Università degli Studi di Firenze

Facoltà di Scienze Matematiche, Fisiche e Naturali

Type Reconstruction

Corso di Fondamenti dei Linguaggi di Programmazione

A cura di

Jacopo Santoni

Docente

Battistina Venneri

Anno Accademico 2009/2010

1 Introduzione

In generale, nel cercare di dimostrare proprietà di *type safety* per i programmi, possiamo effettuare una scelta di livello strutturale che modifica come il programmatore si approcca all'utilizzo del linguaggio e come il *type checker* esegue i suoi controlli, possiamo:

- imporre al programmatore di definire i tipi dei termini del linguaggio nel programma;
- utilizzare, viceversa, un algoritmo di *type reconstruction* in grado di decidere da solo il tipo dei termini a partire dall'utilizzo che ne viene fatto nel programma.

Mentre il primo metodo richiede lo sforzo al programmatore di annotare i tipi nelle varie parti del programma, in modo che il compilatore sia in grado di verificare, partendo da questi, che i termini siano sempre ben tipati; il secondo metodo permette libertà totale nel senso che chi scrive il programma non è costretto a comunicare al compilatore i tipi degli oggetti utilizzati, ma sarà il compilatore che cercherà di ricostruirli valutando l'ambito in cui vengono utilizzati per garantire la *type safety*, in questo caso, infatti, parliamo proprio di *type reconstruction* o *type inference* in quanto il compilatore stesso cerca di inferire i tipi basandosi sul contesto in cui le variabili ed i metodi vengono definiti ed utilizzati.

In [1] viene descritto ed analizzato un algoritmo di *type reconstruction* in grado di inferire il *tipo principale* per un termine nel quale nessuna o solo alcune annotazioni sono state specificate.

La trattazione è suddivisa in due parti, la prima descrive il sistema di tipi nel quale l'algoritmo dovrà operare, mentre la seconda descrive l'elegante risoluzione alla ricostruzione dei tipi attraverso un algoritmo che genera i vincoli intrinsecamente definiti dai termini del programma e li risolve attraverso un algoritmo di unificazione.

È interessante riassumere la storia di questa famiglia di algoritmi che nasce da **Haskell Curry** e **Robert Feys** nel 1958 con un'implementazione per il $\lambda-$calcolo con tipi semplici. Successivamente il lavoro è stato esteso e portato avanti da **Hindley** e **Miller** in [2]. Negli anni sono state dimostrate ulteriori proprietà ed estensioni, fino ad arrivare anche all'idea di **Pierce** descritta in [1], che verrà analizzata nel documento.

Questo algoritmo è alla base di molti linguaggi, tradizionalmente utlizzato nei linguaggi funzionali, ad esempio in **Haskell** o **ML**.

2 Variabili di tipo e Sostituzioni

2.1 Scelta del sistema di tipi

Scegliamo un sistema di tipi semplici contenente un insieme infinito di tipi primitivi ed i tipi freccia. Ulteriori estensioni sono ovviamente definibili anche se bisogna effettuare scelte accurate data la complessità di alcuni passaggi.

Abbiamo quindi un insieme di tipi della forma:

$$T = T_1 \cup T_2$$

con

$$T_1 = \{\text{Bool}, \text{Nat}, \cdots\} \qquad T_2 = \{X_1 \to X_2 : X_1, X_2 \in T_1 \cup T_2\}$$

Non ci interessa quali siano gli effettivi tipi contenuti nell'insieme T_1, l'unica cosa necessaria è che siano tipi primitivi ma dal punto di vista della *type reconstruction* sono tutti uguali.

A partire da questo sistema di tipi siamo interessati a poter escludere le annotazioni degli stessi all'interno dei termini per i quali il *type reconstructor* dovrà operare, è necessario introdurre un concetto che permetta di definire che un termine ha un tipo senza esplicitamente dire quale esso sia.

Definiamo quindi la variabile di tipo come un *placeholder*, ovvero una variabile di cui non conosciamo il tipo specifico ma che utilizziamo all'interno dei termini; chiaramente da sola non può darci alcuna informazione aggiuntiva ma possiamo supporre ad esempio di avere un termine t contenente la variabile di tipo X e ci domandiamo se sostituendo la variabile di tipo ad un tipo primitivo (es. Nat) otteniamo un termine tipabile.

Conviene inoltre separare concettualmente il *mapping* che associa ad una variabile di tipo un tipo specifico dall'applicazione della stessa ad un termine che a tutti gli effetti sostituisce le variabili di tipo contenute in esso con i tipi definiti dal *mapping*.

2.2 Definizione formale

Definizione Sia σ una funzione che associa variabili di tipo a tipi. Definiamo σ come funzione di *type substitution*, definiamo come $\text{dom}(\sigma)$ (dominio) l'insieme delle variabili di tipo che compaiono a sinistra nel mapping e come $\text{range}(\sigma)$ l'insieme dei termini che compaiono a destra.

