively,

$$\Gamma_t \vdash_t \mathbf{O}_t : (\alpha{\to}\alpha){\to}\gamma{\to}\gamma, \quad \text{and} \quad \Gamma_t \vdash_t \mathbf{O}_t : (\beta{\to}\beta){\to}\gamma{\to}\gamma,$$

such that $\mathcal{E}(\mathbf{O}_t) \equiv \mathbf{O}$. But this is not possible, since this would imply that:

$$\mathbf{O}_t \equiv \lambda x{:}\alpha{\to}\alpha.\lambda y{:}\gamma.y, \quad \text{and} \quad \mathbf{O}_t \equiv \lambda x{:}\beta{\to}\beta.\lambda y{:}\gamma.y,$$

at the same time, while $\alpha$ and $\beta$ are different constructor variables.

*Remark 4.1.6* Notice however, that we *can* derive $\Gamma_t, u{:}a(\mathbf{O}_t\mathbf{I}_t){\to}\gamma, v{:}a(\mathbf{O}_t\mathbf{I}_t) \vdash_t uv : \gamma$, because in constructing a derivation we are not forced to construct different types $\alpha{\to}\alpha$ and $\beta{\to}\beta$ for $\mathbf{I}_t$, but are free to choose $\alpha \equiv \beta$; therefore, this example is not a counterexample against similarity.

After the negative result of Theorem *4.1.4 (iii)*, it is natural to ask if the corresponding systems in the $\mathcal{TS}$ and $\mathcal{TAS}$ cubes are at least similar. This property will be shown to hold in Theorem *4.2.5*, but only for those systems with dependences that are without polymorphism, namely, for $DF1$ versus $\lambda P$, and for $F\underline{\omega}$ versus $\lambda\underline{\omega}$. Unfortunately, adding polymorphism makes a difference: the systems with both polymorphism and dependences are not similar.

*Theorem 4.1.7 Let $\mathcal{S}_t$ be either $\lambda P2$, or $\lambda P\omega$, and let $\mathcal{S}_u$ be, respectively, DF2 and DF$\omega$. Then $\mathcal{S}_t$ and $\mathcal{S}_u$ are not similar.*

*Proof:* As a counterexample, we show a derivable judgment of $DF2$, that cannot be obtained as an erasure of any derivable judgment in $\lambda P2$. Let $\Gamma_0$ be a context consisting of the following declarations:

$$\Gamma_0 = \alpha{:}*, \beta{:}*, \gamma{:}*, \delta{:}*, \epsilon{:}\beta{\to}*, u : \Pi\eta{:}*.((\eta{\to}\eta){\to}\alpha){\to}\beta, x{:}\alpha, y{:}\gamma, z{:}\delta.$$

Let $\mathbf{K}$ denote the $\lambda$-term $(\lambda xy.x)$, and let the untyped $\lambda$-terms $M$, $M^0$ and $M^1$ be defined by:

$$M \equiv u(\lambda f.x), \quad M^0 \equiv u(\lambda f.\mathbf{K}x(fy)), \quad \text{and} \quad M^1 \equiv u(\lambda f.\mathbf{K}x(fz)).$$

Clearly, both $M^0$ and $M^1$ $\beta$-reduce to $M$, and all these terms can correctly be assigned the type $\beta$ in the context $\Gamma_0$. Thus, we can assert:

$$\Gamma_0 \vdash \epsilon M^0{\to}\alpha : *, \quad \text{and} \quad \Gamma_0 \vdash \epsilon M^1 : *.$$

Then apply the rules (*Proj*), (*Conv*), and (*Weak*) to the first judgment as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma_0 \vdash \epsilon M^0{\to}\alpha : * \qquad p \notin \mathcal{D}om\,(\Gamma_0)}{\Gamma_0, p{:}\epsilon M^0{\to}\alpha \vdash p : \epsilon M^0{\to}\alpha}\ (Proj) \qquad \Gamma_0 \vdash \epsilon M{\to}\alpha : * \quad \epsilon M^0{\to}\alpha =_\beta \epsilon M{\to}\alpha}{\Gamma_0, p{:}\epsilon M^0{\to}\alpha \vdash p : \epsilon M{\to}\alpha}\ (Conv) \qquad \Gamma_0, p{:}\epsilon M^0{\to}\alpha \vdash \epsilon M^1{\to}\alpha : * \quad q \notin \mathcal{D}om\,(\Gamma_0, p{:}\epsilon M^1)}{\Gamma_0, p{:}\epsilon M^0{\to}\alpha, q{:}\epsilon M^1 \vdash p : \epsilon M{\to}\alpha}\ (Weak)$$

and the rules (*Weak*), (*Proj*), and (*Conv*) to the second judgment as follows:

$$\dfrac{\Gamma_0 \vdash \epsilon M^1 : * \qquad \Gamma_0 \vdash \epsilon M^0 {\rightarrow} \alpha : * \qquad p \notin \mathcal{D}om\,(\Gamma_0)}{\Gamma_0, p{:}\epsilon M^0{\rightarrow}\alpha \vdash \epsilon M^1 : *}\ (Weak)$$

$$\dfrac{q \notin \mathcal{D}om\,(\Gamma_0, p{:}\epsilon M^0{\rightarrow}\alpha)}{\Gamma_0, p{:}\epsilon M^0{\rightarrow}\alpha, q{:}\epsilon M^1 \vdash q : \epsilon M^1}\ (Proj)$$

$$\dfrac{\Gamma_0, p{:}\epsilon M^0{\rightarrow}\alpha, q{:}\epsilon M^1 \vdash \epsilon M : * \qquad \epsilon M^1 =_\beta \epsilon M}{\Gamma_0, p{:}\epsilon M^0{\rightarrow}\alpha, q{:}\epsilon M^1 \vdash q : \epsilon M}\ (Conv)$$

Finally, we can apply an $(E)$ rule, and obtain:

$$\dfrac{\Gamma \vdash p : \epsilon M{\rightarrow}\alpha \quad \Gamma \vdash q : \epsilon M}{\Gamma \vdash pq : \alpha}\ (E)$$

where $\Gamma = \Gamma_0, p{:}\epsilon M^0{\rightarrow}\alpha, q{:}\epsilon M^1$. We claim that the above judgment cannot be obtained as an erasure of any judgment derivable in $\lambda P2$, i.e., that we cannot have $\Gamma_t$,$N_t$, and $\phi_t$, such that $\Gamma_t \vdash_t N_t : \phi_t$ is derivable, with $\mathcal{E}(\Gamma_t) = \Gamma$, $\mathcal{E}(N_t) \equiv pq$, and $\mathcal{E}(\phi_t) \equiv \alpha$.

To justify our claim, let us assume the opposite. First note that $\phi_t \equiv \alpha$, since no terms occur in $\alpha$ (the erasing function can only modify types containing occurrences of terms, in which case the result must also contain terms). Similarly, $\Gamma_t$ may differ from $\Gamma$ only in the declarations of $p$ and $q$, which must be of the form:

$$p{:}\epsilon M_t^0{\rightarrow}\alpha, \quad \text{and} \quad q{:}\epsilon M_t^1,$$

for some $M_t^0$, and $M_t^1$ such that $\mathcal{E}(M_t^0) \equiv M^0$, and $\mathcal{E}(M_t^1) \equiv M^1$.
We can also assume that $N_t$ is of the form $P_tQ_t$, where $\mathcal{E}(P_t) \equiv p$ and $\mathcal{E}(Q_t) \equiv q$ (otherwise we consider an appropriate subterm of $N_t$ instead). Since $P_t$ is applied to $Q_t$, and the type of $P_tQ_t$ is $\alpha$, it follows that $P_t$ has a type of the form $\epsilon M_t^{0'} \rightarrow \alpha$ (for some $M_t^0$), where $\mathcal{E}(M_t^{0'}) \equiv M^0$. Similarly, $Q_t$ has a type of the form $\epsilon M_t^{1'}$, (for some $M_t^1$), where $\mathcal{E}(M_t^{1'}) \equiv M^1$. In order to make the application well-typed (after a possible series of applications of rule $(Conv)$), it must be that $M_t^{0'} =_\beta M_t^{1'}$.
It follows that we have $\beta$-convertible $\lambda$-terms $M_t^{0'}$ and $M_t^{1'}$, that erase to $M^0$ and $M^1$, respectively, and both are of type $\beta$. Without loss of generality, we can assume that these $\lambda$-terms have no $\beta$-redexes involving polymorphic abstraction and/or application, and thus we may write:

$$M_t^{0\prime} \equiv u\gamma(\lambda f{:}\gamma{\to}\gamma.\mathbf{K}_t x(fy)), \quad \text{and} \quad M_t^{1\prime} \equiv u\delta(\lambda f{:}\delta{\to}\delta.\mathbf{K}'_t x(fz)),$$

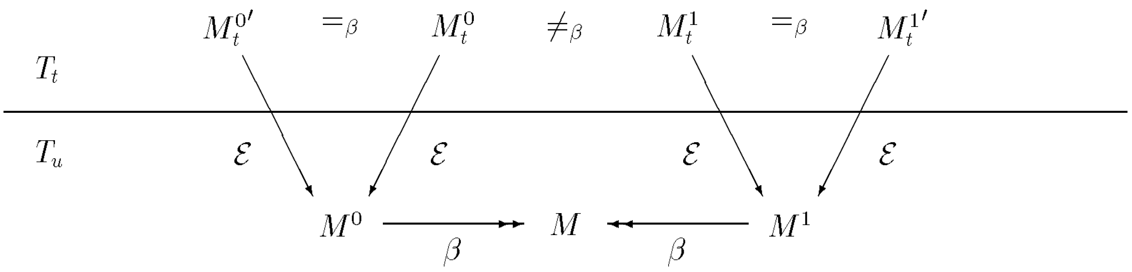
for some $\mathbf{K}_t$, and $\mathbf{K}'_t$, such that $\mathcal{E}(\mathbf{K}_t) \equiv \mathbf{K}$, and $\mathcal{E}(\mathbf{K}'_t) \equiv K$. The types of $f$ used above are forced by the applications $fy$ and $fz$. Note that the type of $f$ may not be externally quantified: if the polymorphic variable $u$ is applied to a type $\mu$, then $f$ must be of type $\mu{\to}\mu$, and no constructor application $f\phi$ is possible. So, the $\beta$-normal forms of these terms are as follows:

$$M_t^{0\prime} \twoheadrightarrow_\beta u\gamma(\lambda f{:}\gamma{\to}\gamma.x), \quad \text{and} \quad M_t^{1\prime} \twoheadrightarrow_\beta u\delta(\lambda f{:}\delta{\to}\delta.x).$$

But these $\beta$-normal forms are different, and this contradicts the previous claim that:

$$M_t^{0\prime} =_\beta M_t^{1\prime}.$$

The following picture graphically summarizes this proof.




∎

*Remark 4.1.8* In the above system, we have shown the existence of two typed $\lambda$-terms, namely $M_t^{0\prime}$ and $M_t^{1\prime}$, and of a context $\Gamma$ and type $\beta$, such that:

*i)* $M_t^{0\prime} \neq_\beta M_t^{1\prime}$;

*ii)* $\mathcal{E}(M_t^{0\prime}) =_\beta \mathcal{E}(M_t^{1\prime})$;

*iii)* $\Gamma \vdash_t M_t^{0\prime} : \beta$ and $\Gamma \vdash_t M_t^{1\prime} : \beta$.

This allows the construction of the counterexample to the similarity.

The heart of the counterexample lies in both the polymorphic rules, and the fact that it is possible to abstract with respect to variables not occurring in the body. In fact, in the proof above, the polymorphic behaviour of the variable $u$ makes that this term can be applied to both the terms $\lambda f{:}\gamma{\to}\gamma.\mathbf{K}_t x(fy)$ and $\lambda f{:}\delta{\to}\delta.\mathbf{K}'_t x(fz)$. Also the use of the $\lambda$-terms $\mathbf{K}_t$ and $\mathbf{K}'_t$ is essential in order to obtain the correct final typing;

because $\mathbf{K}_t$ and $\mathbf{K}'_t$ are cancelling terms that both erase to $\mathbf{K}$, the type assumed for the variable $f$ has no effect on the type of the full terms $M_t^{0'}$ and $M_t^{1'}$.

It is natural to ask if this result allows some comparison between the power (with respect to typability and inhabitation) of the corresponding systems, respectively $DF2$ and $\lambda P2$, $DF\omega$ and $\lambda P\omega$. Recall that a (closed) type $\phi$ is *inhabited* in a system $\mathcal{S}$, if and only if there is a (closed) term $M$ such that $\varepsilon \vdash M : \phi$.

The following corollary states that the set of types inhabited in $\mathcal{S}_u$ includes properly those types that are obtained through $\mathcal{E}$ from inhabited types in $\mathcal{S}_t$, and states that also the set of types assignable to a term in $\mathcal{S}_u$ is larger than its corresponding set in $\mathcal{S}_t$.

*Corollary 4.1.9 Let $\mathcal{S}_t$ be either $\lambda P2$ or $\lambda P\omega$, and $\mathcal{S}_u$ be, respectively, $DF2$ or $DF\omega$.*

*i)* $\{\phi \mid \exists\, M\ [\varepsilon \vdash_{\mathcal{S}_u} M : \phi]\} \supsetneq \{\phi \mid \exists\, \phi_t, M_t\ [\varepsilon \vdash_{\mathcal{S}_t} M_t : \phi_t\ \&\ \mathcal{E}(\phi_t) = \phi]\}$.

*ii) Let $M$ be a closed term. Then:*
$\{\phi \mid \varepsilon \vdash_{\mathcal{S}_u} M : \phi\} \supsetneq \{\phi \mid \exists\, \phi_t, M_t\ [\varepsilon \vdash_{\mathcal{S}_t} M_t : \phi_t\ \&\ \mathcal{E}(M_t) = M\ \&\ \mathcal{E}(\phi_t) = \phi]\}$.

*Proof:* *i)* The inclusion follows immediately from the fact that $\mathcal{S}_t$ and $\mathcal{S}_u$ are consistent via $\mathcal{E}$. To prove that the inclusion is proper, take the derivation for $\Gamma \vdash pq : \alpha$ as constructed in the proof of Theorem *4.1.7*. Clearly, $\varepsilon \vdash \lambda xyzpq.pq : \phi$, where $\phi$ is the closure of $\alpha$ with respect to the context $\Gamma$. Then there is a derivation proving $\xi{:}\phi{\to}* \vdash \xi(\lambda xyzpq.pq) : *$, and, therefore, $\varepsilon \vdash \Pi\xi{:}\phi{\to}*.\xi(\lambda xyzpq.pq) : *$. Let $\psi$ be short for $\Pi\xi{:}\phi{\to}*.\xi(\lambda xyzpq.pq)$, then obviously $\psi{\to}\psi$ is a type inhabited in $\mathcal{S}_u$ by the term $\lambda w.w$, while it is not the erasure of an inhabited type in $\mathcal{S}_t$.

*ii)* Also in this case, the inclusion follows from the consistency via $\mathcal{E}$ between $\mathcal{S}_u$ and $\mathcal{S}_t$. To prove that the inclusion is proper, it is sufficient to observe that, for every closed term $M$ typable in $\mathcal{S}_u$, there is a typing for $M$ of the shape $\alpha{:}* \vdash M : \xi[\alpha]$. Then $\varepsilon \vdash M : \Pi\alpha{:}*.\xi[\alpha]$, and, by rule $(E_K)$, $\varepsilon \vdash M : \xi[\psi]$, where $\psi$ is defined as in part *(i)*. Clearly this type cannot be the erasure of an inhabited type in $\mathcal{S}_t$. ■

In the next section, we will prove that, for systems without polymorphism, the corresponding systems in $\mathcal{TS}$ and $\mathcal{TAS}$ are similar.

## 4.2 Systems without polymorphism

In case polymorphism is not permitted, we can prove that the corresponding $\mathcal{TS}$ and $\mathcal{TAS}$ are similar. In what follows, the symbol $\vdash$ denotes $\vdash_{\mathcal{S}}$, for $S \in \{F1, F\underline{\omega}, DF1, DF\underline{\omega}\}$, while $\vdash_t$ refers to the corresponding $\mathcal{TS}$. That is, we consider only systems without polymorphism. It is important to point out that, restricting the systems in this way, the derivation rules $(I_K)$, $(E_K)$ and $(C\text{-}F_K)$ are eliminated. Moreover, the syntax of terms is limited by no longer allowing for terms of the shape $M\phi$, $\lambda\alpha{:}K.\phi$, and $\Pi\alpha{:}K.\phi$ of Definition *2.4.1* and *3.1.1*.

Before we come to the main proof, we need some preliminary definitions and lemmas; the next lemma says that the erasing function does not change the shape of the term.

*Lemma 4.2.1* *i) If $\Gamma \vdash_t A : B$ and $A$ is in $\beta$-normal form, then so is $\mathcal{E}(A)$.*

*ii) If $\mathcal{E}(A)$ is of the form $*$, variable, application, abstraction, or product, then so is $A$.*

*Proof:* Direct, using Definition *3.1.1*, and Definition *3.1.2*. ∎

The following lemma formulates that, in the absence of polymorphism, the erasing function $\mathcal{E}$ is injective on terms in normal form that can be assigned the same predicate.

*Lemma 4.2.2* *Let $\Gamma \vdash_t A_1 : B$ and $\Gamma \vdash_t A_2 : B$, and let both $A_1$ and $A_2$ be $\beta$-normal forms. If $\mathcal{E}(A_1) \equiv \mathcal{E}(A_2)$, then $A_1 \equiv A_2$.*

*Proof:* By induction on the structure of terms.

[Variable or sort constant]: this case is immediate.

[Abstractions]: We only consider the case that $A_1$ and $A_2$ are $\lambda$-terms, since the other are essentially the same. Let $A_1 \equiv \lambda x{:}\phi_1.M_1$, with $\phi_1$, and $M_1$ in $\beta$-normal form. By Property *2.1.17*, we have $B =_\beta \Pi x{:}\phi_1.\psi_1$, for some $\psi_1$, such that $\Gamma, x{:}\phi_1 \vdash_t M_1 : \psi_1$. Since $\mathcal{E}(A_1) \equiv \mathcal{E}(A_2) \equiv \lambda x.\mathcal{E}(M_1)$, it must be that $A_2 \equiv \lambda x{:}\phi_2.M_2$, for some $\phi_2$, and $M_2$ in $\beta$-normal form. Again by Property *2.1.17*, we have that $\Gamma, x{:}\phi_2 \vdash_t M_2 : \psi_2$, for some $\psi_2$, such that $\Pi x{:}\phi_1.\psi_1 =_\beta B =_\beta \Pi x{:}\phi_2.\psi_2$. By the Church-Rosser Property *2.1.13*, this implies $\phi_1 =_\beta \phi_2$ and $\psi_1 =_\beta \psi_2$. Since $\phi_1$ and $\phi_2$ are $\beta$-normal forms, they must be identical. Let $\phi$ denote both $\phi_1$ and $\phi_2$. By Property *2.1.18*, we get $\Gamma, x{:}\phi \vdash \psi_1 : s$. Then we can apply a $(Conv)$ rule to obtain:

$$\frac{\Gamma, x{:}\phi \vdash M_2 : \psi_2 \quad \Gamma, x{:}\phi \vdash \psi_1 : s \quad \psi_1 =_\beta \psi_2}{\Gamma, x{:}\phi \vdash M_2 : \psi_1} \; (Conv)$$

Since $\mathcal{E}(A_1) \equiv \mathcal{E}(A_2)$, by Definition *3.1.2*, we get $\mathcal{E}(M_1) \equiv \mathcal{E}(M_2)$. Finally, we can apply the induction hypothesis to get $M_1 \equiv M_2$, so the thesis $A_1 \equiv A_2$ follows.

[Applications]: Since $A$ is in $\beta$-normal form, there are $C_1, \ldots, C_n$ in $\beta$-normal forms and a variable $a$ such that $A_1 \equiv aC_1 \ldots C_n$. Since $a \in \mathcal{FV}(A_1)$, by Lemma *3.2.7 (i)*, also $a{:}D \in \Gamma$, for some $D$. By Property *2.1.17*, there are $F_1, \ldots,$ $F_n$, and $G$, such that:

*a)* $D =_\beta \Pi_{i=1}^n b_i{:}F_i.G$;

*b)* $B =_\beta G[C_1/b_1] \ldots [C_n/b_n]$;

*c)* $\Gamma \vdash_t C_i : F_i[C_1/b_1] \ldots [C_{i-1}/b_{i-1}]$, for all $1 \leq i \leq n$.

Since $\mathcal{E}(A_1) \equiv \mathcal{E}(A_2)$, we have $A_2 \equiv aC'_1 \ldots C'_n$, for some $C'_1, \ldots, C'_n$, such that $\mathcal{E}(C_i) \equiv \mathcal{E}(C'_i)$. Repeating for $A_2$ the same argument as above, we get:

*d)* $D =_\beta \Pi_{i=1}^n b_i{:}F'_i.G'$;

*e)* $B =_\beta G'[C'_1/b_1] \ldots [C'_n/b_n]$;

*f)* $\Gamma \vdash_t C'_i : F'_i[C'_1/b_1] \ldots [C'_{i-1}/b_{i-1}]$, for all $1 \leq i \leq n$.

We have $\Pi_{i=1}^n b_i{:}F_i.G =_\beta D =_\beta \Pi_{i=1}^n b_i{:}F'_i.G'$. Using the Church-Rosser Property *2.1.13*, we get $G =_\beta G'$, and $F_i =_\beta F'_i$, for all $1 \leq i \leq n$. It remains to show that $C_i \equiv C'_i$, for all $i$. Note that, by induction, and by the above properties *(c)* and *(f)*, the terms $C_i$ and $C'_i$ have the same type (the same kind), and are in $\beta$-normal form. Since $C_i$ is a subterm of $A_1$, by induction, we obtain $C_i \equiv C'_i$.

[Products]: Let $A_1 \equiv \Pi c{:}C_1.D_1$, for some $c, C_1$, and $D_1$. Since $\mathcal{E}(A_1) \equiv \mathcal{E}(A_2)$, we have $A_2 \equiv \Pi c{:}C_2.D_2$, for some $c, C_2$, and $D_2$. By Property *2.1.17*, we have $\Gamma, c{:}C_1 \vdash_t D_1 : s$, and $\Gamma, c{:}C_2 \vdash_t D_2 : s$. Since the contexts are legal, we have also $\Gamma \vdash_t C_1 : s$, and $\Gamma \vdash_t C_2 : s$. By induction, we obtain $C_1 \equiv C_2$. Thus, $\Gamma, c{:}C_1 \vdash_t D_2 : s$ and, once more by induction, we get $D_1 \equiv D_2$, as desired. ∎

Using this result, in the following lemma we will prove that, in the absence of polymorphism, the erasing function $\mathcal{E}$ is injective on terms, modulo $\beta$-equality, that can be assigned the same predicate.

*Lemma 4.2.3 Let* $\Gamma \vdash_t A_1 : B$ *and* $\Gamma \vdash_t A_2 : B$. *If* $\mathcal{E}(A_1) =_\beta \mathcal{E}(A_2)$, *then* $A_1 =_\beta A_2$.

*Proof:* By Theorem *2.1.19*, both terms are strong normalizable, so let $A'_1$ be the $\beta$-normal form of $A_1$, and let $A'_2$ be the $\beta$-normal form of $A_2$. By Lemma *4.1.3 (ii)*, we have $\mathcal{E}(A_1) =_\beta \mathcal{E}(A'_1)$, and $\mathcal{E}(A_2) =_\beta \mathcal{E}(A'_2)$. Thus, by Lemma *4.2.1 (ii)*, we have $\mathcal{E}(A'_1) \equiv \mathcal{E}(A'_2)$, as they are $\beta$-normal forms. Finally, by Lemma *4.2.2*, we have $A'_1 \equiv A'_2$, and thus $A_1 =_\beta A_2$, as desired. ∎

Now, we can prove similarity for systems without polymorphism.

*Lemma 4.2.4* *If* $\Gamma \vdash A : B$, *then the following conditions hold:*

*i) there exists a typed legal context* $\Gamma_t$, *and typed terms* $A_t$, $B_t$, *satisfying* $\mathcal{E}(\Gamma_t) = \Gamma$, $\mathcal{E}(A_t) \equiv A$, *and* $\mathcal{E}(B_t) \equiv B$, *such that* $\Gamma_t \vdash_t A_t : B_t$, *and*

*ii) for every typed legal context* $\Gamma_t$, *and every typed term* $B_t$, *satisfying* $\mathcal{E}(\Gamma_t) = \Gamma$, $\mathcal{E}(B_t) \equiv B$, *and* $\Gamma_t \vdash_t B_t : s$, *there exists a typed term* $A_t$, *such that* $\Gamma_t \vdash_t A_t : B_t$ *and* $\mathcal{E}(A_t) \equiv A$.

*Proof:* By mutual induction on derivations.

[(*Proj*)]: then the derivation has the following shape:

$$\frac{\Gamma' \vdash B : s \quad b \notin \mathcal{D}om\,(\Gamma')}{\Gamma', b{:}B \vdash b : B}\ (Proj)$$

for some $\Gamma'$, and $b$.

*i)* By induction.

*ii)* Let $\Gamma_t$ and $B_t$ be such that $\mathcal{E}(\Gamma_t) = \Gamma', b{:}B$, $\mathcal{E}(B_t) \equiv B$, and $\Gamma_t \vdash_t B_t : s$. Observe that $\Gamma_t = \Gamma'_t, b{:}B'_t$, for some $B'_t$, such that $\mathcal{E}(B'_t) \equiv B$. By Lemma *4.2.3*, we get $B'_t =_\beta B_t$. Since $\Gamma'_t, b{:}B'_t \vdash_t B_t : s$, by Property *2.1.16(ii)*, we know that $\Gamma'_t \vdash_t B'_t : s$. The result is obtained by the following derivation:

$$\frac{\dfrac{\Gamma'_t \vdash_t B'_t : s \quad b \notin \mathcal{D}om\,(\Gamma'_t)}{\Gamma'_t, b{:}B'_t \vdash_t b : B'_t}(Proj) \quad \Gamma'_t, b{:}B'_t \vdash_t B_t : s \quad B'_t =_\beta B_t}{\Gamma'_t, b{:}B'_t \vdash_t b : B_t}(Conv)$$

[(*Weak*)]: then the derivation has the following shape:

$$\frac{\Gamma' \vdash A : B \quad \Gamma' \vdash C : s \quad c \notin \mathcal{D}om\,(\Gamma')}{\Gamma', c{:}C \vdash A : B}\ (Weak)$$

for some $\Gamma'$,$c$, and $C$.

*i)* By induction on part *(i)* to the first premise, we find $\Gamma'_t$, $A_t$, and $B_t$, such that $\Gamma'_t \vdash_t A_t : B_t$, with $\mathcal{E}(\Gamma'_t) = \Gamma$, $\mathcal{E}(A_t) \equiv A$, and $\mathcal{E}(B_t) \equiv B$. Then, by induction on part *(ii)* to the second premise, we find $C_t$, such that $\Gamma'_t \vdash_t C_t : s$, and $\mathcal{E}(C_t) \equiv C$. Finally, apply a $(Weak)$ rule and obtain:

$$\frac{\Gamma'_t \vdash_t A_t : B_t \quad \Gamma'_t \vdash_t C_t : s \quad c \notin \mathcal{D}om\,(\Gamma'_t)}{\Gamma'_t, c{:}C_t \vdash_t A_t : B_t}\ (Weak)$$

as desired.

*ii)* Assume, by induction on part *(i)*, the existence of $\Gamma'_t, C_t$, and $B_t$, such that $\Gamma'_t, c{:}C_t \vdash_t B_t : s$. So, by induction on part *(ii)* to the first premise, there exists a term $A'_t$, such that $\Gamma'_t \vdash_t A'_t : B_t$, and $\mathcal{E}(A'_t) \equiv A$. By Property *2.1.16 (ii)*, also $\Gamma'_t \vdash_t C_t : s$. Finally, apply a $(Weak)$ rule and obtain the thesis, as desired.

[$(Conv)$]: then the derivation has the following shape:

$$\frac{\Gamma \vdash A : C \quad \Gamma \vdash B : s \quad C =_\beta B}{\Gamma \vdash A : B}\ (Conv)$$

for some $C$.

*i)* By induction on part *(i)*, we find $\Gamma_t, A_t$, and $B_t$, such that $\Gamma_t \vdash_t A_t : B_t$, and $\mathcal{E}(\Gamma_t) = \Gamma_t$, $\mathcal{E}(A_t) \equiv A$, and $\mathcal{E}(B_t) \equiv B$. By Property *2.1.18*, $\Gamma_t \vdash_t B_t : s$. Then, by induction on part *(ii)* to the second premise, we find $B_t$ such that $\Gamma_t \vdash_t B_t : s$ and $\mathcal{E}(B_t) \equiv B$. Since $\mathcal{E}(B_t) \equiv B =_\beta C$ and $C \equiv \mathcal{E}(C_t)$, by Lemma *4.2.3*, we have $B_t =_\beta C_t$, and we can apply a $(Conv)$ rule to obtain:

$$\frac{\Gamma_t \vdash_t A_t : C_t \quad \Gamma_t \vdash_t B_t : s \quad B_t =_\beta C_t}{\Gamma_t \vdash_t A_t : B_t}\ (Conv)$$

as desired.

*ii)* Similar to part *(i)*.

[(Introduction rules)]: part *(i)* follows immediately by induction, and similarly part *(ii)*, but here we use Lemma *4.2.1 (ii)*.

[(Elimination rules)]: then the derivation has the following shape:

$$\frac{\Gamma \vdash C : \Pi e{:}E.F \quad \Gamma \vdash D : E}{\Gamma \vdash CD : F[D/e]} \; (Elim)$$

for some $C, D, e, E$, and $F$, and $(Elim)$ stands for any elimination rule.

*i)* Easy, with help of Lemmas *4.1.3 (i)*, and *4.2.1 (ii)*.

*ii)* Assume, by induction on part *(i)*, the existence of $\Gamma_t$, and $H_t$, such that $\mathcal{E}(\Gamma_t) = \Gamma$, $\mathcal{E}(H_t) \equiv F[D/e]$, and $\Gamma_t \vdash_t H_t : s$, for some $s$. By induction on part *(ii)*, we find $C_t$ and $D_t$, such that $\Gamma_t \vdash_t C_t : \Pi e{:}E_t.F_t$, and $\Gamma_t \vdash_t D_t : E_t$. Then, by applying an elimination rule we get:

$$\frac{\Gamma_t \vdash_t C_t : \Pi e{:}E_t.F_t \quad \Gamma_t \vdash_t D_t : E_t}{\Gamma_t \vdash_t C_t D_t : F_t[D_t/e]} \; (Elim)$$

By Lemma *4.1.3 (i)*, we have $\mathcal{E}(F_t[D_t/e]) \equiv F[D/e] \equiv \mathcal{E}(H_t)$, and by Property *2.1.18*, also $\Gamma_t \vdash_t F_t[D_t/e] : s'$, for some $s'$. From this, and by Property *3.2.1*, it follows that $s \equiv s'$. Then, apply Lemma *4.2.3*, to get $F_t[D_t/e] =_\beta H_t$. Finally, apply a $(Conv)$ rule, and obtain:

$$\frac{\Gamma_t \vdash_t C_t D_t : F_t[D_t/e] \quad \Gamma_t \vdash_t H_t : s \quad F_t[D_t/e] =_\beta H_t}{\Gamma_t \vdash_t C_t D_t : H_t} \; (Conv)$$

and $\mathcal{E}(C_t D_t) \equiv CD$, as desired. ∎

With the last lemma, we can prove the main theorem of this section.

*Theorem 4.2.5 Let $\mathcal{S}_t$ be a $\mathcal{TS}$ system whose set of rules does not contain* Polymorphism *as subset, and let $\mathcal{S}_u$ be the corresponding $\mathcal{TAS}$ system. Then $\mathcal{S}_t$ and $\mathcal{S}_u$ are similar.*

*Proof:* By Theorem *4.1.4 (i)*, and Lemma *4.2.4 (i)*. ∎

## 4.3 How to obtain an isomorphism

In this section, we will briefly discuss a way to define a cube of type assignment systems that is isomorphic to $\mathcal{TS}$. As discussed above, the main problem that causes loss of isomorphism between $\mathcal{TS}$ and $\mathcal{TAS}$ is that the erasure, through $\mathcal{E}$, of two typed terms can be $\beta$-equivalent, while the originals were not (a thorough investigation on the possible alternative definitions of the (*Conv*) rule on typed systems can be found in [GW94]). We will show that it is possible to define another erasing function, named $\mathcal{E}'$, that gives rise to a second type assignment cube $\mathcal{TAS}'$ which is isomorphic to the $\mathcal{TS}$ cube (via $\mathcal{E}'$).

Remember that the behaviour of $\mathcal{E}$ was to erase type information from $\lambda$-terms. So, in case of dependences, if $A$ is a typed constructor, occurring in a typed kind, $\mathcal{E}(A)$ can either coincide with $A$ (in case $A$ does not contain occurrences of $\lambda$-terms), or $\mathcal{E}(A)$ can be partially typed. The new erasing function $\mathcal{E}'$ we will present below has a context-dependent behaviour, in the sense that it erases type information from $\lambda$-terms, but *not* when these occur as subterms of constructors or kinds. The systems in the cube $\mathcal{TAS}'$ uses (copies of) all rules from $\mathcal{TAS}$, but in the two rules where dependent-types are created, a *typed subderivation* is required. The systems without the *Dependencies* rules coincide exactly with the corresponding systems in the $\mathcal{TAS}$ cube. Also, with *Dependencies* or not, the provable judgments are the same as long as their subjects are either constructors, or kinds.

Now we will define a new type assignment cube $\mathcal{TAS}'$. Note that, in contrast to the $\mathcal{TAS}$ cube, this cube is not obtained by applying an erasing function to all rules of $\mathcal{TS}$. Instead, the new derivation rules are defined independently; however, the objects in the conclusion of each rule are in the codomain of $\mathcal{E}'$.

