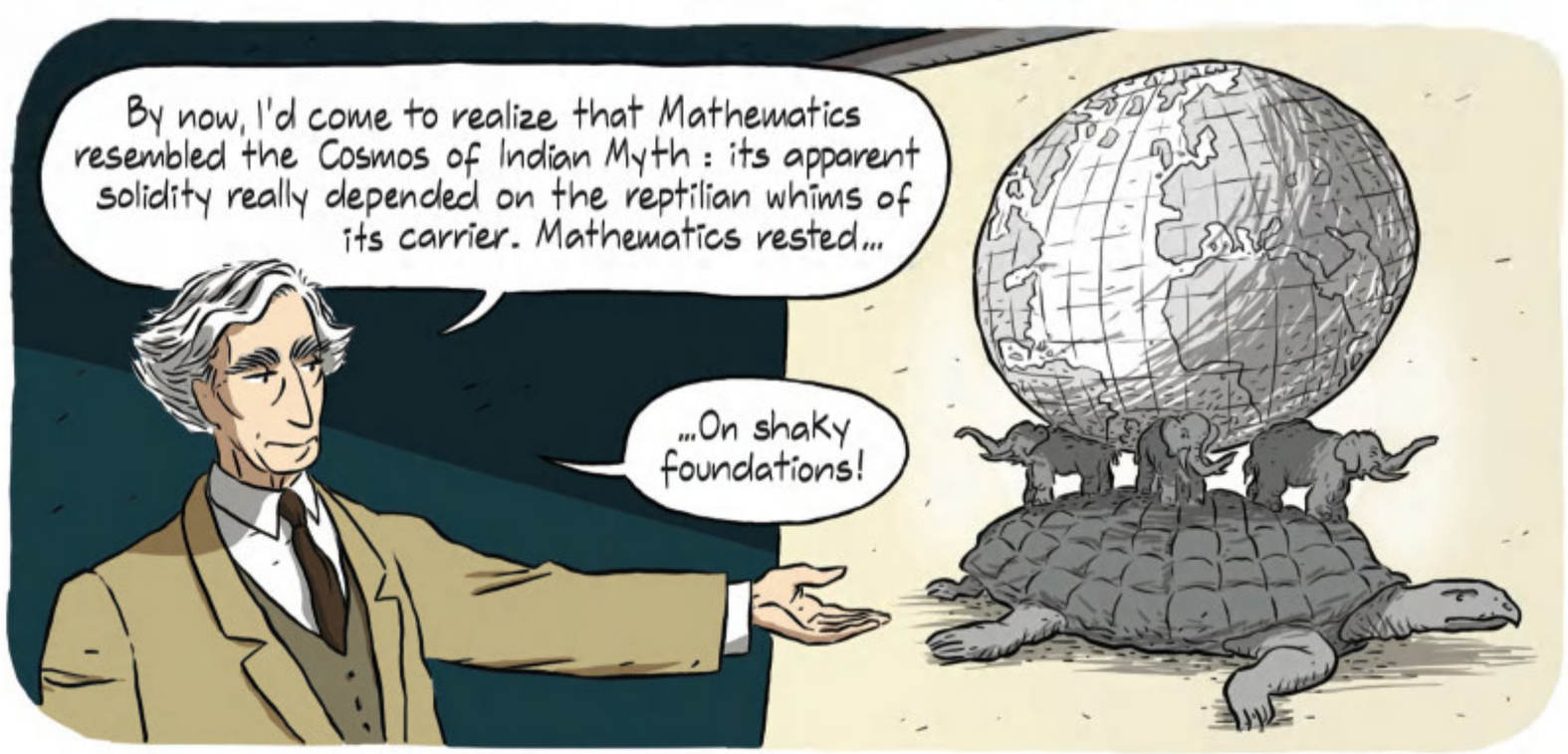
Introduzione ai Tipi

Un po' di storia

Il termine **sistema di tipi** (o teoria dei tipi) si riferisce a un ampio campo di studi in logica, matematica e filosofia. I primi ricercatori hanno formalizzato i sistemi di tipi in questo senso all'inizio del 1900 per evitare i **paradossi logici**, come il paradosso di Russell, che minacciavano i fondamenti della matematica.