Ad esempio possiamo definire $\sigma = [X \mapsto T, Y \mapsto U]$ per la funzione che associa alla variabile di tipo X il tipo T e alla variabile Y il tipo U; $\text{dom}(\sigma) = \{X, Y\}$ e $\text{range}(\sigma) = \{T, U\}$.

Osservazione Si noti che una variabile di tipo può appartenere sia al dominio di σ sia al suo codominio.

Consideriamo l'applicazione della *type substitution* come un'operazione che avviene applicando ogni singola clausola contemporaneamente, quindi ad esempio:

$$\sigma = [X \mapsto \text{Bool}, Y \mapsto X \to X]$$

tiperà Y proprio come $X \to X$, non come $\text{Bool} \to \text{Bool}$.

Definizione Definiamo l'*applicazione* della *type substitution* nel seguente modo:

$$\sigma(X) = \begin{cases} T & \text{if } (X \mapsto T) \in \sigma \\ X & \text{if } X \notin \text{dom}(\sigma) \end{cases}$$

$$\sigma(\text{Nat}) = \text{Nat}$$

$$\sigma(\text{Bool}) = \text{Bool}$$

$$\sigma(T_1 \to T_2) = \sigma T_1 \to \sigma T_2$$

Quindi, se l'associazione tra X e T è presente in σ procediamo con la sostituzione, altrimenti viene lasciata invariata. La sostituzione di un *tipo freccia* avviene sostituendo i singoli termini a sinistra e destra, infine la sostituzione applicata ai tipi primitivi li lascia invariati.

Possiamo facilmente estendere la definizione anche ai contesti:

$$\sigma(x_1 : T_1, x_2 : T_2, \cdots, x_n : T_n) = (x_1 : \sigma T_1, x_2 : \sigma T_2, \cdots, x_n : \sigma T_n)$$

Definiamo infine la sostituzione di un termine t come l'applicazione di σ a tutti i tipi definiti nel termine stesso.

Definizione Siano σ e γ due sostituzioni di tipo. Definiamo la loro composizione $\sigma \circ \gamma$ come:

$$\sigma \circ \gamma = \begin{cases} X \mapsto \sigma(T) & \forall (X \mapsto T) \in \gamma \\ X \mapsto T & \forall (X \mapsto T) \in \sigma \text{ con } X \notin \text{dom}(\gamma) \end{cases}$$

Si noti che la composizione di due sostituzioni è considerabile come una normale composizione di funzioni e quindi che

$$(\sigma \circ \gamma)S = \sigma(\gamma S)$$

È necessario dimostrare che le sostituzioni sono operazioni trasparenti per quanto riguarda la tipabilità dei termini, altrimenti il loro utilizzo non sarebbe valido.

Dobbiamo quindi dimostrare che dato un termine t ben tipato, l'applicazione di una sostituzione al termine restituisce allo stesso modo un termine ben tipato: quindi il *typing* viene preservato dalla sostituzione.

Teorema 2.1. *Preservation of typing under type substitution*
Sia Γ un contesto, sia σ una funzione di sostituzione di tipo, sia t un termine tale che

$$\Gamma \vdash t : T$$

allora

$$\sigma\Gamma \vdash \sigma t : \sigma T$$

Dimostrazione. La dimostrazione procede per induzione sulle derivazioni di tipo. Dobbiamo dimostrare la validità per ogni regola di *typing* assumendo che valga induttivamente per le sotto-derivazioni.

Mostriamola per qualche regola, le altre si dimostrano in maniera del tutto simile.

T-VAR

$$\frac{x : X \in \Gamma}{\Gamma \vdash x : X}$$

dove X potrà essere

1. una variabile di tipo, per la quale la sostituzione si applica sia al termine che al contesto entro il quale stiamo facendo le nostre assuzioni, mantenendo la validità del *typing*;

2. un tipo primitivo, sul quale la sostituzione non ha effetto;

3. un tipo freccia, per il quale la sostituzione viene applicata su entrambi i sottotipi per i quali abbiamo le stesse tre stesse tre alternative che stiamo descrivendo.

T-ABS

$$\frac{\Gamma, x : X \vdash y : Y}{\Gamma \vdash \lambda x : X.y : X \to Y}$$

Come nel caso precedente abbiamo che una qualsiasi sostituzione che contiene un *mapping* per X o Y viene applicata sia al contesto assunto $\Gamma, x : X$ che sui termini coinvolti. Dato inoltre che l'applicazione della sostituzione ad un tipo freccia sostituisce indipendentemente entrambi i tipi abbiamo $\sigma\ X \to Y = \sigma X \to \sigma Y$, dopo la sostituzione dei tipi nel contesto il nuovo λ-termine avra un *typing* corretto. □

3 Approcci alle variabili di tipo

Nell'ambito dell'inferenza sui tipi ci sono due approcci diversi che corrispondo a due diversi tipi di problemi nei quali possiamo incorrere e che sono interessanti da analizzare.