*Definition 4.3.1* (The $\mathcal{TAS}'$ Cube) *i) The untyped and typed $\lambda$-terms, typed constructors, and typed kinds are defined as in Definitions 2.4.1 and 3.1.1. Let $T'_u$ be the union of the sets $\Lambda$, $Cons_t$, and $Kind_t$.*

*ii) The new erasing function $\mathcal{E}'$: $T_t \to T'_u$ is defined as follows:*

$$
\begin{aligned}
\mathcal{E}'(M) &= \mathcal{E}(M),\\
\mathcal{E}'(\phi) &= \phi,\\
\mathcal{E}'(K) &= K.
\end{aligned}
$$

*iii) Let $M$ range over $\Lambda$, and $\phi$ range over typed constructors, and $K$ ranges over typed kinds; $A$, $B$, and $C$ range over $T'_u$. The General Type Assignment System, induced by $\mathcal{E}'$, called $\mathcal{TAS}'$, proves judgments of the following form:*

$$\Gamma \vdash' M : \phi, \quad \Gamma \vdash' \phi : K, \quad or \quad \Gamma \vdash' K : \Box,$$

*where $\phi \in Cons_t$, and $K \in Kind_t$. In what follows, we will take the liberty to write $\vdash'$, as $\vdash$.*

*iv) The type assignment rules are:*

$(Axiom)$ $$\frac{}{\varepsilon \vdash * : \Box}$$

$(Conv)$ $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad B =_\beta C}{\Gamma \vdash A : C}$$

$(Proj)$ $$\frac{\Gamma \vdash A : s \quad a \notin \mathcal{D}om\,(\Gamma)}{\Gamma, a{:}A \vdash a : A}$$

$(Weak)$ $$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad c \notin \mathcal{D}om\,(\Gamma)}{\Gamma, c{:}C \vdash A : B}$$

$(I)$ $$\frac{\Gamma, x{:}\phi \vdash M : \psi}{\Gamma \vdash \lambda x.M : \Pi x{:}\phi.\psi}$$

$(E)$ $$\frac{\Gamma \vdash M : \Pi x{:}\phi.\psi \quad \Gamma \vdash_t N : \phi}{\Gamma \vdash M\mathcal{E}(N) : \psi[N/x]}$$

$(I_K)$ $$\frac{\Gamma, \alpha{:}K \vdash M : \phi}{\Gamma \vdash M : \Pi\alpha{:}K.\phi}$$

$(E_K)$ $$\frac{\Gamma \vdash M : \Pi\alpha{:}K.\phi \quad \Gamma \vdash \psi : K}{\Gamma \vdash M : \phi[\psi/\alpha]}$$

$(C\text{-}I_C)$ $$\frac{\Gamma, x{:}\phi \vdash \psi : K}{\Gamma \vdash \lambda x{:}\phi.\psi : \Pi x{:}\phi.K}$$

$(C\text{-}E_C)$ $$\frac{\Gamma \vdash \psi : \Pi x{:}\phi.K \quad \Gamma \vdash_t M : \phi}{\Gamma \vdash \psi M : K[M/x]}$$

$(C\text{-}I_K)$ $$\frac{\Gamma, \alpha{:}K_1 \vdash \psi : K_2}{\Gamma \vdash \lambda\alpha{:}K_1.\psi : \Pi\alpha{:}K_1.K_2}$$

$(C\text{-}E_K)$ $$\frac{\Gamma \vdash \phi : \Pi\alpha{:}K_1.K_2 \quad \Gamma \vdash \psi : K_1}{\Gamma \vdash \phi\psi : K_2[\psi/\alpha]}$$

$(C\text{-}F_C)$ $$\frac{\Gamma, x{:}\phi \vdash \psi : *}{\Gamma \vdash \Pi x{:}\phi.\psi : *}$$

$(C\text{-}F_K)$ $$\frac{\Gamma, \alpha{:}K \vdash \phi : *}{\Gamma \vdash \Pi\alpha{:}K.\phi : *}$$

$(K\text{-}F_C)$ $$\frac{\Gamma, x{:}\phi \vdash K : \Box}{\Gamma \vdash \Pi x{:}\phi.K : \Box}$$

$(K\text{-}F_K)$ $$\frac{\Gamma, \alpha{:}K_1 \vdash K_2 : \Box}{\Gamma \vdash \Pi\alpha{:}K_1.K_2 : \Box}$$

*Notice that the derivation rules (E) and ($C$-$E_C$) require derivations in $\mathcal{TS}$, although restricted to typed λ-terms. This means that, officially, all rules of $\mathcal{TS}$ belong to the set of rules. Notice, moreover, that* only *types dependent on* typed *λ-terms are created in this way.*

*v) As in Definition 2.4.3 (iii) (iv), the rules can be grouped in sets, and, again, eight type assignment systems can be defined, whose relationships can be represented as before by drawing a cube.*

The main result on the relationship between the $\mathcal{TS}$ cube and the $\mathcal{TAS}'$ cube is:

*Theorem 4.3.2 Let $\mathcal{S}_t$ be any typed system in the $\mathcal{TS}$ cube, and let $\mathcal{S}_u$ be the corresponding system in the $\mathcal{TAS}'$ cube. Then $\mathcal{S}_t$ and $\mathcal{S}_u$ are isomorphic (via $\mathcal{E}'$).*

*Proof:* *i)* The function $\mathcal{F}$: $\mathcal{D}er_t \rightarrow \mathcal{D}er_u$ can be defined by induction on $\mathcal{D} \in \mathcal{D}er_t$ in the following way. We treat only the rules $(E)$, and $(C\text{-}E_C)$: the other rules are defined by straightforward induction. Let the last rule applied be:

[$(E)$]: then the typed derivation has the following shape:

$$\mathcal{D}\text{:}\ \frac{\mathcal{D}' : \Gamma \vdash_t M : \Pi x{:}\phi.\psi \quad \Gamma \vdash_t N : \phi}{\Gamma \vdash_t MN : \psi[N/x]}\ (E)$$

By induction, $\mathcal{F}(\mathcal{D}') : \mathcal{E}'(\Gamma) \vdash \mathcal{E}'(M) : \mathcal{E}'(\Pi x{:}\phi.\psi)$. Since $\mathcal{E}'(\Pi x{:}\phi.\psi) \equiv \Pi x{:}\phi.\psi$, and $\mathcal{E}'(\Gamma) = \Gamma$, and $\mathcal{E}'(M) \equiv M'$, we can define:

$$\mathcal{F}(\mathcal{D}) : \frac{\Gamma \vdash M' : \Pi x{:}\phi.\psi \quad \Gamma \vdash_t N : \phi}{\Gamma \vdash M'\mathcal{E}'(N) : \psi[N/x]}\ (E)$$

[$(C\text{-}E_C)$]: then the typed derivation has the following shape:

$$\mathcal{D}\text{:}\ \frac{\mathcal{D}' : \Gamma \vdash_t \psi : \Pi x{:}\phi.K \quad \Gamma \vdash_t M : \phi}{\Gamma \vdash_t \psi M : K[M/x]}\ (C\text{-}E_C)$$

By induction, $\mathcal{F}(\mathcal{D}') : \mathcal{E}'(\Gamma) \vdash \mathcal{E}'(\psi) : \mathcal{E}'(\Pi x{:}\phi.K)$. Since $\mathcal{E}'(\psi) \equiv \psi$, $\mathcal{E}'(\Gamma) = \Gamma$, and $\mathcal{E}'(\Pi x{:}\phi.K) \equiv \Pi x{:}\phi.K$, we can define:

$$\mathcal{F}(\mathcal{D}) : \frac{\Gamma \vdash \psi : \Pi x{:}\phi.K \quad \Gamma \vdash_t M : \phi}{\Gamma \vdash \psi M : K[M/x]}\ (C\text{-}E_C)$$

*ii)* The function $\mathcal{G}$: $\mathcal{D}er_u \rightarrow \mathcal{D}er_t$ can be defined by induction on $\mathcal{D} \in \mathcal{D}er_u$ in a similar way. We treat only the rules $(E)$, and $(C\text{-}E_C)$: the other rules are defined by straightforward induction.

[$(E)$]: then, the type assignment derivation has the following shape:

$$\mathcal{D}\text{:}\ \frac{\mathcal{D}' : \Gamma \vdash M : \Pi x{:}\phi.\psi \quad \Gamma \vdash_t N : \phi}{\Gamma \vdash M\mathcal{E}'(N) : \psi[N/x]}\ (E)$$

By induction, there exists $\Gamma'$, $M'$, and $\xi$, such that $\mathcal{G}(\mathcal{D}') : \Gamma' \vdash_t M' : \xi$, where $\mathcal{E}'(\Gamma') = \Gamma$, $\mathcal{E}'(M') \equiv M$, and $\mathcal{E}'(\xi) \equiv \Pi x{:}\phi.\psi$. This implies that $\Gamma' = \Gamma$, and

$\xi \equiv \Pi x{:}\phi.\psi$. So, we can define:

$$\mathcal{G}(\mathcal{D}) : \frac{\Gamma \vdash_t M' : \Pi x{:}\phi.\psi \quad \Gamma \vdash_t N : \phi}{\Gamma \vdash_t M'N : \psi[N/x]} \, (E)$$

[$(C\text{-}E_C)$]: then, the type assignment derivation has the following shape:

$$\mathcal{D}\text{:} \; \frac{\mathcal{D}' : \Gamma \vdash \psi : \Pi x{:}\phi.K \quad \Gamma \vdash_t M : \phi}{\Gamma \vdash \psi M : K[M/x]} \, (C\text{-}E_C)$$

By induction, there exists $\Gamma', \theta$, and $\xi$, such that $\mathcal{G}(\mathcal{D}') : \Gamma' \vdash_t \theta : \xi$, where $\mathcal{E}'(\Gamma') = \Gamma$, $\mathcal{E}'(\theta) \equiv \psi$, and $\mathcal{E}'(\xi) \equiv \Pi x{:}\phi.K$. This implies that $\Gamma' = \Gamma$, $\theta \equiv \psi$, and $\xi \equiv \Pi x{:}\phi.\psi$. So, we can define:

$$\mathcal{G}(\mathcal{D}) : \frac{\Gamma \vdash_t \psi : \Pi x{:}\phi.K \quad \Gamma \vdash_t M : \phi}{\Gamma \vdash_t \psi M : K[M/x]} \, (C\text{-}E_C)$$

Looking at Definition *4.1.1 (iii)*, it is easy to verify that these two functions realize an isomorphism between the corresponding systems in the two cubes. ■

So, we obtain another cube of type assignment systems that is isomorphic to the $\mathcal{TS}$ cube. Observe that the systems without the *Dependencies* set of rules coincide exactly with the corresponding systems in the $\mathcal{TAS}$ cube, as is shown in Figure *4.2*.

While the definition of the erasing function $\mathcal{E}'$ is (apparently) easy, the definition of the related cube is rather involved. This is a consequence of the fact that, for systems with dependences, derivations are *not* compositional. Namely, if $\mathcal{D}$ is a derivation and $\mathcal{D}'$ is a subderivation of $\mathcal{D}$ that ends with a judgment of the form $\Gamma \vdash_t M : \phi$, for $M \in \Lambda_t$, then $\mathcal{D}'$ need not be a valid derivation; this is because $E'$ has a context dependent behaviour. This is the price paid for defining a cube that is isomorphic to the typed one.

## 4.4 Conclusions and Future Work

This chapter, and the previous one, together with [GHR93, vBLRU94, vBLRU95], can be seen as the first attempt to study type assignment systems with dependent-types. The

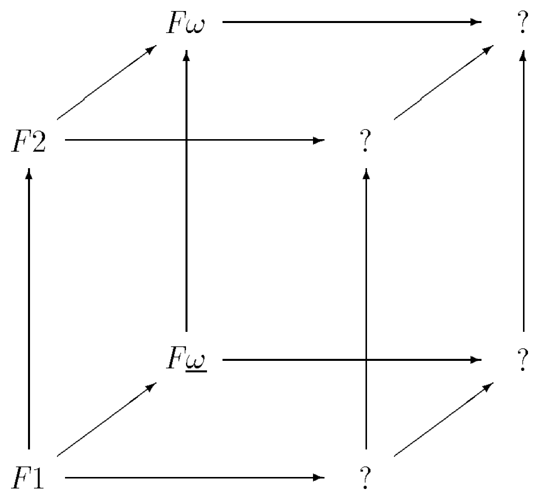



Figure 4.2: The Cube of Type Assignment Systems, $\mathcal{TAS}'$

systems in the dependency-free part of the cubes $\mathcal{TAS}$ and $\mathcal{TAS}'$ have been extensively studied in the literature. We showed that all systems with dependences we defined enjoy good computational properties, such as the Church-Rosser, subject reduction and strong normalization property. We also focused on the relationship between typed and type assignment systems.

The only type assignment system with dependent-types already defined in the literature is the system of Dowek [Dow93], which is based on the typed system $\lambda P$. Strictly speaking, this is not a type assignment system in the usual sense. In [Dow93], there is no formal system to derive judgments; instead, a valid judgment of this system is defined as one of the form $\Gamma \vdash \mathcal{E}(M) : B$, where $\Gamma \vdash_t M : B$ is a valid judgment of $\lambda P$. The type-checking problem for Dowek's system was shown to be undecidable in that paper. Dowek's system is equivalent to the system corresponding to $\lambda P$ in the $\mathcal{TAS}'$ cube. We conjecture that this undecidability result is true for all our systems with dependences.

Topics for further research are:

**$\lambda$I-Terms.** In Section *4.1*, Theorem *4.1.7* shows the failure of similarity between the $\mathcal{TS}$ and $\mathcal{TAS}$ cubes, by showing the existence of two typed $\lambda$-terms, namely $M_t^{0'}$ and $M_t^{1'}$, and of a context $\Gamma$ and type $\beta$, such that:

*i)* $M_t^{0\prime} \neq_\beta M_t^{1\prime}$;

*ii)* $\mathcal{E}(M_t^{0\prime}) =_\beta \mathcal{E}(M_t^{1\prime})$;

*iii)* $\Gamma \vdash_t M_t^{0\prime} : \beta$ and $\Gamma \vdash_t M_t^{1\prime} : \beta$.

This allows the construction of the counterexample to the similarity.

The existence of such terms is possible due to both the polymorphic rules, and the fact that it is possible to abstract with respect to variables not occurring in the body. The use of the $\lambda$-term $\mathbf{K}$ is essential, in order to obtain the correct final typing. A possible direction of research is to study the possibility of reach similarity between the typed and the type assignment cubes by limiting the syntax of terms, by allowing only $\lambda I$-terms, where a $\lambda I$-term is defined as follows:

$$M_I ::= x \mid M_I M_I \mid \lambda x.M_I \text{ if } x \in \mathcal{FV}(M_I).$$

**Principal Typing.** The system $DF1$in the $\mathcal{TAS}$ cube is the smallest type assignment system using dependent-types. In [GHR93], it was proved that the set of pure $\lambda$-terms that are typable in $DF1$ is known to coincide with the set of terms typable in $F1$: hence it is recursive. In [Dow93], G. Dowek has proved that the set of pure $\lambda$-terms typable *in a given context* in the type assignment $\lambda\Pi$, which is the system equivalent to $\lambda P$ in the $\mathcal{TAS}'$ cube is however not recursive.

The counterpart of system $\lambda P$ in the $\mathcal{TAS}$ cube is the type assignment system $DF1$. Because of the properties of the mapping $\mathcal{E}$, the set of pure $\lambda$-terms typable in $DF1$, is the same of the set of pure $\lambda$-terms in $\lambda\Pi$, and also Dowek's theorem adapts to $DF1$ typability in a given context.

It is still open if $DF1$ enjoys the principal type property. If a type $\phi$ is a *principal type* for a $\lambda$-term $M$, then it must be the smallest type such that all types that can be assigned to $M$ can be obtained from $\phi$ by means of suitable operations. For example, the principal type for the identity function $\lambda x.x$ in $F1$is $\alpha\to\alpha$, and all other derivable types for the identity should be obtained by applying a suitable subtitution to the principal type. Obviously, the substitution is not the right operation to capture the principal type property in $DF1$: for example, we can assign to the identity the type $\beta M\to\beta M'$, for suitable $\beta$ and $M =_\beta M'$, and this type cannot be obtained as a substitution instance of $\alpha\to\alpha$.

We conjecture that there are two useful mapping: the first one is $(\mathcal{E}d)$, introduced in Definition *3.5.1*, that maps $DF1$ derivations to $F1$ derivations. Consequently, $\mathcal{E}$ erases all type-dependences. The second one, called $\mathcal{D}e$, reconstructs dependences in the most general way, depending only on a fixed ordering of the free $\lambda$-term variables occurring in the subject and in the predicate of the conclusion (here assumed to be a pure $\lambda$-term). Since system $F1$ enjoys the principal type property of Curry's system, we are able to compute for $DF1$ a pair $(\Gamma; \phi)$ (for a given $\lambda$-term $M$), the pair for $M$ principal up to $\beta$-conversion. Thanks to Theorem *3.5.5*, this conjecture can be used in a type reconstruction algorithm *without a given context* also for system $\lambda P$. Observe that this conjecture does not contradict Dowek's theorem for system $\lambda\Pi$.

**$\eta$-Rule.** In a cube of type assignment systems with $\eta$-rule, some properties of Section *3.2* can be complicated. This was so, for example, the Church-Rosser property for the system $LF$ with $\eta$-reduction, that was first proved by A. Salvesen in [Sal90], or the same theorem in normalizing P.T.S., proved in the thesis of H. Geuvers [Geu93]. As is well-known in the literature, when the $\eta$-rule is added to system $F2$, the subject reduction property for typable terms is lost, as is shown in the next example:

*Example 4.4.1* Consider the following $\eta$-reduction between well typed $\lambda$-terms in $F2$:

$$\lambda x.yx \rightarrow_\eta y.$$

It is easy to verify that $\lambda x.yx$ has type $(\forall\alpha.\alpha) \rightarrow \beta$, in the context $\Gamma = \beta{:}*, y{:}\beta{\rightarrow}\beta$, where the polymorphic-type $\forall\alpha.\alpha$ stands for the product type $\Pi\alpha{:}*.\alpha$. Unfortunately, we cannot do the same for $y$.

# Chapter 5

# What is Object-Orientation

## Preface

In this chapter, we introduce some fundamental concepts of *Object-Orientation*. What follows here is strongly inspired by the very nice paper [FM94]: in that paper, an introduction on the fundamental concepts of Object-Orientation was given, together with an overview of the main type-theoretic solutions studied in the past decade. In particular, in Section *5.1*, we present a description of the most important features of object-orientation. In Section *5.2*, we present a review of the *class-based* model of [PT94]. This is interesting, because it can be compared with the model presented in the subsequent Chapters 5, and 6, where the *axiomatic*, *delegation-based* model of [FHM94] will be presented in full details. We conclude this chapter with a description of the Abadi and Cardelli Calculus of Objects [AC94], a calculus which is very close to the one of [FHM94].

## 5.1 What is Object-Orientation

Object-Orientation is often referred to as a new programming *paradigm*. Other programming paradigms include imperative programming, logic programming, and functional programming. Usually, the word *paradigm* means a way of organizing knowledge: in this sense, object-oriented programming is a new paradigm. In fact, it forces the programmer to reconsider his thinking about computation, and about how programs and data should be structured.

Object-Orientation can be also viewed as a *design methodology* to develop programs: this methodology is an iterative process consisting of the following steps:

*i)* Identify some units of computation at a given level of abstraction;

*ii)* Identify the semantics (intended behaviour) of these units;

*iii)* Identify the relationships between the units;

*iv)* Implement these units.

The process is iterative because one unit is usually implemented using some "sub-units", just as in top-down programming. One difference between Object-Orientation and Structured-Programming is that both functionality and data representation may be refined in the design process, while in Structured-Programming the data structures remain invariant under refinements of the program.

Writing a program requires some ability to split a big problem into some smaller units: the smaller units can be more easily understood and implemented. The units are usually logically interconnected with some *interfaces*; each unit encapsulates and isolates design and execution information, increasing the level of abstraction of the whole program.

Object-Oriented techniques can be seen as a natural outcome of a refinement process starting from defining procedures, modules, abstract-data-types, and finally defining objects. We briefly outline the goals of this abstraction mechanism:

**Procedures and Functions.** They are the first abstraction mechanism by which it is possible to define tasks that can be executed repeatedly, collected in libraries, and reusable. They realize the first primitive notion of *information hiding*. In fact, programmers using libraries do not know the exact details of the implementation, but only need the necessary interface. Procedures do not solve the problem of multiple programmers making use of the same names. In any programming language with a block-scoping mechanism, if some procedures use the same global structures, then there is no effective mechanism to forbid the direct access on those structures.

**Modules.** They can be viewed as an improvement of procedures in order to create and manage name spaces. A module provides the ability to divide a name space into two parts: the *public part* which is accessible from outside the module, and the *private* part which is accessible only within the module. Modules provide an effective method of *information hiding* of the internal structures, but do not allow to perform *instantiations*, which is the ability to make multiple copies of the data areas.

**Abstract Data Types (ADT's).** They are programmer-defined data-types that can be manipulated in a manner similar to the system-defined data-types. To define an ADT corresponds to defining an *initial algebra*, i.e., a set of legal data values and a number of primitive operations that can be performed on those values. Users can create instances of that ADT (create variables of that ADT). Modules are frequently used as an implementation technique for ADT. To build an ADT we must be able to:

*i)* Export a type definition.
*ii)* Build a set of operations that work on instances of the ADT.
*iii)* Protect the internal representation of the ADT to be defined.
*iv)* Make multiple instances of the type possible.

**Object-Types.** They are ADT's equipped with a hidden local *state*. Instances of object-types are called *Objects*. The objects have some interesting mechanisms such as *message passing*, *inheritance*, and *polymorphism*. We will describe all these features in the next subsections.

Thus, we can say that a language is patented "Object-Oriented" if and only if it satisfies the following requirements [CW85]:

*i) It supports objects that are data abstractions with an interface of named operations and a hidden local state.*

*ii) Objects have an associated object-type.*

*iii) Object-types may inherit methods from supertypes.*

*iv) Objects can be subsumed in some contexts if their types are related via an inclusion relation, or can have an (explicit/implicit) quantification over types.*

These requirements may be summarized as follows [CW85]:

*Object-Oriented = data abstraction + object-types + inheritance + polymorphism.*

### 5.1.1 Objects and Message Passing

The basic unity of any proper object-oriented programming language is the *object*. An object can be usually viewed as a set of procedures, called *methods*, which represent the basic actions that the object can perform, and a local hidden *state*. All methods in the object are labeled by a name, the *message name*.

The message encodes the request to perform the procedure labeled by that name. In any programming language with objects, there is some syntax for invoking a method associated with an object. To give a neutral syntax, following [FM94], we write:

$$[\mathtt{object} \Leftarrow \mathtt{message\ arguments}],$$

for invoking the `message` on `object` with some (additional) `arguments`. If the object accepts the message, then the object contains a method associated with the label `message`, and it will carry out the indicated action. All interactions with objects occur via message passing. Another way to see message passing is to consider a message name as an identifier of some kind of *overloaded* function that is not part of the object. A typical example of the object point `p`, can be written [FM94] as follows:

$$\begin{aligned} \mathtt{object\ p\ is}\ \langle \mathtt{x} &= 0, \\ \mathtt{y} &= 0, \\ \mathtt{mv_x} &= \lambda dx \ldots (\mathtt{p} \Leftarrow \mathtt{x}) + dx, \\ \mathtt{mv_y} &= \lambda dy \ldots (\mathtt{p} \Leftarrow \mathtt{y}) + dy \\ \rangle. \end{aligned}$$

Considering the computation as a sequences of *message passing*, when an object sends a message to another object, means that it is not necessary to know the actual means by which the message will be executed. This behaviour of objects models the, so called, *information hiding* (the sender does not know anything but the name of the message of the receiver). Note that the interpretation of any message could be, in principle, different for different receivers: this mechanism makes message passing more general than the usual procedure-call mechanism. Usually, the specific receiver is unknown until run-time, so the determination of which method to invoke cannot be made until run-time. For this reason, we say that there is *late binding*, or *dynamic lookup* between procedure names (messages) and procedure bodies (methods). This is in contrast with the compile-time binding of names to code in conventional procedure calls.

An important feature of objects is that the interpretation of the message, that is, the method used to respond to the message, can depend only on the receiver (in this case the selection of code is *single dispatch*), or can depend also from the arguments supplied to the message (in this case the selection is *multiple dispatch*). A method that behaves differently on different inputs is polymorphic (or, to be precise, it is *ad-hoc* polymorphic).

### 5.1.2 Objects and Encapsulation

Another advantage of the object-oriented paradigm, which directly follows from the information-hiding feature of objects, is the so called *encapsulation*. Typically, an object contains some private data that can be accessed only by its methods. Such encapsulation helps to insure that programs can be written in a modular fashion and that the implementation of objects can be changed without forcing changes in the rest of the system. These features are the same as those realized by ADT.

For example [FM94] defines an ADT called `queue` with some internal structure and suitable operations `add`, `first`, `lenght`, `empty`, and `rest` to access and update the queue. If we want to change the behaviour of some of the operations in the signature of `queue` (e.g. to modify the first-in/first-out order of elements), and to introduce a priority rule to catch the first elements of that queue, then we are obliged to define another ADT, say `queue`′, with the same signature *but* different behaviours. If we use `queue` and `queue`′ ADT's in the same scope (for example a program that counts the total number of items in both queues), then we are obliged to rename the signatures of `queue`′, otherwise one definition will hide the other. It follows that the collection of operation names in the signatures of different queues must be different, even though we might wish to treat `queue` and `queue`′ uniformly. This shortcoming will be eliminated by the *dynamic lookup* of method names for objects; in fact, the interpretation of a message depends on the receiver and can vary for different receivers.

As we said in the previous subsection, if the interpretation of the message depends also on the arguments supplied to the message, then the selection is called *multiple dispatch*. With multiple dispatch there seems to be some loss of encapsulation. In fact, in order to define a method that works on different kinds of arguments, the method must have access to the internal data (if any) of each argument: if the arguments are objects, then we can write:

$$[\mathtt{object}_1 \Leftarrow \mathtt{message}\ \mathtt{object}_2, \ldots, \mathtt{object}_\mathtt{n}].$$

Then the $\mathtt{object}_1$ may need to know the structure of the arguments, breaking the encapsulation of the $\mathtt{object}_\mathtt{i}$'s. The problem of loss of encapsulation is not inherent to multiple dispatch, but instead to the possibility to treat objects as *first-class entities*, i.e., entities that can be passed as arguments of functions, or can be results of some computation. A good example of methods that receive objects as arguments are the *binary methods*, i.e. methods that perform some actions between the receiver and another object. A typical example is the equal method `eq` for object points:

$$\mathtt{p} \Leftarrow \mathtt{eq}\ \mathtt{p}',$$

that checks if the receiver `p` is equal to the argument $\mathtt{p}'$.

### 5.1.3 Object and Types

Object-oriented languages can be classified as either *class-based* or *delegation-based* languages. In class-based languages, such as SMALLTALK [GR83] and C$^{++}$ [ES90], the implementation of an object is specified by its class. Objects are created by instantiating their classes. In delegation-based languages, objects are defined directly from other objects by adding new methods via *method addition*, and replacing old method bodies with new ones via *method override*. Adding or overriding a method produces a new object that inherits all the properties of the original one.

Usually the meanings of the word *class* is overloaded. In the following section, we will informally present the notion of *class-type*, and the notion of *object-type*. These two notions strongly depend on what language we are considering.

In class-based languages, there are two general schools of thought on the issue of defining objects. Languages, such as C$^{++}$ and OBJECT-PASCAL [Tes85], consider a class to be a type (like for any other built-in type), while other languages, such as SMALLTALK and OBJECTIVE-C, consider a class to be a true object.

**Classes-as-Types.** In class based languages, that support the analogy "classes-as-types" view, all objects are instances of a more general type structure, called *class*. In this case, classes are an extension of the concept of the structure of record. Like a record, a class defines fields, and the instances of that class have their own values for these fields. Unlike a record, a class can also have fields that represent functions or procedures that are shared (in terms of code allocation) by all instances of that class. This point of view becomes more complex when the mechanism of *inheritance* is introduced. Modifying (or overriding) methods does not guarantee in general that the behaviour or the effect of an overridden method will have any relationship to the behaviour of the method in the original class. Moreover, adding methods to a class means to allocate more space for that instance, and this can complicate some trivial task such as assignment.

**Classes-as-Objects** In the object oriented jargon, creating an instance of a given class can be viewed as an activity that must be "delegated" to some object. Then the following question arises: which object should have the responsibility for this activity? A solution places a layer of management between the user, who desires the creation of a new object, and the code that performs the allocation of the memory. It follows that, for each class, say `MyClass`, to is be defined, we have a proper *object*, say $\texttt{Object}_{\texttt{MyClass}}$, that has the responsibility of creating instances of `MyClass`. This object must have all the information about the size of the class it represents and the methods to which instances of this class will respond.

Since every object must be an instance of some class, also `MyClass` is an instance of a class called `MetaClass` that represents the class of all classes, containing a method, usually called `new`, that performs the creation of an instance of any subclass of `MetaClass`, and all methods used to performs message passing. By convention, there is a variable named as the class itself (in our example `MyClass`) which maintains this object as a value. On the other hand, $\texttt{Object}_{\texttt{MyClass}}$ is an instance of `MetaClass`, because it is also an object.

We must distinguish between the relationship of being a *subclass* and that of being an *instance*. The class `Myclass` to be defined is a subclass of $\texttt{Object}_{\texttt{MyClass}}$. The class `MetaClass` is both a class and an instance of itself. A particular object of `MyClass` (called `Object-of-MyClass`) is an instance of the class `MyClass`.

The situation is showed in Figure *5.1*, where the thin arrows represents the *instance* relation, the thick arrows the *subclass* relation, and the double circle represents both a

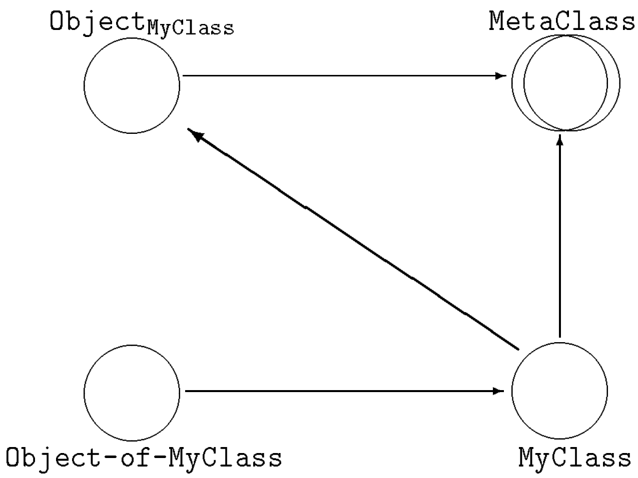



Figure 5.1: Classes and Class instances

class and an object.

An example of a class declaration is the following class of points:

$$
\begin{array}{llll}
\texttt{class Point is} & \texttt{private} & \texttt{x}{:}int = 0; \texttt{y}{:}int = 0; & \\
 & \texttt{public} & \texttt{mv}_\texttt{x}{:}int \rightarrow \texttt{Point} = \lambda dx \ldots (\texttt{Point} \Leftarrow \texttt{x}) + dx; & \\
 & & \texttt{mv}_\texttt{y}{:}int \rightarrow \texttt{Point} = \lambda dy \ldots (\texttt{Point} \Leftarrow \texttt{y}) + dy; & \\
 & & \texttt{eq}{:}\texttt{Point} \rightarrow bool = \lambda \texttt{p}.(\texttt{Point} \Leftarrow \texttt{x}) = (\texttt{p} \Leftarrow \texttt{x})\& & \\
 & & \qquad\qquad (\texttt{Point} \Leftarrow \texttt{y}) = (\texttt{p} \Leftarrow \texttt{y}); & \\
\quad \texttt{end}, & & &
\end{array}
$$

where the notation `name:type=body` means that the method (or the instance variable) `name` has type `type` and body (or value) `body`. The instance (private) variables are initialized to 0, and the public procedures share the same behaviour every time an instance of that class is created.

### 5.1.4 Delegation-Based Languages

In delegation-based languages, such as SELF [US87, CU89], and CLOS there are no classes at all: the programmer creates specific instances of objects, and the intended behaviour is associated with individual objects. Building an object from another object means to "delegate" a portion of its behaviour to the original object. Any message not understood by the new object will be passed on to the delegated object.

Thus, we can build a language out of objects, which possess variables and methods, by means of the delegation-principle, by which an object can defer responsibility for any unrecognized method to another object. Sharing takes place in such language by the use of common delegates. In delegation-based languages, the interface of an object is called *object-type*: it contains the signature of the methods belonging to the object itself. An interesting feature of delegation-based languages is the ability to change delegates dynamically, just by adding methods or overriding existing methods to an object. The relationship between objects and delegates is different from the relationship between instances and classes, since objects and delegates are entities belonging to the same level of abstraction, while instances and classes are not. An example of a object-type is the following:

$$\texttt{object-type Point is}$$
$$\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int, \texttt{mv}_{\texttt{x}}{:}int\to\texttt{Point}, \texttt{mv}_{\texttt{y}}{:}int\to\texttt{Point}, \texttt{eq}{:}\texttt{Point}\to bool\rangle\!\rangle$$

In what follows, for sake of simplicity, we refer to an object-type as "the type of the object", and this word does refers both to class-based and to delegation-based types.

### 5.1.5 Type Inheritance

Type inheritance is recognized as a fundamental concept in object-oriented languages. One can, for example, view inheritance as the basis for a programming methodology where types are constructed in a conceptual *hierarchy*. Inheritance is an "implementation technique"; for every object or object-type defined using inheritance, there is an equivalent definition that does not use it, obtained by expanding the definition so that inherited code is duplicated. In fact, inheritance may be viewed as a type-composition mechanism that allows the properties of one or more types to be reused in the definition of a new type, with the (positive) consequences of saving the effort of duplicating code, and improving both maintenance and modification of programs. The specification "object-type $\tau$ inherits from object-type $\sigma$" may be viewed as an abbreviation mechanism that avoids redefining the methods of object-type $\sigma$ in the definition of object-type $\tau$. As example, suppose to have the following class-type declaration:

$$
\begin{array}{l}
\texttt{class C\_Point is inherit Point} \\
\qquad\qquad\quad \texttt{private col} : Col = white; \\
\qquad\qquad\quad \texttt{public mv}_{\texttt{col}}{:}Col{\to}\texttt{C\_Point} = \lambda col \ldots \\
\qquad\qquad\qquad\qquad \texttt{eq}{:}\texttt{C\_Point}{\to}bool = \lambda\texttt{cp}.(\texttt{C\_Point} \Leftarrow \texttt{x}) = (\texttt{cp} \Leftarrow \texttt{x})\& \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{C\_Point} \Leftarrow \texttt{y}) = (\texttt{cp} \Leftarrow \texttt{y})\& \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{C\_Point} \Leftarrow \texttt{col}) = (\texttt{cp} \Leftarrow \texttt{col}); \\
\qquad \texttt{end.}
\end{array}
$$

When some methods are inherited, we can also need to *redefine* the procedural behaviour of that method and (maybe) the typing interface. In this case we say that some methods are rewritten or *overridden*. In the previous example, the `eq` method is overridden to take into account the color component. When we override some methods, the hierarchy induced by inheritance is modified. In fact, the modified method can behave totally different from the inherited one.

Inheritance can also be "multiple", i.e. the specification "object-type $\rho$ inherits from object-type $\sigma$ and object-type $\tau$" avoids redefining the methods of object-type $\sigma$ and $\tau$ in the definition of object-type $\rho$. Some care must be taken when allowing multiple-inheritance, especially when the inherited types allow methods that share common names.

So, inheritance can be distinguished into:

**Inheritance by "Extension".** A class (or an object) inherits the methods of another class and adds some other methods.

**Inheritance by "Override".** A class rewrite some methods of the superclass.

The override operation can be also classified into:

**"Kamikaze" Override.** override, in a totally arbitrary way, body and type of the method.

**"Strong" Override.** override, in a totally arbitrary way, the body of the method, but with the restriction that the type of the redefined method is a subtype of the corresponding type of the superclass.

**"Light" Override.** override, (in a totally arbitrary way), only the body of the method, leaving invariant its type.

**"Ultra-Light" Override.** override only the body of the method, without loosing the old functionality and signature. This, roughly, corresponds to have the multiple dispatching for methods, where the selection of the correct body to execute depends on the types of the arguments given in input on the method.

### 5.1.6 Method Specialization

It is usual to give the types of some methods of an object in terms of recursive definitions. As example, take the following object-type definition:

$$\texttt{object-type Point is}$$
$$\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int, \texttt{mv}_\texttt{x}{:}int{\to}\texttt{Point}, \texttt{mv}_\texttt{y}{:}int{\to}\texttt{Point}, \texttt{eq}{:}\texttt{Point}{\to}bool \rangle\!\rangle.$$

When colored points are defined in terms of points,

$$\texttt{object-type C\_Point is Point with } \langle\!\langle \texttt{col}{:}Col \rangle\!\rangle,$$

we expect that the types of the recursive inherited methods, namely $\texttt{mv}_\texttt{x}$, $\texttt{mv}_\texttt{y}$, and `eq`, to be specialized to return or use colored points instead of points. Otherwise, we effectively loose type information about the object we are dealing with. In this case, we say that *method specialization*, or *mytype specialization* occurs. So in the object-type of `C_Point`, the inherited methods of `Point` will have the following types:

$$\texttt{mv}_\texttt{x}{:}int{\to}\texttt{C\_Point},\ \texttt{mv}_\texttt{y}{:}int{\to}\texttt{C\_Point},\ \texttt{eq}{:}\texttt{C\_Point}{\to}bool.$$

### 5.1.7 Subtyping

The basic principle associated with subtyping is *substitutivity*. If $\sigma$ is a subtype of $\tau$, usually written $\sigma \preceq \tau$, then any expression of type $\sigma$ can be used without type-errors in any context that requires an expression of type $\tau$. If substitutivity is allowed, then one may improve the program by adding functionalities without any other modification to the original program. The subtyping induces also a hierarchy between types and subtypes. Perhaps the most common confusion surrounding object-oriented programming is the difference between subtyping and inheritance hierarchy. Some languages try to *overlap* the two hierarchies, but some care must be taken in order to avoid an insecure type system.

There are two forms of subtyping on object-types:

**"Width" Subtyping.** This refers to the possibility of considering an object-type to be subtype of another object-type if the former has more methods than the the latter. The previously examples of object-types `C_Point` and `Point` without the `eq` method are in width subtyping relation.

**"Depth" Subtyping.** In this case, the subtyping is made *by component*, i.e. we consider an object-type to be subtype of another object-type if both have the same number of methods but the type of the methods of the former are in the subtype relation with the latter. An example is:

$$\begin{array}{lll} \texttt{object-type Point} & \texttt{is} \ \langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int \rangle\!\rangle & \preceq_{depth} \\ \texttt{object-type R\_Point} & \texttt{is} \ \langle\!\langle \texttt{x}{:}real, \texttt{y}{:}real \rangle\!\rangle, & \end{array}$$

because $int \preceq real$.

In C$^{++}$, the hierarchy induced by the subtyping relation coincides with the one induced by inheritance. This choice can lead to type unsecurities, as it is showed in the following example, taken from [BCG$^+$95]:

```
procedure breackit (p:Point)
  var q:Point
  begin
    q := newPoint(3,4)
    if (p ⇐ equal)(q) then
    .....
  end
```

where the statement `q := newPoint(3, 4)` creates an object (i.e. an instance) of the class `Point` and allocate this instance to the local variable `q`. Then, when `breackit` is invoked with an actual parameter of class `C_Point`, if we allow that `C_Point`$\preceq$ `Point`we will obtain a run-time error, because in the statement (`p` $\Leftarrow$ `eq`)`q`, the `col` component will be required to `q` which obviously does not have it.

### 5.1.8 Polymorphic Types

In this section, we briefly recall various kinds of polymorphism, following the classification of Strachey.Strachey distinguished between two major kinds of polymorphism.

**Universal Polymorphism.** It allows to write functions whose code can work on different types. Among this kind of polymorphism we recall:

**Parametric Polymorphism.** It allows to write functions that accept types as parameters (implicit or explicit). These parameters determine the types of the arguments for each application of the functions. Functions with parametric polymorphism are usually called *generic functions*. An example is the `lenght` operation, which takes a list composed of elements of any type and returns the number of elements in the list. Another example is the `poly-identity` function, which maps an argument of any typing to itself.

**Inclusion Polymorphism or Subtyping.** We consider an object as belonging to many different types that need not be disjoint; this implies that there may be inclusion of types. Functions can work on different inputs if the type of the input is a subtype of a given type.

**Ad-hoc Polymorphism or Overloading.** It allows to write functions that execute different code for inputs of different types. Functions with this kind of polymorphism exhibit different behaviours according to the context in which they are applied. Functions that exhibit this behaviour are usually called *overloaded functions*. The basic principle associated with overloading is *dynamic typing*, i.e., types must be computed during execution of the program, and this computation must affect the final value of the whole execution. A method with multiple dispatch is an example of an overloaded function.