Doxiadis, Apostolos, and Christos Papadimitriou. ***Logicomix: An epic search for truth***. Bloomsbury Publishing USA, 2015.

Un po' di storia

Il termine **sistema di tipi** (o teoria dei tipi) si riferisce a un ampio campo di studi in logica, matematica e filosofia. I primi ricercatori hanno formalizzato i sistemi di tipi in questo senso all'inizio del 1900 per evitare i **paradossi logici**, come il paradosso di Russell, che minacciavano i fondamenti della matematica.

Sia $R = \{x \mid x \notin x\}$

allora $R \in R \iff R \notin R$

Doxiadis, Apostolos, and Christos Papadimitriou. ***Logicomix: An epic search for truth***. Bloomsbury Publishing USA, 2015.

Un po' di storia

Il termine **sistema di tipi** (o teoria dei tipi) si riferisce a un ampio campo di studi in logica, matematica e filosofia. I primi ricercatori hanno formalizzato i sistemi di tipi in questo senso all'inizio del 1900 per evitare i **paradossi logici**, come il paradosso di Russell, che minacciavano i fondamenti della matematica.

Let $R = \{x \mid x \notin x\}$
then $R \in R \iff R \notin R$

L'intuizione di Russel sull'introduzione dei tipi: l'inclusione degli insiemi è condizionata ai tipi.

Sia $Set \vdash x, R = \{x \mid x \notin x\} \implies Set(Set) \vdash R \wedge R \notin R$

Sia $Set(Set) \vdash x, R = \{x \mid x \notin x\} \implies Set(Set(Set)) \vdash R \wedge R \notin R$

. . .

tipo di

Un po' di storia

Nel corso del XX secolo, i tipi sono diventati strumenti standard della logica, in particolare della teoria delle prove, e hanno permeato il linguaggio della filosofia e della scienza.
In informatica esistono due grandi branche di ricerca sui sistemi di tipi:

- quella più **pratica** riguarda le applicazioni ai linguaggi di programmazione;
- quella più **astratta** si concentra sulle connessioni con le varietà della logica.

Entrambi i rami utilizzano concetti, notazioni e tecniche simili, ma con alcune importanti differenze di orientamento: ad esempio, la ricerca astratta di solito riguarda sistemi in cui ogni computazione *ben tipata* è garantita terminare, mentre la maggior parte delle applicazioni pratiche sacrifica questa proprietà a vantaggio di caratteristiche come le definizioni ricorsive.

Un po' di storia

Una definizione plausibile di sistema di tipi in informatica è:

Un metodo sintattico praticabile per dimostrare l'assenza di determinati comportamenti del programma, fatto classificando le unità sintattiche in base ai tipi di valore che assumono.

Un elemento importante della definizione precedente è l'enfasi posta sulla classificazione dei termini — le unità sintattiche — in base alle proprietà dei valori che assumeranno durante l'esecuzione.

Un sistema di tipi può essere considerato come una sorta di approssimazione statica al comportamento in fase di esecuzione dei termini di un programma.

Un po’ di storia

Guardando all'aspetto più pratico dei tipi, possiamo interpretarli come

collezioni di valori omogenei e rappresentabili

Quindi, un tipo è un insieme di valori, ad esempio, numeri interi. L'aggettivo **omogeneo**, un po' informale, suggerisce che tali valori devono condividere alcune proprietà strutturali, che li rendono tutti simili tra loro.

La parte sulla **rappresentazione** dei valori dell'interpretazione si riferisce all'aspetto pratico della manipolazione di valori che possono essere rappresentati (scritti, nominati) in modo finito. Ad esempio, i numeri reali non sono effettivamente rappresentabili, perché esistono numeri reali con espansione decimale infinita che non possono essere ottenuti con alcun algoritmo. Pertanto, le loro approssimazioni nei linguaggi di programmazione (ad esempio, real e float) sono il loro sottoinsieme rappresentabile.