Sia t un termine che contiene variabili di tipo e sia Γ un contesto (contenente, auspicabilmente, variabili di tipo a sua volta), possiamo domandarci:

- Qualsiasi sostituzione di t rende il termine tipato?
  Ovvero $\forall\sigma$ abbiamo $\sigma\Gamma \vdash \sigma t : T$ per un qualche t

- Esiste una o più sostituzioni di t che lo rendono ben tipato?
  Ovvero $\exists\sigma$ t.c. $\sigma\Gamma \vdash \sigma t : T$ per un qualche t

Le due differenti situazioni sono concettualmente diverse, nel primo caso ci chiediamo se il termine t è tipabile a prescindere dal tipo specifico che possono assumere le sue componenti; mentre nel secondo vengono utilizzate delle

sostituzioni *ad hoc* che rendono tipabile il termine ma solo nei casi previsti, magari con un termine non sarebbe tipabile sotto qualsiasi condizione.

Nello sviluppo di un algoritmo di *type inference* è chiaro che scegliere un approccio piuttosto che un altro porta a considerazioni diverse: seguendo il primo approccio notiamo come le variabili di tipo debbano rimanere astratte durante il *type checking*, non ci dev'essere bisogno di una sostituzione in quanto vogliamo verificare che il termine sia tipabile a prescindere dalla sostituzione (di fatto per ogni sostituzione), l'approccio sfocia nel *polimorfismo parametrico* in cui controlliamo la *type safety* di termini senza preoccuparci dei tipi effettivi che entreranno in gioco al momento dell'esecuzione del codice parametrico.

Un esempio di quanto detto in relazione al fatto di mantenere astratte le variabili di tipo è il seguente:

$$\lambda f : X \to X.\big(\lambda a : X.f(f\ a)\big)$$

Il tipo del termine è $(X \to X) \to (X \to X)$ e una qualsiasi sostituzione che rimpiazza X con un tipo concreto T trasformerà il termine:

$$\lambda f : T \to T.\big(\lambda a : T.f(f\ a)\big)$$

Il nuovo termine ottenuto è ben tipato a sua volta.

Per quanto riguarda il secondo approccio non ci interessa il tipo specifico che può assumere il termine t ma se è possibile istanziarlo in un termine ben tipato scegliendo precisamente le sue variabili di tipo. Ad esempio il termine

$$\lambda f : Y.\lambda a : X.f(fa)$$

non è direttamente tipabile ma se sostituiamo Y con $X \to X$ otteniamo il termine

$$\lambda f : X \to X.\lambda a : X.f(fa)$$

che è ben tipato nonostante contenga variabili. Nello specifico il termine è l'*istanza più generale* del termine precedente che assume il meno possibile ma lo rende allo stesso tempo ben tipato.

Cercare istanze valide per le variabili di tipo di un termine ci porta al problema generale definito come *type inference* o *type reconstruction* in cui facciamo sì che il compilatore si occupi di recuperare i tipi delle variabili utilizzate nel programma lasciando al programmatore la libertà di non specificare parzialmente o totalmente i tipi.

Definizione (soluzione)

Sia t un termine e Γ un contesto. Una *soluzione* di (Γ, t) è una coppia (σ, T) tale che

$$\sigma\Gamma \vdash \sigma t : T$$

Esempio

Supponiamo di avere un contesto $\Gamma = f : X, a : Y$ ed il termine $t = f\ a$. Le seguenti coppie sono soluzioni di (Γ, t):

- $([X \longmapsto Y \to Nat], Nat)$: il tipo di f è un tipo freccia da Y a Nat, Y è il tipo del "parametro" quindi il tipo di $f\ a$ è Nat.
- $([X \longmapsto Y \to Z], Z)$: il tipo di f è un tipo freccia da Y a Z, Y è il tipo del "parametro" dell'applicazione quindi il tipo di $f\ a$ è Z (anche se è semplicemente un *placeholder*).
- $([X \longmapsto Y \to Z, Z \longmapsto Nat], Z)$: come il caso precedente, ma in più associamo alla variabile Z il tipo base Nat. Chiaramente questa soluzione è meno generale della precedente.
- $([X \longmapsto Y \to Nat \to Nat], Nat \to Nat)$: in questo caso f viene vista come una funzione che, dato un parametro di tipo Y, restituisce a sua volta una funzione che va da numeri naturali a numeri naturali. Chiaramente il tipo di $f\ a$ è $Nat \to Nat$.
- $([X \longmapsto Nat \to Nat, Y \longmapsto Nat], Nat \to Nat)$: il tipo di f viene definito come tipo freccia da naturali a naturali. Il tipo di a è Nat, dunque il tipo di fa è un tipo freccia da Nat a Nat.