## 5.2 Abstract Data Types and Existential-Types

Abstract Data Type declarations, from now on called ADT [Rey83, MP88], appear in typed programming languages like ADA, CLU, ML [LSS77, MMM91]. This form of declaration binds a list of identifiers to a type with associated operations. We call this composite value a *Data Algebra*. The access to a data algebra is restricted to the explicit declared operations. Formally, a data algebra is a composed value, built from a set of elements, and some operations on this set. For example, consider the following ADT declaration in ML:

$$
\begin{array}{l}
\texttt{abstype}\ complex = real{\times}real\ \texttt{with} \\
\qquad \texttt{create} : real{\to}real{\to}complex = \lambda x{:}real.\lambda y{:}real.\langle x, y\rangle, \\
\qquad \texttt{plus} : complex{\to}complex = \\
\qquad\qquad \lambda z{:}real{\times}real.\lambda w{:}real{\times}real.\langle fst(z) + fst(w), sdn(z) + snd(w)\rangle, \\
\qquad \texttt{re-part} : complex{\to}real = \lambda z{:}real{\times}real.fst(z), \\
\qquad \texttt{im-part} : complex{\to}real = \lambda z{:}real{\times}real.snd(z) \\
\texttt{end}.
\end{array}
$$

Notice that the interface types of the operations differs from the types of the implementing functions: this is because operations are defined using the *concrete* representation of values (in this case $real{\times}real$). This property of typing encapsulation allows the data algebra to hide both the internal structure of the data, and the implementation of the operations on the algebra itself. In general, an abstract data type declaration has the following structure:

$$\texttt{abstype}\ t\ \texttt{with}\ \texttt{x}_1{:}\sigma_1, \ldots, \texttt{x}_n{:}\sigma_n\ \texttt{is}\ \langle\tau; M_1, \ldots, M_n\rangle\ \texttt{in}\ N,$$

where $t$ is the name of the ADT, $\texttt{x}_1, \ldots, \texttt{x}_n$ are the names of the operations, $\sigma_1, \ldots, \sigma_n$ are the signatures of that operations (notice that $t$ can occur in the $\sigma_i$'s), $\langle\tau; M_1, \ldots, M_n\rangle$ is the data algebra (i.e. the type of a particular implementation, and the implementation of the operations on the algebra), and finally $N$ is the scope of the declaration. Using cartesian products, we may put each ADT declaration in the more concise form:

$$\texttt{abstype}\ t\ \texttt{with}\ \texttt{x}{:}\sigma\ \texttt{is}\ \langle\tau; M\rangle\ \texttt{in}\ N.$$

This description is sufficiently abstract to characterize ADT's, without giving any information about particular implementations. For this reason we introduce the *existential-types* of the form:

$$\exists t.\sigma.$$

Intuitively, each element of an existential-type $\exists t.\sigma$ consists of a type $\tau$, and an element of $[\tau/t]\sigma$, (where $[\tau/t]\sigma$ means the substitution of every occurrence of $t$ in $\sigma$ with $\tau$). In the previous example for the ADT *complex*, the related existential-type would be:

$$\exists t.[(real{\rightarrow}real{\rightarrow}t){\times}(t{\rightarrow}t){\times}(t{\rightarrow}real){\times}(t{\rightarrow}real)].$$

The existential-type was part of Girard's System $F$ [Gir86].

The next step to introduce abstract data type declarations in a typed language is to give the type-checking rules associated to them. To do this, let us consider an apparently redundant type information of a data algebra $\langle\tau, \mathtt{M}\rangle$ of the form:

$$\langle t = \tau, M{:}\sigma\rangle,$$

where $t$ can occur in $\sigma$. The bound variable $t$ and the type $\sigma$ serve to disambiguate the type of the expression. The type of a well formed expression $\langle t = \tau, M{:}\sigma\rangle$ is $\exists t.\sigma$, according to the following rule:

$$\frac{M : [\tau/t]\sigma}{\langle t = \tau, M : \sigma\rangle : \exists t.\sigma} \quad (\exists\ Intro)$$

The reason of this redundancy is now clear: since a type $\sigma$ cannot be uniquely determined by the form of a substitution instance $[\tau/t]\sigma$, the type of the simpler implementation form $\langle\tau, M\rangle$ would not be determined uniquely.

It remains to define the typing rule that introduce the declaration of abstract data type:

$$\frac{M : \exists t.\tau \quad \mathtt{x}{:}\tau \vdash N : \sigma}{\mathtt{abstype}\ t\ \mathtt{with}\ \mathtt{x}{:}\tau\ \mathtt{is}\ M\ \mathtt{in}\ N : \sigma} \quad (\exists\ Elim)$$

Informally, this rule binds the type variable $t$ and the operation names to the type implementation and body of the operation part respectively, with scope $N$.

### 5.2.1 The Existential Model of Pierce and Turner

The paper [PT94] presents a functional model of class-based object-oriented languages on an encoding into $F^{\omega}_{\leq}$, an explicit typed, polymorphic $\lambda$ calculus with subtyping. The System $F_{\omega}$ was introduced by Girard in his thesis [Gir72], and it is classified as a vertice in the $\lambda$-cube of Barendregt [Bar92] (see Chapter*2*).

The existential model provides a type-theoretic account of the basic mechanisms of object-oriented programming: encapsulation, message passing, subtyping, and multiple inheritance. The underlying type-theoretic framework is that of *existential-types*, as introduced in [MP88]. The key step in this approach is an alternative treatment of the encapsulation mechanism. Reynolds, in [Rey83], identified two kinds of encapsulation:

**Procedural Abstraction** which relies on hiding the value of some nullary methods (i.e. private variables) inside some objects, (these variables being common to a collection of procedures), and

**Type Abstraction** which reveals the existence of a private state externally, but forbids the access by hiding the type.

Roughly speaking, the first kind of abstraction hides the value of a method, while the second hides the type of the method itself.

In the existential model to be defined below, we consider an object-type as an existential-type of the following form:

$$\texttt{object-type Point is } \exists\texttt{Rep}.\langle\!\langle \texttt{state:Rep}, \texttt{methods:}\langle\!\langle \texttt{x:Rep}, \texttt{mv}_{\texttt{x}}\texttt{:}\mathit{int}\to\texttt{Rep}\rangle\!\rangle\rangle\!\rangle : *,$$

where $*$ is the kind of well-formed types. Each object has an internal `state` component, and a `methods` component, which is a record of all methods that operates on that state. Both the state, and the methods are visible in this encoding, with the existential-type protecting the state from external access. This feature allows to treat abstract data type as existential-types, and realize the type abstraction encapsulation.

The main benefit of this change is a neat simplification of the underlying type-theory: it yields a clean separation between different aspects of encapsulation and subtyping, but, unfortunately, a more difficult treatment of binary methods. The reason of this difficulty is clear: the type abstraction is stronger than the procedural one, and the binary methods usually need to break such abstraction. For example, to check if two points $\mathtt{p}_1$ and $\mathtt{p}_2$ are equal, we can send a message to $\mathtt{p}_1$, namely:

$$(\mathtt{p}_1 \Leftarrow \mathtt{eq})\ \mathtt{p}_2.$$

It follows that the body of the method `eq` of the object $\mathtt{p}_1$ needs to know the types of the coordinates of $\mathtt{p}_2$, and this would be denied because of the type abstraction of the object $\mathtt{p}_2$.

### 5.2.2 Methods and Object-Types

Since the internal state of an object is accessible only through its methods, the type of a method can be expressed by replacing the type of the state by an abstract token, `Rep`: e.g., the type of $\mathtt{mv}_\mathtt{x}$ would be $int \rightarrow \mathtt{Rep}$. Conversely, when a method is called, it needs to access its internal structure: then a type operator, usually called *interface function*, maps the internal representation type into a record of method types. In the case of the object-type `Point` such a function is the following one:

$$\mathtt{P}_{int} \stackrel{def}{=} \lambda\mathtt{Rep}.\langle\!\langle \mathtt{x}{:}\mathtt{Rep}\rightarrow int, \mathtt{mv}_\mathtt{x}{:}\mathtt{Rep}\rightarrow int \rightarrow \mathtt{Rep}\rangle\!\rangle : * \rightarrow *,$$

where $*\rightarrow*$ means that $\mathtt{P}_{int}$ is a type-functional. Using the above interface function, we can write the object-type `Point` as follows:

$$\mathtt{object{-}type\ Point\ is}\ \exists\mathtt{Rep}.\langle\!\langle \mathtt{state}{:}\mathtt{Rep}, \mathtt{methods}{:}\mathtt{P}_{int}\ \mathtt{Rep}\rangle\!\rangle : *.$$

Abstracting $\mathtt{P}_{int}$ from `Point` yields a higher-order type-operator that, given an interface specification, forms the type of objects satisfying it:

$$\mathtt{Object} \stackrel{def}{=} \lambda M{:}* \rightarrow *.\exists\mathtt{Rep}.\langle\!\langle \mathtt{state}{:}\mathtt{Rep}, \mathtt{methods}{:}M\ \mathtt{Rep}\rangle\!\rangle : (*\rightarrow*)\rightarrow*.$$

The type of points objects can be now expressed more concisely:

$$\mathtt{Point} = \mathtt{Object}\ \mathtt{P}_{int}.$$

### 5.2.3 Methods and Objects

In this model, a method is a function that implements transformations on the state: to do this, it needs to know the representation of the internal state. An object can be seen as a pair $\langle \mathtt{Rep}, r\rangle$, where `Rep` is a "witness" for the existential-type, and `r` is the implementation of that object. For example, an object `p` with type `Point` might be implemented as follows:

$$
\begin{array}{lll}
\mathtt{p} \stackrel{def}{=} & \multicolumn{2}{l}{\langle \langle\!\langle \mathtt{X}{:}int \rangle\!\rangle,} \\
& \langle \mathtt{state} & = \langle \mathtt{X} = 3\rangle, \\
& \mathtt{methods} & = \langle \mathtt{x} = \lambda s{:}\langle\!\langle \mathtt{X}{:}int \rangle\!\rangle.s.\mathtt{X}), \mathtt{mv_X} = \lambda s{:}\langle\!\langle \mathtt{X}{:}int \rangle\!\rangle.\lambda dx{:}int.\langle \mathtt{X} = s.\mathtt{X} + dx\rangle\rangle \\
& \rangle\rangle, &
\end{array}
$$

where the notation $s.\mathtt{X}$ denote the selection of the `X` field in the record $\langle \mathtt{X} = 3\rangle$. In this example, $\langle\!\langle \mathtt{X}{:}int \rangle\!\rangle$ is the type of the internal state component of `p`, and the internal value of that state is $\langle \mathtt{X} = 3\rangle$. The methods of the object `p` are represented as a record of functions, each one taking as first argument the internal representation of the object itself (in this case $\langle\!\langle \mathtt{X}{:}int \rangle\!\rangle$). Note that, unlike some object-oriented languages, the elements of an object type here may have *different* internal representations and implementations of their methods. Another implementation for the above object `p` would use the following internal representation

$$\langle\!\langle \mathtt{X}{:}real \rangle\!\rangle,$$

making a real point instead of an integer one.

### 5.2.4 Methods and Message Send

In the existential model, sending a message is modeled by polymorphic functions that take an object as parameter, open it, apply the appropriate method from its `methods` components, and finally repackage it. Such functions are polymorphic to insure that they will work on all objects derived from the one for which they were originally intended. The form of polymorphism used here is *higher-order bounded universal quantification*, and it is modelled by the following rule:

$$\frac{\vdash A : \forall b{:}B \preceq C.D \quad \vdash E \preceq C : *{\to}*}{\vdash AE : [E/b]D} \quad (\forall{-}bound_{\preceq})$$

The bound here is the interface specification. For the above example of object points, here is the function that performs the call on the $\mathtt{mv_x}$ method:

$$\begin{aligned}\mathtt{P}_{mv} \stackrel{def}{=}\ & \lambda M \preceq \mathtt{P}_{int}.\lambda \mathtt{p}{:}\mathtt{Object}\ M. \\ & \mathtt{open\ p\ as}\langle\mathtt{Rep},\mathtt{r}\rangle\ \mathtt{in}\ \lambda dx{:}int.\langle\mathtt{Rep},\langle\mathtt{state} = (\mathtt{r.methods} \Leftarrow \mathtt{mv_x})\ (\mathtt{r.state})\ dx, \\ & \qquad\qquad\qquad\qquad\qquad\qquad \mathtt{methods} = \mathtt{r.methods}\rangle\rangle. \\ & \mathtt{end},\end{aligned}$$

where `r.state`, and `r.methods` select the `state` and the `methods` component, respectively, from the implementation of the point object. The type of $\mathtt{P}_{mv}$ will be:

$$\mathtt{P}_{mv} : \forall M \preceq \mathtt{P}_{int}.(\mathtt{Object}\ M)\rightarrow int \rightarrow (\mathtt{Object}\ M).$$

This function works as follows: it takes an interface function $M$, which is a subtype of $\mathtt{P}_{int}$, an object `p`, and a displacement $dx$, and returns a point exactly like `p` except for the value of `X` incremented by $dx$. This is done by first opening the point `p`, and binding the internal representation to `Rep`, and its implementation to `r`, and then calculating a new object point in the new location, repackaging the old internal representation and implementation, with the modified state. Since the interface function of colored point is:

$$\mathtt{CP}_{int} \stackrel{def}{=} \lambda\mathtt{Rep}.\langle\!\langle \mathtt{x}{:}\mathtt{Rep}\rightarrow int, \mathtt{col}{:}\mathtt{Rep}\rightarrow Col, \mathtt{mv_x}{:}\mathtt{Rep}\rightarrow int\rightarrow\mathtt{Rep}\rangle\!\rangle : *\rightarrow *,$$

and since $\mathtt{CP}_{int}$ is a subtype of $\mathtt{P}_{int}$, it follows that the message sending function $\mathtt{P}_{mv}$ will work on colored points as well as on points: i.e.:

$$\begin{aligned}\mathtt{P}_{mv}\ \mathtt{P}_{int}\ &:\ \mathtt{Object}\ \mathtt{P}_{int}\rightarrow int\rightarrow\mathtt{Object}\ \mathtt{P}_{int}, \\ \mathtt{P}_{mv}\ \mathtt{CP}_{int}\ &:\ \mathtt{Object}\ \mathtt{CP}_{int}\rightarrow int\rightarrow\mathtt{Object}\ \mathtt{P}_{int}.\end{aligned}$$

### 5.2.5 Classes and Inheritance

In the existential-model, as in any other class-based model, we can think of classes as *templates functions* which can either be used to create objects, or extended to create new classes. The objects created from a particular class all have the same type, but not all objects of a particular type need be created from the same class.

A class fixes the internal representation, i.e. the state of the objects it creates, and the methods that can access the internal representation. Since classes can be extended, future subclasses can also modify the internal state (by adding new fields), and add new methods. For this reason, each method defined in a class must be polymorphic in the final representation type.

If the body of a method in a class refer to some other methods of the class itself, then such methods must be accessed via an "indirection" to guarantee the dynamic lookup of methods bodies. To do this, methods refer to other methods by a bound parameter *self*, containing all the methods of the object to be instantiated. This happens when, for example, the body of the $\mathtt{mv_x}$ method uses another method, to set the state of the point.

So, a *class* is essentially an object parameterized in both the internal representation and the used methods. When the class is instantiated to form an object, its representation and methods are fixed, so all reference to *self* becomes references to the current object. For example, let the current representation of a point be $\mathtt{P}_{rep}$, and let:

$$\mathtt{P}_{rep} \stackrel{def}{=} \langle\!\langle \mathtt{X}{:}int \rangle\!\rangle.$$

The function $\mathtt{P}_{class}$ below can be thought of as a class function interface, which is a record of methods, abstracted on a record *self* of methods with the same type. We suppose, as a non-trivial simplification, that point and color point classes will have the same internal representation type. This is obviously not true in reality, but it simplifies the theory. We refer the reader who is interested to [PT94].

$$\begin{aligned}
\mathtt{P}_{class} \stackrel{def}{=} \; & \lambda self{:}\mathtt{P}_{int}\ \mathtt{P}_{rep} \\
& \langle \mathtt{x} = \lambda s{:}\mathtt{P}_{rep}.s.\mathtt{X}, \\
& \mathtt{mv_x} = \lambda s{:}\mathtt{P}_{rep}.\lambda dx{:}int.\langle \mathtt{X} = ((s.\mathtt{X}) + dx)\rangle \\
& \rangle : (\mathtt{P}_{int}\ \mathtt{P}_{rep}) \to (\mathtt{P}_{int}\ \mathtt{P}_{rep}).
\end{aligned}$$

Using the fixed point operator

$$\mathtt{rec} : \forall\alpha.(\alpha\to\alpha)\to\alpha,$$

we can create a point object from the $\mathtt{P}_{class}$ definition as follows:

$$\mathtt{p} \stackrel{def}{=} \langle \mathtt{P}_{rep}, \langle \mathtt{state} = \langle \mathtt{x} = 3 \rangle, \mathtt{methods} = \mathtt{rec}(\mathtt{P}_{int}\ \mathtt{P}_{rep})\mathtt{P}_{class} \rangle : \mathtt{Object}\ \mathtt{P}_{int}.$$

$$\begin{tikzcd}
\begin{array}{c}\mathtt{P}_{class}\\ :\mathtt{Class}\ \mathtt{P}_{int}\ \mathtt{P}_{rep}\end{array} \arrow[r, "\mathtt{new}"] \arrow[d, "\mathtt{extend}"] & \mathtt{p}:\mathtt{Object}\ \mathtt{P}_{int} \\
\begin{array}{c}\mathtt{CP}_{class}\\ :\mathtt{Class}\ \mathtt{CP}_{int}\ \mathtt{CP}_{rep}\end{array} \arrow[r, "\mathtt{new}"] & \mathtt{cp}:\mathtt{Object}\ \mathtt{CP}_{int}
\end{tikzcd}$$

Figure 5.2: Classes and Objects

Abstracting the method interface $\mathtt{P}_{int}$, and the local type representation $\mathtt{P}_{rep}$ in the type of $\mathtt{P}_{class}$ yields a higher-order type-operator describing the types of arbitrary class definitions:

$$\mathtt{Class} \stackrel{def}{=} \lambda M{:}\,* \to *\,.\lambda R{:}\,*\,.(M\ R)\to(M\ R) : (*\to*)\to * \to *.$$

The type of the $\mathtt{P}_{class}$ can be rewritten, using `Class`, as:

$$\mathtt{P}_{class} : \mathtt{Class}\ \mathtt{P}_{int}\ \mathtt{P}_{rep} : *,$$

for suitable internal representation, and method interface function. The Figure *5.2* shows the situation when we extend the class of points with a color component; here `new` is the operation that instantiates an object from a class.

### 5.2.6 Conclusions

As we have showed in this section, the existential model of Pierce and Turner [PT94] allows a rich account of the basic mechanisms of object-oriented programming languages. Among other properties we recall the possibility of subtyping. This is possible keeping separate the subtype and the inheritance hierarchy. The latter hierarchy is defined by the programmer as classes are defined, while the subtype hierarchy is *inferred* from types according to the subtyping rules. The main result about subtyping is the *decidability* of the subtyping relation [Com94]. This model features both subtyping in "width" and in "depth".

Due to the use of existential-types, the treatment of binary methods is very difficult. In [PT94] such methods where divided into two different categories:

**"Strong" Binary Methods,** whose implementation depends upon the ability to obtain a concrete access to the internal state of several objects at the same time. An example of this category is the `union` operation on sets of integers:

$$\texttt{set}_1 \Leftarrow \texttt{union set}_2.$$

This operation must access the internal representation of both the sets of integers, which normally are not visible outside. This necessarily will break the encapsulation of the argument $\texttt{set}_2$.

**"Weak" Binary Methods,** which accept as input one or more arguments of the same type of the receiver, but need neither to know, nor to access the internal representation of these arguments. For example the, so called, *equality* methods are in this category. In fact, to compare two points it sufficies to ask the argument for the values of its his coordinates and this can be done by sending messages.

The existential model does not support binary methods directly. [PT93] proposes an easy generalization of the Cardelli and Wegner *partially abstract data types* [CW85], that supports strong binary methods.

## 5.3 The Primitive Object Calculus of Abadi and Cardelli

In this section, we present the calculus of Abadi and Cardelli [AC94]. This calculus supports method override, method specialization, and "width" subtyping. No object extension is provided, since the objects have fixed size.

The calculus is untyped, and a sound static type system is given. The only operation allowed on objects are method invocation and method override. The objects are very simple, with just four syntactic forms, and without functions. The expressivity of the calculus is given via an encoding of the $\lambda$-calculus. In addition, a denotational semantics, based on partial equivalence relations, is given.

### 5.3.1 Syntax and Operational Semantics

The Calculus of Primitive Objects is defined as follows:

$$\mathtt{e} ::= s \mid \langle \mathtt{m}_1 = \varsigma(s_1)\mathtt{e}_1, \ldots, \mathtt{m}_k = \varsigma(s_k)\mathtt{e}_k \rangle \mid \mathtt{e.m} \mid \mathtt{e.m} \prec\!\!\!- \varsigma(s)\mathtt{e},$$

where $\mathtt{m}_i$ $(i \leq k)$ are method names, $\mathtt{e}_i$ $(i \leq k)$ are the bodies of methods, $s_i$ $(i \leq k)$ are bound parameters, referring to the object itself, and $\varsigma$ is a binder for $s_i$s. So, an object is a collection of pairs of method-names and method-bodies; the order of these pairs does not matter. The expression $\mathtt{e.m}$ means method invocation, and the expression $\mathtt{e.m} \prec\!\!\!- \varsigma(s)\mathtt{e}'$ means method override. If we identify the binder $\varsigma$ with the $\lambda$-binder, then bodies of methods are functions. Instead, the authors of [AC94] argue that with $\varsigma$-binders a more direct rewrite semantics can be given for the calculus, and moreover, the $\varsigma$-binders can be as expressive as the $\lambda$-binders, by producing a direct translation of the untyped $\lambda$-calculus into primitive objects (see Section 2.2 of [AC94]).

Let $\mathtt{e}\{s \leftarrow \mathtt{e}'\}$ denote the substitution of the term $\mathtt{e}'$ for the free occurrences of $s$ in $\mathtt{e}$. The operational semantics is defined as the least congruence relation generated by the following rules:

$$\begin{array}{lll} (Select) & \langle \ldots \mathtt{m}_i = \varsigma(s_i)\mathtt{e}_i \ldots \rangle.\mathtt{m}_i & \stackrel{eval}{\to} \mathtt{e}_i\{s_i \leftarrow \langle \ldots \mathtt{m}_i = \varsigma(s_i)\mathtt{e}_i \ldots \rangle\} \\ (Override) & \langle \ldots \mathtt{m}_i = \varsigma(s_i)\mathtt{e}_i \ldots \rangle.\mathtt{m}_i \prec\!\!\!- \varsigma(s_i)\mathtt{e} & \stackrel{eval}{\to} \langle \ldots \mathtt{m}_i = \varsigma(s_i)\mathtt{e} \ldots \rangle. \end{array}$$

To send the message $\mathtt{m}$ to the object $\mathtt{e}$ means to substitute the object itself (i.e. $\mathtt{e}$) in the body of $\mathtt{m}$. We can derive an untyped equational theory from the untyped reduction rules, by simply adding rules for symmetry, transitivity, and congruence, as follows:

$$\frac{\mathtt{e}_2 \stackrel{eval}{=} \mathtt{e}_1}{\mathtt{e}_1 \stackrel{eval}{=} \mathtt{e}_2}\ (Symm) \qquad \frac{\mathtt{e}_1 \stackrel{eval}{=} \mathtt{e}_2 \quad \mathtt{e}_2 \stackrel{eval}{=} \mathtt{e}_3}{\mathtt{e}_1 \stackrel{eval}{=} \mathtt{e}_3}\ (Trans)$$

$$\frac{}{s \stackrel{eval}{=} s}\ (Var) \qquad \frac{\mathtt{e}_i \stackrel{eval}{=} \mathtt{e}'_i \quad \forall i \in 1 \dots k}{\langle \mathtt{m}_i = \varsigma(s_i)\mathtt{e}_i \rangle^{i \in 1 \dots k} \stackrel{eval}{=} \langle \mathtt{m}_i = \varsigma(s_i)\mathtt{e}'_i \rangle^{i \in 1 \dots k}}\ (Object)$$

$$\frac{\mathtt{e}_1 \stackrel{eval}{=} \mathtt{e}_2}{\mathtt{e}_1.\mathtt{m} \stackrel{eval}{=} \mathtt{e}_2.\mathtt{m}}\ (Select) \qquad \frac{\mathtt{e}_1 \stackrel{eval}{=} \mathtt{e}_2 \quad \mathtt{e}'_1 \stackrel{eval}{=} \mathtt{e}'_2}{\mathtt{e}_1.\mathtt{m} \Leftarrow\!\!\!- \varsigma(s)\mathtt{e}'_1 \stackrel{eval}{=} \mathtt{e}_2.\mathtt{m} \Leftarrow\!\!\!- \varsigma(s)\mathtt{e}'_2}\ (Override)$$

$$\frac{j \in 1 \dots k}{\langle \mathtt{m}_i = \varsigma(s_i)\mathtt{e}_i \rangle^{i \in 1 \dots k}.\mathtt{m}_j \stackrel{eval}{=} \mathtt{e}_j\{s_j \leftarrow \langle \mathtt{m}_i = \varsigma(s_i)\mathtt{e}_i \rangle^{i \in 1 \dots k}\}}\ (Select^{\stackrel{eval}{\to}})$$

$$\frac{j \in 1 \dots k}{\langle \dots \mathtt{m}_i = \varsigma(s_i)\mathtt{e}_i \dots \rangle.\mathtt{m}_i \Leftarrow\!\!\!- \varsigma(s_i)\mathtt{e} \stackrel{eval}{=} \langle \dots \mathtt{m}_i = \varsigma(s_i)\mathtt{e} \dots \rangle}\ (Override^{\stackrel{eval}{\to}})$$

The connection between equality $\stackrel{eval}{=}$ and $\stackrel{eval}{\to}$ is given by the fact that the $\stackrel{eval}{\to}$ reduction rule satisfies the Church-Rosser property.

The semantics of method update is functional: an override produces another object where the overridden method has been replaced by the new body. Let see, with help of some examples, how the operational semantics works.

*Example 5.3.1* *i)* Let $\mathtt{e}_1 \stackrel{def}{=} \langle \mathtt{m} = \varsigma(s)s.\mathtt{m} \rangle$. If we send message $\mathtt{m}$ to $\mathtt{e}_1$, then we have the following infinite computation:

$$\mathtt{e}_1.\mathtt{m} \stackrel{eval}{\to} s.\mathtt{m}\{s \leftarrow \mathtt{e}_1\} \equiv \mathtt{e}_1.\mathtt{m} \stackrel{eval}{\to} \dots$$

*ii)* Let $\mathtt{e}_2 \stackrel{def}{=} \langle \mathtt{m} = \varsigma(s)s \rangle$. If we send message $\mathtt{m}$ to $\mathtt{e}_2$, then we have the following computation:

$$\mathtt{e}_2.\mathtt{m} \stackrel{eval}{\to} s\{s \leftarrow \mathtt{e}_2\} \equiv \mathtt{e}_2.$$

*iii)* Let $\mathtt{e}_3 \stackrel{def}{=} \langle \mathtt{m} = \varsigma(s)(s.\mathtt{m} \Leftarrow\!\!\!- \varsigma(s')s') \rangle$. If we send message $\mathtt{m}$ to $\mathtt{e}_3$, then we have the following computation:

$$\mathtt{e}_3.\mathtt{m} \stackrel{eval}{\to} s\{(s.\mathtt{m} \Leftarrow\!\!\!- \varsigma(s')s') \leftarrow \mathtt{e}_2\} \equiv \mathtt{e}_2.\mathtt{m} \Leftarrow\!\!\!- \varsigma(s')s' \stackrel{eval}{\to} \mathtt{e}_2.$$

*iv)* (Primitive objects of Points) The primitive object expressing a point with a $\mathtt{mv_x}$ methods is as follows:

$$\mathtt{p} \stackrel{def}{=} \langle \mathtt{x} = 0, \mathtt{mv_x} = \varsigma(s)\lambda dx.s.\mathtt{x} \Leftarrow s.\mathtt{x} + dx \rangle,$$

whereas a bi-dimensional point can be expressed as primitive object as follows:

$$\mathtt{q} \stackrel{def}{=} \langle \mathtt{x} = 0, \mathtt{y} = 0, \mathtt{mv_x} = \varsigma(s)\lambda dx.s.\mathtt{x} \Leftarrow s.\mathtt{x} + dx, \mathtt{mv_y} = \varsigma(s)\lambda dy.s.\mathtt{y} \Leftarrow s.\mathtt{y} + dy \rangle.$$

*v)* The primitive object expressing a point with an `eq` methods is as follows:

$$\mathtt{p}' \stackrel{def}{=} \langle \mathtt{x} = 0, \mathtt{eq} = \varsigma(s)\lambda p.equal(s.\mathtt{x}, p.\mathtt{x}) \rangle.$$

*vi)* (Backup Methods) These primitive objects allow to "store" two different versions of self at once, i.e. we define an object that is able to keep backup copies of itself. Let the simple object `e` be defined as follows:

$$\mathtt{e} \stackrel{def}{=} \langle \mathtt{retrieve} = \varsigma(s)s, \mathtt{backup} = \varsigma(s)s.\mathtt{retrieve} \Leftarrow \varsigma(s')s \rangle.$$

If we send a `backup` message to `e`, then we have the following computation:

$$\begin{aligned}
\mathtt{e.backup} &\stackrel{eval}{\to} (s.\mathtt{retrieve} \Leftarrow \varsigma(s')s)\{s \leftarrow \mathtt{e}\} \\
&\equiv \mathtt{e.retrieve} \Leftarrow \varsigma(s')\mathtt{e} \\
&\stackrel{eval}{\to} \langle \mathtt{retrieve} = \varsigma(s')\mathtt{e}, \mathtt{backup} = \varsigma(s)s.\mathtt{retrieve} \Leftarrow \varsigma(s')s \rangle \\
&\stackrel{def}{=} \mathtt{e}_1.
\end{aligned}$$

If we want to extract the object that was most recently backed up, then we can send the `retrieve` message to $\mathtt{e}_1$, which gives the following computation:

$$\mathtt{e}_1.\mathtt{retrieve} \stackrel{eval}{\to} \mathtt{e}\{s \leftarrow \mathtt{e}_1\} \equiv \mathtt{e}.$$

### 5.3.2 The Type System: a Survey

The type system of the Calculus of Primitive Object is composed by several fragments, each of them necessary to give a correct type to different objects of this calculus. For example, to give a type to those objects which contain only methods whose results are not the object itself, a first-order fragment of the type system would suffice. On the other hand, to give a type of the point objects with move methods also recursion and, hence, recursive-types are needed. Finally, if we want to include a subsumption relation between objects, then we must extend this type system with polymorphic-types.

We start this brief survey of the Abadi Cardelli type system with the definition of an object-type: in its simpler form (first-order) it has the following shape:

$$\langle\!\langle \mathtt{m}_1{:}\tau_1, \ldots, \mathtt{m}_k{:}\tau_k \rangle\!\rangle \ (k \geq 1),$$

where the $\tau_i$s are constant-types or arrow-types. We assume that the $\mathtt{m}_i$ are distinct and that permutations do not matter. When a method is invoked, it produces a result having the corresponding type $\tau_i$. Worthy noticing is that the $\varsigma$-bound type of the bound variable is not explicitly listed in the object-type. The judgments have the following form:

$$\Gamma \vdash \tau, \text{ and } \Gamma \vdash \mathtt{e} : \tau,$$

where $\Gamma$ is a context which gives types to the free variables of $\tau$ and $\mathtt{e}$. The first judgment checks the well-formations of the type $\tau$, and the second assigns a type $\tau$ to the expression $\mathtt{e}$. The main typing rules are:

$$\frac{\Gamma \vdash \tau_i \qquad \forall i \in 1 \ldots k}{\Gamma \vdash \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i \in 1 \ldots k}} \ (Object{-}Type)$$

$$\frac{\tau \equiv \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i \in 1 \ldots k} \qquad \Gamma, s_i{:}\tau \vdash \mathtt{e}_i : \tau_i \qquad \forall i \in 1 \ldots k}{\Gamma \vdash \langle \mathtt{m}_i = \varsigma(s_i{:}\tau)\mathtt{e}_i \rangle^{i \in 1 \ldots k} : \tau} \ (Object)$$

$$\frac{\Gamma \vdash \mathtt{e} : \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i \in 1 \ldots k} \qquad j \in 1 \ldots k}{\Gamma \vdash \mathtt{e.m}_j : \tau_j} \ (Select)$$

$$\frac{\tau \equiv \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i \in 1 \ldots k} \qquad \Gamma \vdash \mathtt{e} : \tau \qquad \Gamma, s{:}\tau \vdash \mathtt{e}'{:}\tau_j \qquad j \in 1 \ldots k}{\Gamma \vdash \mathtt{e.m}_j \Leftarrow\!\!\!\!- \varsigma(s{:}\tau)\mathtt{e}' : \tau} \ (Override)$$

In addition to the previously showed rules, we have also rules for well-formation of contexts and rules for dealing with constant-type and function-types. Note that the above typing system is given for a “fully-typed” variant of the primitive calculus. This calculus enjoys the nice property of unicity of types:

*Property 5.3.2* (Unicity of Typing) *If* $\Gamma \vdash \mathtt{e} : \tau$ *then* $\tau$ *is unique.* ∎

For this calculus, we can reformulate the (previously showed) equational theory where the types of the object are explicitly listed in the theory. For example, the $(Select^{\overset{eval}{\rightarrow}})$ rule can be reformulated as follows:

$$\frac{\tau \equiv \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i\in 1\ldots k} \qquad \mathtt{e} \equiv \langle \mathtt{m}_i = \varsigma(s_i{:}\tau)\mathtt{e}_i \rangle^{i\in 1\ldots k} \qquad \Gamma \vdash \mathtt{e} : \tau \qquad j \in 1\ldots k}{\Gamma \vdash \mathtt{e.m}_j \overset{eval}{=} \mathtt{e}_j\{s_j{\leftarrow}\mathtt{e}\} : \tau_j} \quad (Select^{\overset{eval}{\rightarrow}})$$

The resulting equational theory is too fine. In fact we can easily find two objects that are not equal in the theory, that have the same type, that gives equal results for all their methods, and still are distinguishable. As an example, look at the following two objects:

$$\begin{aligned} \tau &\overset{def}{=} \langle\!\langle \mathtt{x}{:}nat, \mathtt{f}{:}nat \rangle\!\rangle \\ \mathtt{e}_1{:}\tau &\overset{def}{=} \langle \mathtt{x} = 1, \mathtt{f} = \varsigma(s{:}\tau)1 \rangle \\ \mathtt{e}_2{:}\tau &\overset{def}{=} \langle \mathtt{x} = 1, \mathtt{f} = \varsigma(s{:}\tau)s.\mathtt{x} \rangle. \end{aligned}$$

### 5.3.3 Adding Subtyping

In dealing with subtyping, it is useful to add a type-constant $Top$, such that $\sigma \preceq Top$, for all $\sigma$. The subtyping relation allows to use an object of type $\sigma$ in any context expecting an object of type $\tau$, provided that $\sigma \preceq \tau$. The subtyping rule for objects allows an object with more methods to be a subtype of another object with less methods.

$$\frac{\Gamma \vdash \tau_i \qquad \forall i \in 1\ldots k+h}{\Gamma \vdash \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i\in 1\ldots k+h} \preceq \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i\in 1\ldots k}} \quad (Type_{\preceq})$$

The rules for arrow-type are standard (contravariant in the domain). The typing rules must take the introduction of a subtyping relation into account. To have a system with the minimum type property, Abadi and Cardelli do not introduce a subsumption rule, but modify the $(Override)$ and $(Object)$ rules. For example, the $(Override)$ rule is rewritten to:

$$\frac{\begin{array}{lll} \tau \equiv \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i\in 1\ldots k} & \Gamma \vdash \mathtt{e} : \tau' & \Gamma \vdash \tau' \preceq \tau \\ \Gamma, s{:}\tau \vdash \mathtt{e}'{:}\tau_j' & \Gamma \vdash \tau_j' \preceq \tau_j & j \in 1\ldots k \end{array}}{\Gamma \vdash \mathtt{e.m}_j \Leftarrow\!\!\!- \varsigma(s{:}\tau)\mathtt{e}' : \tau} \quad (Override)$$

The first-order Primitive Calculus with subtyping enjoys the minimum type property.

