

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

**Analisi del linguaggio x10 per
architetture parallele: il caso di studio
dell'algoritmo Gift Wrapping**

Tesi in
High Performance Computing

Relatore:
Prof. Moreno Marzolla

Candidato:
Simone Romagnoli

Anno Accademico 2019 – 2020

Alla mia famiglia

Indice

Introduzione	1
1 Introduzione al calcolo parallelo	3
1.1 High Performance Computing	3
1.2 Valutazione delle prestazioni	3
1.2.1 Speed-up	4
1.2.2 Efficienza	5
2 Il problema dell'inviluppo convesso	9
2.1 L'algoritmo	9
2.2 Gift Wrapping	10
3 Analisi di x10	13
3.1 Nascita e sviluppo del linguaggio x10	13
3.2 Componenti	14
3.2.1 Activity	15
3.2.2 Place	16
3.3 Potenzialità del linguaggio	18
3.4 Contenuto dell'elaborato	18
3.4.1 Versione a memoria condivisa	20
3.4.2 Versione a memoria distribuita	22
4 Valutazione delle prestazioni	25
4.1 Versione a memoria condivisa	26
4.1.1 Speed-up	27
4.1.2 Strong Scaling Efficiency	29
4.1.3 Weak Scaling Efficiency	29
4.2 Versione a memoria distribuita	31
4.2.1 Speed-up	31
4.2.2 Strong Scaling Efficiency	32
4.2.3 Weak Scaling Efficiency	33
Conclusioni	35

Ringraziamenti **37**

Bibliografia **39**

Introduzione

L'obiettivo di questa tesi consiste nello studio di x10, un linguaggio di programmazione orientato agli oggetti le cui componenti estendono linguaggi sequenziali come Java e C++ per permettere il calcolo parallelo. Per trarre delle conclusioni pratiche, questa tesi analizza l'algoritmo per il calcolo dell'inviluppo convesso, uno degli algoritmi fondamentali usati in geometria computazionale, e ne illustra una versione parallela implementata in x10.

Il concetto alla base del calcolo parallelo è quello di avere più unità di calcolo che si suddividono delle computazioni equamente, in modo da ridurre il tempo d'esecuzione di quest'ultime: l'High Performance Computing è la disciplina che studia il calcolo parallelo, e, negli anni, sono stati sviluppati diversi linguaggi e protocolli con i quali è possibile parallelizzare programmi sequenziali rendendoli estremamente più veloci.

X10 è un linguaggio basato su classi e staticamente tipizzato, e rientra fra i linguaggi di High Performance Computing: è stato progettato per ottenere calcoli ad alte prestazioni su computer che supportino circa 10^{15} operazioni al secondo e circa 10^5 thread hardware [6]. Un fattore importante che ha contribuito allo sviluppo di x10 è stato quello di voler aumentare la produttività dei programmati; infatti, è stato concepito per avere una semantica semplice, in modo da essere facilmente compreso da persone familiari con linguaggi orientati agli oggetti.

Il lavoro di tesi è stato suddiviso nei seguenti capitoli:

- **Capitolo 1** - Introduzione al calcolo parallelo;
- **Capitolo 2** - Il problema dell'inviluppo convesso;
- **Capitolo 3** - Analisi di x10;
- **Capitolo 4** - Valutazione delle prestazioni.

Capitolo 1

Introduzione al calcolo parallelo

In questo capitolo viene brevemente introdotto il calcolo parallelo per fornire una visione più specifica del lavoro che verrà svolto.

1.1 High Performance Computing

Il calcolo parallelo nasce con l'obiettivo di incrementare le prestazioni delle computazioni che hanno un costo elevato, ad esempio calcoli di fisica quantistica o simulazioni di eventi astronomici.

Alla base dell'High Performance Computing, vi è il concetto di concorrenza: la computazione viene divisa in flussi di dati, comunemente chiamati *thread*, che eseguono in maniera indipendente ripartendosi una porzione del carico di lavoro, operando sullo stesso insieme di dati. Tuttavia, gestire un gruppo di *thread* non è semplice: bisogna tener conto della possibilità di sovrascrittura dei dati da parte di *thread* diversi (*race condition*) oppure della necessità di sincronizzarsi per mantenere il parallelismo temporale. Di fatto, esistono una serie di costrutti che i linguaggi di HPC utilizzano per risolvere i problemi del calcolo parallelo: un esempio sono le barriere, che bloccano i *thread* finché tutti non le raggiungono in modo da sincronizzarli, oppure l'utilizzo di operazioni atomiche - che non vengono mai interrotte dall'inizio di altre istruzioni, e quindi possono essere considerate sequenziali - per evitare *race condition*.

1.2 Valutazione delle prestazioni

Nella parallelizzazione di un programma, è essenziale comprendere quanto esso possa migliorare con il progressivo aumentare dei processori a disposizione. La valutazione delle prestazioni serve a capire la scalabilità del programma,

ovvero la sua capacità di migliorare o peggiorare in funzione delle risorse. Per valutare le prestazioni si fa riferimento a due tipi di misure: lo *speed-up* e l'efficienza.

1.2.1 Speed-up

Lo scopo primario nella progettazione di un programma parallelo è quello di ottenere un miglioramento di tempo rispetto alla versione seriale: se effettivamente la versione parallela risulta più veloce, allora si può dire di aver ottenuto uno *speed-up*, ovvero un miglioramento delle prestazioni del programma, in termini di velocità.