4 Type Reconstruction con constraint-based typing

L'algoritmo che introduciamo adesso permette, a partire da un termine t ed un contesto Γ, di generare la sostituzione più generale che contiene il *mapping* che tipa correttamente il termine coinvolto.

Questo algoritmo, proposto in [1], suddivide il problema di ricostruire i tipi in due fasi separate: la prima viene utilizzata per spostarsi nell'ambito dell'unificazione di predicati, la seconda si appoggia ad un algoritmo già

esistente nella letteratura e che sappiamo già restituire una sostituzione più generale che unifica i predicati generati.

Abbiamo quindi le seguenti fasi:

- la prima fase, denominata *constraint-based typing*, attraversa l'albero di derivazione del termine che ci interessa analizzare generando ad ogni passo dei vincoli logici sui tipi dei termini coinvolti;
- la seconda fase, attraverso un *algoritmo di unificazione*, cerca la sostituzione più generale in grado di soddisfare il precedente insieme, nel caso in cui non venga trovata allora il termine risulta non tipabile.

Prima di tutto abbiamo dobbiamo generare l'insieme di predicati da unificare, abbiamo che dato un contesto Γ e un termine t si determinano i *vincoli* (*constraints*) che devono essere rispettati per poter ben tipare il termine (vincoli che devono essere rispettati quindi da ogni soluzione (Γ, t)). Chiaramente l'idea che sta dietro a questo algoritmo è la stessa del *type checking*, la differenza principale è che nel primo caso i vincoli vengono determinati e controllati mentre in questo caso vengono solamente determinati per poterli trasformare in predicati da unificare.

Un esempio chiarificatore è il seguente, supponiamo di avere un termine di tipo applicazione $t_1\ t_2$, con

$$\Gamma \vdash t_1 : T_1 \quad \text{e} \quad \Gamma \vdash t_2 : T_2$$

Il normale algoritmo di *type checking* controlla che t_1 abbia tipo $T_2 \to T_{12}$ e assegna al risultato dell'applicazione il tipo T_{12}, il *constraint-based typing* introduce, invece, una nuova variabile di tipo X, determina il vincolo $T_1 = T_2 \to X$ ed il tipo di $t_1\ t_2$ con X.

Definizione (insieme di vincoli)

Un *insieme di vincoli* è un insieme di equazioni della forma

$$C = \{S_i = T_i\} \qquad\qquad i = 1, \cdots, n$$

Diciamo che una sostituzione σ unifica un'equazione $S = T$ se le istanze corrispondono (ossia $\sigma S = \sigma T$). Diciamo che una sostituzione σ unifica (o soddisfa) un set di vincoli C se unifica ogni equazione presente in esso.

Definizione (constraint typing relation)
Una *constraint typing relation* è una relazione della forma

$$\Gamma \vdash t : T \mid_{\chi} C$$

che informalmente sta a significare che il termine t ha tipo T, nel contesto Γ, quando l'insieme di vincoli C è soddisfatto.

Indicare la variabile X serve per poter distinguere le variabili di tipo che vengono introdotte in ogni passaggio della derivazione dei vincoli; questo per sapere sempre a quale variabile ci stiamo riferendo ed evitare *name clashes* via via che applichiamo le regole.

L'utilizzo di questa notazione permette due cose fondamentali:

- Quando una variabile di tipo viene selezionata da una regola terminale in una derivazione siamo certi che sia differente da ogni altra variabile introdotta nelle sottoderivazioni;
- Quando una regola utilizza due o più sottoderivazioni siamo sicuri che gli insiemi di variabili delle stesse siano disgiunti.

Infine queste regole permettono sempre di costruire una derivazione per un termine. I vincoli imposti limitano unicamente l'applicazione delle regole in cui una stessa variabile *fresh* appare più di una volta in posizioni differenti ma dato che possiamo arbitrariamente definire sempre nuove variabili esisterà sempre una derivazione che soddisfi quest'unico requisito di *freshness*.

L'algoritmo utilizza regole simili a quelle usate per il normale *type checking* per il lambda calcolo. La differenza principale è proprio il fatto che questo algoritmo non può mai fallire e che, dato un contesto Γ ed un termine t, è sempre possibile determinare unicamente un tipo T e un insieme di vincoli C tale che

$$\Gamma \vdash t : T \mid C$$

4.1 Regole per il *constraint typing*

La relazione è definita utilizzando le seguenti regole:

CT-VAR

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_{\varnothing} \{\}}$$

La regola dice che, avendo una variabile x di tipo T definita nel contesto Γ, è vera la stessa relazione in cui l'insieme di vincoli è vuoto (trivialmente, dato che non dobbiamo vincolare alcun tipo).