*Property 5.3.3 If* $\Gamma \vdash \mathtt{e} : \tau$, *then there exists* $\sigma$ *such that* $\Gamma \vdash \sigma \preceq \tau$ *and* $\Gamma \vdash \mathtt{e} : \sigma$, *and, for all* $\sigma'$ *such that* $\Gamma \vdash \mathtt{e} : \sigma'$, $\Gamma \vdash \mathtt{e} : \sigma'$. ∎

The equational theory still can be updated, in order to take into account the presence of subsumption. The first two rules we need are the following:

$$\frac{\Gamma \vdash \mathtt{e} \stackrel{eval}{=} \mathtt{e}' : \sigma \qquad \Gamma \vdash \sigma \preceq \tau}{\Gamma \vdash \mathtt{e} \stackrel{eval}{=} \mathtt{e}' : \tau} \ (Sub_{\preceq}) \quad \frac{\Gamma \vdash \mathtt{e} : \sigma \qquad \Gamma \vdash \mathtt{e} : \tau}{\Gamma \vdash \mathtt{e} \stackrel{eval}{=} \mathtt{e}' : Top} \ (Top)$$

In addition, we need to reformulate some rules of the equational theory to compare objects with differents number of methods. For example, the $(Select)$ rule can be reformulated as follows:

$$\frac{\tau \equiv \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i \in 1 \ldots k} \qquad \mathtt{e} \equiv \langle \mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathtt{e}_i \rangle^{i \in 1 \ldots k+h} \qquad \Gamma \vdash \mathtt{e} : \tau \qquad j \in 1 \ldots k}{\Gamma \vdash \mathtt{e}.\mathtt{m}_j \stackrel{eval}{=} \mathtt{e}_j\{s_j \leftarrow \mathtt{e}\} : \tau_j} \ (Select^{\stackrel{eval}{\rightarrow}})$$

whereas the $(Object)$ rule can be reformulated as follows:

$$\frac{\begin{array}{ll} \tau \equiv \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i \in 1 \ldots k} & \tau' \equiv \langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i \in 1 \ldots k+h} \\ \Gamma, s_i{:}\tau \vdash \mathtt{e}_i : \tau_i & \forall i \in 1 \ldots k \\ \Gamma, s_i{:}\tau' \vdash \mathtt{e}_j : \tau_j & \forall j \in k+1 \ldots k+h \end{array}}{\Gamma \vdash \langle \mathtt{m}_i = \varsigma(s_i{:}\tau)\mathtt{e}_i \rangle^{i \in 1 \ldots k} \stackrel{eval}{=} \langle \mathtt{m}_i = \varsigma(s_i{:}\tau')\mathtt{e}_i \rangle^{i \in 1 \ldots k+h} : \tau} \ (Object_{\preceq})$$

This rule is sound only if the the remaining methods do not depend on the forgotten ones, otherwise the truncated object will produce run-time errors.

### 5.3.4 Adding Recursive-Types

In order to give a type to the primitive objects "point" with "move" methods, we need to add recursive-types to the type system. In its simpler form, a recursive object-type has the following form:

$$\mu(t)\langle\!\langle \mathtt{m}_i{:}\tau_i \rangle\!\rangle^{i \in 1 \ldots k},$$

where the bound type-variable $t$ can occur in the $\tau_i$s. We add to the primitive calculus one syntactic form, namely:

$$\mu(s{:}\tau)\mathtt{e},$$

where the $\mu$-bound variable $s$ can occur in $\mathtt{e}$. The typing rules are enriched with some rules of type formation, and with the following rules:

$$\frac{\Gamma \vdash \mathtt{e} : \tau\{t{\leftarrow}\mu(t)\tau\}}{\Gamma \vdash fold(\mu(t)\tau, \mathtt{e}) : \mu(t)\tau}\ (Fold) \qquad \frac{\Gamma \vdash \mathtt{e} : \mu(t)\tau}{\Gamma \vdash unfold(\mathtt{e}) : \tau\{t{\leftarrow}\mu(t)\tau\}}\ (Unfold)$$

$$\frac{\Gamma, s{:}\tau \vdash \mathtt{e} : \tau}{\Gamma \vdash \mu(s{:}\tau)\mathtt{e} : \tau}\ (Rec)$$

Again the equational theory has to be redefined, taking into account the presence of recursive-types (see Section 5.1 of [AC94]). The obtained equational theory allows to move around the isomorphism of types $\mu(t)\tau$ and $\tau\{t{\leftarrow}\mu(t)\tau\}$ by means of $Fold/Unfold$ coercions on terms.

If we still want to have subtyping, we need to reformulates the rules, since type-variables in environments require subtype bounds. Moreover, we must add the following rule, which determines the variance behaviour of the recursive-types:

$$\frac{\Gamma \vdash \mu(t)\tau \qquad \Gamma \vdash \mu(t')\tau' \qquad \Gamma, t' \preceq Top, t \preceq t' \vdash \tau \preceq \tau'}{\Gamma \vdash \mu(t)\tau \preceq \mu(t')\tau'}\ (Rec_{\preceq})$$

It is easy to verify that with this definition of subtyping we get:

$$\mu(t)\langle\!\langle \mathtt{x}{:}int, \mathtt{y}{:}int, \mathtt{mv_x}{:}int{\to}t, \mathtt{mv_y}{:}int{\to}t\rangle\!\rangle \preceq \mu(t)\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int{\to}t\rangle\!\rangle.$$

Unfortunately, if we accept subtyping between these two types, then we obtain an inconsistency (see Section 5.4 of [AC94], and Subsection *7.2.4* of this thesis). Starting from this inconsistency, Abadi and Cardelli present a second-order extension of the first-order calculus with recursion to produce the formalization of the type of *self*, i.e. the "mytype" specialization. We do not repeat here that second-order extension. The interested reader is referred to Section 6 of [AC94].

# Chapter 6

# The Lambda Calculus of Objects

## Introduction

As we saw in the previous chapter, Object-Oriented languages can be classified either as *class-based* or *delegation-based* languages. In class-based languages, such as SMALLTALK [GR83] and $C^{++}$ [ES90], the implementation of an object is specified by its class and objects are created by instantiating their classes. In delegation-based languages, like SELF [US87, CU89], objects are defined directly from other objects by adding new methods via *method addition* and replacing old methods bodies with new ones via *method override*. Adding or overriding a method produces a new object that inherits all the properties of the original. In this chapter, we consider the delegation-based axiomatic model developed by Fisher, Honsell and Mitchell, and, in particular, we refer to the model in [FHM94], and in [FM94]. This calculus, from now on called $\lambda_{obj}$, offers:

*i)* a very simple and effective inheritance mechanism,

*ii)* a straightforward *mytype* method specialization,

*iii)* dynamic lookup of methods, and

*iv)* easy definition of binary methods.

The $\lambda_{obj}$-calculus is essentially an untyped lambda calculus enriched with object primitives. There are three operations on objects: *method addition* ($\langle \mathtt{e}_1 \leftarrow\!\!+\ \mathtt{m}{=}\mathtt{e}_2 \rangle$) to define methods, *method override* ($\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle$) to re-define methods, and *method call* ($\mathtt{e} \Leftarrow \mathtt{m}$) to send a message $\mathtt{m}$ to an object $\mathtt{e}$. In $\lambda_{obj}$, the method addition makes sense only if method $\mathtt{m}$ does not occur in the object $\mathtt{e}$, while method override can be done only if $\mathtt{m}$ occurs in $\mathtt{e}$. If the expression $\mathtt{e}_1$ denotes an object without method $\mathtt{m}$, then $\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$ denotes a new object obtained from $\mathtt{e}_1$ by adding the method body $\mathtt{e}_2$ for $\mathtt{m}$. When the message $\mathtt{m}$ is sent to $\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$, the result is obtained by applying $\mathtt{e}_2$ to $\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$ (similarly for $\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle$).

This form of *self-application* allows to model the special symbol *self* of object-oriented languages directly by lambda abstraction. Intuitively, the method body $\mathtt{e}_2$ is a function and the first actual parameter of $\mathtt{e}_2$ will always be the object itself. The type system of this calculus allows methods to be specialized appropriately as they are inherited.

This chapter is organized as follows: Section *6.1* contains the syntax of the $\lambda_{obj}$-calculus, its operational semantics together with some examples useful to understand the method specialization. Section *6.2* contains the presentation of the type system of $\lambda_{obj}$. In Section *6.3*, we present the subject reduction and the type-soundness theorems. Section *6.4* shows the expressiveness of this calculus. The last section is devoted conclusions and to relate $\lambda_{obj}$ with the calculus of Abadi and Cardelli [AC94].

## 6.1 The $\lambda_{obj}$: Syntax and Semantics

In this section, we will present the syntax of the $\lambda_{obj}$-calculus, its operational semantics, and some examples useful to understand the method specialization.

### 6.1.1 Syntax of the Core Language

The untyped lambda calculus enriched with object-related syntactic forms is defined as follows:

$$\mathtt{e} ::= x \mid \lambda x.\mathtt{e} \mid \mathtt{e}_1\mathtt{e}_2 \mid \langle\rangle \mid \langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle \mid \langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle \mid \mathtt{e} \Leftarrow \mathtt{m},$$

where $x$ is a term variable, and $\mathtt{m}$ is a method name. The object forms are:

| | |
|---|---|
| $\langle\rangle$ | the empty object; |
| $\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}{=}\mathtt{e}_2 \rangle$ | extends object $\mathtt{e}_1$ with a new method $\mathtt{m}$ having body $\mathtt{e}_2$; |
| $\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle$ | replaces the body of method $\mathtt{m}$ in $\mathtt{e}_1$ by $\mathtt{e}_2$; |
| $\mathtt{e} \Leftarrow \mathtt{m}$ | sends message $\mathtt{m}$ to the object $\mathtt{e}$. |

Let $\leftarrow\!\!*$ denote either $\leftarrow\!\!+$ or $\leftarrow$. The description of an object via $\leftarrow\!\!*$ operations is intentional, and the object corresponding to a sequence of $\leftarrow\!\!*$ can be extensionally defined as follows.

*Definition 6.1.1 Let* $\mathtt{m}_1, \ldots, \mathtt{m}_k$, *and* $\mathtt{n}$ *be distinct method names.* $\langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k \rangle$ *is defined as:*

$$\begin{aligned}
\langle \ldots \langle \langle\rangle \leftarrow\!\!+\, \mathtt{m}_1{=}\mathtt{e}_1 \rangle \ldots \leftarrow\!\!+\, \mathtt{m}_k{=}\mathtt{e}_k \rangle &\stackrel{\text{ext}}{=} \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k \rangle \\
\langle \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_i{=}\mathtt{e}_i, \ldots, \mathtt{m}_k{=}\mathtt{e}_k \rangle \leftarrow \mathtt{m}_i{=}\mathtt{e}'_i \rangle &\stackrel{\text{ext}}{=} \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_i{=}\mathtt{e}'_i, \ldots, \mathtt{m}_k{=}\mathtt{e}_k \rangle \\
\langle \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k \rangle \leftarrow\!\!+\, \mathtt{n}{=}\mathtt{e}' \rangle &\stackrel{\text{ext}}{=} \langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k, \mathtt{n}{=}\mathtt{e}' \rangle.
\end{aligned}$$

So $\langle \mathtt{m}_1{=}\mathtt{e}_1, \ldots, \mathtt{m}_k{=}\mathtt{e}_k \rangle$ represents, extensionally, the behaviour of an object in which each method body corresponds to the outermost assignment (by addition or by override) performed on the method.

As we will see in the next subsection, this extensional behaviour will not be useful to define an operational semantics, because this would imply to treat objects as *set of methods*, rather than *ordered sequences*. This complication arises also in the typing rules, since our typing rules allow us to type object expressions when methods are added in an appropriate order.

### 6.1.2 The Operational Semantics of $\lambda_{obj}$

The operational semantics of the $\lambda_{obj}$ -calculus is mainly based on the following evaluation rules for $\beta$-reduction and for message sending $\Leftarrow$-reduction:

$$\begin{array}{lll}
(\beta) & (\lambda x.\mathtt{e}_1)\mathtt{e}_2 & \stackrel{eval}{\to} [\mathtt{e}_2/x]\mathtt{e}_1 \\
(\Leftarrow) & \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2 \rangle \Leftarrow \mathtt{m} & \stackrel{eval}{\to} \mathtt{e}_2 \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2 \rangle.
\end{array}$$

Sending a message $\mathtt{m}$ to the object $\mathtt{e}$ means applying the body of $\mathtt{m}$ to the object itself (i.e. $\mathtt{e}$); in fact, the body of $\mathtt{m}$ is a lambda abstraction whose bound variable will be substituted by the full object in the next step of $\beta$-reduction.

The problem that arises in the calculus of objects is how to extract the appropriate method out of an object; the more natural rule to move the required method in a accessible position is a "free-permutation" rule that allows to treat objects more as sets of methods. Unfortunately, this approach is not possible: in fact, the typing rules give type only to object expression when methods are added in an appropriate order: this means that an object $\langle\langle \mathtt{e}_1 \leftarrow\!\!* \, \mathtt{m} = \mathtt{e}_2\rangle \leftarrow\!\!* \, \mathtt{n} = \mathtt{e}_3\rangle$ is correctly typed if and only if all proper "subobjects", namely $\mathtt{e}_1$ and $\langle \mathtt{e}_1 \leftarrow\!\!* \, \mathtt{m} = \mathtt{e}_2\rangle$, are correctly typed too. In fact, the following extensional rule

$$\langle\langle \mathtt{e}_1 \leftarrow\!\!* \, \mathtt{m} = \mathtt{e}_2\rangle \leftarrow\!\!* \, \mathtt{n} = \mathtt{e}_3\rangle = \langle\langle \mathtt{e}_1 \leftarrow\!\!* \, \mathtt{n} = \mathtt{e}_3\rangle \leftarrow\!\!* \, \mathtt{m} = \mathtt{e}_2\rangle,$$

is *unsound*, because it treats objects as sets of methods, instead of ordered sequences. The approach chosen by [FHM94] to solve the problem of method order was to add to the $\stackrel{eval}{\to}$ relation a *bookeeping* relation, denoted by $\stackrel{book}{\to}$, that leads to a *standard form* object, in which each method is defined *exactly* once (with the extension operation) using, perhaps, some "dummy" bodies, and redefined *exactly* once (with the override operation), giving it the desired body. The standard form and the bookeeping rules are presented in the next definition.

*Definition 6.1.2 Let* m *and* n *be distinct method names.*

*i) The* standard form *of an object is defined as follows:*

$$\langle \ldots \langle\langle\rangle \leftarrow\!\!+ \, \mathtt{m}_1 = \mathtt{e}_1\rangle \ldots\rangle \leftarrow\!\!+ \, \mathtt{m}_n = \mathtt{e}_n \ldots\rangle \leftarrow \mathtt{m}_1 = \mathtt{e}'_1\rangle \ldots\rangle \leftarrow \mathtt{m}_n = \mathtt{e}'_n \ldots\rangle.$$

*ii) The bookeeping reduction rules, that rewrite an object to its standard form, are:*

$$\begin{array}{lll}
\text{(switch)} & \langle\langle \mathtt{e}_1 \leftarrow \mathtt{n} = \mathtt{e}_2\rangle \leftarrow\!\!* \, \mathtt{m} = \mathtt{e}_3\rangle & \stackrel{book}{\to} \langle\langle \mathtt{e}_1 \leftarrow\!\!* \, \mathtt{m} = \mathtt{e}_3\rangle \leftarrow \mathtt{n} = \mathtt{e}_2\rangle \\
\text{(add)} & \langle \mathtt{e}_1 \leftarrow\!\!+ \, \mathtt{m} = \mathtt{e}_3\rangle & \stackrel{book}{\to} \langle\langle \mathtt{e}_1 \leftarrow\!\!+ \, \mathtt{m} = \mathtt{e}_3\rangle \leftarrow \mathtt{m} = \mathtt{e}_3\rangle \\
\text{(cancel)} & \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2\rangle \leftarrow \mathtt{m} = \mathtt{e}_3\rangle & \stackrel{book}{\to} \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_3\rangle.
\end{array}$$

The evaluation relation is the congruence closure of the union of the $\stackrel{eval}{\to}$, and $\stackrel{book}{\to}$ reduction rules. The original bookkeeping rules were expressed in [FHM94] as follows:

$$\begin{array}{lll}
\text{(switch ext ov)} & \langle\langle \mathtt{e}_1 \leftarrow \mathtt{n} = \mathtt{e}_2\rangle \leftarrow\!\!+ \, \mathtt{m} = \mathtt{e}_3\rangle & \stackrel{book}{\to} \langle\langle \mathtt{e}_1 \leftarrow\!\!+ \, \mathtt{m} = \mathtt{e}_3\rangle \leftarrow \mathtt{n} = \mathtt{e}_2\rangle \\
\text{(perm ov ov)} & \langle\langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2\rangle \leftarrow \mathtt{n} = \mathtt{e}_3\rangle & \stackrel{book}{\to} \langle\langle \mathtt{e}_1 \leftarrow \mathtt{n} = \mathtt{e}_3\rangle \leftarrow \mathtt{m} = \mathtt{e}_2\rangle \\
\text{(add ov)} & \langle \mathtt{e}_1 \leftarrow\!\!+ \, \mathtt{m} = \mathtt{e}_3\rangle & \stackrel{book}{\to} \langle\langle \mathtt{e}_1 \leftarrow\!\!+ \, \mathtt{m} = \mathtt{e}_3\rangle \leftarrow \mathtt{m} = \mathtt{e}_3\rangle \\
\text{(cancel ov ov)} & \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2\rangle \leftarrow \mathtt{m} = \mathtt{e}_3\rangle & \stackrel{book}{\to} \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_3\rangle.
\end{array}$$

### 6.1.3 Examples of Objects, Inheritance and Self-References

To provide some intuition for this calculus, we give a few short examples. To simplify notation, these example are, sometimes, presented extensionally, according to Definition *6.1.1*.

*Example 6.1.3* (Bi-dimensional Point) Let the object $\mathtt{p}$ be defined as follows:

$$\mathtt{p} \stackrel{def}{=} \langle\langle\langle\rangle \leftarrow\!\!+\, \mathtt{x} = \lambda self.3\rangle \leftarrow\!\!+\, \mathtt{y} = \lambda self.3\rangle \stackrel{ext}{=} \langle \mathtt{x} = \lambda self.3, \mathtt{y} = \lambda self.3\rangle.$$

Here, we have encoded a record with two components, by adding two methods $\mathtt{x}$ and $\mathtt{y}$ to the empty object $\langle\rangle$. Note that the order of this addition is not important. If we send a message $\mathtt{x}$ to the object $\mathtt{p}$, then the result will be the application of the body of $\mathtt{x}$ in $\mathtt{p}$ to the object $\mathtt{p}$ itself, i.e.

$$\mathtt{p} \Leftarrow \mathtt{x} \stackrel{eval}{\to} (\lambda self.3)\mathtt{p} \stackrel{eval}{\to} 3.$$

*Example 6.1.4* (Point with $\mathtt{mv_x}$ method) Let the object $\mathtt{p}$ be defined as follows:

$$\begin{aligned} \mathtt{p} &\stackrel{def}{=} \langle\langle\langle\rangle \leftarrow\!\!+\, \mathtt{x} = \lambda self.3\rangle \leftarrow\!\!+\, \mathtt{mv_x} = \lambda self.\lambda dx.\langle self \leftarrow \mathtt{x} = \lambda s.(self \Leftarrow x) + dx\rangle\rangle \\ &\stackrel{ext}{=} \langle \mathtt{x} = \lambda self.3, \mathtt{mv_x} = \lambda self.\lambda dx.\langle self \leftarrow \mathtt{x} = \lambda s.(self \Leftarrow x) + dx\rangle\rangle. \end{aligned}$$

The order in which we introduce the $\mathtt{x}$ and the $\mathtt{mv_x}$ methods is important: since the body of the $\mathtt{mv_x}$ method "uses" the method $\mathtt{x}$, it cannot be introduced *before* the introduction of $\mathtt{x}$. If we send a $\mathtt{mv}$ message to $\mathtt{p}$ then we have the following computation:

$$\begin{aligned} \mathtt{p} \Leftarrow \mathtt{mv_x} 4 &\stackrel{eval}{\to} (\lambda self.\lambda dx.\langle \ldots \rangle)\mathtt{p}4 \\ &\stackrel{eval}{\to} \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda s.(\mathtt{p} \Leftarrow \mathtt{x}) + 4\rangle \\ &\stackrel{eval}{\to} \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda s.((\lambda self.3)\mathtt{p}) + 4\rangle \\ &\stackrel{eval}{\to} \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda s.3 + 4\rangle \\ &\stackrel{eval}{\to} \langle \mathtt{p} \leftarrow \mathtt{x} = \lambda s.7\rangle. \end{aligned}$$

Using again the extensionality (Definition *6.1.1*), we may say:

$$\mathtt{p} \Leftarrow \mathtt{mv_x}\ 4 \stackrel{ext}{=} \langle \mathtt{x} = \lambda self.7, \mathtt{mv_x} = \ldots \rangle.$$

*Example 6.1.5* (Diagonal-Point) Let the object $\mathtt{p}$ be defined as follows:

$$\mathtt{p} \stackrel{def}{=} \langle\langle\langle\rangle \leftarrow\!\!+\, \mathtt{x} = \lambda self.3\rangle \leftarrow\!\!+\, \mathtt{y} = \lambda self.self \Leftarrow \mathtt{x}\rangle \stackrel{ext}{=} \langle \mathtt{x} = \lambda self.3, \mathtt{y} = \lambda self.self \Leftarrow \mathtt{x}\rangle.$$

Again the order of methods addition is important. In fact, the type system will not type the following expression:

$$\mathtt{wrong} \stackrel{def}{=} \langle\langle\langle\rangle \leftarrow\!\!+\, \mathtt{y} = \lambda self.self \Leftarrow \mathtt{x}\rangle \leftarrow\!\!+\, \mathtt{x} = \lambda self.3\rangle.$$

If we want to type `wrong`, then we need to give a correct type to all subexpressions of the whole object. But we cannot give any type to the subexpression

$$\langle\langle\rangle \leftarrow\!\!+\, \mathtt{y} = \lambda self.self \Leftarrow \mathtt{x}\rangle,$$

because the empty object $\langle\rangle$ does not have any `x` method.

*Example 6.1.6* (A "funny" object) Let the object `funny` be defined as follows:

$$\mathtt{funny} \stackrel{def}{=} \langle\langle\rangle \leftarrow\!\!+\, \mathtt{m} = \lambda self.\langle self \leftarrow \mathtt{m} = \lambda s.s \Leftarrow \mathtt{m}\rangle\rangle.$$

This is an object whose evaluation may be infinite. If we send the message `m` to `funny`, then we have the following computation:

$$\begin{aligned} \mathtt{funny} \Leftarrow \mathtt{m} &\stackrel{eval}{\to} (\lambda self.\langle self \leftarrow \mathtt{m} = \lambda s.s \Leftarrow \mathtt{m}\rangle)\mathtt{funny} \\ &\stackrel{eval}{\to} \langle \mathtt{funny} \leftarrow \mathtt{m} = \lambda s.s \Leftarrow \mathtt{m}\rangle, \end{aligned}$$

where $\langle \mathtt{funny} \leftarrow \mathtt{m} = \lambda s.s \Leftarrow \mathtt{m}\rangle \stackrel{ext}{=} \langle \mathtt{m} = \lambda s.s \Leftarrow \mathtt{m}\rangle$. If we send the message `m` twice to `funny`, then the computation becomes infinite:

$$\begin{aligned} (\mathtt{funny} \Leftarrow \mathtt{m}) \Leftarrow \mathtt{m} &\stackrel{eval^*}{\to} \langle \mathtt{funny} \leftarrow \mathtt{m} = \lambda s.s \Leftarrow \mathtt{m}\rangle \Leftarrow \mathtt{m} \\ &\stackrel{eval}{\to} (\lambda s.s \Leftarrow \mathtt{m})\langle \mathtt{funny} \leftarrow \mathtt{m} = \lambda s.s \Leftarrow \mathtt{m}\rangle \\ &\stackrel{eval}{\to} \langle \mathtt{funny} \leftarrow \mathtt{m} = \lambda s.s \Leftarrow \mathtt{m}\rangle \Leftarrow \mathtt{m} \\ &\stackrel{eval^*}{\to} \langle \mathtt{funny} \leftarrow \mathtt{m} = \lambda s.s \Leftarrow \mathtt{m}\rangle \Leftarrow \mathtt{m} \\ &\stackrel{eval}{\to} \ldots \end{aligned}$$

In Section *6.4*, a simpler object with an infinite computation, called $\Omega$, will be presented.

## 6.2 Static Type System

In this section, we will present the type system of $\lambda_{obj}$, together with its most important features, i.e. the "mytype" method specialization. The basic calculus we use is $\lambda$-calculus with records in the general style of [CM91]. In addition to function-types we have also polymorphism and recursive-type definitions.

### 6.2.1 Static and Strong Typing

We will briefly discuss *static typing* and *strong typing*, following [Car95]. First we must distinguish between checked and unchecked errors. *Checked* errors cause the computation to stop immediately. *Unchecked* errors are those errors that go unnoticed (for a while) during a computation. The run-time error *message–not–understood* is a checked error: when it occurs, it cause the computation to stop immediately. Also the *division–by–zero* is a checked error. Examples of an unchecked errors are improperly accessing a legal address, or jumping to the wrong address. Each language designates a subset of the program errors aas *type errors*: this set must include all unchecked errors, plus a subset of the checked errors. With respect to a type system, a strongly-typed language is such that:

*i)* no unchecked errors occur (this is often called "safeness" or "soundness");

*ii)* none of the checked errors designated as type error occur;

*iii)* other checked errors may occur; it is the programmer's responsibility to handle or prevent them.

A statically typed language performs compile-time checks to prevent run-time type errors. Thus, the $\lambda_{obj}$-calculus is a statically typed language whose type system is sound and catches the *message-not-understood* run-time error.

### 6.2.2 Message-Send and Method Specialization

The central part of the type system of an object oriented language consists of the types of objects: in [FHM94] an object-type has the form:

$$\texttt{class}\, t.\langle\!\langle \texttt{m}_1{:}\tau_1, \ldots, \texttt{m}_k{:}\tau_k \rangle\!\rangle,$$

where $\langle\!\langle \mathtt{m}_1{:}\tau_1, \ldots, \mathtt{m}_k{:}\tau_k \rangle\!\rangle$, called a *row* expression, is essentially a record-type. In what follows, we denote `class` by `obj`, in the style of [FM94]. The object-type describes the properties of any object `e` that can receive messages $\mathtt{m}_i$ ($\mathtt{e} \Leftarrow \mathtt{m}_i$), producing results of type $\tau_i$, for $1 \leq i \leq k$. The bound variable $t$ may appear in $\tau_i$, referring to the object itself. Thus, an object-type is a form of recursively-defined type. The previously showed Example *6.1.4* of point with move method will have the following object-type:

$$Point \stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int \rightarrow t \rangle\!\rangle.$$

The type system gives a type to a message send, according to the following (although informally described) typing rule:

$$\frac{\mathtt{e} : \mathtt{obj}\, t.\langle\!\langle \ldots \mathtt{m}{:}\tau \rangle\!\rangle}{\mathtt{e} \Leftarrow \mathtt{m} : [\mathtt{obj}\, t.\langle\!\langle \ldots \mathtt{m}{:}\tau \rangle\!\rangle / t]\tau} \quad (send)$$

where the substitution of $t$ in $\tau$ reflects an unfolding step of the recursive-type `obj`. A significant aspect of the type system is that the type ($int \rightarrow t$) of the method $\mathtt{mv_x}$ does not change syntactically if we perform a method addition of a method `col`

$$\mathtt{cp} \stackrel{def}{=} \langle \mathtt{p} \leftarrow\!\!+\; \mathtt{col} = \lambda self.black \rangle,$$

to build a colored point object from an object point. Instead, the meaning of the type changes, since, before the `col` addition, the bound variable $t$ referred to an object of type $Point$, and, after, $t$ refers to an object of type:

$$CPoint \stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int \rightarrow t, \mathtt{col}{:}Col \rangle\!\rangle.$$

### 6.2.3 Operational Equivalence and Objects-Types

The type system of $\lambda_{obj}$ has as important peculiarity that is it may assign different types to different objects that behave identically, with respect to an observational congruence. The following two object are taken from [FHM94]:

$$\begin{aligned} \mathtt{p} &\stackrel{def}{=} \langle \mathtt{x} = \lambda self.3, \mathtt{mv_x} = \lambda self.\lambda dx.\langle self \leftarrow \mathtt{x} = \lambda s.(self \Leftarrow x) + dx \rangle \rangle \\ \mathtt{q} &\stackrel{def}{=} \langle \mathtt{x} = \lambda self.3, \mathtt{mv_x} = \lambda self.\lambda dx.(\mathtt{p} \Leftarrow \mathtt{mv_x}\; dx) \rangle. \end{aligned}$$

It is easy to verify that `p` and `q` return the same results for any sequence of message sends, but they are *not* equivalent if we extend them with additional methods. The reason is that the `p` object preserves method specialization of the $\mathtt{mv_x}$ method, whereas the `q` object not. The type system will derive for the above objects the following types:

$$\begin{aligned} Point &\stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int{\to}t \rangle\!\rangle \\ Q\_Point &\stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int{\to}Point \rangle\!\rangle, \end{aligned}$$

and this reflects the fact that the object-type gives also information about the types of methods when they are inherited by other classes.

### 6.2.4 Syntax of the Type System

The type expressions include type-constants, type-variables, function-types and object-types. In what follows, a *term* will be either an *object* of the calculus, a *type*, a *row*, or a *kind*.

Some notational convention is needed: the symbols $\sigma$, $\tau$, $\rho$, ... are metavariables over types; $t$, *self*,... range over type variables; $R$, $r$ range over rows and row variables respectively; $\mathtt{m}$, $\mathtt{n}$, ... range over method names and $\kappa$ ranges over kinds.

*Definition 6.2.1 The set of types, rows and kinds are mutually defined by the following grammar:*

$$\begin{array}{lll} Types\ \tau & ::= & t \mid \tau{\to}\tau \mid \mathtt{obj}\, t.R \\ Rows\ \ R & ::= & r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid \mathtt{m}{:}\tau \rangle\!\rangle \mid \lambda t.R \mid R\tau \\ Kinds\ \kappa & ::= & T \mid T^p{\to}[\mathtt{m}_1, \ldots, \mathtt{m}_k]\ (p \geq 0, k \geq 1). \end{array}$$

Rows occur as components of object-types, with rows and types distinguished by kinds. The order of methods inside rows is not important; we consider $\alpha$-conversion of type variables bound by $\lambda$ or `obj`, and application of the igma'(t)*rinciple*

$$\langle\!\langle \langle\!\langle R \mid \mathtt{n}{:}\tau_1 \rangle\!\rangle \mid \mathtt{m}{:}\tau_2 \rangle\!\rangle = \langle\!\langle \langle\!\langle R \mid \mathtt{m}{:}\tau_2 \rangle\!\rangle \mid \mathtt{n}{:}\tau_1 \rangle\!\rangle,$$

within types or row expressions to be conventions of syntax, rather than explicit rules of the system. Additional equations between types and rows arise as result of $\beta$-reduction, written $\to_\beta$, or $=_\beta$ . Intuitively, the elements of kind $[\mathtt{m}_1, \ldots, \mathtt{m}_k]$ are rows that do not include method names $\mathtt{m}_1, \ldots, \mathtt{m}_k$. We need this information to guarantee statically that methods are not multiply defined. We write $\overline{\mathtt{m}}{:}\overline{\tau}$ to abbreviate $\mathtt{m}_1{:}\tau_1, \ldots, \mathtt{m}_k{:}\tau_k$, for some $k$.

*Definition 6.2.2 Context and judgment are defined as follows:*

*i) Contexts are ordered lists* (not sets)*:*

$$\Gamma \ ::= \ \varepsilon \mid \Gamma, x{:}\tau \mid \Gamma, t{:}T \mid \Gamma, r{:}\kappa.$$

*ii) Judgments have the following form:*

$$\Gamma \vdash *, \ \ \textit{or} \ \ \Gamma \vdash R : \kappa, \ \ \textit{or} \ \ \Gamma \vdash \tau : T, \ \ \textit{or} \ \ \Gamma \vdash \mathtt{e} : \tau.$$

*The judgment $\Gamma \vdash *$ can be read as "$\Gamma$ is a well-formed context". The meaning of the other judgments is the usual one.*

### 6.2.5 Analysis of the Main Typing Rules

In this subsection, we discuss the rules that are crucial to understand this typing system, namely the rules that allows to build, extend and override an object and the rule for message send.

The empty object $\langle\rangle$ has the object-type $\mathtt{obj}\, t.\langle\!\langle\rangle\!\rangle$: this object cannot respond to any message, but can be extended with other methods. The typing rule is

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathtt{obj}\, t.\langle\!\langle\rangle\!\rangle} \quad (\textit{empty}{-}\textit{obj})$$

The rule to give a type to a message send is simple:

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau \rangle\!\rangle}{\Gamma \vdash \mathtt{e} \Leftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau \rangle\!\rangle / t]\tau} \quad (\textit{send})$$

This rule says that we can give a type to a message send provided that the receiver has the method we require in its object-type. Since the order of method in rows is irrelevant, we can write $\mathtt{n}$ as the last method listed in the object-type. The result of a message send will have a type in which every occurrence of the type variable $t$ has to be substituted by the full type of the object itself, thus reflecting the recursive nature of the object-type.

The most subtle and intriguing rules are the rule to extend and to override an existing object. We first introduce the $(obj{-}ext)$ rule.

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_1 : \mathtt{obj}\, t.\langle\!\langle R \mid \overline{\mathtt{m}}{:}\overline{\tau}\rangle\!\rangle \quad \Gamma, t{:}T \vdash R : [\overline{\mathtt{m}}, \mathtt{n}] \qquad r \text{ not in } \tau \\ \Gamma, r{:}T{\to}[\overline{\mathtt{m}}, \mathtt{n}] \vdash \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{m}}{:}\overline{\tau}, \mathtt{n}{:}\tau\rangle\!\rangle/t](t{\to}\tau)\end{array}}{\Gamma \vdash \langle \mathtt{e}_1 \leftarrow\!\!\!+\, \mathtt{n}{=}\mathtt{e}_2\rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \overline{\mathtt{m}}{:}\overline{\tau}, \mathtt{n}{:}\tau\rangle\!\rangle} \quad (obj{-}ext)$$

In this rule, we assume the type of the object $\mathtt{e}_1$ to be an object-type which does not contain the method we want to add, and we require also that the subrow $R$ in that object-type does not contain methods $\overline{\mathtt{m}}$ and $\mathtt{n}$, for some list of method $\overline{\mathtt{m}}$ in the type of $\mathtt{e}$.

The meaning of the explicitly listed methods $\overline{\mathtt{m}}$ in the type of $\mathtt{e}$ is crucial: they represent the methods which are *useful* to type $\mathtt{n}$'s body. The exact meaning of those methods will become clear when, in the next subsection, we will present an example of derivations. The final judgment is a typing for $\mathtt{e}_2$, i.e. the body of the method we want to add. Two observations are needed to understand this judgment:

**Higher-Order.** Note that the type of $\mathtt{e}_2$ contains the row variable $r$, which is implicitly quantified in the context. Because of this quantification, for every substitution of $r$ with any row $R'$, provided that $R'$ has the correct kind, $\mathtt{e}_2$ will have the indicated type in which $r$ will be substituted by $R'$. This means that for every possible future extension of $\langle \mathtt{e}_1 \leftarrow\!\!\!+\, \mathtt{n}{=}\mathtt{e}_2\rangle$, the method $\mathtt{n}$ will have the required functionality. This "mutability" of the type of a method is referred as *method specialization*.

**Self-Application.** The typing of $\mathtt{e}_2$ is an arrow-type of the shape $(t{\to}\tau)$ with $t$ substituted by an object-type. Since $t$ is hidden in the final typing of $\langle \mathtt{e}_1 \leftarrow\!\!\!+\, \mathtt{n}{=}\mathtt{e}_2\rangle$, it is necessary in the typing of $\mathtt{e}_2$, because the semantics of sending messages would give as result the application of the body of the message applied to the full object itself.

The rule to override a method $\mathtt{n}$ in the object $\mathtt{e}_1$ with a new body $\mathtt{e}_2$ is simpler:

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_1 : \mathtt{obj}\, t.\langle\!\langle R \mid \overline{\mathtt{m}}{:}\overline{\tau}, \mathtt{n}{:}\tau\rangle\!\rangle \qquad r \text{ not in } \tau \\ \Gamma, r{:}T{\to}[\overline{\mathtt{m}}, \mathtt{n}] \vdash \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{m}}{:}\overline{\tau}, \mathtt{n}{:}\tau\rangle\!\rangle/t](t{\to}\tau)\end{array}}{\Gamma \vdash \langle \mathtt{e}_1 \leftarrow \mathtt{n}{=}\mathtt{e}_2\rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \overline{\mathtt{m}}{:}\overline{\tau}, \mathtt{n}{:}\tau\rangle\!\rangle} \quad (obj{-}over)$$

The first premise of this rule requires that the method n we want to override must be in the object-type of $\texttt{e}_1$, and the new body we want to substitute has the same functionality as the older one.

### 6.2.6 Example of Typing Derivations

In this subsection, we will give some derivations of the previously showed examples of `funny`and colored points.