Tipi di Dato • Supporto all'Organizzazione Concettuale

I tipi possono aiutare a disciplinare l'organizzazione concettuale dei programmi.

I tipi consentono ai programmatori di esprimere la differenza tra le entità che formano la soluzione di un determinato problema. Ad esempio, un programma di prenotazione alberghiera contiene probabilmente concetti come *clienti*, *date*, *prezzi*, *camere*, ecc. Il programmatore può definire un tipo per ciascuno di questi concetti. L'uso dei tipi aiuta a separare logicamente elementi concettualmente diversi, come una camera e un prezzo. Questi possono avere implementazioni simili, ad esempio possono essere entrambi numeri interi, ma a livello di progettazione sono considerati tipi distinti.

L'uso di tipi distinti è uno strumento di **documentazione** e di **progettazione**. Conoscere il tipo di una variabile ci aiuta a capire che ruolo ha nel programma. In questo senso, i tipi svolgono un ruolo simile a quello dei commenti. Tuttavia, a differenza dei commenti, possiamo usare i tipi per ragionare sui programmi che annotano, ad esempio segnalando l'assegnazione errata di una variabile dichiarata come "Stanza" a un valore annotato come "Prezzo" (si veda la parte sulla "correttezza", nel seguito).

Tipi di Dato • Supporto all'Astrazione

Una particolare declinazione del supporto all'organizzazione concettuale offerto dai tipi è quella di dare supporto concreto ai sistemi di moduli nei linguaggi—dove i moduli "impacchettano" e legano insieme diverse unità software.

L'esempio tipico di questo tipo di utilizzo dei tipi sono le **interfacce**, che **associano un tipo** (per esempio, l'intero) **ad operazioni** che si possono applicare su di esso (per esempio, +, -, %, conversioni). In effetti, possiamo considerare un'interfaccia come una sorta di "menù" delle strutture fornite dal modulo o come parte di un contratto tra implementatori e utenti.

Avere unità software in termini di tipi (moduli con interfacce) porta a uno stile di progettazione più astratto. Poiché i programmatori possono astrarre dall'implementazione di ciascuna unità, possono concentrarsi sulla progettazione del software dall'alto verso il basso: dai contratti che i moduli si offrono reciprocamente alla loro implementazione indipendente (questo supporta anche il riutilizzo e la ristrutturazione coerente del codice).

Tipi di Dato • Supporto alla Correttezza

Il vantaggio più famoso dei tipi è il **type-checking**, cioè la possibilità di usare i tipi per rilevare errori di programmazione.

Ad esempio, se riusciamo a scoprire un errore prima dell'esecuzione del nostro programma, possiamo risolverlo immediatamente, invece di scoprirlo mentre usiamo il programma (o peggio, quando lo usano i nostri utenti). Inoltre, le segnalazioni di errore derivanti dal controllo dei tipi sono spesso più accurate e più facili da gestire rispetto all'analisi dello stack-trace dell'errore (se presente) nel runtime.

I sistemi di tipi semplici possono impedirci di assegnare il valore sbagliato a una variabile, ma quelli più avanzati possono individuare, ad esempio, la mancata corrispondenza tra i tipi di ritorno delle clausole if-then-else, che si manifestano come inconsistenze a livello di tipi.

Tipi di Dato • Supporto alla Correttezza

Naturalmente, l'impatto dei tipi sulla correttezza del programma dipende dall'espressività del sistema di tipi e dal compito di programmazione in questione. Ad esempio, se codifichiamo tutte le nostre strutture di dati come elenchi, il compilatore non ci aiuterà tanto quanto se definissimo un tipo diverso per ciascuna di esse.