CT-ABS

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid_{\chi} C}{\Gamma \vdash \lambda x : T_1.t2 : T_1 \to T_2 \mid_{\chi} C}$$

In questo caso tipiamo la *lambda funzione*, assegnandogli tipo $T1 \to T2$, esattamente come viene fatto per il normale *type checking*. L'insieme dei vincoli non viene modificato dall'applicazione della regola perché anche in questo caso tutto deriva dal contesto e dalle assunzioni già fatte sui tipi.

CT-APP

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid_{\chi_1} C_1 \qquad \Gamma \vdash t_2 : T_2 \mid_{\chi_2} C_2 \\ \chi_1 \cap \chi_2 = \chi_1 \cap FV(T_2) = \chi_2 \cap FV(T_1) = \emptyset \\ X \notin \chi_1, \chi_2, T_1, T_2, C_1, C_2, \Gamma, t_1 \text{ or } t_2 \\ C' = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}\end{array}}{\Gamma \vdash t_1 t_2 : X \mid_{\chi_1 \cup \chi_2 \cup \{X\}} C'}$$

Il caso dell'applicazione di funzione è semplice in sé ma richiede una serie di *side conditions* necessarie ad evitare *clashes* tra le variabili di tipo che ovviamente non possono verificarsi onde inficiare l'intera risoluzione dell'algoritmo.

Fondamentalmente avendo un termine t_1 di tipo freccia al quale applichiamo un parametro t_2 di tipo T_2 il vincolo che si genera è che il tipo dell'applicazione è $T_2 \to X$ con X variabile *fresh* di tipo che non deve essere già stata utilizzata precedentemente.

Questo è un tipico esempio di un concetto che a livello implementativo si gestisce facilmente (generando variabili univoche ad ogni passaggio) ma che, all'interno di una trattazione formale, richiede una precisione accurata nella scelta delle condizioni aggiuntive.

CT-ZERO

$$\Gamma \vdash 0 : Nat \mid_{\varnothing} \{\}$$

Il termine 0 viene direttamente tipato con *Nat*. È chiaro che un tipo primitivo di un valore non viene mai considerato dalla *type reconstruction*, dato che di fatto non vi è nulla da inferire.

CT-SUCC (CT-PREC)

$$\frac{\Gamma \vdash t_1 : T \mid_{\chi} C \qquad C' = C \cup \{T = Nat\}}{\Gamma \vdash \text{succ } t_1 : Nat \mid_{\chi} C'}$$

La regola viene utilizzata quando troviamo il predicato successore, è chiaro che dobbiamo aggiungere il vincolo $T = Nat$ perché il successore accetta come parametro una variabile di tipo numero naturale, quindi per assicurarci che sia tipabile dobbiamo vincolarla ad essere proprio di questo tipo.

Viene espressa proprio l'idea della *type inference* in cui inferiamo il tipo di T considerando l'ambito in cui viene utilizzato (in questo caso un predicato che accetta numeri come parametri).

CT-ISZERO

$$\frac{\Gamma \vdash t_1 : T \mid_{\chi} C \qquad C' = C \cup \{T = Nat\}}{\Gamma \vdash \text{iszero } t_1 : Bool \mid_{\chi} C'}$$

Anche questa regola è simile alla precedente, l'unica differenza sta nel fatto che il predicato *iszero* invece di essere tipato come *Nat* viene banalmente tipato come *Bool*.

CT-TRUE (CT-FALSE)

$$\Gamma \vdash true : Bool \mid_{\varnothing} \{\}$$

Il predicato tipa il valore costante *true*. Chiaramente, come per il valore costante 0, l'insieme dei vincoli è vuoto.

CT-IF

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid_{\chi_1} C_1 \\ \Gamma \vdash t_2 : T_2 \mid_{\chi_2} C_2 \qquad \Gamma \vdash t_3 : T_3 \mid_{\chi_3} C_3 \\ \chi_1 \cap \chi_2 \cap \chi_3 = \emptyset \\ C' = C_1 \cup C_2 \cup C_3 \cup \{T_1 = Bool, T_2 = T_3\}\end{array}}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \mid_{\chi_1 \cup \chi_2 \cup \chi_3} C'}$$

Nel caso del costrutto *if* partiamo dai tre termini che lo compongono, ognuno con il proprio insieme di vincoli determinato. Imponendo, per evitare *clashes* tra i nomi delle variabili di tip,o che non vi siano variabili in comune possiamo determinare i vincoli che si generano: il tipo del termine t_1, corrispondente alla condizione, dovrà essere di tipo $Bool$ mentre i tipi T_2 e T_3 dovranno essere uguali.

Notiamo come l'algoritmo eviti di assegnare un tipo specifico, imponendo il vincolo più generale possibile: che essi siano lo stesso.

Osservazione

La possibilità di ommettere le annotazioni di tipo per le λ-astrazioni deriva dal fatto che il *parser* che analizza il programma semplicemente riempie ogni annotazione mancante con una variabile di tipo *fresh* generata in quel momento.