*Example 6.2.3* Recall that `funny` is the following object:

$$\texttt{funny} \stackrel{ext}{=} \langle \texttt{m} = \lambda \mathit{self}.\langle \mathit{self} \leftarrow \texttt{m} = \lambda s.s \Leftarrow \texttt{m}\rangle.$$

We give a derivation for the judgment $\varepsilon \vdash \texttt{funny} : \texttt{obj}\,t.\langle\!\langle \texttt{m}{:}t\rangle\!\rangle$. Take the context:

$$\begin{array}{rcl} \Gamma_1 & = & r_1{:}T{\to}[\texttt{m}], \mathit{self}{:}\texttt{obj}\,t.\langle\!\langle r_1t \mid \texttt{m}{:}t\rangle\!\rangle \\ \Gamma_2 & = & r_2{:}T{\to}[\texttt{m}], s{:}\texttt{obj}\,t.\langle\!\langle r_2t \mid \texttt{m}{:}t\rangle\!\rangle. \end{array}$$

then the following is a legal derivation:

$$
\dfrac{
\begin{array}{c}
\dfrac{
\begin{array}{c}
\dfrac{
\dfrac{\Gamma_1,\Gamma_2 \vdash s : \texttt{obj}\,t.\langle\!\langle r_2t \mid \texttt{m}{:}t\rangle\!\rangle}{\Gamma_1,\Gamma_2 \vdash s \Leftarrow \texttt{m} : \texttt{obj}\,t.\langle\!\langle r_2t \mid \texttt{m}{:}t\rangle\!\rangle}\;(\mathit{send})
}{\Gamma_1, r_2{:}T{\to}[\texttt{m}] \vdash \lambda s.s \Leftarrow \texttt{m} : [\texttt{obj}\,t.\langle\!\langle r_2t \mid \texttt{m}{:}t\rangle\!\rangle/t](t{\to}t)}\;(\mathit{exp{-}abs}) \\
\Gamma_1 \vdash \mathit{self} : \texttt{obj}\,t.\langle\!\langle r_1t \mid \texttt{m}{:}t\rangle\!\rangle \qquad \Gamma_1, t{:}T \vdash \langle\!\langle r_1t\rangle\!\rangle : [\texttt{m}] \qquad r_2 \ \mathit{not\ in}\ t
\end{array}
}{\Gamma_1 \vdash \langle \mathit{self} \leftarrow \texttt{m} = \lambda s.s \Leftarrow \texttt{m}\rangle : \texttt{obj}\,t.\langle\!\langle r_1t \mid \texttt{m}{:}t\rangle\!\rangle}\;(\mathit{obj{-}over})
\\ \hline
r_1{:}T{\to}[\texttt{m}] \vdash \lambda \mathit{self}.\langle \mathit{self} \leftarrow \texttt{m} = \lambda s.s \Leftarrow \texttt{m}\rangle : [\texttt{obj}\,t.\langle\!\langle r_1t \mid \texttt{m}{:}t\rangle\!\rangle/t](t{\to}t) \;\;(\mathit{exp{-}abs}) \\
\varepsilon \vdash \langle\rangle : \texttt{obj}\,t.\langle\!\langle\rangle\!\rangle \qquad t{:}T \vdash \langle\!\langle\rangle\!\rangle : [\texttt{m}] \qquad r_1 \ \mathit{not\ in}\ t
\end{array}
}{\varepsilon \vdash \texttt{funny} : \texttt{obj}\,t.\langle\!\langle \texttt{m}{:}t\rangle\!\rangle}\;(\mathit{obj{-}ext})
$$

*Example 6.2.4* To build a color point with move object that is obtained by adding to the object point the color component, i.e. from:

$$\texttt{p} \stackrel{def}{=} \langle\langle\langle\rangle \leftarrow\!\!+\, \texttt{x} = \lambda \mathit{self}.3\rangle \leftarrow\!\!+\, \texttt{mv}_\texttt{x} = \lambda s_1.\lambda dx.\langle s_1 \leftarrow \texttt{x} = \lambda s_2.(s_1 \Leftarrow x) + dx\rangle\rangle,$$

we build

$$\begin{aligned}\mathtt{cp} &\stackrel{def}{=} \langle \mathtt{p} \leftarrow\!\!+ \,\mathtt{col} = \lambda self.red\rangle \\ &\stackrel{ext}{=} \langle \mathtt{x} = \lambda self.3, \mathtt{mv_x} = \lambda s_1.\lambda dx.\langle s_1 \leftarrow \mathtt{x} = \lambda s_2.(s_1 \Leftarrow x) + dx\rangle, \mathtt{col} = \lambda self.red\rangle.\end{aligned}$$

We give a derivation for the judgment $\varepsilon \vdash \mathtt{cp} : \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t, \mathtt{col}{:}Col\rangle\!\rangle$. Take the context:

$$\begin{aligned}\Gamma_1 &= r_1{:}T \to [\mathtt{x}, \mathtt{mv_x}], s_1{:}\mathtt{obj}\, t.\langle\!\langle r_1 t \mid \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t\rangle\!\rangle, dx{:}int \\ \Gamma_2 &= r_2{:}T \to [\mathtt{x}], s_2{:}\mathtt{obj}\, t.\langle\!\langle r_2 t \mid \mathtt{x}{:}int\rangle\!\rangle,\end{aligned}$$

and let $body_{mv}$ be the subobject $\langle s_1 \leftarrow \mathtt{x} = \lambda s_2.(s_1 \Leftarrow x) + dx\rangle$, and take the following derivation (to short the notation we write $T_{[\overline{\mathtt{m}}]}$ for $T \to [\overline{\mathtt{m}}]$):

$$\dfrac{\dfrac{\dfrac{\Gamma_1, \Gamma_2 \vdash (s_1 \Leftarrow \mathtt{x}) + dx : int}{\Gamma_1, r_2{:}T_{[\mathtt{x}]} \vdash \lambda s_2.(s_1 \Leftarrow \mathtt{x}) + dx : [\mathtt{obj}\, t.\langle\!\langle r_2 t \mid \mathtt{x}{:}int\rangle\!\rangle/t](t \to int) \quad \Gamma_1 \vdash s_1 : \mathtt{obj}\, t.\langle\!\langle r_1 t \mid \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t\rangle\!\rangle \quad \Gamma_1, t{:}T \vdash \langle\!\langle r_1 t \mid \mathtt{mv_x}{:}int \to t\rangle\!\rangle : [\mathtt{x}]}\,(exp{-}abs)}{\dfrac{\dfrac{\Gamma_1 \vdash body_{mv} : \mathtt{obj}\, t.\langle\!\langle r_1 t \mid \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t\rangle\!\rangle}{\Gamma_1 - dx \vdash \lambda dx.body_{mv} : [\mathtt{obj}\, t.\langle\!\langle r_1 t \mid \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t\rangle\!\rangle/t](int \to t)}\,(exp{-}abs)}{r_1{:}T_{[\mathtt{x},\mathtt{mv_x}]} \vdash \lambda s_1.\lambda dx.body_{mv} : [\mathtt{obj}\, t.\langle\!\langle r_1 t \mid \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t\rangle\!\rangle/t](t \to int \to t) \quad \varepsilon \vdash \langle \mathtt{x} = \lambda self.3\rangle : \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int\rangle\!\rangle \quad t{:}T \vdash \langle\!\langle\rangle\!\rangle : [\mathtt{x}, \mathtt{mv_x}]}\,(exp{-}abs)}\,(obj{-}ext)}{\varepsilon \vdash \mathtt{p} : \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t\rangle\!\rangle \quad t{:}T \vdash \langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t\rangle\!\rangle : [\mathtt{col}] \quad r_3{:}T_{[\mathtt{col}]} \vdash \lambda self.red : [\mathtt{obj}\, t.\langle\!\langle r_3 t \mid \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t, \mathtt{col}{:}col\rangle\!\rangle/t](t \to col)}\,(obj{-}ext)$$

$$\dfrac{\varepsilon \vdash \mathtt{p} : \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t\rangle\!\rangle \quad t{:}T \vdash \langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t\rangle\!\rangle : [\mathtt{col}] \quad r_3{:}T_{[\mathtt{col}]} \vdash \lambda self.red : [\mathtt{obj}\, t.\langle\!\langle r_3 t \mid \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t, \mathtt{col}{:}col\rangle\!\rangle/t](t \to col)}{\varepsilon \vdash \langle \mathtt{p} \leftarrow\!\!+ \,\mathtt{col} = \lambda self.red\rangle : \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int \to t, \mathtt{col}{:}colors\rangle\!\rangle}\,(obj{-}ext)$$

where the conditions on the $r_i$ row variables, with $i = 1, 2, 3$ are omitted.

## 6.3 Subject Reduction and Type-Soundness

The properties of type system of the $\lambda_{obj}$ are proved in [FHM94] in several steps. The first step is the subject reduction theorem proving that types are preserved by the evaluation rules. Then the authors introduce an evaluation strategy, called *Eval* (not to confuse with $\stackrel{eval}{\rightarrow}$) which evaluates expressions, producing an output which is an ordinary value, the *error* value, or fails to terminate. The error value is introduce to catch the run-time error *message-not-understood.* Then they use the subject reduction property to show that if a closed expression $\mathtt{e}$ is well typed, the evaluation of $\mathtt{e}$ ends correctly, i.e. $eval(\mathtt{e}) \neq error$. This property has as nice corollary the type-soundness of the system. In the proofs, the authors focused on derivations in which not all the equality rules of the systems are used and, if so, they are used in a very restricted way. They call such judgments and derivations in *normal form*, denoted by $\vdash_N$ . We will formally introduce the normal form derivations in the next chapter. The main theorems proved are:

*Theorem 6.3.1* *i)* **(Subject Reduction).** *If* $\Gamma \vdash \mathtt{e} : \tau$, *and* $\mathtt{e} \stackrel{eval}{\rightarrow} \mathtt{e}'$, *then also* $\Gamma \vdash \mathtt{e}' : \tau$.

*ii)* **(Type-Soundness).** *If* $\varepsilon \vdash_N \mathtt{e} : \tau$, *then* $Eval(\mathtt{e}) \neq error$.

## 6.4 Expressive Power

The Lambda Calculus of Objects allows to represent all partial recursive functions. This is possible because the object-types are essentially recursive-types. We show how to encode the fixed-point operator.

**Divergent Computation** Consider the object:

$$\Omega \stackrel{def}{=} \langle \mathtt{m} = \lambda self.self \Leftarrow \mathtt{m} \rangle,$$

of type:

$$\mathtt{obj}\, t.\langle\!\langle \mathtt{m}{:}t \rangle\!\rangle.$$

If we send the message $\mathtt{m}$ to $\Omega$, then we have the following infinite computation:

$$\Omega \Leftarrow \mathtt{m} \stackrel{eval}{\rightarrow} (\lambda self.self \Leftarrow \mathtt{m})\Omega \stackrel{eval}{\rightarrow} \Omega \Leftarrow \mathtt{m} \stackrel{eval}{\rightarrow} \ldots.$$

**Fixed Points.** Consider the following object whose method `rec` cause the fixed function `f` to be applied to $\mathtt{X} \Leftarrow \mathtt{rec}$.

$$\mathtt{X} \stackrel{def}{=} \langle \mathtt{rec} = \lambda self.\mathtt{f}(self \Leftarrow \mathtt{rec})\rangle.$$

In fact, if we send a message rec to X, then we have the following infinite computation:

$$\begin{aligned}
\mathtt{X} \Leftarrow \mathtt{rec} &\stackrel{eval}{\to} (\lambda self.\mathtt{f}(self \Leftarrow \mathtt{rec}))\mathtt{X} \\
&\stackrel{eval}{\to} \mathtt{f}(\mathtt{X} \Leftarrow \mathtt{rec}) \\
&\stackrel{eval}{\to} \mathtt{f}((\lambda self.\mathtt{f}(self \Leftarrow \mathtt{rec}))\mathtt{X}) \\
&\stackrel{eval}{\to} \mathtt{f}(\mathtt{f}(\mathtt{X} \Leftarrow \mathtt{rec})) \\
&\stackrel{eval^*}{\to} \mathtt{f}(\cdots \mathtt{f}(\mathtt{X} \Leftarrow \mathtt{rec})\cdots) \\
&\stackrel{eval}{\to} \ldots
\end{aligned}$$

By abstracting over f, we obtain a functional that, for every function g of an arbitrary type $\sigma \to \tau$, gives the fixed point of g, i.e.

$$\mathtt{funct} \stackrel{def}{=} \lambda \mathtt{f}.\mathtt{X} \Leftarrow \mathtt{rec},$$

is such that:

$$\mathtt{funct\ g} \stackrel{eval}{\to} \langle \mathtt{rec} = \lambda self.\mathtt{g}(self \Leftarrow \mathtt{rec}\rangle \Leftarrow \mathtt{rec} \stackrel{eval^*}{\to} \mathtt{g}(\ldots \mathtt{g}((\mathtt{funct\ g}) \Leftarrow \mathtt{rec})\ldots),$$

we can show that:

$$\mathtt{funct} : (\sigma \to \tau) \to \tau.$$

**Object Numerals.** We do not repeat here the encoding of object numerals. The interested reader can look at [FHM94].

## 6.5 Conclusions

The Lambda Calculus of Objects is a computationally expressive typed calculus of functions and objects with a sound type system. The main feature of this type system is method specialization: with method extension and override, we can modify existing objects and build other objects whose behaviour and type change in a very natural way as result of inheritance. The operational semantics needs a special evaluation step in order to access to the method we want to answer. This seems to be a complication that can be dropped by simply giving another semantics which searches method bodies more directly and deals with possible errors. Moreover, this calculus lacks a subtyping relation, which seems to be an important feature for object-oriented programming. The investigation of an easier operational semantics, and the possibility of adding a subtyping relation, will be the topics of the next chapter.

# Chapter 7

# Adding Subtyping to the Calculus of Objects

## Introduction

In this chapter, we will study the possibility of adding some features to the Calculus of Objects. In particular, we will focus on the possibility of adding a subtyping relation. The intuitive definition of subtyping is: type $\sigma$ is a subtype of type $\tau$, usually denoted as $\sigma \preceq \tau$, if every object of type $\sigma$ can be used in any context that expects an object of type $\tau$. This is usually possible by adding to the type system a *subsumption* rule of the following shape:

$$\frac{\Gamma \vdash \mathtt{e} : \sigma \qquad \sigma \preceq \tau}{\Gamma \vdash \mathtt{e} : \tau} \; (subsumption)$$

As we showed in Chapter 5, there are essentially two forms of subtyping, namely *width*, and *depth* subtyping. Consider the type of an object as the collection of the types of its methods: then

**"Width".** The object-type $\mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}_k{:}\overline{\tau}_k \rangle\!\rangle$ is a *width* subtype of $\mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}_h{:}\overline{\tau}_h \rangle\!\rangle$ if $h \leq k$. For example:

$$\mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}_{\mathtt{x}}{:}int{\rightarrow}t, \mathtt{col}{:}colors \rangle\!\rangle \preceq \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}_{\mathtt{x}}{:}int{\rightarrow}t \rangle\!\rangle.$$

**"Depth".** The object-type $\texttt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}_k{:}\overline{\sigma}_k \rangle\!\rangle$ is a *depth* subtype of $\texttt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}_k{:}\overline{\tau}_k \rangle\!\rangle$ if, for all $1 \leq i \leq k$, we have $\sigma_i \preceq \tau_i$. For example:

$$\texttt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{y}{:}int \rangle\!\rangle \preceq \texttt{obj}\, t.\langle\!\langle \mathtt{x}{:}real, \mathtt{y}{:}real \rangle\!\rangle,$$

because $int \preceq real$, and

$$\texttt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}_{\mathtt{x}}{:}int {\to} t \rangle\!\rangle \not\preceq \texttt{obj}\, t.\langle\!\langle \mathtt{x}{:}real, \mathtt{mv}_{\mathtt{x}}{:}real {\to} t \rangle\!\rangle,$$

because $int \preceq real$, but $int {\to} \not\preceq real {\to} t$, assuming, as usual, that the arrow-type constructor is *contra-variant* in its left-hand side.

In [FM94], the authors observed that in pure delegation based languages, no subtyping is possible; let us repeat their arguments.

**"Width"** Since method addition is a legal operation on objects, objects with extra methods cannot be inserted in any context which expect an object with fewer methods. For example a colored point cannot be inserted in the following context:

$$C[\;] \stackrel{def}{=} \langle [\;] \leftarrow\!\!+ \, \mathtt{col} = \lambda self.red \rangle,$$

otherwise we would add twice the color component.

**"Depth"** [AC94] Consider the object-types

$$\sigma \stackrel{def}{=} \texttt{obj}\, t.\langle\!\langle \mathtt{x}{:}posint, \mathtt{y}{:}real \rangle\!\rangle,$$

and

$$\tau \stackrel{def}{=} \texttt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{y}{:}real \rangle\!\rangle.$$

Then, $\sigma \preceq \tau$, since $posint \preceq int$. If consider the object

$$\mathtt{e} \stackrel{def}{=} \langle \mathtt{x} = \lambda self.1, \mathtt{y} = \lambda self.\mathtt{ln}(self \Leftarrow \mathtt{x}) \rangle,$$

then we can assign to $\mathtt{e}$ the type $\sigma$, and, by subsumption, also the type $\tau$. Now we can override $\mathtt{e}$ as in $\langle \mathtt{e} \leftarrow \mathtt{x} = \lambda self. -1\rangle$, but if we send to this object a message $\mathtt{y}$, then we have the following wrong computation:

$$\begin{array}{rcl}
\langle \mathtt{e} \leftarrow \mathtt{x} = \lambda self. -1\rangle \Leftarrow \mathtt{y} & \stackrel{eval}{\rightarrow} & (\lambda self.\mathtt{ln}(self \Leftarrow \mathtt{x}))\langle \mathtt{e} \leftarrow \mathtt{x} = \lambda self. -1\rangle \\
& \stackrel{eval}{\rightarrow} & \mathtt{ln}(\langle \mathtt{e} \leftarrow \mathtt{x} = \lambda self. -1\rangle \Leftarrow \mathtt{x})) \\
& \stackrel{eval}{\rightarrow} & \mathtt{ln}((\lambda self. -1)\langle \mathtt{e} \leftarrow \mathtt{x} = \lambda self. -1\rangle) \\
& \stackrel{eval}{\rightarrow} & \mathtt{ln}(-1) \\
& \stackrel{eval}{\rightarrow} & type\!-\!mismatch\!-\!error.
\end{array}$$

In this chapter, we study subtyping in the Calculus of Objects. In particular we will find a suitable restriction of the "width" subtyping relation that can be integrated into $\lambda_{obj}$. This relation represents a trade-off between the possibility of having a restricted form of "width" subtyping and the features of the delegation-based language itself. "Width" subtyping is constrained by one restriction: $\sigma$ is a subtype of another type $\tau$ if and only if we can assure that the methods of $\sigma$, that are not methods of $\tau$, are not referred to by the methods which remain in $\tau$. This restriction is crucial to avoid that methods of $\tau$ will refer to the *forgotten* methods of $\sigma$, causing a run-time error. This subtyping relation allows to *forget* methods in the type without changing the shape of the object; it follows that we can type programs that accept as actual parameters objects with more methods than one could expect. A first consequence of this relation is that it can be possible to have an object in which a method is, via a new operation, added more than once. For this reason, we introduce a different symbol to indicate the method addition operation on objects, namely

$$\langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_2\rangle.$$

The operation $\leftarrow\!\circ$ behaves exactly as the method addition of $\lambda_{obj}$, but it can be used to add the same method more than once. For example, in the object

$$\langle\langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{m}{=}\mathtt{e}_2\rangle \leftarrow\!\circ\, \mathtt{m} = \mathtt{e}_3\rangle,$$

the first addition of the method $\mathtt{m}$ is forgotten by the type inference system via a subsumption rule.

This extension gives the following (positive) consequences:

*i)* objects with extra methods can be used in any context where an object with fewer methods might be used,

*ii)* we do not loose any feature of $\lambda_{obj}$ .

We also extend the set of objects and we present an alternative operational semantics. The evaluation rules search method bodies more directly and deal with possible errors. The resulting calculus is an extension of the original one, allowing both specialization of the type of an inherited method to the type of the inheriting object and static detection of errors, such as *'message-not-understood'*.

This chapter is organized as follows. In Section *7.1*, we present the language and the evaluation strategy. in Section *7.2*, we give the type inference rules for the calculus and the subtyping relation. Some interesting examples, showing the power of this calculus with respect to the original one, are also illustrated. In Section *7.4*, we prove some structural properties of the system. In Section *7.5*, we give a subject reduction theorem: moreover, we prove the soundness of the type system, namely, we show that it prevents *message-not-understood* errors. The last section is devoted to related papers and future work.

## 7.1 The $\lambda_{obj}^{\preceq}$ : Syntax and Semantics

In this section, we will present the Lambda Calculus of Objects as revisited in the paper [BL95]: we call this language $\lambda_{obj}^{\preceq}$ . The main differences between $\lambda_{obj}$ and $\lambda_{obj}^{\preceq}$ are that we extend the set of objects, and we present an alternative operational semantics that searches method bodies more directly, and deals with possible errors. This semantics was inspired by [Bel94], where its calculus is proved to be Church-Rosser. Thanks to this semantics, we can add to the type system a restricted form of "width" subtyping, preserving the full power of the delegation style of $\lambda_{obj}$ . We show a type inference system that allows both specialization of the type of an inherited method to the type of the inheriting object and static detection of errors, such as *'message-not-understood'*. This extension was made possible by adding a restricted form of "dependent-types", called *Labeled-types*, which collect the information about the methods which are used to type a method body. It follows that the resulting calculus is an extension of the original one. The type soundness follows, easily, from the subject reduction property.

### 7.1.1 Syntax of $\lambda^{\preceq}_{obj}$

The $\lambda^{\preceq}_{obj}$-calculus is defined as follows:

$$\mathtt{e} ::= x \mid \mathtt{c} \mid \lambda x.\mathtt{e} \mid \mathtt{e}_1\mathtt{e}_2 \mid \langle\rangle \mid \langle \mathtt{e}_1 \leftarrow\!\!\circ\, \mathtt{m}{=}\mathtt{e}_2\rangle \mid \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2\rangle \mid \mathtt{e} \Leftarrow \mathtt{m} \mid \mathtt{e} \hookleftarrow \mathtt{m} \mid \mathtt{err},$$

where $x$ is a term-variable, $\mathtt{c}$ belongs to a fixed set of constants, and $\mathtt{m}$ is a method name. The object forms are:

| | |
|---|---|
| $\langle\rangle$ | the empty object; |
| $\langle \mathtt{e}_1 \leftarrow\!\!\circ\, \mathtt{m}{=}\mathtt{e}_2\rangle$ | extends object $\mathtt{e}_1$ with a method $\mathtt{m}$ having body $\mathtt{e}_2$; |
| $\langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2\rangle$ | replaces the body of method $\mathtt{m}$ in $\mathtt{e}_1$ by $\mathtt{e}_2$; |
| $\mathtt{e} \Leftarrow \mathtt{m}$ | sends message $\mathtt{m}$ to the object $\mathtt{e}$; |
| $\mathtt{e} \hookleftarrow \mathtt{m}$ | searches the body of the message $\mathtt{m}$ in the object $\mathtt{e}$; |
| $\mathtt{err}$ | the error object. |

Notice that the last two object forms are not present in $\lambda_{obj}$ , and the $\leftarrow\!\!+$ operator was replaced by the new object operator $\leftarrow\!\!\circ$ , which has a different semantics.

Let $\leftarrow\!\!*$ denote either $\leftarrow\!\!+$ or $\leftarrow$. The description of an object via $\leftarrow\!\!*$ operators is intensional, and the object corresponding to a sequence of $\leftarrow\!\!*$ can be extensionally defined as in Definition *6.1.1*.

*Definition 7.1.1 Let* $\mathtt{m}_1, \ldots, \mathtt{m}_k$, *and* $\mathtt{n}$ *be distinct method names.* $\langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k\rangle$ *is defined as:*

$$\langle \ldots \langle \langle\rangle \leftarrow\!\!\circ\, \mathtt{m}_1 = \mathtt{e}_1\rangle \ldots \leftarrow\!\!\circ\, \mathtt{m}_k = \mathtt{e}_k\rangle \stackrel{\text{ext}}{=} \langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k\rangle$$
$$\langle \langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_i = \mathtt{e}_i, \ldots, \mathtt{m}_k = \mathtt{e}_k\rangle \leftarrow\!\!*\, \mathtt{m}_i = \mathtt{e}'_i\rangle \stackrel{\text{ext}}{=} \langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_i = \mathtt{e}'_i, \ldots, \mathtt{m}_k = \mathtt{e}_k\rangle$$
$$\langle \langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k\rangle \leftarrow\!\!\circ\, \mathtt{n} = \mathtt{e}'\rangle \stackrel{\text{ext}}{=} \langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k, \mathtt{n} = \mathtt{e}'\rangle.$$

So, $\langle \mathtt{m}_1 = \mathtt{e}_1, \ldots, \mathtt{m}_k = \mathtt{e}_k\rangle$ represents, extensionally, the behaviour of an object in which each method body corresponds to the outermost assignment (by extension or by override) performed on the method.

### 7.1.2 The Operational Semantics of $\lambda^{\preceq}_{obj}$

The operational semantics of $\lambda_{obj}$ is mainly based on the following evaluation rules for $\beta$-reduction and for message sending $\Leftarrow$-reduction:

$$
\begin{array}{lll}
(\beta) & (\lambda x.\mathtt{e}_1)\mathtt{e}_2 & \stackrel{eval}{\to} [\mathtt{e}_2/x]\mathtt{e}_1 \\
(\Leftarrow) & \langle \mathtt{e}_1 \leftarrow\!\!* \ \mathtt{m} = \mathtt{e}_2 \rangle \Leftarrow \mathtt{m} & \stackrel{eval}{\to} \mathtt{e}_2\langle \mathtt{e}_1 \leftarrow\!\!* \ \mathtt{m} = \mathtt{e}_2 \rangle.
\end{array}
$$

To send message $\mathtt{m}$ to the object $\mathtt{e}$ means to apply the body of $\mathtt{m}$ to the object itself. In fact, the body of $\mathtt{m}$ is a lambda abstraction whose first bound variable will be substituted by the full object in the next step of $\beta$-reduction.

The problem that arises in $\lambda^{\preceq}_{obj}$ is how to extract the appropriate method out of an object. The most natural way is moving the required method in an accessible position (the most external one). This means to treat objects as sets of methods. Unfortunately, this approach is not possible, as it was showed in Subsection *6.1.3*.

The approach chosen in $\lambda_{obj}$ to solve the problem of method order was to add to the $\stackrel{eval}{\to}$ relation, a *bookeeping* relation, denoted by $\stackrel{book}{\to}$. This relation leads to a *standard form*, in which each method is defined *exactly* once, using some "dummy" bodies, and redefined *exactly* once, giving it the desired body.

In $\lambda^{\preceq}_{obj}$ , the notion of standard form is unuseful, since the *subject reduction* property does not hold for the $\stackrel{book}{\to}$ part of the evaluation rules in presence of subtyping. On the other hand, we can use the extra information contained in types, to type correctly the extraction of the bodies of methods from the objects. Therefore, we propose as operational semantics the least congruence generated by the following rules.

$$
\begin{array}{llll}
(\beta) & (\lambda x.\mathtt{e}_1)\mathtt{e}_2 & \stackrel{eval}{\to} & [\mathtt{e}_2/x]\mathtt{e}_1 \\
(\Leftarrow) & \mathtt{e} \Leftarrow \mathtt{m} & \stackrel{eval}{\to} & (\mathtt{e} \hookleftarrow \mathtt{m})\mathtt{e} \\
(succ \hookleftarrow) & \langle \mathtt{e}_1 \leftarrow\!\!* \ \mathtt{n} = \mathtt{e}_2 \rangle \hookleftarrow \mathtt{m} & \stackrel{eval}{\to} & \mathtt{e}_2 \\
(next \hookleftarrow) & \langle \mathtt{e}_1 \leftarrow\!\!* \ \mathtt{n} = \mathtt{e}_2 \rangle \hookleftarrow \mathtt{m} & \stackrel{eval}{\to} & \mathtt{e}_1 \hookleftarrow \mathtt{m} \\
(fail\ \langle\rangle) & \langle\rangle \hookleftarrow \mathtt{m} & \stackrel{eval}{\to} & \mathtt{err} \\
(fail\ abs) & \lambda x.\mathtt{e} \hookleftarrow \mathtt{m} & \stackrel{eval}{\to} & \mathtt{err} \\
(err \leftarrow\!\!*\ ) & \langle \mathtt{err} \leftarrow\!\!* \ \mathtt{m} = \mathtt{e} \rangle & \stackrel{eval}{\to} & \mathtt{err} \\
(err\ abs) & \lambda x.\mathtt{err} & \stackrel{eval}{\to} & \mathtt{err} \\
(err\ appl) & \mathtt{err}\ \mathtt{e} & \stackrel{eval}{\to} & \mathtt{err} \\
(err \hookleftarrow) & \mathtt{err} \hookleftarrow \mathtt{n} & \stackrel{eval}{\to} & \mathtt{err}.
\end{array}
$$

To send message m to the object e still means applying the body of m to the object itself. The difference is that, in our semantics, the body of the method is recursively searched by the $\hookleftarrow$ operator without modifying the shape of the full object; if such a method does not exist, the object evaluates to error.

*Remark 7.1.2* *i)* The rule

$$(\mathit{fail\ var})\ x \hookleftarrow \mathtt{m} \stackrel{eval}{\rightarrow} \mathtt{err},$$

is unsound, since the variable $x$ could be substituted (by applying a $\beta$-reduction) by an object containing the method m.

*ii)* The rule

$$(\mathit{err} \Leftarrow)\ \mathtt{err} \Leftarrow \mathtt{n} \stackrel{eval}{\rightarrow} \mathtt{err},$$

is derived by the following evaluation chain:

$$\mathtt{err} \Leftarrow \mathtt{n} \stackrel{eval}{\rightarrow} (\mathtt{err} \hookleftarrow \mathtt{n})\mathtt{err} \stackrel{eval}{\rightarrow} (\mathtt{err})\mathtt{err} \stackrel{eval}{\rightarrow} \mathtt{err},$$

using $(\Leftarrow)$, $(\mathit{err} \hookleftarrow)$, and $(\mathit{err\ appl})$.

With help of an easy example, we show an object typable in $\lambda^{\preceq}_{obj}$, but not in $\lambda_{obj}$.

*Example 7.1.3* Take the context:

$$C[\ ] \stackrel{def}{=} \langle [\ ] \hookleftarrow\!\circ\ \mathtt{col} = \lambda \mathit{self}.\mathit{red}\rangle,$$

and the object colored point

$$\begin{aligned} \mathtt{cp} \stackrel{def}{=} \langle \mathtt{x} &= \lambda \mathit{self}.3, \\ \mathtt{mv_x} &= \lambda \mathit{self}.\lambda dx.\langle \mathit{self} \leftarrow \mathtt{x} = \lambda s.(\mathit{self} \Leftarrow x) + dx\rangle, \\ \mathtt{col} &= \lambda \mathit{self}.\mathit{black} \\ &\rangle. \end{aligned}$$

Then, the object:

$$\mathtt{cp}' \stackrel{def}{=} C[\mathtt{cp}] \stackrel{def}{=} \langle \mathtt{cp} \hookleftarrow\!\circ\ \mathtt{col} = \lambda \mathit{self}.\mathit{red}\rangle,$$

is an object belonging to the new syntax that is typable in the $\lambda^{\preceq}_{obj}$ type system (by one application of the subsumption rule) with the type:

$$\mathtt{cp}' : \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv_x}{:}int{\to}t, \mathtt{col}{:}colors \rangle\!\rangle.$$

If we take the syntax of $\lambda_{obj}$ , then we read $\leftarrow\!\circ$ as $\leftarrow\!+$ , and this would have as effect the failure of the typing of $\mathtt{cp}'$ because the color component was added twice, thus violating the condition of the addition operator.

## 7.2 Static Type System of $\lambda_{obj}^{\preceq}$

As we said in the introduction of this chapter, to allow subtyping we add a new sort of types, the labeled-types, that carry on the information about the methods used to type a certain method body. This information is given by a subscript which is a set of method names. The methods used to type a body are roughly the methods which occur in sending or overriding position with the *self* variable. For example, suppose that the object $\mathtt{e}_1$ has a method $\mathtt{m}$ with body $\mathtt{e}_2$, that in $\mathtt{e}_2$ a message $\mathtt{p}$ is sent to the bound variable *self* and a method $\mathtt{q}$ (of $\mathtt{e}_1$) is overridden. Then the type $\tau$ of $\mathtt{e}_2$ is subscripted by the set $\{\mathtt{p}, \mathtt{q}\}$, since $\mathtt{e}_2$ uses $\mathtt{p}$ and $\mathtt{q}$. These labeled-types are written inside the row of the object-type and do not appear externally. Therefore, in this type system the object point will have, as one could expect, the object-type

$$\mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{y}{:}, \mathtt{mv}{:}(int{\to}int{\to}t)_{\mathtt{x},\mathtt{y}} \rangle\!\rangle,$$

where $int$ stands for $int_{\emptyset}$. Moreover, we must insert, in the label of a type, the methods on which the "future" body (i.e. the body which will override the present one) will depend on. We can forget, by subtyping, those methods that are not used by other methods in the object, i.e. a method is *forgettable* if and only if it does not appear in the labels of the types of the remaining methods. This dependency is correctly handled in the typing rules for adding and overriding methods $(obj{-}ext)$ and $(obj{-}over)$, where the labels of types are created. In Section *7.2.4*, some meaningful examples will be presented.

### 7.2.1 Types, Rows, and Kinds in $\lambda_{obj}^{\preceq}$

As in the $\lambda_{obj}$ , the type expressions include type-constants, type-variables, arrow-types, and object-types. We introduce some notational conventions.

**Notational Convention**

In what follows, a *term* will be either an *expression* of the calculus, a *type*, a *row*, or a *kind*. The symbols $\sigma$, $\tau$, $\rho$, ... are metavariables over types; $\iota$ ranges over type-constants; $t$, *self*,... range over type-variables; $\Delta$ ranges over labels; $\alpha$, $\beta$, ... range over labeled-types; $R$, $r$ range over rows and row-variables respectively, and $\kappa$ ranges over kinds. $x$, $y$, ... range over expression-variables; $\mathtt{m}$, $\mathtt{n}$, $\mathtt{p}$, $\mathtt{q}$, ... range over method names. The symbols $a$, $b$, $c$, ... range over term-variables or constants; $u$, $v$, ... range over type- and row-variables; $U$, $V$, ... range over type, row, and kind expressions, and, finally, $A$, $B$, $C$, ... range over terms. All symbols may appear indexed. We also use the notation $\overline{\mathtt{m}}{:}\overline{\alpha}$, as short for $\mathtt{m}_1{:}\alpha_1, \ldots, \mathtt{m}_k{:}\alpha_k$, for some $k$, and $\overline{\alpha}$, as short for $\alpha_1, \ldots, \alpha_k$, for some $k$. We adopt the standard convention that bound variables have different names, if there is danger of confusion.

*Definition 7.2.1 The set of types, rows and kinds are mutually defined by the following grammar:*

$$
\begin{array}{llcl}
\textit{Types} & \tau & ::= & \iota \mid t \mid \tau \to \tau \mid \mathtt{obj}\, t.R \\
\textit{Labels} & \Delta & ::= & \emptyset \mid \Delta \cup \{\mathtt{m}\} \\
\textit{Labeled-Types} & \alpha & ::= & \tau_\Delta \\
\textit{Rows} & R & ::= & r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid \mathtt{m}{:}\alpha \rangle\!\rangle \mid \lambda t.R \mid R\tau \\
\textit{Kinds} & \kappa & ::= & T \mid T^p \to [\mathtt{m}_1, \ldots, \mathtt{m}_k]\ (p \geq 0, k \geq 1).
\end{array}
$$

*Definition 7.2.2 Signatures, contexts and judgments are defined as follows:*

*i) Signatures are* sets*:*

$$\Sigma \ ::= \ \emptyset \mid \Sigma \cup \iota{:}T \mid \Sigma \cup c{:}\iota \mid \Sigma \cup \iota_1 \preceq \iota_2,$$

*ii) Contexts are ordered lists* (not sets)*:*

$$\Gamma \ ::= \ \varepsilon \mid \Gamma, x{:}\tau \mid \Gamma, t{:}T \mid \Gamma, r{:}\kappa,$$

*and the notation $\Gamma \sqsubseteq \Gamma'$ means that $\Gamma$ is a prefix of $\Gamma'$ (see Definition 2.1.7).*

*iii) Judgments are of the following form:*

$$\Gamma \vdash_\Sigma *, \quad or \quad \Gamma \vdash_\Sigma R : \kappa, \quad or \quad \Gamma \vdash_\Sigma \tau : T, \quad or \quad \Gamma \vdash_\Sigma \tau_1 \preceq \tau_2, \quad or \quad \Gamma \vdash_\Sigma \mathtt{e} : \tau,$$

*and the notation* $A \bullet B$ *stands for* $A : B$*, or* $A \preceq B$*.*
*The judgment* $\Gamma \vdash_\Sigma *$ *can be read as "*$\Gamma$ *is a well-formed context in a well formed signature* $\Sigma$*". The meaning of the other judgments is the usual one. We will omit the subscript* $\Sigma$ *in the turnstyle* $\vdash_\Sigma$ *.*

*Remark 7.2.3* Note that in $\lambda_{obj}$, constants, signatures, and subtyping statements were absent.

The following definition agrees with the view of rows as sets of typed methods.

*Definition 7.2.4 Let* $\mathtt{m}{:}\alpha \in R$ *if and only if* $R \equiv \langle\!\langle R' \mid \mathtt{m}{:}\alpha \rangle\!\rangle$ *for some* $R'$*, and define the restriction of a row* $R$ *according to a label* $\Delta$*, denoted by* $R_\Delta$*, as follows:*

$$R_\Delta \stackrel{\text{def}}{=} \{\mathtt{m}{:}\alpha \in R \mid \mathtt{m} \in \Delta\}.$$

The following definition associates to a set of labeled types the labels which is the union of all labels.

*Definition 7.2.5 Define the set* $\mathcal{L}(\overline{\alpha})$ *of methods which occur in the labels of* $\overline{\alpha}$ *as follows:*

$$\begin{aligned} \mathcal{L}(\epsilon) &= \emptyset \\ \mathcal{L}(\overline{\beta}, \overline{\tau}_\Delta) &= \mathcal{L}(\overline{\beta}) \cup \Delta. \end{aligned}$$

### 7.2.2 The Typing Rules

In this subsection, we discuss all the typing rules which are new with respect to $\lambda_{obj}$, except for the subsumption rule which will be discussed in the next subsection. More precisely, we present the rules for extending an object with a new method or for redefining an existing one with a new body, the rule for searching method bodies, and the rule for sending messages. The remaining rules of the type system are presented in Figure *7.2.2*, and *7.2.3*. Let us examine the main rules in details.