I tipi aiutano molto anche il **refactoring** (l'atto di ristrutturare il codice esistente). Per esempio, senza il supporto di un sistema di tipi (statici), se cambiamo la definizione di una struttura di dati dobbiamo cercare e aggiornare tutto il codice che riguarda tale struttura. Con un sistema di tipi (statico), una volta cambiata la dichiarazione del tipo della struttura, un passaggio del type checker indicherà tutti i punti in cui il tipo è usato in modo incoerente, per essere corretti.

Tipi di Dato • Supporto alla Correttezza (safety)

Il supporto per la correttezza menzionato in precedenza si riferisce al concetto più generale di “safety” del linguaggio, come in “well-typed programs do not go wrong" [1] o “safe languages make it impossible to shoot yourself in the foot while programming” [2].

In generale, la “safety” si riferisce alla capacità di un linguaggio di **garantire l'integrità delle sue astrazioni** (e delle astrazioni di livello superiore introdotte dal programmatore utilizzando le strutture di definizione del linguaggio). I linguaggi “safe” sono anche detti a *tipizzazione forte* o *strongly typed* (e quelli non sicuri a *tipizzazione debole* o *weakly typed*).

Ad esempio, un linguaggio può fornire array, con operazioni di accesso e aggiornamento, come astrazione della memoria sottostante. Utilizzando questo linguaggio, ci si può aspettare di modificare un array solo utilizzando esplicitamente l'operazione di aggiornamento e non, ad esempio, scrivendo oltre la fine di un'altra struttura dati. Allo stesso modo, possiamo aspettarci di accedere alle variabili con scoping lessicale solo all'interno dei loro ambiti.

[1] Milner, Robin. "**A theory of type polymorphism in programming.**" *Journal of computer and system sciences* 17.3 (1978): 348-375.

[2] Pierce, Benjamin C. **Types and programming languages**. MIT press, 2002.

Tipi di Dato • Supporto alla Correttezza (safety)

In particolare, possiamo ottenere la sicurezza del linguaggio attraverso la type safety, ma i tipi non sono l'unica freccia al nostro arco. Per esempio, possiamo avere controlli di runtime che rilevano le operazioni senza senso nel momento in cui il programma prova a eseguirle e lo fermano o sollevano un'eccezione.

Al contrario, i linguaggi non-safe forniscono garanzie di sicurezza “best effort” che aiutano i programmatori a eliminare gli errori più evidenti, ma non garantiscono la conservazione delle loro astrazioni.

In questa prospettiva, possiamo considerare “safe” un linguaggio come Java—dove il suo compilatore (utilizzando il suo sistema di tipi) è in grado di rilevare una pletora di problemi, ma il linguaggio si occupa di altre classi di problemi tramite controlli a runtime, ad esempio sollevando eccezioni sull'accesso ad array fuori limite e sui riferimenti a riferimenti inesistenti—ma non un linguaggio come C, poiché, ad esempio, può permettere ai programmi di accedere ad array oltre la loro fine.

Tipi di Dato • Supporto all'Implementazione

I tipi possono contribuire a migliorare **l'efficienza** dei programmi. In effetti, i progettisti hanno introdotto i primi sistemi di tipi negli anni '50 in linguaggi come Fortran per migliorare l'efficienza dei calcoli numerici, distinguendo tra espressioni aritmetiche a valore intero e a valore reale; ciò ha permesso al compilatore di utilizzare rappresentazioni diverse e di generare istruzioni macchina ottimizzate.

Nei linguaggi safe, i tipi aiutano a migliorare l'efficienza eliminando molti dei controlli dinamici per garantire la sicurezza. I moderni compilatori ad alte prestazioni si basano molto sulle informazioni raccolte dal type-checker per ottimizzare la generazione del codice.

Tipi di Dato • Altre Applicazioni

I tipi sono stati utilizzati nella sicurezza dei computer e delle reti. Ad esempio, la tipizzazione è alla base del modello di sicurezza di Java e dell'architettura “plug and play” di JINI per i dispositivi di rete. Allo stesso modo, ricercatori hanno utilizzato algoritmi di controllo dei tipi in strumenti di analisi dei programmi diversi dai compilatori, come ad esempio l'analisi degli alias e delle eccezioni.