Questa soluzione, benché funzionante, presenta una limitazione data dal fatto che nel caso si volessero utilizzare più copie del termine, ognuna verrebbe annotata con una variabile di tipo *invisibile*, facendo sì che tutte le copie condividano lo stesso tipo per l'argomento.

Per risolvere il problema e rendere il sistema più espressivo possiamo introdurre un'apposita regola per le astrazioni non tipate e la relativa regola che ne genera i vincoli:

T-ABSINF

$$\frac{X \notin \chi \qquad \Gamma, x : X \vdash t_1 : T \mid_{\chi} C}{\Gamma \vdash \lambda x.t_1 : X \to T \mid_{\chi \cup \{X\}} C}$$

Questa regola ci permette di utilizzare argomenti diversi per copie dello stesso termine.

4.2 Correttezza e completezza

Per dimostrare la correttezza e completezza di entrambe le fasi dimostriamo prima che l'algoritmo che genera l'insieme dei vincoli funziona bene, che quindi è corretto nei confronti del normale algoritmo di *type checking* in cui i tipi delle variabili vengono effettivamente dichiarati.

Definizione (soluzione)
Supponiamo che $\Gamma \vdash t : S \mid C$, diciamo che (σ, t) è *soluzione* di (Γ, t, S, C) se σ soddisfa C e $\sigma S = T$.

Dato un contesto Γ ed un termine t esistono due approcci differenti con i quali è possibile determinare i tipi delle variabili in Γ e t per produrre un *typing* valido:

- possiamo utilizzare l'insieme di tutte le soluzioni di (Γ, t) della forma (σ, T) per le quali vale che $\sigma\Gamma \vdash \sigma t : T$ (approccio dichiarativo);
- oppure possiamo determinare l'insieme dei vincoli utilizzando la relazione appena definita, trovando S e C tali che $\Gamma \vdash t : S \mid C$ e successivamente trovare tutte le soluzioni di (Γ, t, S, C) (approccio algoritmico).

Dimostriamo adesso la correttezza e la completezza del *typing* con l'utilizzo dei vincoli nelle consuete due direzioni:

- Correttezza: ogni soluzione (Γ, t, S, C) è anche soluzione di (Γ, T);
- Completezza: ogni soluzione (Γ, T) può essere estesa ad una soluzione per (Γ, t, S, C).

Infine dobbiamo verificare che la sostituzione più generale o *tipo principale*:

Definizione (tipo principale)
Definiamo *soluzione principale* di (Γ, t, S, C) la soluzione (σ, T) tale che per ogni (σ', T') soluzione di (Γ, t, S, C) abbiamo che $\sigma \sqsubseteq \sigma'$. Abbiamo inoltre che T è il *tipo principale* di t in Γ.

4.2.1 Correttezza *(contraint-based-typing)*

Teorema 4.1. *Correttezza*

Supponiamo $\Gamma \vdash t : S \mid C$*, allora se* (Γ, T) *è soluzione di* (Γ, t, S, C)*, è anche soluzione di* (Γ, t)*.*

Dimostrazione. La dimostrazione procede per induzione sulla derivazione di $\Gamma \vdash t : S \mid C$, analizzando i singoli casi per le regole che possono essere applicate.

Ci riferiamo alle regole del *constraint-based typing* utilizzando il prefisso CT e alle regole standard utilizzate per il type checking del lambda calcolo con il prefisso T.

Alcuni esempi:

CT-VAR

Abbiamo che (σ, T) è soluzione di $\Gamma \vdash t : S \mid C$ e, dato che l'insieme dei vincoli è vuoto, abbiamo $\sigma S = T$. Utilizzando la regola T-VAR abbiamo $\sigma\Gamma \vdash t : T$ allo stesso modo.

CT-ABS

Abbiamo

$$t = \lambda x : T_1.t_2 \qquad S = T_1 \to S_2$$
$$\Gamma, x : T_1 \vdash t_2 : S_2 \mid C$$

Sappiamo che (σ, T) è soluzione di (Γ, t, S, C) quindi σ unifica l'insieme di vincoli e $T = \sigma S = \sigma T_1 \to \sigma S_2$. A questo punto abbiamo $(\sigma, \sigma S_2)$ soluzione di (Γ, t_2, S_2, C); per l'ipotesi induttiva $(\sigma, \sigma S_2)$ è soluzione di $((\Gamma, x : T_1), t_2)$. Utilizzando la regola T-ABS abbiamo $\sigma\Gamma \vdash \lambda x : \sigma T_1.\sigma t_2 : \sigma T_1 \to \sigma S_2 = \sigma(T_1 \to S_2) = T$.