Look at the shape of rule $(obj{-}ext)$ in Figure *7.2.2*. This rule performs a method addition, producing the new object $\langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n}{=}\mathtt{e}_2 \rangle$. It always adds the method to the syntactic object in case the method is not present or it is present in the object but it was previously forgotten in the type by an application of the subtyping rule $(sub_\preceq)$.

**Rules Expressions**

$$\frac{\Gamma \vdash \tau : T \quad x \notin Dom(\Gamma)}{\Gamma, x{:}\tau \vdash *}\ (exp{-}var) \qquad \frac{\Gamma, x{:}\tau_1 \vdash \mathtt{e} : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\mathtt{e} : \tau_1 \to \tau_2}\ (exp{-}abs)$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \sigma \to \tau \quad \Gamma \vdash \mathtt{e}_2 : \sigma}{\Gamma \vdash \mathtt{e}_1\mathtt{e}_2 : \tau}\ (exp{-}appl) \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle}{\Gamma \vdash \mathtt{e} \Leftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle / t]\tau}\ (send)$$

$$\frac{\begin{array}{ll} \Gamma \vdash \mathtt{e}_1 : \mathtt{obj}\, t.R & \Gamma, t{:}T \vdash R : [\mathtt{n}] \\ \{\overline{\mathtt{m}}{:}\overline{\alpha}\} \subseteq R_\Delta & r \textit{ not in } \tau \\ \multicolumn{2}{l}{\Gamma, r{:}T \to [\overline{\mathtt{m}}, \mathtt{n}] \vdash \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta \rangle\!\rangle / t](t \to \tau)} \end{array}}{\Gamma \vdash \langle \mathtt{e}_1 \leftarrow\!\!\circ\ \mathtt{n}{=}\mathtt{e}_2 \rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle}\ (obj{-}ext)$$

$$\frac{\begin{array}{ll} \Gamma \vdash \mathtt{e}_1 : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle & \{\overline{\mathtt{m}}{:}\overline{\alpha}\} \subseteq R_\Delta \\ \multicolumn{2}{l}{\Gamma, r{:}T \to [\overline{\mathtt{m}}, \mathtt{n}] \vdash \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta \rangle\!\rangle / t](t \to \tau)} \end{array}}{\Gamma \vdash \langle \mathtt{e}_1 \leftarrow \mathtt{n}{=}\mathtt{e}_2 \rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle}\ (obj{-}over)$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle \quad \Gamma, t{:}T \vdash R' : [\mathtt{n}] \quad R_\Delta \subseteq R'_\Delta}{\Gamma \vdash \mathtt{e} \hookleftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R' \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle / t](t \to \tau)}\ (search)$$

$$\frac{\Gamma \vdash \mathtt{e} : \sigma \quad \Gamma \vdash \sigma \preceq \tau}{\Gamma \vdash \mathtt{e} : \tau}\ (sub_{\preceq})$$

Figure 7.1: Type Assignment Rules for Expressions

The condition $\{\overline{\mathtt{m}}{:}\overline{\alpha}\} \subseteq R_\Delta$ assures that the $\overline{\mathtt{m}}$ methods, which are needed to type $\mathtt{e}_2$, are present in $\mathtt{e}_1$. But $\Delta$ can contain also methods which will be useful to type future bodies of $\mathtt{n}$.

In the $(obj{-}over)$ rule we require that the new method body must have the same type and label of the old one.

In the $(obj{-}ext)$ and the $(obj{-}over)$ rules the method $\mathtt{n}$ *uses* some of the methods belonging to the label $\Delta$ associated with the labeled-type of $\mathtt{n}$.

The $(search)$ rule asserts that the type of the extracted method body is an instance of the type we deduced for it when the method was added (by an $(obj{-}ext)$ rule) or overridden (by an $(obj{-}over)$ rule). Note also that labels are essential in this rule, to prevent an unsound choice of $R'$.

The $(send)$ rule is the same as in [FHM94]: it can be considered as an unfolding rule for recursive-types.

*Remark 7.2.6* Note that the following rule is admissible:

$$\frac{\Gamma \vdash (\mathtt{e} \hookleftarrow \mathtt{n})\mathtt{e} : \tau}{\Gamma \vdash \mathtt{e} \Leftarrow \mathtt{n} : \tau}\ (send)$$

In fact, a deduction for $(\mathtt{e} \hookleftarrow \mathtt{n})\ \mathtt{e}$ must have the shape:

$$\frac{\dfrac{\Gamma \vdash \mathtt{e} \hookleftarrow \mathtt{n} : \sigma \to \rho \qquad \Gamma \vdash \mathtt{e} : \sigma}{\Gamma \vdash (\mathtt{e} \hookleftarrow \mathtt{n})\ \mathtt{e} : \rho}\ (exp\ appl)}{\Gamma \vdash (\mathtt{e} \hookleftarrow \mathtt{n})\ \mathtt{e} : \tau}\ (sub_{\preceq})$$

where $\sigma \equiv \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\mu_\Delta \rangle\!\rangle$, and $[\sigma/t]\mu \preceq \rho$. Therefore we have:

$$\frac{\dfrac{\Gamma \vdash \mathtt{e} : \sigma}{\Gamma \vdash \mathtt{e} \Leftarrow \mathtt{n} : [\sigma/t]\mu}\ (send)}{\Gamma \vdash \mathtt{e} \Leftarrow \mathtt{n} : \tau}\ (sub_{\preceq})$$

### 7.2.3 The Subtyping Relation and the Subsumption Rule

The subtyping relation is based on the information given by the labeled-types of methods in rows. Looking at rules $(obj{-}ext)$ and $(obj{-}over)$, it is clear that if the body of the method $\mathtt{n}$ in $\mathtt{e}$ has type $\tau$, and its typing "uses" the methods $\overline{\mathtt{m}}$ of $\mathtt{e}$, then the type of $\mathtt{e}$ will be $\mathtt{obj}\,t.\langle\!\langle R \mid \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta\rangle\!\rangle$, with $\{\overline{\mathtt{m}}\} \subseteq \Delta$, for suitable $R$ and $\overline{\alpha}$. In other words, we label the types of the methods in rows by the sets of methods the typing of their bodies depends on (or will depend on). We discuss the most important rule, namely the rule which forgets methods in object-types and the subsumption rule ($sub_{\preceq}$). The set of all subtyping rules is showed in Figure *7.2*.

The ($width_{\preceq}$) rule says that a type is a subtype of another type if the forgotten methods (i.e. the methods not occurring) in the second type are not in the union of the sets of labels of the remaining methods. The condition $\overline{\mathtt{n}} \notin \mathcal{L}(\overline{\alpha})$ formally assures that the remaining methods do not use the methods $\overline{\mathtt{n}}$. Clearly, we can forget groups of mutually recursive methods with this rule. Observe that subtyping is formally forbidden on "open" object-types of the shape, $\mathtt{obj}\,t.\langle\!\langle \ldots r \ldots\rangle\!\rangle$. If we would allow subtyping on object-types of that shape, then we would loose the subject reduction property. In fact, our subtyping relation would no longer be closed under substitution of row-variables in object-types, as it is showed in the following example:

*Example 7.2.7* Assume to allow the following subtyping between object-types:

$$r{:}T{\to}[\mathtt{m},\mathtt{n}] \vdash \mathtt{obj}\,t.\langle\!\langle rt \mid \mathtt{m}{:}int_{\emptyset}, \mathtt{n}{:}int_{\mathtt{m}}\rangle\!\rangle \preceq \mathtt{obj}\,t.\langle\!\langle rt \mid \mathtt{m}{:}int_{\emptyset}\rangle\!\rangle,$$

because method $\mathtt{n}$ is not used by $\mathtt{m}$, and consider the following derivable judgment:

$$\varepsilon \vdash \langle\!\langle \mathtt{p}{:}int_{\mathtt{n}}\rangle\!\rangle.$$

If we suppose that the subtyping relation is closed under substitution, then we could derive:

$$\varepsilon \vdash \mathtt{obj}\,t.\langle\!\langle \mathtt{p}{:}int_{\mathtt{n}}, \mathtt{m}{:}int_{\emptyset}, \mathtt{n}{:}int_{\mathtt{m}}\rangle\!\rangle \preceq \mathtt{obj}\,t.\langle\!\langle \mathtt{p}{:}int_{\mathtt{n}}, \mathtt{m}{:}int_{\emptyset}\rangle\!\rangle,$$

which is obviouly unsound, because method $\mathtt{n}$ is "unforgettable", since method $\mathtt{p}$ uses it.

We have also the usual subtyping rules for constants, reflexivity, transitivity and for the arrow-type constructor (that behaves contra-variantly with respect to the $\preceq$ relation).

$$\frac{\Gamma \vdash \sigma : T}{\Gamma \vdash \sigma \preceq \sigma}\ (refl_{\preceq}) \qquad \frac{\Gamma \vdash \sigma \preceq \tau \quad \Gamma \vdash \tau \preceq \rho}{\Gamma \vdash \sigma \preceq \rho}\ (trans_{\preceq})$$

$$\frac{\Gamma \vdash \sigma' \preceq \sigma \quad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash \sigma \to \tau \preceq \sigma' \to \tau'}\ (arrow_{\preceq}) \qquad \frac{\Gamma \vdash_N \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \overline{\mathtt{n}}{:}\overline{\beta} \rangle\!\rangle : T \qquad \overline{\mathtt{n}} \notin \mathcal{L}(\overline{\alpha})}{\Gamma \vdash_N \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \overline{\mathtt{n}}{:}\overline{\beta} \rangle\!\rangle \preceq \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha} \rangle\!\rangle}\ (width_{\preceq})$$

Figure 7.2: The Subtyping Rules

Let two types $\sigma$ and $\tau$ be given, such that the judgment $\Gamma \vdash \sigma \preceq \tau$ is derivable and the expression $\mathtt{e}$ is of type $\sigma$. The $(sub_{\preceq})$ rule says that we can derive also type $\tau$ for $\mathtt{e}$. It follows that the expression $\mathtt{e}$ can be used in any context in which an expression of type $\tau$ is required. The possibility of giving more types to the same expression makes the $\lambda_{obj}^{\preceq}$ calculus more expressive than the original $\lambda_{obj}$ .

$$\frac{\Gamma \vdash \mathtt{e} : \sigma \qquad \Gamma \vdash \sigma \preceq \tau}{\Gamma \vdash \mathtt{e} : \tau}\ (sub_{\preceq})$$

Using the $(sub_{\preceq})$ rule we can obtain judgments of the shape $\Gamma \vdash \mathtt{e} : \mathtt{obj}\, t.R$, where $\mathtt{n}$ is a method of $\mathtt{e}$ but $\Gamma \vdash R : [\mathtt{n}]$. In this case we say that this rule *forgets* the method $\mathtt{n}$. It is important to remark that, when a method is forgotten in the type of an object, it is like it was never added.

### 7.2.4 Examples Typing Derivation

In this section we will present three examples: the first shows how the "width" subtyping relation works on a critical example of [AC94]. The second example gives a simple object, typable in $\lambda_{obj}^{\preceq}$ , but not $\lambda_{obj}$ .

*Example 7.2.8* Given the following objects:

$$\begin{aligned}
\mathtt{p}_1 &\stackrel{def}{=} \langle \mathtt{x} = \lambda self.0,\ \mathtt{mv}_{\mathtt{x}} = \lambda self.\lambda dx.\langle self \leftarrow \mathtt{x} = \lambda s.(self \Leftarrow \mathtt{x}) + dx \rangle \rangle \\
\mathtt{p}_2 &\stackrel{def}{=} \langle \mathtt{x} = \lambda self.1,\ \mathtt{y} = \lambda self.0,\ \mathtt{mv}_{\mathtt{x}} = \lambda self.\lambda dx.\langle self \leftarrow \mathtt{x} = \lambda s.(self \Leftarrow \mathtt{x}) + dx \rangle, \\
&\qquad \mathtt{mv}_{\mathtt{y}} = \lambda self.\lambda dy.\langle self \leftarrow \mathtt{y} = \lambda s.(self \Leftarrow \mathtt{y}) + dy \rangle \rangle,
\end{aligned}$$

**Valid Signatures and Axioms**

$$\frac{}{\emptyset\ sig}\ (axiom_\Sigma) \qquad \frac{\Sigma\ sig}{\varepsilon \vdash *}\ (axiom_\Gamma) \qquad \frac{\Sigma\ sig \quad \vdash A * B \quad a \notin \mathcal{D}om\,(\Sigma)}{\Sigma, a{:}A\ sig}\ (var_\Sigma)$$

**Rules for Types**

$$\frac{\Gamma \vdash * \quad t \notin Dom(\Gamma)}{\Gamma, t{:}T \vdash *}\ (type{-}var) \qquad \frac{\Gamma \vdash \tau_1 : T \quad \Gamma \vdash \tau_2 : T}{\Gamma \vdash \tau_1 {\rightarrow} \tau_2 : T}\ (type{-}arr)$$

$$\frac{\Gamma, t{:}T \vdash R : [\overline{\mathtt{m}}]}{\Gamma \vdash \mathtt{obj}\, t.R : T}\ (object)$$

**General Rules**

$$\frac{\Gamma \vdash * \quad a{:}A \in \Sigma}{\Gamma \vdash a : A}\ (proj_\Sigma) \qquad \frac{\Gamma \vdash * \quad a{:}A \in \Gamma}{\Gamma \vdash a : A}\ (proj_\Gamma) \qquad \frac{\Gamma \vdash A : B \quad \Gamma, \Gamma' \vdash *}{\Gamma, \Gamma' \vdash A : B}\ (weak)$$

**Type and Row Equality**

$$\frac{\Gamma \vdash \tau : T \quad \tau \rightarrow_\beta \tau'}{\Gamma \vdash \tau' : T}\ (type{-}\beta) \qquad \frac{\Gamma \vdash \tau' : T \quad \Gamma \vdash \mathtt{e} : \tau \quad \tau =_\beta \tau'}{\Gamma \vdash \mathtt{e} : \tau'}\ (type{-}eq)$$

$$\frac{\Gamma \vdash R : \kappa \quad R \rightarrow_\beta R'}{\Gamma \vdash R' : \kappa}\ (row{-}\beta)$$

**Rules for Rows**

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\!\langle\rangle\!\rangle : [\overline{\mathtt{m}}]}\ (empty{-}row) \qquad \frac{\Gamma \vdash * \quad r \notin Dom(\Gamma)}{\Gamma, r{:}T^p {\rightarrow} [\overline{\mathtt{m}}] \vdash *}\ (row{-}var)$$

$$\frac{\Gamma, t{:}T \vdash R : T^p {\rightarrow} [\overline{\mathtt{m}}]}{\Gamma \vdash \lambda t{:}T.R : T^{n+1} {\rightarrow} [\overline{\mathtt{m}}]}\ (row{-}abs) \qquad \frac{\Gamma \vdash R : T {\rightarrow} \kappa \quad \Gamma \vdash \tau : T}{\Gamma \vdash R\tau : \kappa}\ (row{-}app)$$

$$\frac{\Gamma \vdash R : [\overline{\mathtt{m}}, \mathtt{n}] \qquad \Gamma \vdash \tau : T}{\Gamma \vdash \langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle : [\overline{\mathtt{m}}]}\ (row{-}ext) \qquad \frac{\Gamma \vdash R : T^p {\rightarrow} [\overline{\mathtt{m}}] \qquad \{\overline{\mathtt{n}}\} \subseteq \{\overline{\mathtt{m}}\}}{\Gamma \vdash R : T^p {\rightarrow} [\overline{\mathtt{n}}]}\ (row{-}label)$$

Figure 7.3: Common Rules

we can derive $\varepsilon \vdash \mathtt{p}_1 : P_1$ and $\varepsilon \vdash \mathtt{p}_2 : P_2$, where:

$$
\begin{aligned}
P_1 &\stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}_\mathtt{x}{:}(int \to t)_\mathtt{x} \rangle\!\rangle \\
P_2 &\stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{y}{:}int, \mathtt{mv}_\mathtt{x}{:}(int \to t)_\mathtt{x}, \mathtt{mv}_\mathtt{y}{:}(int \to t)_\mathtt{y} \rangle\!\rangle.
\end{aligned}
$$

It is easy to verify that in our system $P_2 \preceq P_1$. This relation between $P_2$ and $P_1$ is the one we want to have, since it is the intuitive relation between a one-dimensional point and a two-dimensional point. If we modify $\mathtt{p}_1$ and $\mathtt{p}_2$ as follows:

$$
\begin{aligned}
\mathtt{p}_1' &\stackrel{def}{=} \langle \mathtt{p}_1 \leftarrow \mathtt{mv}_\mathtt{x} = \lambda self.\lambda dx.self \rangle \\
\mathtt{p}_2' &\stackrel{def}{=} \langle\langle \mathtt{p}_2 \leftarrow \mathtt{x} = \lambda self.((self \Leftarrow \mathtt{mv}_\mathtt{x}\, 1) \Leftarrow \mathtt{y})\rangle \leftarrow \mathtt{mv}_\mathtt{x} = \lambda self.\lambda dx.self \rangle,
\end{aligned}
$$

we can derive $\varepsilon \vdash \mathtt{p}_1' : P_1'$ and $\varepsilon \vdash \mathtt{p}_2' : P_2'$, where:

$$
\begin{aligned}
P_1' &\stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{mv}_\mathtt{x}{:}(int \to t) \rangle\!\rangle \\
P_2' &\stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int_{\mathtt{mv}_\mathtt{x},\mathtt{y}}, \mathtt{y}{:}int, \mathtt{mv}_\mathtt{x}{:}(int \to t), \mathtt{mv}_\mathtt{y}{:}(int \to t)_\mathtt{y} \rangle\!\rangle.
\end{aligned}
$$

Now $P_2' \not\preceq P_1'$, because we cannot forget the $\mathtt{y}$ method since the $\mathtt{x}$ method uses it. Therefore, we are unable to assign type $P_1'$ to the object $\mathtt{p}_2'$. In this way, we avoid the *message-not-understood* error. In fact, if we allowed $P_2' \preceq P_1'$, we would get $\varepsilon \vdash \mathtt{p}_2' : P_1'$ by subtyping. Then, it would be possible to override the $\mathtt{mv}_\mathtt{x}$ method of $\mathtt{p}_2'$ by a body that has an output of type $P_1'$. Since the $\mathtt{x}$ method of $\mathtt{p}_2'$ uses $\mathtt{y}$, this would produce a run-time error. Let us formalize this situation. The original pattern of this example appears in [AC94], paragraph 5.4. Let us consider the object:

$$
\mathtt{p}_2'' \stackrel{def}{=} \langle \mathtt{p}_2' \leftarrow \mathtt{mv}_\mathtt{x} = \lambda self.\lambda dx.p_1' \rangle.
$$

We cannot give type $P_1'$ to $\mathtt{p}_2''$. If it would be possible, then we could send message $\mathtt{x}$ to $p_2''$, and this will cause a run-time error, since the body of $\mathtt{x}$ sends the message $\mathtt{y}$ to the object $(self \Leftarrow \mathtt{mv}_\mathtt{x}\, 1)$, but this object does not have any $\mathtt{y}$ method.

*Example 7.2.9* (A Geometric Object) Consider the object `draw`, that can receive two messages: `fig`, that describes a geometric figure, and `plot`, that, given a point, colors it black or white, depending on the position of the point with respect to the figure. The object `draw` accepts as input both a colored point or a point. This would be impossible in $\lambda_{obj}$, since there one would have to write two different objects, one for colored points and one for points, with different bodies for the method `plot`. In fact, for colored points we need an override instead of an extension. For the object `draw`:

$$\begin{aligned}\texttt{draw} \stackrel{def}{=} \langle \texttt{fig} &= \lambda self.\lambda dx.\lambda dy.(y = f(x)),\\ \texttt{plot} &= \lambda self.\lambda p.\\ &\quad if\ (self \Leftarrow \texttt{fig})(p \Leftarrow x)(p \Leftarrow y)\\ &\quad then\ \langle p \leftarrow\!\circ\ \texttt{col} = \lambda self.black\rangle\\ &\quad else\ \langle p \leftarrow\!\circ\ \texttt{col} = \lambda self.white\rangle\rangle\\ \rangle,\end{aligned}$$

we can derive $\varepsilon \vdash \texttt{draw} : DR$, where

$$\begin{aligned} DR &\stackrel{def}{=} \texttt{obj}\, t.\langle\!\langle \texttt{fig}{:}int \to int \to bool, \texttt{plot}{:}(P \to CP)_{\texttt{fig}} \rangle\!\rangle\\ P &\stackrel{def}{=} \texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int, \texttt{mv}_{\texttt{x}}{:}(int \to t)_{\texttt{x}}, \texttt{mv}_{\texttt{y}}{:}(int \to t)_{\texttt{y}} \rangle\!\rangle\\ CP &\stackrel{def}{=} \texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int, \texttt{mv}_{\texttt{x}}{:}(int \to t)_{\texttt{x}}, \texttt{mv}_{\texttt{y}}{:}(int \to t)_{\texttt{y}}, \texttt{col}{:}Col \rangle\!\rangle. \end{aligned}$$

*Example 7.2.10* (Labels for Encapsulation) Take a special label *priv*, for the types of methods we want to consider "private", and add to the $(obj{-}over)$ rule the condition $priv \notin \Delta$. Then we obtain a form of *encapsulation* for methods in objects that become "read-only". For example, in the object `point` previously showed, to make the `mv` method read-only, we can add the *priv* dependences to its label, thus obtaining the type:

$$\texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int, \texttt{mv}{:}(int \to int \to t)_{\texttt{x},\texttt{y},priv} \rangle\!\rangle,$$

It follows that the `mv` method cannot be modified by overriding. In Section *7.6*, we will explain out the different approach of [FM95].

## 7.3 Labeled-Types and Method Specialization: Some Problems

In this subsection, we briefly introduce some problems which are related to the method specialization and the addition of the labeled-types to the type system of $\lambda_{obj}$. If we take an object-type $\texttt{obj}\, t.\langle\!\langle \overline{\texttt{m}}{:}\overline{\tau} \rangle\!\rangle$, then every occurrence of the $t$ variable in $\langle\!\langle \overline{\texttt{m}}{:}\overline{\tau} \rangle\!\rangle$ means "the object itself". The problem is that the meaning of the $t$ variable can be different, depending on the label we have attached to the types of the methods. Let us examine, with the help of examples, some problems

*Example 7.3.1* (Binary Methods) If we take a (binary) method eq of type $(t \rightarrow bool)$ inside an object point p, then the occurrence of the $t$ variable means that the eq method will compare the coordinates of p with those of the input argument, say the object point q. To be precise, consider the following typing for p and q, respectively:

$$Point \stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{y}{:}int, \mathtt{eq}{:}(t \rightarrow bool)_{\mathtt{x},\mathtt{y}} \rangle\!\rangle,$$

and

$$Q\_Point \stackrel{def}{=} \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{y}{:}int_{\mathtt{x}}, \mathtt{eq}{:}(t \rightarrow bool)_{\mathtt{x},\mathtt{y}} \rangle\!\rangle.$$

We could apply eq to p and q if we could derive:

$$Point \preceq Q\_Point,$$

but this type inclusion is unsound, as it will be shown in Example *7.3.5*. Fortunately, we can always derive *Q_Point* for an object of type *Point*, since we are free of adding methods names to labels in building rows (see the typing rules in Figure *7.2.2*).

*Example 7.3.2* (The use of labels in applications) A natural question is that we can disregard labels in applications. This would solve also in a different way the problem of Example *7.3.1*. More precisely, one could ask if it is possible to reformulate the rule for typing application as follows:

$$\frac{\Gamma \vdash \mathtt{e}_1 : \sigma \rightarrow \tau \quad \Gamma \vdash \mathtt{e}_2 : \sigma \quad |\sigma| = |\sigma'|}{\Gamma \vdash \mathtt{e}_1 \mathtt{e}_2 : \tau} \;(exp{-}appl)$$

Unfortunately, the interaction of this rule with the subtyping rule in the type system lead to unsoundness. Suppose a point is built as follows:

$$\mathtt{p} \stackrel{def}{=} \langle \mathtt{x} = \lambda self.3, \mathtt{y} = \lambda self.self \Leftarrow \mathtt{x} \rangle : \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{y}{:}int_{\mathtt{x}} \rangle\!\rangle,$$

and consider the following derivation:

$$\dfrac{\dfrac{\begin{array}{c}\varepsilon \vdash \texttt{p} : \texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int_{\texttt{x}}\rangle\!\rangle \\ \varepsilon \vdash \lambda x.x : \texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int\rangle\!\rangle \to \texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int\rangle\!\rangle \\ |\texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int_{\texttt{x}}\rangle\!\rangle| \;=\; |\texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int\rangle\!\rangle|\end{array}}{\dfrac{\varepsilon \vdash (\lambda x.x)\texttt{p} : \texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{y}{:}int\rangle\!\rangle}{\varepsilon \vdash (\lambda x.x)\texttt{p} : \texttt{obj}\, t.\langle\!\langle \texttt{y}{:}int\rangle\!\rangle}\;(sub_{\preceq})}\;(exp\!-\!appl) \qquad \varepsilon, r{:}T\to[\texttt{x}] \vdash \lambda self.true : [\texttt{obj}\, t.\langle\!\langle rt \mid \texttt{x}{:}bool\rangle\!\rangle/t](t\to bool)}{\varepsilon \vdash \langle(\lambda x.x)\texttt{p} \leftarrow\!\circ\; \texttt{x} = \lambda self.true\rangle : \texttt{obj}\, t.\langle\!\langle \texttt{x}{:}bool, \texttt{y}{:}int\rangle\!\rangle}\;(obj\!-\!ext)$$

Then, if we send a message y to the above typable object, we get a *type-mismatch* run-time error, because we first forget the m method by the ($sub \preceq$) application, and then we add again x with a type *different* from the original one.

*Example 7.3.3* (Identity Methods) Consider the following object:

$$\texttt{e} \stackrel{def}{=} \langle \texttt{n} = \lambda self.self, \texttt{m} = \lambda self.\langle self \leftarrow \texttt{n} = \lambda s.s \Leftarrow \texttt{m}\rangle\rangle.$$

Then, what can be the type of e? At as first glance, we can suppose that the type of e could be:

$$\texttt{obj}\, t.\langle\!\langle \texttt{n}{:}t, \texttt{m}{:}t_{\texttt{n}}\rangle\!\rangle.$$

Is it the right choice? The answer comes by sending the message m to e. Then, we have the following computation:

$$\begin{array}{rcl}\texttt{e} \Leftarrow \texttt{m} & \stackrel{eval}{\to} & (\lambda self.\langle self \leftarrow \texttt{n} = \lambda s.s \Leftarrow \texttt{m}\rangle)\texttt{e} \\ & \stackrel{eval}{\to} & \langle \texttt{e} \leftarrow \texttt{n} = \lambda s.s \Leftarrow \texttt{m}\rangle.\end{array}$$

It is easy to verify now that the typing for $\texttt{e} \Leftarrow \texttt{m}$ should be $\texttt{obj}\, t.\langle\!\langle \texttt{n}{:}t_{\texttt{m}}, \texttt{m}{:}t_{\texttt{n}}\rangle\!\rangle$, because the resulting object will have extensionally the following form:

$$\langle \texttt{n} = \lambda self.self \Leftarrow \texttt{m}, \texttt{m} = \lambda self.\langle self \leftarrow \texttt{n} = \lambda s.s \Leftarrow \texttt{m}\rangle\rangle.$$

It follows that, if we could derive the initial choice for the typing of e, we would have lost the subject reduction property. This is essentially due to the fact that the occurrence of the $t$ variable in the object-type of $\mathtt{e}$ does not always means the object itself. In fact, after the call of $\mathtt{m}$ we will get a different type. The same problems appear when we send a message $\mathtt{m}_1$ to the following object:

$$\begin{aligned}\mathtt{e}' \stackrel{def}{=} \langle \mathtt{m}_1 &= \lambda \mathit{self}.\mathit{self} \Leftarrow \mathtt{m}_2,\\ \mathtt{m}_2 &= \lambda \mathit{self}.\langle \mathit{self} \leftarrow \mathtt{m}_3 = \lambda s.s \Leftarrow \mathtt{m}_1\rangle,\\ \mathtt{m}_3 &= \lambda \mathit{self}.\mathit{self},\\ \rangle,\end{aligned}$$

if the type system would assign to $\mathtt{e}'$ the type $\mathtt{obj}\, t.\langle\!\langle \mathtt{m}_1{:}t_{\mathtt{m}_2}, \mathtt{m}_2{:}t_{\mathtt{m}_1,\mathtt{m}_3}, \mathtt{m}_3{:}t\rangle\!\rangle$. In fact, sending $\mathtt{m}_1$ to $\mathtt{e}'$ will produce a new object of type:

$$\langle \mathtt{e}' \leftarrow \mathtt{m}_3 = \lambda \mathit{self}.\mathit{self} \Leftarrow \mathtt{m}_1\rangle : \mathtt{obj}\, t.\langle\!\langle \mathtt{m}_1{:}t_{\mathtt{m}_2}, \mathtt{m}_2{:}t_{\mathtt{m}_3}, \mathtt{m}_3{:}t_{\mathtt{m}_1}\rangle\!\rangle.$$

We can conclude that if we want to keep the property of subject reduction, then the labeled-type assigned to a method must preview *all* the possible modifications of the method itself (from the viewpoint of dependences from other methods).

*Example 7.3.4* (Diagonal Point) Suppose we want to build a diagonal point $\mathtt{dp}$, starting from a bi-dimensional point $\mathtt{p}$ with the move method.

$$\begin{aligned}\mathtt{p} \stackrel{ext}{=} \langle &\mathtt{x} = \lambda \mathit{self}.0,\\ &\mathtt{y} = \lambda \mathit{self}.0,\\ &\mathtt{mv}_\mathtt{x} = \lambda \mathit{self}.\lambda dx.\lambda dy.\langle\langle \mathit{self} \leftarrow \mathtt{x} = \lambda s.(\mathit{self} \Leftarrow \mathtt{x}) + dx\rangle \leftarrow \mathtt{y} = \lambda s.(\mathit{self} \Leftarrow \mathtt{y}) + dy\rangle\\ \rangle.&\end{aligned}$$

Then we first override the $\mathtt{y}$ method in order to set the $\mathtt{x}$ and $\mathtt{y}$ coordinates equal as follows:

$$\langle \mathtt{p} \leftarrow \mathtt{y} = \lambda \mathit{self}.\mathit{self} \Leftarrow x\rangle,$$

and then we also override the $\mathtt{mv}$ method (that modifies the $\mathtt{x}$ and $\mathtt{y}$ coordinates of $\mathtt{dp}$ by the same displacements), even if the $\mathtt{mv}$ method may be directly inherited from the object $\mathtt{p}$, i.e.:

$$
\begin{array}{rl}
\mathtt{dp} \stackrel{ext}{=} & \langle \mathtt{x} = \lambda self.0, \\
& \mathtt{y} = \lambda self.self \Leftarrow \mathtt{x}, \\
& \mathtt{mv} = \lambda self.\lambda dx.\lambda dy.if\ dx \neq dy\ then\ self\ else \\
& \quad \langle\langle self \leftarrow \mathtt{x} = \lambda s.(self \Leftarrow \mathtt{x}) + dx\rangle \leftarrow \mathtt{y} = \lambda s'.(self \Leftarrow \mathtt{y}) + dy\rangle \\
& \rangle.
\end{array}
$$

In other words, mv cannot be inherited from p, since it must also be specialized in accordance with the behaviour of the dp object, obtained from p. So method specialization seems to complicate static analysis. This is essentially due to the presence of the override that modifies the dependences. In the previous examples, the y method in p had no dependences, while in dp it depends on the x method.

Then dp will have the following object-type:

$$\mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{y}{:}int_{\mathtt{x}}, \mathtt{mv}{:}(int{\rightarrow}int{\rightarrow}t)_{\mathtt{x},\mathtt{y}}\rangle\!\rangle.$$

*Example 7.3.5* (Enforcement of Labels and "Depth" Subtyping) In Example *7.3.1*, we presented two object-types, namely $Point$ and $Q\_Point$ which differ only on labels. The typing rules of $\lambda^{\preceq}_{obj}$ allow to insert in labels any number of methods. So, the possibility of deriving an object-type which has "enforced" labels, allows to compare, in binary methods, objects of type $Point$ and $Q\_Point$, respectively. Therefore, it would be interesting to have a subtyping rule such as:

$$\frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_{\Delta}\rangle\!\rangle : T \qquad \Delta \subseteq \Delta'}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_{\Delta}\rangle\!\rangle \preceq \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_{\Delta'}\rangle\!\rangle}\ (deep^{\Delta}_{\preceq})$$

which adds method names in the labels of types. Unfortunately, adding this rule to the typing system will cause the failure of the subject reduction property. The cause of this negative result is the fact that the arrow-type behave contra-variantly.

The same argument can be argued to say that the following rule, introducing a restricted form of "depth" subtyping, is unsound:

$$\frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \overline{\mathtt{n}}{:}\overline{\sigma}_{\Delta}\rangle\!\rangle : T \qquad \Gamma, t{:}T \vdash \overline{\sigma} \preceq \overline{\tau} \qquad \overline{\mathtt{n}} \notin \mathcal{L}(\overline{\alpha})}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \overline{\mathtt{n}}{:}\overline{\sigma}_{\Delta}\rangle\!\rangle \preceq \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \overline{\mathtt{n}}{:}\overline{\tau}_{\Delta}\rangle\!\rangle}\ (deep_{\preceq})$$

### 7.3.1 Subtyping and Binary Methods

In this subsection, we will discuss problems related to interaction between subtyping and the possibility of defining binary methods. Consider the familiar object point p with a binary method eq of type *Point*, and suppose to extend p to build a colored point cp, by overriding the eq method in order to compare also the color component:

$$\texttt{cp} \stackrel{def}{=} \langle\langle \texttt{p} \hookleftarrow \texttt{col} = \lambda \mathit{self}.\mathit{red}\rangle \leftarrow \texttt{eq} = \ldots\rangle,$$

of type:

$$C\_Point \stackrel{def}{=} \texttt{obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{col}{:}Col, \texttt{mv}_{\texttt{x}}{:}(int \to t)_{\texttt{x}}, \texttt{eq}{:}(t \to bool)_{\texttt{x},\texttt{col}}\rangle\!\rangle.$$

Observe that the eq method still has type $t \to bool$, even if the method compares also the color component. The subtyping relation gives:

$$C\_Point \preceq Point.$$

Now consider the following contexts:

$$(\texttt{cp} \Leftarrow \texttt{eq})[\;], \text{ and } (\texttt{p} \Leftarrow \texttt{eq})[\;].$$

In the first context, the eq methods checks the equality looking also to the color component, while in the second context not. So, there are four cases to consider:

$$\begin{array}{lll} 1) & (\texttt{cp} \Leftarrow \texttt{eq}) & [\texttt{cp}] \\ 2) & (\texttt{cp} \Leftarrow \texttt{eq}) & [\texttt{p}] \\ 3) & (\texttt{p} \Leftarrow \texttt{eq}) & [\texttt{cp}] \\ 4) & (\texttt{p} \Leftarrow \texttt{eq}) & [\texttt{p}]. \end{array}$$

The first and the last case have no typing problems. The third case is typable thanks to the subtyping relation; in fact, the argument of $(\texttt{p} \Leftarrow \texttt{eq})$ has type $C\_Point$ that can be subsumed to type *Point*. Instead, the typing of the second case fails: in fact no subsumption is possible because $Point \not\preceq C\_Point$. This is sound. In fact, since the body of the eq method can be:

$$\lambda \mathit{self}.\lambda p.\mathit{equal}(\mathit{self} \Leftarrow \texttt{x})(p \Leftarrow \texttt{x})\ \&\ \mathit{equal}(\mathit{self} \Leftarrow \texttt{col})(p \Leftarrow \texttt{col}),$$

it follows that, if we send `eq` to `cp` with argument `p`, then the argument `p` will receive a `col` message and this causes an error.

So, the very delicate case is when we fill a point object into a context which expects a colored point. We discuss some lines of solutions in an informal way.

**Coercion of the Argument.** This means promoting a point object into a coloured point object, by adding a *default* color. This solution is not realistic.

**Coercion of the Function.** This means to translate the body of the `eq` method into an equivalent function for ordinary point. As example:

$$\begin{aligned} coe_{C\_Point \to Point}(\lambda self.\lambda p.equal(self \Leftarrow \mathtt{x})(p \Leftarrow \mathtt{x})\ \&\ equal(self \Leftarrow \mathtt{col})(p \Leftarrow \mathtt{col})) = \\ \lambda self.\lambda p.equal(self \Leftarrow \mathtt{x})(p \Leftarrow \mathtt{x}). \end{aligned}$$

This solution seems to be reasonable. The resulting language will feature a restricted form of multiple dispatching, since the coercion should be a very simple operation like the elimination of some test. While multiple dispatching implies some loss of encapsulation, this would be a good compromise for having both features.

**Generic Functions.** If we want to type all the above expressions, then we must provide the capability of associating to the `eq` method two bodies, one of them comparing the color component and the other not, and we must add a mechanism to choose the correct body, depending on the type of the argument given in input. This corresponds to add *multiple dispatch* to $\lambda^{\preceq}_{obj}$, and implies a more difficult treatment of override (because of method specialization). This solution seems to integrate the $\lambda^{\preceq}_{obj}$-calculus with the $\lambda\&$-calculus of Castagna and Longo [CGL95]. However, to perform such integration, we need a "fully" typed version of $\lambda^{\preceq}_{obj}$, since the choice of the correct "branch" depends on the types of the bound variables of an overloaded method body.