I theorem provers utilizzano sistemi di tipi — solitamente potenti, basati su tipi dipendenti — per rappresentare proposizioni logiche e prove.

Anche i database e i sistemi di gestione dei dati utilizzano i tipi, ad esempio sotto forma di Document Type Definitions e altri tipi di schemi (schemi XML, JSON) e linguaggi di interrogazione che integrano questi linguaggi per verificare la correttezza delle interrogazioni.

Un po' di storia

1870s	*origins of formal logic*	Frege (1879)
1900s	*formalization of mathematics*	Whitehead and Russell (1910)
1930s	*untyped lambda-calculus*	Church (1941)
1940s	*simply typed lambda-calculus*	Church (1940), Curry and Feys (1958)
1950s	Fortran	Backus (1981)
	Algol-60	Naur et al. (1963)
1960s	*Automath project*	de Bruijn (1980)
	Simula	Birtwistle et al. (1979)
	Curry-Howard correspondence	Howard (1980)
	Algol-68	(van Wijngaarden et al., 1975)
1970s	Pascal	Wirth (1971)
	Martin-Löf type theory	Martin-Löf (1973, 1982)
	System F, F^{ω}	Girard (1972)
	polymorphic lambda-calculus	Reynolds (1974)
	CLU	Liskov et al. (1981)
	polymorphic type inference	Milner (1978), Damas and Milner (1982)
	ML	Gordon, Milner, and Wadsworth (1979)
	intersection types	Coppo and Dezani (1978)
		Coppo, Dezani, and Sallé (1979), Pottinger (1980)

1980s	NuPRL project	Constable et al. (1986)
	subtyping	Reynolds (1980), Cardelli (1984), Mitchell (1984a)
	ADTs as existential types	Mitchell and Plotkin (1988)
	calculus of constructions	Coquand (1985), Coquand and Huet (1988)
	linear logic	Girard (1987) , Girard et al. (1989)
	bounded quantification	Cardelli and Wegner (1985)
		Curien and Ghelli (1992), Cardelli et al. (1994)
	Edinburgh Logical Framework	Harper, Honsell, and Plotkin (1992)
	Forsythe	Reynolds (1988)
	pure type systems	Terlouw (1989), Berardi (1988), Barendregt (1991)
	dependent types and modularity	Burstall and Lampson (1984), MacQueen (1986)
	Quest	Cardelli (1991)
	effect systems	Gifford et al. (1987), Talpin and Jouvelot (1992)
	row variables; extensible records	Wand (1987), Rémy (1989)
		Cardelli and Mitchell (1991)
1990s	higher-order subtyping	Cardelli (1990), Cardelli and Longo (1991)
	typed intermediate languages	Tarditi, Morrisett, et al. (1996)
	object calculus	Abadi and Cardelli (1996)
	translucent types and modularity	Harper and Lillibridge (1994), Leroy (1994)
	typed assembly language	Morrisett et al. (1998)

Dynamic vs Static Typing

Un linguaggio è “tipato **staticamente**” (statically typed) se possiamo controllare i tipi sul testo del programma, senza doverlo eseguire, e.g., in fase di compilazione. A meno che il runtime non abbia bisogno di informazioni a livello di tipo (vedremo alcuni esempi in cui potremmo voler conservare alcune di queste informazioni), il compilatore può eliminare le annotazioni di tipo (e i relativi controlli) dal codice generato.

Si parla di linguaggi “tipati **dinamicamente**” (dynamically typed) quando il controllo dei tipi avviene durante l'esecuzione del programma. In particolare, il controllo dinamico dei tipi richiede che ogni valore abbia un descrittore di esecuzione che ne specifichi il tipo e che, a ogni operazione, il runtime verifichi che il programma esegua operazioni solo su operandi del tipo corretto.