CT-APP

Abbiamo

$$t = t_1\ t_2 \qquad S = x$$
$$\Gamma \vdash t_1 : S_1 \mid C_1 \qquad \Gamma \vdash t_2 : S_2 \mid C_2$$
$$C = C_1 \cup C_2 \cup \{S_1 = S_2 \to x\}$$

Quindi $(\sigma, \sigma S_1)$ è soluzione di (Γ, t_1, S_1, C_1) e $(\sigma, \sigma S_2)$ è soluzione di (Γ, t_2, S_2, C_2). Per l'ipotesi induttiva applicata sui sottotermini abbiamo $\sigma\Gamma \vdash \sigma t_1 : \sigma S_1$ e $\sigma\Gamma \vdash \sigma t_2 : \sigma S_2$. Dato che $\sigma S_1 = \sigma S_2 \to \sigma x$ otteniamo $\sigma\Gamma \vdash \sigma t_1 : \sigma S_2 \to \sigma x$, infine secondo la regola T-APP abbiamo $\sigma\Gamma \vdash \sigma(t_1\ t_2) : \sigma x = T$ dimostrando la correttezza per questa regola.

Si dimostrano in maniera simile le altre regole, concludendo che l'algoritmo che genera i vincoli è corretto.

□

4.2.2 Completezza *(constraint-based typing)*

Utilizziamo la notazione $\sigma\backslash X$ per indicare la sostituzione che non è definita per le variabili contenute nell'insieme X, altrimenti si comporta esattamente come la σ originale.

Teorema 4.2. *Completezza Vogliamo dimostrare che, assunto* $\Gamma \vdash t : S \mid C$, *se* (σ, T) *è una soluzione di* (Γ, t, S, C), *allora è anche soluzione di* (Γ, t).

Sia $\Gamma \vdash t : S \mid_\chi C$. *Se* (σ, T) *è una soluzione per* (Γ, t) *e* $dom(\sigma) \cap \chi = \emptyset$, *significa che esiste una soluzione* (σ', T) *di* (Γ, t, S, C) *tale che* $\sigma'\backslash\chi = \sigma$.

Dimostrazione. La dimostrazione procede, come la precedente, per induzione sulla lunghezza della derivazione nell'applicazione delle regole del *constraint typing*. Dobbiamo quindi dimostrare la completezza per ogni singola regola.

CT-VAR

$$t = x, \qquad x : S \in \Gamma$$

Per ipotesi abbiamo che (σ, T) soluzione di (Γ, t, S, C). Applicando il *lemma di inversione* otteniamo $T = \sigma S$. Dunque (σ, T) è soluzione di $(\Gamma, x, S, \{\})$.

CT-ABS

$$t = \lambda x : T_1.T_2 \qquad \Gamma, x : T_1 \vdash t_2 : S_2 \mid_X C \qquad S = T_1 \to S_2$$

Per ipotesi abbiamo (σ, T) soluzione di $(\Gamma, \lambda x : T_1.t_2)$. Applicando il *lemma di inversione* otteniamo $\sigma\Gamma, x : \sigma T_1 \vdash \sigma t_2 : T_2$ e $T = \sigma T_2 \to T_2$ per un qualche T_2.

Per ipotesi induttiva esiste una soluzione (σ', T_2) per $\big((\Gamma, x : T_1), t_2, S_2, C\big)$ tale che $\sigma'\backslash\chi$ concorda con σ; Ovviamente χ non può includere alcuna variabile di tipo già presente in T_1.

In conclusione abbiamo $\sigma' T_1 = \sigma T_1$, $\sigma'(S) = \sigma'(T_1 \to S_2) = \sigma T_1 \to \sigma' S_2 = \sigma T_1 \to T_2 = T$, quindi (σ', T) è soluzione di $\big(\Gamma, (\lambda x : T_1.t_2), T_1 \to S_2, C\big)$.

CT-APP

$$t = t_1 t_2 \qquad \Gamma \vdash t_1 : S_1 \mid_{\chi_1} C_1 \qquad \Gamma \vdash t_2 : S_2 \mid_{\chi_2} C_2$$
$$\chi_1 \cap \chi_2 = \emptyset \qquad \chi_1 \cap FV(S_2) = \emptyset \qquad \chi_2 \cap FV(S_1) = \emptyset$$
$$X \text{ non viene usata in } \chi_1, \chi_2, S_1, S_2, C_1, C_2$$
$$S = X \qquad \chi = \chi_1 \cup \chi_2 \cup \{X\} \qquad C = C_1 \cup C_2 \cup \{S_1 = S_2 \to X\}$$

Per ipotesi abbiamo (σ, T) soluzione di $(\Gamma, t_1\ t_2)$. Grazie al *lemma di inversione* otteniamo $\sigma\Gamma \vdash \sigma t_1 : T_1 \to T$ e $\sigma\Gamma \vdash \sigma t_2 : T_1$. Per l'ipotesi induttiva esistono le soluzioni

$$(\sigma_1, T_1 \to T) \text{ di } (\Gamma, t_1, S_1, C_1)$$
$$(\sigma_2, T_1) \text{ di } (\Gamma, t_2, S_2, C_2)$$

con $\sigma_1\backslash X_1 = \sigma = \sigma_2\backslash X_2$.