We refer to [BCG$^+$95] for more discussion on binary methods.

## 7.4 Basic Properties of the System

In this section, we will show that $\lambda^{\preceq}_{obj}$ has all the good properties of the original system. We follow the same pattern of [FHM94]: first, we introduce some substitution lemmas and then the notion of derivation in *normal form* that simplifies the proofs of some technical lemmas; then we prove a subject reduction theorem, which will have as a corollary the type-soundness of the system.

### 7.4.1 Substitution Properties

The following lemmas are useful to show both a substitution property on type and kind derivations and to specialize object-types with additional methods.

The first lemma shows that the contexts are well-formed in every judgment and allows to treat contexts, which are lists, more like sets.

*Lemma 7.4.1* *i)* (Prefix) *If* $\Gamma \sqsubseteq \Gamma'$ *and* $\Gamma' \vdash *$, *then* $\Gamma \vdash *$.

*ii)* (Legality) *If* $\Gamma \vdash A \bullet B$, *then* $\Gamma \vdash *$.

*iii)* (Thinning) *If* $\Gamma, \Gamma' \vdash A \bullet B$ *and* $\Gamma, c{:}C, \Gamma' \vdash *$, *then* $\Gamma, c{:}C, \Gamma' \vdash A \bullet B$.

*Proof:* By induction on derivations.

*i)* Easy.

*ii)* The $(weak)$, $(proj)$, $(empty{-}row)$, and $(empty{-}obj)$ rules are immediate, the $(object)$, $(row{-}abs)$, and $(exp{-}abs)$ rules follow by induction from part *(i)*. All other cases follow by induction.

*iii)* The $(obj{-}ext)$ and $(obj{-}over)$ rules use the easy property that permits consistent renaming of row-variables in a judgment. All other cases follow by. ∎

The following lemma shows that the notion of $\beta$-reduction is closed under substitution.

*Lemma 7.4.2* *if* $U_1 \rightarrow_\beta U_2$, *then* $[U/u]U_1 \rightarrow_\beta [U/u]U_2$.

*Proof:* By easy induction on the definition of $\rightarrow_\beta$. ∎

The next lemma states a substitution property for type and kind derivations; the extra assumption $\Gamma, [U_2/u_2]\Gamma' \vdash *$ is needed in the rules $(proj_\Sigma)$, $(proj_\Gamma)$, $(weak)$, and $(empty{-}row)$.

*Lemma 7.4.3* *If* $\Gamma, u_2{:}V_2, \Gamma' \vdash U_1 \bullet V_1$ *and* $\Gamma \vdash U_2 : V_2$ *and* $\Gamma, [U_2/u_2]\Gamma' \vdash *$, *then* $\Gamma, [U_2/u_2]\Gamma' \vdash [U_2/u_2]U_1 \bullet [U_2/u_2]V_1$.

*Proof:* We distinguish two cases:

*i)* $U_1 \bullet V_1$ is a statement of the form $R : \kappa$ or $\tau : T$. This part is proven by induction on a derivation of $\Gamma, u_2{:}V_2, \Gamma' \vdash U_1 \bullet V_1$. Notice that $[U_2/u_2]V_1 \equiv V_1$.

[$(proj_\Gamma)$]: we distinguish two cases, namely, if the projected variable is $u_2$ or not. In the first case, $U_1 \equiv u_2$, $V_1 \equiv V_2$, and the derivation has the following shape:

$$\frac{\Gamma, u_2{:}V_2, \Gamma' \vdash * \quad u_2{:}V_2 \in \Gamma, u_2{:}V_2, \Gamma'}{\Gamma, u_2{:}V_2, \Gamma' \vdash u_2 : V_2}\ (proj_\Gamma)$$

Since $\Gamma \vdash U_2 : V_2$ is $\Gamma \vdash [U_2/u_2]u_2 : V_2$, the thesis follows by an application of $(weak)$ rule, i.e.:

$$\frac{\Gamma \vdash [U_2/u_2]u_2 : V_2 \qquad \Gamma, [U_2/u_2]\Gamma' \vdash *}{\Gamma, [U_2/u_2]\Gamma' \vdash [U_2/u_2]u_2 : V_2}\ (weak)$$

If the projected variable is not $u_2$, then $U_1 \equiv u_1$, for some $u_1 \not\equiv u_2$, and the derivation has the following shape:

$$\frac{\Gamma, u_2{:}V_2, \Gamma' \vdash * \quad u_1{:}V_1 \in \Gamma, u_2{:}V_2, \Gamma'}{\Gamma, u_2{:}V_2, \Gamma' \vdash u_1 : V_1}\ (proj_\Gamma)$$

Observe that if $u_1{:}V_1 \in \Gamma$, or $u_1{:}V_1 \in \Gamma'$, then $u_1{:}V_1 \in \Gamma, [U_2/u_2]\Gamma'$. So, we can apply a $(proj_\Gamma)$ rule to obtain the thesis.

[$(weak)$]: then the derivation has the following shape:

$$\frac{\Gamma'' \vdash U_1 : V_1 \qquad \Gamma, u_2{:}V_2, \Gamma' \vdash *}{\Gamma, u_2{:}V_2, \Gamma' \vdash U_1 : V_1}\ (weak)$$

being $\Gamma'' \sqsubseteq \Gamma, u_2{:}V_2, \Gamma'$. We have to distinguish two cases, namely, if $u_2{:}V_2 \in \Gamma''$ or not. For the first case, we have that $\Gamma'' = \Gamma', u_2{:}V_2, \Gamma'''$, for some $\Gamma'''$. By induction, we get $\Gamma, [U_2/u_2]\Gamma''' \vdash [U_2/u_2]U_1 : V_2$. Then apply a $(weak)$ rule to this statement and to the assumption $\Gamma, [U_2/u_2]\Gamma' \vdash *$, and we get $\Gamma, [U_2/u_2]\Gamma' \vdash [U_2/u_2]U_1 : V_2$.

[$(row{-}\beta)$, and $(type{-}\beta)$]: take as example the rule $(row{-}\beta)$:

$$\frac{\Gamma, u_2{:}V_2, \Gamma' \vdash U : V_1 \qquad U \to_\beta U_1}{\Gamma, u_2{:}V_2, \Gamma' \vdash U_1 : V_1}\ (row{-}\beta)$$

this case follows directly from Lemma *7.4.2*.

[$(object)$ and $(row{-}abs)$]: these cases follow from the fact that bound variables may be consistently renamed when necessary.

[(*other rules*)]: most of the cases are either vacuous or follow immediately by induction hypothesis.

*ii)* $U_1 \bullet V_1$ is a statement of the shape $\sigma \preceq \tau$. By induction on the definition of $\preceq$. The cases are either vacuous or follow immediately by induction hypothesis. ■

*Lemma 7.4.4* *If* $\Gamma, u_2{:}V_2, \Gamma' \vdash *$ *and* $\Gamma \vdash U_2 : V_2$, *then* $\Gamma, [U_2/u_2]\Gamma' \vdash *$.

*Proof:* By induction on the length of $\Gamma'$.

[$\Gamma' = \varepsilon$]: then $\Gamma \vdash *$, as desired.

[$\Gamma' = \Gamma'', a{:}A$]: by cases on the the last applied rule. We consider the more interesting case, i.e. that of rule $(exp{-}var)$. Then the derivation has the following shape:

$$\frac{\Gamma, u_2{:}V_2, \Gamma'' \vdash \tau : T \qquad x \notin Dom(\Gamma)}{\Gamma, u_2{:}V_2, \Gamma'', x{:}\tau \vdash *} \ (exp{-}var)$$

By induction $\Gamma, [U_2/u_2]\Gamma'' \vdash *$, so by Lemma *7.4.3*, we get $\Gamma, [U_2/u_2]\Gamma'' \vdash [U_2/u_2]\tau : T$. By an application of a $(exp{-}var)$ rule we conclude $\Gamma, [U_2/u_2]\Gamma'', x{:}[U_2/u_2]\tau \vdash *$, as desired. ■

The following lemma gives the desired substitution property on type and kind derivations.

*Lemma 7.4.5* *If* $\Gamma, u_2{:}V_2, \Gamma' \vdash U_1 \bullet V_1$ *and* $\Gamma \vdash U_2 : V_2$, *then* $\Gamma, [U_2/u_2]\Gamma' \vdash [U_2/u_2]U_1 \bullet [U_2/u_2]V_1$.

*Proof:* By Lemmas *7.4.1 (ii)*, *7.4.3*, and *7.4.4*. ■

The aim of the following lemma is to remove unuseful assumptions in the context.

*Lemma 7.4.6* *If* $\Gamma, a{:}A, \Gamma' \vdash B \bullet C$ *and* $a \notin \mathcal{FV}(B) \cup \mathcal{FV}(C) \cup \mathcal{FV}(\Gamma')$, *then* $\Gamma, \Gamma' \vdash B \bullet C$.

*Proof:* Immediate. ■

The following lemma is used together with the previous ones in order to specialize object-types to contain additional methods.

*Lemma 7.4.7* (Substitutions) *i) If* $\Gamma, r{:}T^p {\rightarrow} [\overline{\mathtt{m}}], \Gamma' \vdash \mathtt{e} : \tau$ *and* $\Gamma \vdash R : T^p {\rightarrow} [\overline{\mathtt{m}}]$*, then* $\Gamma, [R/r]\Gamma' \vdash \mathtt{e} : [R/r]\tau$.

*ii) If* $\Gamma, x{:}\tau_1, \Gamma' \vdash_N \mathtt{e}_2 : \tau_2$ *and* $\Gamma \vdash_N \mathtt{e}_1 : \tau_1$*, then* $\Gamma, \Gamma' \vdash_N [\mathtt{e}_1/x]\mathtt{e}_2 : \tau_2$.

*Proof:* By induction on derivations.

*i)* We show the most important cases.

$[(proj_\Gamma)]$: the premises give $\Gamma, r{:}T^p {\rightarrow} [\overline{\mathtt{m}}], \Gamma' \vdash *$, and $x{:}\tau \in Dom(\Gamma, r{:}T^p {\rightarrow} [\overline{\mathtt{m}}], \Gamma')$. By Lemma *7.4.4*, we get $\Gamma, [R/r]\Gamma' \vdash *$, and, since $x{:}\tau \in Dom(\Gamma, r{:}T^p {\rightarrow} [\overline{\mathtt{m}}], \Gamma')$ implies $x{:}[R/r]\tau \in Dom(\Gamma, [R/r]\Gamma')$, we can apply the $(proj_\Gamma)$ rule to get $\Gamma, [R/r]\Gamma', x{:}[R/r]\tau \vdash *$, as desired.

$[(obj{-}ext)]$: this case needs Lemma *7.4.5*.

*ii)* We consider just one case, namely if the last applied rule is $(obj{-}ext)$ and $x$ occurs in $\mathtt{e}_2$. Then the derivation has the following shape:

$$\frac{\begin{array}{ll} \Gamma, x{:}\tau, \Gamma' \vdash \mathtt{e}_3 : \mathtt{obj}\, t.R & \Gamma, x{:}\tau, \Gamma', t{:}T \vdash R : [\mathtt{n}] \\ \{\overline{\mathtt{m}}{:}\overline{\alpha}\} \subseteq R_\Delta & \quad r \textit{ not in } \tau \\ \multicolumn{2}{l}{\Gamma, x{:}\tau, \Gamma', r{:}T {\rightarrow} [\overline{\mathtt{m}}, \mathtt{n}] \vdash \mathtt{e}_4 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\sigma_\Delta \rangle\!\rangle/t](t {\rightarrow} \sigma)} \end{array}}{\Gamma, x{:}\tau, \Gamma' \vdash \langle \mathtt{e}_3 \leftarrow\!\circ\, \mathtt{n} = \mathtt{e}_4 \rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\sigma_\Delta \rangle\!\rangle} \; (obj{-}ext)$$

By induction, we get:

$$\Gamma, \Gamma' \vdash [\mathtt{e}_2/x]\mathtt{e}_3 : \mathtt{obj}\, t.R,$$

and

$$\Gamma, \Gamma', r{:}T {\rightarrow} [\overline{\mathtt{m}}, \mathtt{n}] \vdash [\mathtt{e}_2/x]\mathtt{e}_4 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\sigma_\Delta \rangle\!\rangle/t](t {\rightarrow} \sigma).$$

By Lemma *7.4.6*, also

$$\Gamma, \Gamma', t{:}T \vdash R : [\mathtt{n}].$$

So we can conclude the proof by applying an $(obj{-}ext)$ rule to get:

$$\Gamma, \Gamma' \vdash [\mathtt{e}_2/x]\langle \mathtt{e}_3 \leftarrow\!\circ\, \mathtt{n} = \mathtt{e}_4 \rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle,$$

as desired. ■

### 7.4.2 Normal Form Derivations

It is well known that equality-rules in proof systems usually complicate derivations, and make theorems and lemmas difficult to prove. In this section, we introduce the notion of *normal form derivation* and of type and row in *normal form*, respectively denoted by $\vdash_N$ and $\tau nf$ in [FHM94]. Although it is not possible to derive all judgments of the system by means of these derivations, we will show that all judgments whose rows and type expressions are in $\tau nf$ are $\vdash_N$ derivable. Using this, we can prove the subject reduction theorem using only $\vdash_N$ derivations.

*Definition 7.4.8* *i)* $\Gamma \vdash_N \mathtt{e} : \tau$ *is a normal form derivation only if the only appearance of an equality rule is as* $(row-\beta)$ *immediately following an occurrence of a* $(row-appl)$ *rule.*

*ii) The* $\tau nf$ *of a type and of a row are their* $\beta$*-normal form.*

It is easy to show that $\tau nf$ satisfies the following identities:

*Fact 7.4.9* *For* $\tau nf$*, the following equivalences holds:*

$$\begin{aligned} \tau nf(\mathtt{obj}\, t.R) &\equiv \mathtt{obj}\, t.\tau nf(R) \\ \tau nf(\langle\!\langle R \mid \mathtt{m}{:}\tau_\Delta \rangle\!\rangle) &\equiv \langle\!\langle \tau nf(R) \mid \mathtt{m}{:}\tau nf(\tau_\Delta) \rangle\!\rangle \\ \tau nf(\tau_\Delta) &\equiv \tau nf(\tau)_\Delta. \end{aligned}$$

In [FHM94], the type and row parts of $\lambda_{obj}$ are translated into the typed $\lambda$-calculus with arrow-types over an assigned signature $\Theta$ and subtyping (called $\lambda_{\Theta}^{\preceq}\!\rightarrow$). We present an extension of the translation function $tr$ that takes labeled-types into account. This extension is done by deleting the labels in labeled-types.

*Definition 7.4.10* *Given the following signature* $\Theta$*:*

$$\begin{array}{lll} \textit{Type Constant} & : & typ, meth \\ \textit{Term Constant} & : & \mathtt{iot}_i : typ, \ \textit{for each constant type } \iota_i \\ & & \mathtt{er} : meth \\ & & \mathtt{ar} : typ \rightarrow typ \rightarrow typ \\ & & \mathtt{ob} : (typ \rightarrow meth) \rightarrow typ \\ & & \mathtt{br}_{\mathtt{m}} : meth \rightarrow typ \rightarrow meth, \ \textit{for each method name } \mathtt{m}. \end{array}$$

*i) The translation function* $tr : Types \cup Row \rightarrow \lambda_{\Theta}^{\preceq}\!\rightarrow$ *is inductively defined as follows:*

$$
\begin{aligned}
tr(\iota_i) &= \mathtt{iot}_i \\
tr(t) &= t \\
tr(\tau_1 {\rightarrow} \tau_2) &= \mathtt{ar}\ tr(\tau_1)\ tr(\tau_2) \\
tr(\mathtt{obj}\, t.R) &= \mathtt{ob}\ (\lambda t{:}typ.tr(R)) \\
tr(r) &= r \\
tr(\langle\!\langle\rangle\!\rangle) &= \mathtt{er} \\
tr(\langle\!\langle R \mid \mathtt{m}{:}\tau_\Delta \rangle\!\rangle) &= \mathtt{br}_\mathtt{m}\ tr(R)\ tr(\tau) \\
tr(\lambda t.R) &= \lambda t{:}typ.tr(R) \\
tr(R\tau) &= tr(R)\ tr(\tau).
\end{aligned}
$$

*ii) We extend the function tr to kinds, signatures and contexts in the standard way.*

$$
\begin{aligned}
tr(T) &= typ \\
tr(T^n {\rightarrow} [\overline{\mathtt{m}}]) &= typ^n {\rightarrow} meth \\
tr(\emptyset) &= \emptyset \\
tr(\Sigma, \iota{:}T) &= tr(\Sigma) \cup \{tr(\iota){:}tr(T)\} \\
tr(\Sigma, \iota_1 \preceq \iota_2) &= tr(\Sigma) \cup \{tr(\iota_1) \preceq tr(\iota_2)\} \\
tr(\Sigma, \mathtt{c}{:}\iota) &= tr(\Sigma) \\
tr(\varepsilon) &= \emptyset \\
tr(\Gamma, t{:}T) &= tr(\Gamma) \cup \{tr(t){:}tr(T)\} \\
tr(\Gamma, r{:}\kappa) &= tr(\Gamma) \cup \{tr(r){:}tr(\kappa)\} \\
tr(\Gamma, x{:}\tau) &= tr(\Gamma).
\end{aligned}
$$

Some explanations are needed here: the term constants in the signature $\Theta$ stand for:

$$
\begin{aligned}
\mathtt{iot}_i : typ \ &: \ \text{one term constant in } \Sigma \text{ for all “type-constants”} \\
\mathtt{er} : meth \ &: \ \text{“empty row” constant} \\
\mathtt{ar} : typ \ &: \ \text{“arrow type” constant} \\
\mathtt{ob} : (typ {\rightarrow} meth) {\rightarrow} typ \ &: \ \text{“object-type” constant} \\
\mathtt{br}_\mathtt{m} : meth {\rightarrow} typ {\rightarrow} meth, \ &: \ \text{“build-row” constant, for each method name } \mathtt{m}.
\end{aligned}
$$

Observe also that the translation function $tr$ is conservative with respect to the set of free and bound variables, and, furthermore, if $U_1 =_\alpha U_2$, then $tr(U_1) =_\alpha tr(U_2)$, under the same renaming of bound variables.

To show that the row and type portion of this calculus is strongly normalizing and confluent, we must show the strong normalization and the confluence of $\lambda_{\Theta}^{\preceq}{\to}$. We follow the proof in [FHM94]. To do this, we need to prove two important properties of the translation function $tr$. The first show that the translation of any two terms, related via $\to_\beta$ in our system, are related via $\to_\beta$ in $\lambda_{\Theta}^{\preceq}{\to}$, and the second that the translation of every well-kinded term in our system is a well-typed term in $\lambda_{\Theta}^{\preceq}{\to}$, as it is proved in the next two lemmas.

*Lemma 7.4.11 The function* $tr$ *enjoys the following properties:*

*i)* $[tr(\tau)/t]tr(U) \equiv tr([\tau/t]U)$.

*ii) If* $U_1 \to_\beta U_2$, *then* $tr(U_1) \to_\beta tr(U_2)$.

*iii) If* $tr(U) \to_\beta W$, *then there is a unique expression* $U'$ *such that* $U \to_\beta U'$ *and* $tr(U') \equiv W$, *where* $W$ *is a term in* $\lambda_{\Theta}^{\preceq}{\to}$.

*Proof:* *i)* By induction on the structure of $U$.

*ii)* By induction on the structure of $U_1$. The only interesting case is when $U_1$ is a redex, i.e. $U_1 \equiv (\lambda t.R)\tau$ and $U_2 \equiv [\tau/t]R$. This case follows from part *(i)*.

*iii)* Since, by part *(ii)*, the function $tr$ preserves redexes, then proof can be done by induction on the structure of $U$. As before, we show only the case when $U$ is a redex, i.e. $U \equiv (\lambda t.R)\tau$. Then
$tr((\lambda t.R)\tau) \equiv tr(\lambda t.R)tr(\tau) \equiv (\lambda t{:}typ.tr(R))tr(\tau) \to_\beta [tr(\tau)/t]tr(R)$, which is equal, by part *(i)*, to $tr([\tau/t]R)$. Then, $W \equiv tr([\tau/t]R)$, and $U' \equiv [\tau/t]R$. ∎

The next lemma states that the translation function $tr$ produces typable $\lambda_{\Theta}^{\preceq}{\to}$terms.

*Lemma 7.4.12 If* $\Gamma \vdash U : V$, *then* $tr(\Gamma) \vdash_{\lambda_{\Theta}^{\preceq}\to} tr(U) : tr(V)$.

*Proof:* By induction on a derivation of $\Gamma \vdash U : V$. We show only the most interesting cases, namely, if the last rule is $(row{-}\beta)$ or $(type{-}\beta)$. We consider the second case, the first being similar. Then the derivation has the following shape:

$$\frac{\Gamma \vdash \tau : T \quad \tau \to_\beta \tau'}{\Gamma \vdash \tau' : T}\ (type{-}\beta)$$

Then, by induction, we get $tr(\Gamma) \vdash_{\lambda^{\preceq}_{\Theta}\rightarrow} tr(\tau) : tr(T)$, and $\tau \rightarrow_\beta \tau'$. By Lemma *7.4.11 (ii)*, also $tr(\tau) \rightarrow_\beta tr(\tau')$. Since the subject reduction property holds for $\lambda^{\preceq}_{\Theta}\rightarrow$, also $tr(\Gamma) \vdash_{\lambda^{\preceq}_{\Theta}\rightarrow} tr(\tau') : tr(T)$, as desired. ∎

The following theorem states that the row and type fragment of our calculus is strongly normalizing and confluent.

*Theorem 7.4.13* (Strong Normalization and Confluence for Row and Type Portion)

*i) If $\Gamma \vdash U : V$, then there is no infinite sequence of $\rightarrow_\beta$ reductions out of $U$.*

*ii) If $\Gamma \vdash U_1 : V_1$ and $U_1 \twoheadrightarrow_\beta U_2$ and $U_1 \twoheadrightarrow_\beta U_3$, then there exists $U_4$, such that $U_2 \twoheadrightarrow_\beta U_4$ and $U_3 \twoheadrightarrow_\beta U_4$.*

*Proof:* *i)* By Lemmas *7.4.11 (ii)*, *7.4.12*, and the strong normalization of $\lambda^{\preceq}_{\Theta}\rightarrow$.

*ii)* By Lemmas *7.4.11 (ii)*, *(iii)*, *7.4.12*, and the confluence of $\lambda^{\preceq}_{\Theta}\rightarrow$. ∎

The following lemma states that, for each derivation in our system, it is possible to find a corresponding derivation in normal form. Let $\tau nf(\mathsf{e}) = \mathsf{e}$, and let $\tau nf(\Gamma)$ be the context listing the $\tau nf$'s of the elements of $\Gamma$.

*Lemma 7.4.14 If $\Gamma \vdash A \bullet B$, then $\tau nf(\Gamma) \vdash_N \tau nf(A) \bullet \tau nf(B)$.*

*Proof:* By induction on a derivation of $\Gamma \vdash A \bullet B$. We distinguish three cases:

*i)* $A \bullet B$ is a statement of the form $R : k$ or $\tau : T$. Equality rules may be eliminated in the $\vdash_N$ derivations because of the unicity of the $\tau nf$. Most cases follow directly by induction, while an interesting case is the $(row{-}app)$ rule. In this case the derivation has the following shape:

$$\frac{\Gamma \vdash R : T{\rightarrow}\kappa \quad \Gamma \vdash \tau : T}{\Gamma \vdash R\tau : \kappa}\ (row{-}app)$$

By induction, we get:

$$\tau nf(\Gamma) \vdash_N \tau nf(R) : \tau nf(T{\rightarrow}\kappa),$$

and

$$\tau nf(\Gamma) \vdash_N \tau nf(\tau) : \tau nf(T).$$

Apply again a $(row{-}app)$ rule to obtain:

$$\tau nf(\Gamma) \vdash_N \tau nf(R)\tau nf(\tau) : \kappa,$$

since $\tau nf(T \to \kappa) \equiv T \to \kappa$, and $\tau nf(T) \equiv T$.
Now there are two cases to consider, namely, if the $\tau nf(R)$ is an abstraction or not. In the second case the thesis follows immediately by induction, while in the first case we have $R \equiv \lambda t.R'$, for some $R'$. Then we have:

$$\tau nf(\Gamma) \vdash_N (\lambda t.\tau nf(R'))\tau nf(\tau) : \kappa.$$

Since

$$(\lambda t.\tau nf(R'))\tau nf(\tau) \to_\beta [\tau nf(\tau)/t]\tau nf(R'),$$

we can apply a $(row{-}\beta)$ rule to get:

$$\tau nf(\Gamma) \vdash_N [\tau nf(\tau)/t]\tau nf(R') : \kappa.$$

We conclude the proof by observing that:

$$[\tau nf(\tau)/t]\tau nf(R') \equiv \tau nf(R\tau),$$

since if we substitute types in normal form for type-variables, we cannot introduce new redexes. So we conclude:

$$\tau nf(\Gamma) \vdash_N \tau nf(R\tau) : \tau nf(\kappa),$$

as desired.

*ii)* $A \bullet B$ is a statement of the form $\sigma \preceq \tau$. Most cases follow from the induction hypothesis. The only interesting case is when the last rule applied is $(width_\preceq)$:

$$\frac{\Gamma \vdash_N \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \overline{\mathtt{n}}{:}\overline{\beta} \rangle\!\rangle : T \qquad \overline{\mathtt{n}} \notin \mathcal{L}(\overline{\alpha})}{\Gamma \vdash_N \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \overline{\mathtt{n}}{:}\overline{\beta} \rangle\!\rangle \preceq \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha} \rangle\!\rangle} \ (width_\preceq)$$

By part *(i)* we get:

$$\tau nf(\Gamma) \vdash_N \tau nf(\mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha} \mid \overline{\mathtt{n}}{:}\overline{\beta} \rangle\!\rangle) : \tau nf(T),$$

that is equal, by Fact *7.4.9*, to:

$$\tau nf(\Gamma) \vdash_N \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\tau nf(\overline{\alpha}) \mid \overline{\mathtt{n}}{:}\tau nf(\overline{\beta}) \rangle\!\rangle : T.$$

The hypothesis gives us $\overline{\mathtt{n}} \notin \mathcal{L}(\overline{\alpha})$, so $\overline{\mathtt{n}} \notin \mathcal{L}(\tau nf(\overline{\alpha}))$, by Fact *7.4.9*. We can apply a $(width_\preceq)$ rule to get:

$$\tau nf(\Gamma) \vdash_N \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\tau nf(\overline{\alpha}) \mid \overline{\mathtt{n}}{:}\tau nf(\overline{\beta}) \rangle\!\rangle \preceq \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\tau nf(\overline{\alpha}) \rangle\!\rangle,$$

that is, again by Fact *7.4.9*:

$$\tau nf(\Gamma) \vdash_N \tau nf(\texttt{obj}\, t.\langle\!\langle \overline{\texttt{m}}{:}\overline{\alpha} \mid \overline{\texttt{n}}{:}\overline{\beta} \rangle\!\rangle) \preceq \tau nf(\texttt{obj}\, t.\langle\!\langle \overline{\texttt{m}}{:}\overline{\alpha} \rangle\!\rangle).$$

*iii)* $A \bullet B$ is a statement of the form $\texttt{e} : \tau$. If the last applied rule is $(sub_{\preceq})$, then the thesis follows from part *(ii)*. All other cases come by straightforward induction. ■

### 7.4.3 Technical Lemmas

We are going to show some technical lemmas, necessary to prove the *subject reduction* theorem. The proofs in this subsection are extensions of the corresponding proofs in [FHM94]. They essentially say that each component of a judgment is well-formed.

The first lemma is a generation lemma for rows and object-types.

*Lemma 7.4.15* *i) If* $\Gamma \vdash_N \lambda t_1 \ldots t_p.\langle\!\langle R \mid \overline{\texttt{m}}{:}\overline{\tau}_\Delta \rangle\!\rangle : T^p \to [\overline{\texttt{n}}]$*, with* $p \geq 0$*, then* $\Gamma, t_1{:}T, \ldots, t_p{:}T \vdash_N \tau_i : T$*, for each* $\tau_i$*, and* $\Gamma, t_1{:}T, \ldots, t_p{:}T \vdash_N R : [\overline{\texttt{m}}, \overline{\texttt{n}}]$.

*ii) If* $\Gamma \vdash_N \texttt{obj}\, t.\langle\!\langle R \mid \overline{\texttt{m}}{:}\overline{\tau}_\Delta \rangle\!\rangle : T$*, then* $\Gamma, t{:}T \vdash_N \tau_i : T$*, for each* $\tau_i$*, and* $\Gamma, t{:}T \vdash_N R : [\overline{\texttt{m}}]$.

*iii) If* $\Gamma \vdash_N \sigma \to \tau : T$*, then* $\Gamma \vdash_N \sigma : T$*, and* $\Gamma \vdash_N \tau : T$. ■

*Proof:* By induction on derivations.

*i)* We show the most interesting cases, namely if the last applied rule is $(weak)$ or $(row{-}\beta)$.

[$(weak)$]: then the derivation has the following shape:

$$\frac{\Gamma_1 \vdash_N \lambda t_1 \ldots t_p.\langle\!\langle R \mid \overline{\texttt{m}}{:}\overline{\tau}_\Delta \rangle\!\rangle : T^p \to [\overline{\texttt{n}}] \qquad \Gamma_1, \Gamma_2 \vdash_N *}{\Gamma_1, \Gamma_2 \vdash_N \lambda t_1 \ldots t_p.\langle\!\langle R \mid \overline{\texttt{m}}{:}\overline{\tau}_\Delta \rangle\!\rangle : T^p \to [\overline{\texttt{n}}]} \ (weak)$$

for suitable $\Gamma_1$, and $\Gamma_2$, such that $\Gamma \equiv \Gamma_1, \Gamma_2$. By induction we get:

$$\Gamma_1, t_1{:}T, \ldots, t_p{:}T \vdash_N \tau_i : T,$$

for each $\tau_i$, and

$$\Gamma_1, t_1{:}T, \ldots, t_p{:}T \vdash_N R : [\overline{\texttt{m}}, \overline{\texttt{n}}].$$

By repeated application of Lemma *7.4.1 (iii)* we obtain the thesis.

[$(row{-}\beta)$]: then the (normal form) derivation has the following shape:

$$\frac{\begin{array}{l}\Gamma \vdash_N \tau'' : T \\ \Gamma \vdash_N \lambda t.t_1 \ldots t_p.\langle\!\langle R' \mid \overline{\mathtt{m}}{:}\overline{\tau}'_\Delta \rangle\!\rangle : T^{p+1} {\rightarrow} [\overline{\mathtt{n}}]\end{array}}{\Gamma \vdash_N (\lambda t.t_1 \ldots t_p.\langle\!\langle R' \mid \overline{\mathtt{m}}{:}\overline{\tau}'_\Delta \rangle\!\rangle)\tau'' : T^p {\rightarrow} [\overline{\mathtt{n}}]} \quad (row{-}appl)$$

$$\frac{(\lambda t.t_1 \ldots t_p.\langle\!\langle R' \mid \overline{\mathtt{m}}{:}\overline{\tau}'_\Delta \rangle\!\rangle)\tau'' \rightarrow_\beta \lambda t_1 \ldots t_p.\langle\!\langle R \mid \overline{\mathtt{m}}{:}\overline{\tau}_\Delta \rangle\!\rangle}{\Gamma \vdash_N \lambda t_1 \ldots t_p.\langle\!\langle R \mid \overline{\mathtt{m}}{:}\overline{\tau}_\Delta \rangle\!\rangle : T^p {\rightarrow} [\overline{\mathtt{n}}]} \quad (row{-}\beta)$$

We can suppose, without loss of generality, that $t_1, \ldots, t_p \notin \mathcal{FV}(\tau'') \cup \{t\}$, and this means that:

$$\lambda t_1 \ldots t_p.\langle\!\langle R \mid \overline{\mathtt{m}}{:}\overline{\tau}_\Delta \rangle\!\rangle \equiv \lambda t_1 \ldots t_p.\langle\!\langle [\tau''/t]R' \mid \overline{\mathtt{m}}{:}[\tau''/t]\overline{\tau}'_\Delta \rangle\!\rangle),$$

so giving $R \equiv [\tau''/t]R'$, and $\overline{\tau}_\Delta \equiv [\tau''/t]\overline{\tau}'_\Delta$. By induction, for each $\tau'_i$, we get:

$$\Gamma, t{:}T, t_1{:}T, \ldots, t_p{:}T \vdash_N \tau'_i : T,$$

and

$$\Gamma, t{:}T, t_1{:}T, \ldots, t_p{:}T \vdash_N R' : [\overline{\mathtt{m}}, \overline{\mathtt{n}}].$$

By Lemma *7.4.5*, for each $\tau'_i$, we get:

$$\Gamma, t_1{:}T, \ldots, t_p{:}T \vdash \tau'_i : T,$$

and

$$\Gamma, t_1{:}T, \ldots, t_p{:}T \vdash R : [\overline{\mathtt{m}}, \overline{\mathtt{n}}].$$

Since the original derivation is in $\tau nf$, so $\Gamma$, $\overline{\tau}$, and $R$ are all in $\tau nf$. We can, finally, apply Lemma *7.4.14* to get:

$$\Gamma, t_1{:}T, \ldots, t_p{:}T \vdash_N \tau_i : T,$$

and

$$\Gamma, t_1{:}T, \ldots, t_p{:}T \vdash_N R : [\overline{\mathtt{m}}, \overline{\mathtt{n}}],$$

as desired.

*ii)* The last rule cannot be $(type{-}\beta)$, since the derivation is in normal form. The case of rule (*weak*) follows by induction. The case of rule (*object*) follows directly by applying part *(i)*. The other cases are immediate.

*iii)* Easy ■

The last lemma assures us that we can deduce only well-formed object-types for objects.

*Lemma 7.4.16* *i) If* $\Gamma \vdash_N \sigma \preceq \tau$*, then* $\Gamma \vdash_N \sigma : T$ *and* $\Gamma \vdash_N \tau : T$*.*
*ii) If* $\Gamma \vdash_N \mathtt{e} : \tau$*, then* $\Gamma \vdash_N \tau : T$*.*

*Proof:* By induction on derivations.

*i)* Most cases are trivial, or come from the induction hypothesis. We consider the case in which the last applied rule is $(width_{\preceq})$. The hypothesis of the rule gives us:

$$\Gamma \vdash_N \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\alpha}, \overline{\mathtt{n}}{:}\overline{\beta} \rangle\!\rangle : T.$$

Let $\overline{\alpha} \equiv \overline{\tau}_\Delta$, for some $\overline{\tau}_\Delta$. By Lemmas *7.4.15 (ii)* and *7.4.1 (iii)*, we have that:

$$\Gamma, t{:}T \vdash_N \; * \, .$$

We can apply an $(empty{-}row)$ rule to get:

$$\Gamma, t{:}T \vdash_N \langle\!\langle\rangle\!\rangle : [\overline{\mathtt{m}}, \overline{\mathtt{n}}].$$

By Lemma *7.4.15 (ii)*, we have that:

$$\Gamma, t{:}T \vdash_N \tau_i : T,$$

for each $\tau_i$ in $\overline{\tau}_\Delta$. By applying a sequence of $(row{-}ext)$ rules to the above judgments we get:

$$\Gamma, t{:}T \vdash_N \langle\!\langle \overline{\mathtt{m}}{:}\overline{\tau}_\Delta \rangle\!\rangle : [\overline{\mathtt{n}}].$$

Finally, using the $(object)$ rule, we can conclude:

$$\Gamma \vdash_N \mathtt{obj}\, t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\tau}_\Delta \rangle\!\rangle : T.$$

*ii)* If the last applied rule is $(sub_{\preceq})$, then the thesis follows immediately from part *(i)*. We show the case when the last applied rule is $(obj{-}ext)$. Then the derivation has the following shape:

$$\frac{\begin{array}{ll} \Gamma \vdash_N \mathtt{e}_1 : \mathtt{obj}\, t.R & \Gamma, t{:}T \vdash_N R : [\mathtt{n}] \\ \{\overline{\mathtt{m}}{:}\overline{\alpha}\} \subseteq R_\Delta & r \textit{ not in } \tau \\ \multicolumn{2}{l}{\Gamma, r{:}T{\to}[\overline{\mathtt{m}}, \mathtt{n}] \vdash_N \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta \rangle\!\rangle/t](t{\to}\tau)} \end{array}}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n}{=}\mathtt{e}_2 \rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle} \; (obj{-}ext)$$

By induction on the first premise, we get:

$$\Gamma \vdash_N \mathtt{obj}\, t.R : T.$$

Applying the induction hypothesis to the last premise of the $(obj{-}ext)$ rule gives:

$$\Gamma, r{:}T{\to}[\overline{\mathtt{m}}, \mathtt{n}] \vdash_N [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta \rangle\!\rangle/t](t{\to}\tau) : T.$$

By Lemma *7.4.15 (iii)* this implies:

$$\Gamma, r{:}T \to [\overline{\mathtt{m}}, \mathtt{n}] \vdash_N \mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{m}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta \rangle\!\rangle : T.$$

By Lemma *7.4.15 (ii)*, we get:

$$\Gamma, r{:}T \to [\overline{\mathtt{m}}, \mathtt{n}], t{:}T \vdash_N \tau : T.$$

By the side condition of the $(obj{-}ext)$ rule we know that $r \notin \mathcal{FV}(\tau)$, so by applying Lemma *7.4.6*, we derive:

$$\Gamma, t{:}T \vdash_N \tau : T.$$

We finish the proof by first applying a $(row{-}ext)$ rule to the premises $\Gamma, t{:}T \vdash_N R : [\mathtt{n}]$, and $\Gamma, t{:}T \vdash_N \tau : T$, which gives:

$$\Gamma, t{:}T \vdash_N \langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle : [\,],$$

and, finally, by applying an $(object)$ rule to get:

$$\Gamma \vdash_N \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle : T,$$

as desired. ∎

## 7.5 The Subject Reduction Theorem

We are going to prove the *subject reduction* property for our calculus. The next lemma is used to prove that $\beta$-reduction preserves types.