Manifest vs Inferred typing

La tipizzazione statica o dinamica riguarda il **momento** in cui un linguaggio (interpretato o compilato) esegue il controllo dei tipi. Il tipaggio manifesto (manifest) o inferito (inferred) determina la **quantità di informazioni** sui tipi che il programmatore deve inserire nei suoi programmi.

Un linguaggio con **manifest typing** richiede che il programmatore annoti tutti i tipi di variabili e operazioni. Un linguaggio con inferred typing non ha bisogno di annotazioni, in quanto dispone di algoritmi che deducono i tipi dal contesto (dichiarazioni, operazioni).

Mentre il tipaggio statico e quello dinamico lasciano poco spazio (utile) all'ibridazione, il **tipaggio manifesto e inferito sono uno spettro** determinato da fattori ergonomici e algoritmici. Ad esempio, leggendo il programma `x=5` noi (come programmatori) possiamo dedurre che il tipo di `x` potrebbe essere intero. Tuttavia, con programmi più complessi, potremmo faticare a tenere a mente tutti i dettagli e preferiremmo beneficiare delle informazioni documentali aggiuntive fornite dai tipi. D'altra parte, chiedere di annotare tutto può rallentare sensibilmente la programmazione. Per questi motivi, il bilanciamento tra tipi manifesti e tipi inferiti non è una decisione netta e riguarda il bilanciamento tra supporto/impegno che un linguaggio offre/richiede ai programmatori.

Dynamic vs Static Typing

Quando si parla dei compromessi tra l'uso di linguaggi con dynamic e static typing si tende a fare un po' di confusione.

Un esempio è il presupposto che i linguaggi tipati staticamente siano a tipaggio manifesto e quelli a tipaggio dinamico a tipaggio inferito. Questo è vero per molti linguaggi, ma in misura diversa. Per esempio, Java è un linguaggio famoso per la suo verboso tipaggio statico, ma le nuove versioni hanno perfezionato le capacità di inferenza del suo type-checker per consentire agli utenti di omettere molte annotazioni di tipo.

Un altro mito da sfatare è che i linguaggi a tipaggio dinamico siano interpretati. Questo deriva dal folklore che i linguaggi determinino il modo in cui si eseguono i programmi. Possiamo avere compilatori per linguaggi a tipaggio dinamico che dotano le operazioni eseguite dal programma sorgente del codice di controllo necessario per controllare i tipi in fase di esecuzione. Anche l'altra direzione è vera, ad esempio le proiezioni di Futamura [1] utilizzano la valutazione parziale per ottenere compilatori dagli interpreti.

[1] Futamura, Yoshihiko. "**Partial evaluation of computation process—an approach to a compiler-compiler**." *Higher-Order and Symbolic Computation* 12.4 (1999): 381-391.

Dynamic vs Static Typing

Quindi, la richiesta di scegliere tra un linguaggio a tipaggio dinamico e uno a tipaggio statico non riguarda la quantità di informazioni che i programmatori devono fornire sui tipi (manifesti o inferiti) o se i programmi sono interpretati o compilati (implementazione). Invece, i fattori principali che possono orientare la scelta tra i due approcci sono la correttezza, le prestazioni e l'espressività del programma.

Il tipaggio statico significa trovare gli errori prima di eseguire il programma. Questo elimina la necessità di annotare i termini e di eseguire controlli a tempo di esecuzione e permette alcune ottimizzazioni, aumentando le prestazioni. Tuttavia, i linguaggi a tipaggio statico hanno un “difetto” comune: a seconda dell'espressività dei tipi e dell'accuratezza dell'algoritmo di type-checking, possono rifiutare programmi che verrebbero eseguiti correttamente (ad esempio, il codice sulla destra). Ciò è dovuto all'indecidibilità dei programmi (terminazione), che rende impossibile sapere quali rami verranno eseguiti o meno e costringe il type-checking a considerare tutti i possibili stati di calcolo.

```
int x
if(e) x=“A”
else x = 5
```