A questo punto è necessario definire una sostituzione σ' tale che:

- $\sigma'\backslash X$ concorda con σ;
- $\sigma' X = T$;
- σ' unifica entrambi C_1 e C_2;
- σ' unifica $\{S_1 = S_2 \to x\}$.

Definiamo σ' per casi in questo modo:

$$\sigma' = \begin{cases} Y \mapsto U & \text{se } Y \notin X \wedge (Y \mapsto U) \in \sigma \\ Y_1 \mapsto U_1 & \text{se } Y_1 \in X_1 \wedge (Y_1 \mapsto U_1) \in \sigma_1 \\ Y_2 \mapsto U_2 & \text{se } Y_2 \in X_2 \wedge (Y_2 \mapsto U_2) \in \sigma_2 \\ X \mapsto T & \end{cases}$$

Il primo caso è banalmente verificato. Lo sono anche il secondo e il terzo considerando che per ipotesi abbiamo che χ_1 non ha variabili in comune con χ_2. Infine per l'ultimo caso osserviamo che le condizioni ci garantiscono che $FV(S_1) \cap (\chi_2 \cup \{X\}) = \emptyset$ quindi per passaggi abbiamo $\sigma' S_1 = \sigma_1 S_1 = T_1 \to T = \sigma_2 S_2 \to T = \sigma' S_2 \sigma' X = \sigma'(S_2 \to X)$.

La dimostrazione per le altre regole procede, al solito, in maniera simile. □

Corollario 4.3. *Sia* $\Gamma \vdash t : s \mid C$*, allora esiste una soluzione per* (Γ, t) *se e solo se esiste una soluzione per* (Γ, t, S, C).

Dimostrazione. Deriva trivialmente dai due precedenti teoremi. □

5 Unificazione

Generato l'insieme dei vincoli possiamo cercare la sostituzione con lo unifica, se esiste.

Per fare questo possiamo avvalerci di uno strumento della logica matematica, chiamato *unificazione*, che trova una sostituzione che è soluzione dell'insieme di vincoli e che rappresenta inoltre la migliore soluzione, nel senso che tutte le altre sono determinabili da questa.

Vi sono varie metodologie per unificare un insieme di predicati, ma in gerale una *unificazione* è una classificazione in base a criteri di specializzazione tra i vari predicati. Avremo quindi che un vincolo di tipo specializza il mapping che ci interessa trovare.

5.1 Unificazione di Robinson

Utilizziamo l'unificazione di Robinson, che funziona **unificando** più espressioni e risolvendo l'assegnazione delle variabili all'interno delle stesse in modo da rispettarle tutte.

L'algoritmo seleziona uno ad uno i vincoli $\{S = T\} \in C$:

unify(C) = if $(C = \emptyset)$ then []
else let $\{S = T\} \cup C' = C$ in
if $S = T$
then *unify(C')*
else if $S = X$ and $X \notin FV(T)$

then *unify(*$[X \mapsto T]C') \circ [X \mapsto T]$
else if $T = X$ and $X \notin FV(S)$
then *unify(*$[X \mapsto S]C') \circ [X \mapsto SW]$
else if $S = S_1 \to S_2$ and $T = T_1 \to T_2$
then *unify(*$C' \cup \{S_1 = T_1, S_2 = T_2\}$*)*
else
fail

È interessante osservare che per evitare di generare delle sostituzioni cicliche vengono utilizzati i due controlli $X \notin FV(T)$ e $X \notin FV(S)$ in modo che l'algoritmo non possa generare una sostituzione della forma $X \mapsto X \mapsto X$. Questo test viene definito come *occur check* ed è necessario nel caso di espressioni di tipo finite, chiaramente potrebbe essere omesso in estensioni particolari come i *tipi ricorsivi*.

L'algoritmo banalmente termina dato che ad ogni iterazione rimuove un vincolo dall'insieme prima di invocarsi ricorsivamente. La sua correttezza, invece, è stata ampiamente dimostrata in letteratura[3].

5.2 Correttezza e completezza

Dimostrazione. Abbiamo già la correttezza dei predicati rispetto al *type checker*, aggiungendo il fatto che l'algoritmo di unificazione restituisce la sostituzione più generale per ipotesi possiamo dire che l'algoritmo completo è corretto. □

Riferimenti bibliografici

1 Benjamin C.Pierce (2002), *"Types and Programming Languages"*, MIT Press.

2 Milner, Robin (1978), *"A Theory of Type Polymorphism in Programming"* , Jcss 17: 348-375.

3 Robinson, J. Alan., *"Computational logic: The unification computation"*, Machine Intelligence, 6:63-72, 1971.