*Lemma 7.5.1* (Subject Reduction for $\beta$) *If* $\Gamma \vdash_N \mathtt{e} : \tau$ *and* $\mathtt{e} \twoheadrightarrow_\beta \mathtt{e}'$, *then* $\Gamma \vdash_N \mathtt{e}' : \tau$.

*Proof:* By induction on the number of $\beta$-reductions steps in $\mathtt{e} \twoheadrightarrow_\beta \mathtt{e}'$. For the basic step (i.e. one $\beta$-reduction step), we proceed again by induction on the structure of terms. Let us consider the case $\mathtt{e} \equiv (\lambda x.\mathtt{e}_1)\mathtt{e}_2$, and $\mathtt{e}' \equiv [\mathtt{e}_2/x]\mathtt{e}_1$. Then, the derivation has the following shape:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\Gamma'', x{:}\sigma'' \vdash_N \mathtt{e}_1 : \tau''}{\Gamma'' \vdash_N \lambda x.\mathtt{e}_1 : \sigma''\to\tau''}\ (exp{-}abs)}{\vdots\ (weak) \text{ and } (sub_{\preceq})}}{\Gamma' \vdash_N \lambda x.\mathtt{e}_1 : \sigma'\to\tau'}\ (weak) \qquad \Gamma' \vdash_N \mathtt{e}_2 : \sigma'}{\Gamma' \vdash_N (\lambda x.\mathtt{e}_1)\mathtt{e}_2 : \tau'}\ (exp{-}appl)
$$

$$
\dfrac{\vdots\ (weak) \text{ and } (sub_{\preceq})}{\Gamma \vdash_N (\lambda x.\mathtt{e}_1)\mathtt{e}_2 : \tau}\ (weak)
$$

where $\sigma' \preceq \sigma''$, $\tau'' \preceq \tau' \preceq \tau$, and $\Gamma'' \sqsubseteq \Gamma' \sqsubseteq \Gamma$. We apply Lemma *7.4.1 (iii)* a number of times to $\Gamma'', x{:}\sigma'' \vdash_N \mathtt{e}_1 : \tau''$ getting:

$$\Gamma', x{:}\sigma'' \vdash_N \mathtt{e}_1 : \tau''.$$

The application of rule $(sub_{\preceq})$ to $\Gamma' \vdash_N \mathtt{e}_2 : \sigma'$ gives $\Gamma' \vdash_N \mathtt{e}_2 : \sigma''$. Then, by Lemma *7.4.7 (ii)*, we derive:

$$\Gamma' \vdash_N [\mathtt{e}_2/x]\mathtt{e}_1 : \tau''.$$

We apply again Lemma *7.4.1 (iii)* a number of times, and one $(sub_{\preceq})$ rule to conclude:

$$\Gamma \vdash_N [\mathtt{e}_2/x]\mathtt{e}_1 : \tau,$$

as desired. Then the general case follows by induction. The induction step for many $\beta$-reductions is obvious. ■

Now we can state the Subject Reduction theorem for $\stackrel{eval}{\to}$.

*Theorem 7.5.2* (Subject Reduction for *eval*) *If* $\Gamma \vdash_N \mathtt{e} : \sigma$ *and* $\mathtt{e} \stackrel{eval}{\twoheadrightarrow} \mathtt{e}'$, *then* $\Gamma \vdash_N \mathtt{e}' : \sigma$.

*Proof:* By induction on the number of *eval*-reductions steps in $\mathtt{e} \stackrel{eval}{\twoheadrightarrow} \mathtt{e}'$. For the basic step (i.e. one *eval*-reduction step), we proceed again by induction on the structure of terms. The proof for $\beta$-reduction follows from Lemma *7.5.1*. We show the derivation for the left-hand side of each rule (considering the most difficult cases, in which the $(sub_{\preceq})$ rule is applied after each other rule) and then we build the correct derivation for the right-hand side. We do not consider instead applications of rule $(weak)$, which can be dealt with in the standard way.

$[(\Leftarrow)\ (\mathtt{e} \Leftarrow \mathtt{n}) \overset{eval}{\to} (\mathtt{e} \hookleftarrow \mathtt{n})\mathtt{e}]$ In this case, the left-hand side is:

$$\dfrac{\dfrac{\Gamma \vdash_N \mathtt{e} : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle}{\Gamma \vdash_N \mathtt{e} \Leftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle/t]\tau}\ (send)}{\Gamma \vdash_N \mathtt{e} \Leftarrow \mathtt{n} : \sigma}\ (sub_\preceq)$$

Then we can build the derivation for the right-hand side as follows. The judgment $\Gamma \vdash_N \mathtt{e} : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle : T$ implies, by Lemma *7.4.16 (ii)*:

$$\Gamma \vdash_N \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle : T.$$

By Lemma *7.4.15 (ii)*, $\Gamma, t{:}T \vdash_N R : [\mathtt{n}]$ is derivable. Therefore, we build the following derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash_N \mathtt{e} : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle \qquad \Gamma, t{:}T \vdash_N R : [\mathtt{n}] \qquad R_\Delta \subseteq R_\Delta}{\Gamma \vdash_N \mathtt{e} \hookleftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle/t](t \to \tau)}\ (search) \quad \Gamma \vdash_N \mathtt{e} : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle}{\Gamma \vdash_N (\mathtt{e} \hookleftarrow \mathtt{n})\mathtt{e} : [\mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle/t]\tau}\ (exp{-}appl)}{\Gamma \vdash_N (\mathtt{e} \hookleftarrow \mathtt{n})\mathtt{e} : \sigma}\ (sub_\preceq)$$

$[(succ \hookleftarrow)\ \langle \mathtt{e}_1 \hookleftarrow\!\!* \ \mathtt{n} = \mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} \overset{eval}{\to} \mathtt{e}_2]$ There are two cases to consider, according to the shape of the $\hookleftarrow\!\!*$ operation.

*Case 1:* $\hookleftarrow\!\!*$ *is* $\hookleftarrow\!\!\circ$ . Then the left-hand side is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash_N \mathtt{e}_1 : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle \qquad \{\overline{\mathtt{p}}{:}\overline{\alpha}\} \subseteq R_\Delta \qquad \Gamma, r{:}T \to [\overline{\mathtt{p}}, \mathtt{n}] \vdash_N \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{p}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta \rangle\!\rangle/t](t \to \tau)}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow \mathtt{n}{=}\mathtt{e}_2 \rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle}\ (obj{-}over)}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow \mathtt{n}{=}\mathtt{e}_2 \rangle : \mathtt{obj}\, t.\langle\!\langle R' \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle}\ (sub_\preceq) \quad R'_\Delta \subseteq R''_\Delta \qquad \Gamma, t{:}T \vdash_N R'' : [\mathtt{n}]}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow \mathtt{n}{=}\mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R'' \mid \mathtt{n}{:}\tau_\Delta \rangle\!\rangle/t](t \to \tau)}\ (search)}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow \mathtt{n}{=}\mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : \sigma}\ (sub_\preceq)$$

Observe that it is not possible to forget method $\mathtt{n}$ in the first application of the $(sub_\preceq)$ rule, because afterwards we have to type its search. Moreover, it is not possible to forget any of the $\overline{\mathtt{p}}$ methods, since $\mathtt{n}$ uses them. Then $\{\overline{\mathtt{p}}{:}\overline{\alpha}\} \subseteq R''_\Delta$, which implies $R'' \equiv \langle\!\langle R''' \mid \overline{\mathtt{p}}{:}\overline{\alpha} \rangle\!\rangle$, for some $R'''$. Moreover, since

$\Gamma, t{:}T \vdash_N \langle\!\langle R''' \mid \overline{\mathtt{p}}{:}\overline{\alpha}\rangle\!\rangle : [\mathtt{n}]$, we derive $\Gamma, t{:}T \vdash_N R''' : [\overline{\mathtt{p}}, \mathtt{n}]$, by Lemma *7.4.15 (i)*. From $\Gamma, t{:}T \vdash_N R''' : [\overline{\mathtt{p}}, \mathtt{n}]$, by applying a $(row{-}abs)$ rule, we get:

$$\Gamma \vdash_N \lambda t.R''' : T \rightarrow [\overline{\mathtt{p}}, \mathtt{n}].$$

This implies, by Lemma *7.4.7 (i)*:

$$\Gamma \vdash \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle (\lambda t.R''')t \mid \overline{\mathtt{p}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta\rangle\!\rangle/t](t \rightarrow \tau).$$

So, by Lemma *7.4.14*, we can derive:

$$\Gamma \vdash_N \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle R''' \mid \overline{\mathtt{p}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta\rangle\!\rangle/t](t \rightarrow \tau).$$

Finally, we build a derivation for the right-hand side as follows:

$$\dfrac{\Gamma \vdash_N \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle R'' \mid \mathtt{n}{:}\tau_\Delta\rangle\!\rangle/t](t \rightarrow \tau)}{\Gamma \vdash_N \mathtt{e}_2 : \sigma}\ (sub_{\preceq})$$

*Case 2:* $\leftarrow\!\!*$ *is* $\leftarrow$. Then the left-hand side is:

$$\dfrac{\dfrac{\dfrac{\begin{array}{ll}\Gamma \vdash_N \mathtt{e}_1 : \mathtt{obj}\, t.R & \Gamma, t{:}T \vdash_N R : [\mathtt{n}] \\ \{\overline{\mathtt{p}}{:}\overline{\alpha}\} \subseteq R_\Delta & r \textit{ not in } \tau \\ \multicolumn{2}{l}{\Gamma, r{:}T \rightarrow [\overline{\mathtt{p}}, \mathtt{n}] \vdash_N \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{p}}{:}\overline{\alpha}, \mathtt{n}{:}\tau_\Delta\rangle\!\rangle/t](t \rightarrow \tau)}\end{array}}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n} = \mathtt{e}_2\rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{n}{:}\tau_\Delta\rangle\!\rangle}\ (obj{-}ext)}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n} = \mathtt{e}_2\rangle : \mathtt{obj}\, t.\langle\!\langle R' \mid \mathtt{n}{:}\tau_\Delta\rangle\!\rangle \quad R'_\Delta \subseteq R''_\Delta \quad \Gamma, t{:}T \vdash_N R'' : [\mathtt{n}]}\ (sub_{\preceq})}{\dfrac{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n} = \mathtt{e}_2\rangle \hookleftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R'' \mid \mathtt{n}{:}\tau_\Delta\rangle\!\rangle/t](t \rightarrow \tau)}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\, \mathtt{n} = \mathtt{e}_2\rangle \hookleftarrow \mathtt{n} : \sigma}\ (sub_{\preceq})}\ (search)$$

We apply the same arguments as in case 1 in order to obtain a correct derivation for $\mathtt{e}_2$. The $(succ \hookleftarrow)$ rule shows the usefulness of labeled-types. Thanks to the label $\Delta$ (that contains all the methods used by $\mathtt{n}$), it is possible to reconstruct the correct type of $\mathtt{e}_2$ for the right-hand side of the rule.

[$(next \hookleftarrow)$ $\langle \mathtt{e}_1 \leftarrow\!* \,\mathtt{m} = \mathtt{e}_2\rangle \hookleftarrow \mathtt{n} \stackrel{eval}{\rightarrow} \mathtt{e}_1 \hookleftarrow \mathtt{n}$] There are two cases to consider, according to the shape of the $\leftarrow\!*$ operation.

*Case 1:* $\leftarrow\!*$ *is* $\leftarrow\!\circ$ . Then the left-hand side is:

$$
\dfrac{
\dfrac{
\dfrac{
\begin{array}{ll}
\Gamma \vdash_N \mathtt{e}_1 : \mathtt{obj}\, t.R & \Gamma, t{:}T \vdash_N R : [\mathtt{m}] \\
\{\overline{\mathtt{p}}{:}\overline{\alpha}\} \subseteq R_\Delta & r \ \mathit{not\ in}\ \tau \\
\multicolumn{2}{l}{\Gamma, r{:}T \to [\overline{\mathtt{p}}, \mathtt{m}] \vdash_N \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{p}}{:}\overline{\alpha}, \mathtt{m}{:}\tau_\Delta \rangle\!\rangle / t](t \to \tau)}
\end{array}
}{
\dfrac{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\ \mathtt{m} = \mathtt{e}_2 \rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{m}{:}\tau_\Delta \rangle\!\rangle}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\ \mathtt{m} = \mathtt{e}_2 \rangle : \mathtt{obj}\, t.\langle\!\langle R' \mid \mathtt{n}{:}\rho_{\Delta'} \rangle\!\rangle}\;(sub_{\preceq})
}\;(obj{-}ext)
\quad
\begin{array}{l} R'_{\Delta'} \subseteq R''_{\Delta'} \qquad \Gamma, t{:}T \vdash_N R'' : [\mathtt{n}] \end{array}
}{
\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\ \mathtt{m} = \mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R'' \mid \mathtt{n}{:}\rho_{\Delta'} \rangle\!\rangle / t](t \to \rho)
}\;(search)
}{
\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow\!\circ\ \mathtt{m} = \mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : \sigma
}\;(sub_{\preceq})
$$

The correctness of the first $(sub_{\preceq})$ rule implies that $R \equiv \langle\!\langle R''' \mid \mathtt{n}{:}\rho_{\Delta'} \rangle\!\rangle$, for some $R'''$, such that $R'''_{\Delta'} \subseteq R'_{\Delta'}$. Then we can build the derivation for the right-hand side as follows:

$$
\dfrac{
\dfrac{
\Gamma \vdash_N \mathtt{e}_1 : \mathtt{obj}\, t.\langle\!\langle R''' \mid \mathtt{n}{:}\rho_{\Delta'} \rangle\!\rangle \qquad \Gamma, t{:}T \vdash_N R'' : [\mathtt{n}] \qquad R'''_{\Delta'} \subseteq R''_{\Delta'}
}{
\Gamma \vdash_N \mathtt{e}_1 \hookleftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R'' \mid \mathtt{n}{:}\rho_{\Delta'} \rangle\!\rangle / t](t \to \rho)
}\;(search)
}{
\Gamma \vdash_N \mathtt{e}_1 \hookleftarrow \mathtt{n} : \sigma
}\;(sub_{\preceq})
$$

*Case 2:* $\leftarrow\!*$ *is* $\leftarrow$. Then the left-hand side is:

$$
\dfrac{
\dfrac{
\dfrac{
\begin{array}{ll}
\Gamma \vdash_N \mathtt{e}_1 : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{m}{:}\tau_\Delta \rangle\!\rangle & \{\overline{\mathtt{p}}{:}\overline{\alpha}\} \subseteq R_\Delta \\
\multicolumn{2}{l}{\Gamma, r{:}T \to [\overline{\mathtt{p}}, \mathtt{m}] \vdash_N \mathtt{e}_2 : [\mathtt{obj}\, t.\langle\!\langle rt \mid \overline{\mathtt{p}}{:}\overline{\alpha}, \mathtt{m}{:}\tau_\Delta \rangle\!\rangle / t](t \to \tau)}
\end{array}
}{
\dfrac{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2 \rangle : \mathtt{obj}\, t.\langle\!\langle R \mid \mathtt{m}{:}\tau_\Delta \rangle\!\rangle}{\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2 \rangle : \mathtt{obj}\, t.\langle\!\langle R' \mid \mathtt{n}{:}\rho_{\Delta'} \rangle\!\rangle}\;(sub_{\preceq})
}\;(obj{-}over)
\quad
\begin{array}{l} R'_{\Delta'} \subseteq R''_{\Delta'} \qquad \Gamma, t{:}T \vdash_N R'' : [\mathtt{n}] \end{array}
}{
\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : [\mathtt{obj}\, t.\langle\!\langle R'' \mid \mathtt{n}{:}\rho_{\Delta'} \rangle\!\rangle / t](t \to \rho)
}\;(search)
}{
\Gamma \vdash_N \langle \mathtt{e}_1 \leftarrow \mathtt{m} = \mathtt{e}_2 \rangle \hookleftarrow \mathtt{n} : \sigma
}\;(sub_{\preceq})
$$

Then we apply the same arguments as in case 1 in order to obtain a correct derivation for $\mathtt{e}_1 \hookleftarrow \mathtt{n}$.

[*Other evaluation rules*] Clearly, the left-hand sides of all other evaluation rules cannot be typed.

The general case follows by induction. The induction step for many $\overset{eval}{\to}$-reductions is obvious. ■

The subject reduction proof shows the power of the typing system. Labeled-types not only allow a restricted form of subtyping that enriches the set of types of typable objects, but they also lead to a simpler and more natural operational semantics, in which no transformations on the objects are necessary to get the body of a message. In fact, the typing rule for the $\hookleftarrow$ operation is strictly based on the information given by the labels. Moreover, since an $\stackrel{eval}{\to}$ produces the object $\mathtt{err}$ when a message $\mathtt{m}$ is sent to an expression which does not define an object with a method $\mathtt{m}$, the type-soundness follows from Theorem *7.5.2*.

*Theorem 7.5.3* (Type-Soundness) *If* $\varepsilon \vdash_N \mathtt{e} : \tau$ *is derivable for some* $\tau$*, then the evaluation of* $\mathtt{e}$ *cannot produce* $\mathtt{err}$*, i.e.* $\mathtt{e} \stackrel{eval}{\not\to} \mathtt{err}$.

Notice that in [FHM94] the type soundness was proved by introducing a structural operational semantics and showing suitable properties.

## 7.6 Related Papers and Future Work

In a recent paper [FM95], Fisher and Mitchell propose a very interesting solution for adding subtyping to $\lambda_{obj}$. The novelty is the introduction of a new type, the *protype*, denoted by $\mathtt{pro}\, t.R$, which can be intended as the $\mathtt{class}\, t.R$ type in [FHM94]. If we can assign a prototype to an object, then we can add new methods or override existing ones. At this level, only trivial subtyping is possible, as in was showed in [FM94], and in [AC94]. Then we can "convert" the object into a different kind of object where methods cannot be altered, i.e. the only operation on objects is message send, by "sealing" a prototype into a real object-type. Even if from the outside of the object the only operation is message send, the internal methods can override other methods of their host object. Preventing from the outside extension and override allows "width" and "depth" subtyping, provided that the bound type-variable of an object-type do not occur in contra-variant position. This solution gives profound knowledge about how classes can be understood and implemented: however, the full power of the delegation-based philosophy seems to be lost. In particular, the $\mathtt{draw} \Leftarrow \mathtt{plot}$ of Example *7.2.9* can no longer accept a coloured point as input. We briefly explain why. To simplify, we consider one dimension points and the following body for $\mathtt{plot}$:

$$\mathtt{e} \stackrel{def}{=} \lambda self.\lambda p.\langle \mathtt{p} \leftarrow\!\circ\; \lambda self.white \rangle.$$

In [FM95] we can derive:

$$\varepsilon \vdash \mathtt{e} : \sigma \rightarrow \mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int \rangle\!\rangle \rightarrow \mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{col}{:}Col \rangle\!\rangle,$$

for all types $\sigma$. Notice that we cannot derive:

$$\varepsilon \vdash \mathtt{e} : \mu \rightarrow \mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int \rangle\!\rangle \rightarrow \nu,$$

for any $\mu, \nu$, since e adds a method to its second argument. Moreover if we define:

$$\mathtt{draw} \stackrel{def}{=} \langle \mathtt{plot} = \mathtt{e} \rangle,$$

we can derive:

$$\varepsilon \vdash \mathtt{draw} : \mathtt{pro}\, t'.\langle\!\langle \mathtt{plot} = \mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int \rangle\!\rangle \rightarrow \mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{col}{:}Col \rangle\!\rangle \rangle\!\rangle.$$

this implies that

$$\varepsilon \vdash \mathtt{draw} \Leftarrow \mathtt{plot} : \mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int \rangle\!\rangle \rightarrow \mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{col}{:}Col \rangle\!\rangle.$$

Now if cp is a colored point, then it has the pro-type $\mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int, \mathtt{col}{:}Col \rangle\!\rangle$, which can be "seal" obtaining the object-type $\mathtt{obj}\, t.\langle\!\langle \mathtt{x}{:}int \rangle\!\rangle$. But we cannot derive the type $\mathtt{pro}\, t.\langle\!\langle \mathtt{x}{:}int \rangle\!\rangle$ for cp. The above discussion shows that we cannot type the application $(\mathtt{draw} \Leftarrow \mathtt{plot})\ \mathtt{cp}$.

Another interesting solution, orthogonal to the previous one, is the one of Abadi and Cardelli [AC94] (see Section *5.3*): in their Calculus of Primitive Objects, the only operations allowed are method override and message send. In fact, this calculus is based on fixed-size objects, instead of open-ended extensible objects. This calculus features "width" subtyping. Our solution of using labeled-types lives in-between those two solutions.

We can summarize the different solutions as follows:

*i)* $\lambda_{obj}$ of [FHM94] allows extension and override.

*ii)* The Calculus of Primitive Objects of [AC94] allows override and "width"subtyping".

*iii)* The $\lambda_{obj}^{\preceq}$ of [BL95] allows extension, override, and a restricted form of "width" subtyping, provided that the forgotten methods are not referred to by the remaining ones.

*iv)* The objects of [FM95] support "internal" override, "width", and "depth" subtyping, provided that the bound type-variable of an object-type do not occurs in contra-variant position.

Among the further possible researches about the Lambda Calculus of Objects, we briefly recall:

*i)* The addition of suitable mechanism for modeling "encapsulation", along the lines showed in Example *7.2.10*.

*ii)* The definition of a fully-typed version of $\lambda_{obj}^{\preceq}$. We studied an explicitly typed version of the Lambda Calculus of Objects of [FHM94], which is a development of the object-calculi defined in [Mit90, BL95]. This calculus supports *object extension* in presence of *object subsumption.* Extension is the ability of modify the behavior of an object by adding new methods (and inheriting the existing ones). Object subsumption allows to use objects with a bigger interface in a context expecting another object with a smaller interface. This calculus has a sound and decidable type system, which allows for static detection of run-time errors such as *message-not-understood*,

However, we conjecture that the type systems of [FHM94, BL95] are undecidable. More precisely, recall that those type systems are defined for an untyped calculus. Therefore, in order to use those type systems in practice, it is necessary to have a type assignment algorithm that returns the principal type (supposing that a notion of principality can be defined) of every typable untyped term. Hitherto, no such an algorithm exists. Furthermore, by our experience, we believe that this algorithm cannot exist, the reason being the impossibility of assigning the right type to *self*, whose semantics, as clearly stated in [PJ95], is context dependent.

For these reasons in [LC96, Liq95] we define an explicitly typed version of $\lambda_{obj}^{\preceq}$, that we call $\lambda_{objT}^{\preceq}$. We define a type system for $\lambda_{objT}^{\preceq}$, we prove that it statically ensures type safety of well-typed programs and we define a type-checking algorithm that is sound and complete with respect to the type system, and terminating (which implies the decidability of the type system). What we obtain, then, is the first known-to-be decidable type system for an object calculus that supports object extension. The introduction of explicit universal quantification (in contrast with the implicit universal quantification of [FHM94, BL95]) allows to write *first-order* method bodies that can be passed as function arguments. This increase the expressiveness of the language, since it allows to write "portable methods".

Then we studied the relation between $\lambda^{\preceq}_{obj}$ and $\lambda^{\preceq}_{objT}$: we show that $\lambda^{\preceq}_{objT}$ is at least as expressive as $\lambda^{\preceq}_{obj}$ (since there exists a *type erasing function* from typed to untyped terms, such that every untyped expression typeable in $\lambda^{\preceq}_{obj}$ is the erasure of some typed expression typeable in $\lambda^{\preceq}_{objT}$) and that $\lambda^{\preceq}_{obj}$ can be used as a target language to compile $\lambda^{\preceq}_{objT}$ (since the computation of the type-erasure of a term $\mathsf{e}$ of $\lambda^{\preceq}_{objT}$, returns the erasure of a reductum of $\mathsf{e}$).

From a pragmatic point of view the situation can be described as follows. If the programmer writes an untyped program in $\lambda^{\preceq}_{obj}$, then we cannot define a type-checking algorithm that can statically ensure type safety of the program. Therefore, we ask the programmer to add some "type annotations" to his program. Thanks to these annotations it is possible to type check the program by using the algorithm we describe in this paper. Once that the safety of the program is statically ensured, type annotations can be erased and the execution can be computed on the untyped term: a theorem ensures that each step of the untyped computation is the erasure of the typed computation.

*iii)* The work of [LC96] has however some deeper motivations. The calculus described here stems from the research done by the authors on the addition of multi-methods to object-based languages. Multi-methods are methods whose code is selected according not only to the class of the receiver but also to the class of possible further arguments. In [Cas94], one of the authors showed that multi-methods can be used to overcome the longstanding problem of specializing binary methods (see [BCG$^+$95]). According to [Cas94] this can be obtained by adding special overloaded functions that use a late binding discipline. An overloaded function is a function that selects some code according to the type of its argument. Therefore, in order to execute an overloaded function one needs to compute the type of its argument. Since this was not possible in $\lambda^{\preceq}_{obj}$, we defined $\lambda^{\preceq}_{objT}$, the calculus we describe in this paper. The extension of $\lambda^{\preceq}_{objT}$ with multi-methods is described in [CL96].

*iv)* We want to extend the Lambda Calculus of Objects with a new support for *incomplete* objects [BBL96]. Incomplete objects behave operationally as "standard" objects;

their typing, instead, is different, as they may be typed even though they contain references to methods that are yet to be added. As a byproduct, incomplete objects may be typed independently of the order of their methods and, consequently, the operational semantics of the untyped calculus may be soundly defined relying on a permutation rule that treats objects as sets of methods.

This extension is based on a new encoding of the types of objects that allows us to treat objects as *sets of methods* (as opposed to ordered sequences) and, consequently, to rely on a simpler operational semantics. The new encoding also gives additional flexibility to the type system, by allowing a method invocation to be typed correctly even though the receiver of the message is an *incomplete* object, i.e. an object that contains references to methods that are yet to be added. This flexibility appears to be desirable for prototyping languages, such as delegation-based languages, where prototypes may reasonably be defined, and operated with as well, while part of their implementation (i.e. their methods) are yet to be defined.

The new type system should be a conservative extension of the system of [FHM94] that retains the *mytype* specialization property for inherited methods peculiar to [FHM94], as well as the ability to statically detect run-time errors such as *message not understood.*

*v)* We also plan to study [BDL+96] a "linear logic" programming language, called $O_{\multimap}$, that gives a complete account of an object-based calculus with inheritance and override.

The logical foundations of $O_{\multimap}$ should lie in a fragment of intuitionistic higher-order Linear Logic consisting of the connectives $\multimap$, $\Rightarrow$, $\&$, and $\forall$. The higher-order nature of the syntax allows a direct encoding of objects as first-class values with attribute and method fields: the linear connectives, on the other hand, allow methods to be selected and applied according to the self-application semantics of method invocation. $O_{\multimap}$ also provides primitives for method addition and redefinition, peculiar to the companion functional calculi, and it encompasses the same mechanism of method and attribute inheritance by delegation. Method redefinition relies on a functional form of update whereby overriding a method (or attribute) is realized by computing a modified copy of the object containing that method.

The design of $O_{\multimap}$ shares ideas with previous proposals of linear object-oriented languages [AP91, DM95, HM94, KY94], but the resulting language exhibits several novel features. As in in these languages, computation in $O_{\multimap}$ is a process of logical deduction in which, however, objects are computed values rather than consumable resources. Accordingly, the result of a program is a set of answer substitutions as it is customary in logic programming. In fact, $O_{\multimap}$ can readily be extended to allow writing program clauses (actually meta-programs with respect to objects) in the usual logic programming style, using predicates which have objects as data.

*vi)* We finally plan to investigate in [Liq96] an extension of the Calculus of Primitive Objects of [AC94] that supports *method extension* in presence of *object subsumption*. The challenge is to provide, for this extended calculus, a sound type system which allows for static detection of run-time errors such as *message-not-understood*, *width* subtyping and a typed equational theory on objects.

# Bibliography


[AC94] M. Abadi and L. Cardelli. A Theory of Primitive Objects. In *Proceedings of Theoretical Aspect of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, 1994.

[AP91] J. M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. *New Generation Computing*, 9:445–473, 1991.

[Bar91] H. Barendregt. Introduction to Generalised Type Sytems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, Dov.M. Gabbai, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 118–310. Oxford University Press, 1992.

[BBL96] V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. *Proc. International Conference of MFCS-96, Mathematical Foundation of Computer Science, 1996*, *Lecture Notes in Computer Science* Springer-Verlag, 1996.

[BCD83] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Cmpleteness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[BCG+95] K. Bruce, L. Cardelli, Castagna G., The Hopkins Object Group, G. T. Leavens, and B Pierce. On Binary Methods. *TODS*, 1995. To appear.

[BDL+96] M. Bugliesi, G.Delzanno, L. Liquori, and M. Martelli. A Linear Logic Calculus of Objects. *Proc. of JICSLP-96, Joint International Conference and Symposium on Logic Programming*, The MIT Press, 1996.

[Bel94] G. Bellè. Some Remarks on Lambda Calculus of Objects. Technical report, University of Udine, 1994.

[Ber88] S. Berardi. Towards a Mathematical Analysis of Type Dependence in Coquand–Huet Calculus of Constructions and the Other Systems in Barendregt's Cube. Technical report, Department of Computer Science, CMU, and Dipartimento di Matematica, Torino, 1988.

[BI82] A.H. Borning and D.H. Ingalls. A Type Declaration and Inference System for Smalltalk. In *Proc. of the 9th ACM Symposium on Principles of Programming Languages*, pages 133–141. The ACM Press, 1982.

[BL95] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In L. Pacholsky and J. Tiuryn, editors, *Proceedings of International Conference of Computer Science Logic 1994*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1995.

[BY81] C.B. Ben-Yelles. $g$-stratification is Equivalent to $f$-stratification. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, bd.27:141–150, 1981.

[Car95] L. Cardelli. Type System. Personal Notes, August 1995.

[Cas94] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[CC90] F. Cardone and M. Coppo. Type Inference with Recursive Types: Syntax and Semantics. *Information and Computation*, 1990.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, Amsterdam, 1958.

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A Calculus for Overloaded Functions with Subtyping. *Information and Computation*, 117:115–135, 1995.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[CHS72] H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic*, volume 2. North-Holland, Amsterdam, 1972.

[Chu41] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1941.

[CL96] G. Castagna and L. Liquori. Multi-methods in delegation-based object-oriented languages. Manuscript, 1996.

[CM91] L. Cardelli and J. C. Mitchell. Operations on Records. *Mathematical Structures in Computer Science*, 1(1):3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[Com94] A. Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, Department of Computer Science, University of Nijmegen, December 1994.

[Con86] Robert L. Constable, *et. al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice–Hall, Englewood Cliffs, NJ, 1986.

[Coq91] T. Coquand. Metamathematical Investigations of a Calculus of Constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic press, 1991.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for Self, a Dinamically-typed Object-Oriented Programming Language. In *SIGPLAN-89 Conference on Programming Language Design and Implementation*, pages 146–160, 1989.

[Cur34] H.B. Curry. Functionality in Combinatory Logic. In *Proc. Nat. Acad. Sci. U.S.A.*, volume 20, pages 584–590, 1934.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Survey*, 17(4):471–522, December 1985.

[dB80] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[DM95] G. Delzanno and M. Martelli. Objects in Forum. In *Proceedins of the International Logic Programming Symposium, Portland, Oregon*, pages 115–129. The MIT Press, 1995.

[Dow93] Gilles Dowek. The Undecidability of Typability in the Lambda-Pi-Calculus. In *Proceedings of TLCA '93 International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 139–145. Springer-Verlag, 1993.

[ES90] E. Ellis and B. Stroustrop. *The Annotated $C^{++}$ Reference Manual*. The ACM Press, 1990.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[FM94] K. Fisher and J. C. Michell. Notes on Typed Object-Oriented Programming. In *Proceedings of Theoretical Aspect of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 844–885. Springer-Verlag, 1994.

[FM95] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proceedings of FCT-95*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[Geu93] Herman Geuvers. *Logics and Type Systems*. PhD thesis, Department of Mathematics and Computer Science, University of Nijmegen, 1993.

[GHR93] P. Giannini, F. Honsell, and S. Ronchi della Rocca. Type inference: some results, some problems. *Fundamenta Informaticae*, 19((1,2)):87–126, 1993.

[Gir72] J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures dans l'aritmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[Gir86] J.Y. Girard. The System F of Variable Types, Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[GN91] H. Geuvers and M.J. Nederhof. A modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, April 1991.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1983.

[GR88] P. Giannini and S. Ronchi della Rocca. Characterization of Typings in Polymorphic Type Discipline. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 61–70, 1988.

[GW94] H. Geuvers and B. Werner. On the Church-Rosser Property for Expressive Type Systems and its Consequences for their Metatheoretic Study. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science*, pages 320–329, 1994.

[HHP92] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1992. Preliminary version in LICS'87.

[How80] W. A. Howard. The Formulae-as-Types Notion of Construction. In J. P. Seldin and J. R. Hyndley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[HM94] J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, pages 110(2):327–365, 1994.

[KY94] N. Kobayashi and A. Yonezawa. Type-Theoretic Foundations for Concurrent Object-Oriented Programming. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '94)*, 1994.

[Klo80] W. Klop, J. *Combinatory Reduction Systems.* PhD thesis, Department of Computer Science, Rijksuniversiteit Utrecht, 1980.

[Lei83] D. Leivant. Polymorphic Type Inference. In *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, pages 88–98. The ACM Press, 1983.

[Liq95] L. Liquori. A Typed Axiomatic Object Calculus with Subtyping. Presented to Advances in Type Systems for Computing, Newton Institute, Cambridge. Also avalaible as Technical Report, University of Turin, August 1995.

[Liq96] L. Liquori. An Extended Theory of Primitive Objects. Technical Report CS-23-96, Computer Science Department, Turin University, Italy, 1996.

[LC96] L. Liquori, and B. Castagna. A Typed Lambda Calculus of Objects. Proc. of *Int. Conf. ASIAN-96, Asian Computing Science Conference. Lecture Notes in Computer Science*, Springer-Verlag.

[LSS77] B. Liskov, A. Snyder, and C. Shaffert. Abstraction Mechanism in Clu. *Communications of the ACM*, (20):564–576, 1977.

[Luo90] Z. Luo. ECC: an extended calculus of constructions. In *Proceedings of the Fourth Annual Conference on Logic in Computer Science*, pages 385–395. IEEE, 1990.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1984.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit90] J. C. Mitchell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proc. of POPL*, pages 109–124. The ACM Press, 1990.

[MMM91] C. Mitchell, J., S. Meldav, and N Madhav. An Extension of Standard ML with Subtyping and Inheritance. In *Proc. of the 18th ACM Symposium on Principles of Programming Languages*, pages 270–278. The ACM Press, 1991.

[MP88] C. Mitchell, J. and G. Plotkin, G. Abstract Data Types Habe Existential Types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[PJ95] J. Palsberg and T. Jim. Type Inference for Simple Object Types is NP-Complete. Manuscript, 1995.

[PM89] C. Paulin-Mohring. Extracting $F_\omega$'s programs from proofs in the Calculus of Constructions. In *Proc. of the 6th ACM Symposium on Principles of Programming Languages*. The ACM Press, 1989.

[Pra65] D. Prawitz. *Natural Deduction*. Almqwist and Wiksell, Stockolm, 1965.

[PT93] C. Pierce, B. and N. Turner, D. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edimburgh, 1993. Also avalaible as INRIA-Roquencourt, Rapport de Recherche No. 1899.

[PT94] C. Pierce, B. and N. Turner, D. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[Rey74] J.C. Reynolds. Towards a Theory of Type Structures. In B. Robinet, editor, *Proceedings of Programming Symposium, Paris*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[Rey83] J.C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*, pages 513–523. North-Holland, 1983.

[Sal90] A. Salvesen. The Church-Rosser theorem for LF with beta/eta reduction. Talk given at First Workshop on Logical Frameworks, Antibes, 1990.

[Sel79] P. Seldin, J. Progress Report on Generalised Functionality. *Annals of Mathematical Logic*, 17:29–59, 1979.

[Tak89] M. Takahashi. Parallel reductions in $\lambda$-calculus. *Information and Computation*, 7:113–123, 1989.

[Tes85] L. Tesler. *Object Pascal Report.* Apple Computer, 1985.

[US87] D. Ungar and B. Smith, R. Self: The Power of Simplicity. In *Proceeding ACM Symposium in Object-Oriented Programming:Systems, Languages, and Applications*, pages 227–241. The ACM Press, 1987.

[vBJ93] L.S. van Benthem Jutting. Typing in Pure Type Sytems. *Information and Computation*, 105(1):30–41, July 1993.

[vBLRU94] S. van Bakel, L. Liquori, S. Ronchi della Rocca, and P. Urzyczyn. Comparing Cubes. In Nerode A. and Matiyasevich Yu.V., editors, *Proceedings of Third International Symposium on Logical Foundations of Computer Science*, volume 813 of *Lecture Notes in Computer Science*, pages 353–365. Springer-Verlag, 1994.

[vBLRU95] S. van Bakel, L. Liquori, S. Ronchi della Rocca, and P. Urzyczyn. Comparing Cubes of Typed and Type Assignment System. Annals of Pure and Applied Logics, 199?.