Lo *speed-up* non è solo un concetto, ma è anche misurabile in maniera precisa; definendo p come il numero di processori utilizzati, T_s il tempo d'esecuzione del programma seriale, $T_p(p)$ quello del programma parallelo usando p processi o *thread*, lo *speed-up* $S(p)$ è:

$$S(p) = \frac{T_s}{T_p} \simeq \frac{T_p(1)}{T_p(p)}$$

Nel caso ideale, un programma parallelo impiega $\frac{1}{p}$ del tempo della versione seriale: ad esempio, un calcolo che impiega 60 secondi, se parallelizzato ed eseguito con 2 processori, il tempo di computazione diventa $\frac{60}{2} = 30$; nel caso i processori siano 3, il tempo calerebbe a 20 secondi, con 6 processori a 10 secondi, e così via (tempi illustrati nel grafico 1.1). Pertanto, dalla formula si ottiene che il caso ideale di *speed-up* è $S(p) = p$, ovvero uno *speed-up* lineare (mostrato in figura 1.2).

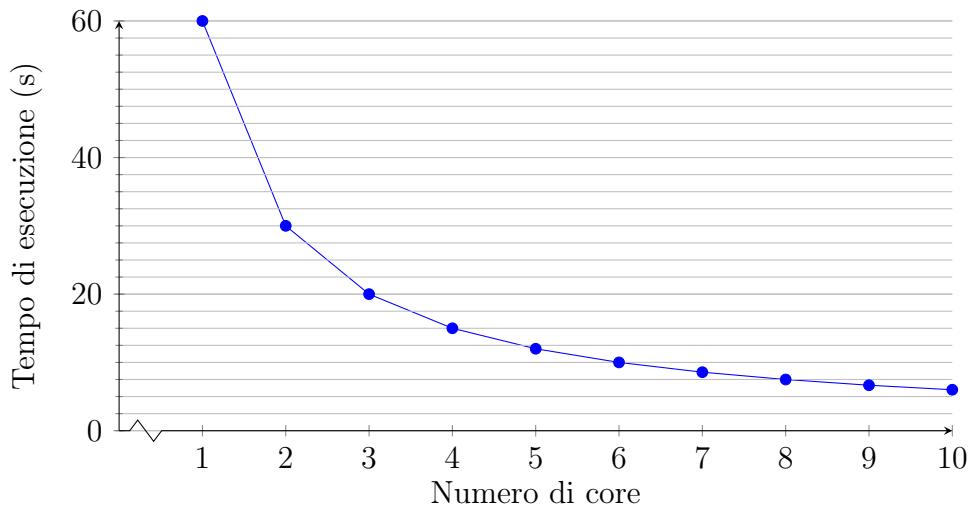


Figura 1.1: esempio di grafico dei tempi con *speed-up* lineare

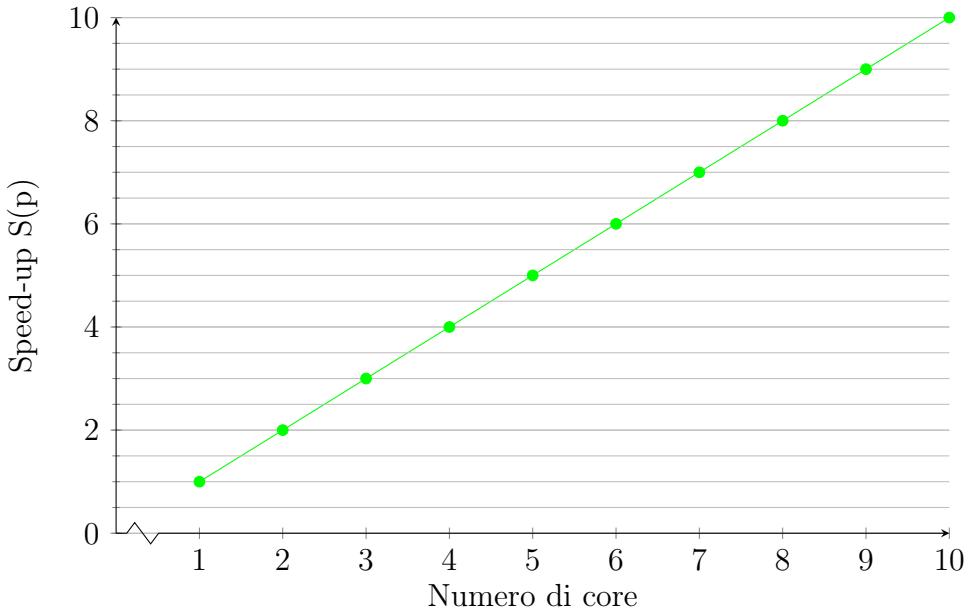


Figura 1.2: grafico dello *speed-up* lineare relativo alla figura 1.1

Nella pratica è difficile ottenere uno *speed-up* lineare in quanto spesso si riscontrano porzioni di codice non parallelizzabili, ma anche per questioni di livello più basso; quindi, realisticamente, lo *speed-up* risulta essere $S(p) \leq p$. Nel caso in cui vi sia una porzione di codice α non parallelizzabile, e supponendo che il restante $(1 - \alpha)$ lo sia, si ottiene un tempo d'esecuzione parallelo rappresentato dalla seguente equazione:

$$T_p(p) = \alpha T_s + \frac{(1 - \alpha)T_s}{p}$$

Non sempre α corrisponde a porzioni di codice non parallelizzabile: spesso ci sono dei ritardi dovuti a comunicazioni tra i *thread* o dipendenze dei dati insite negli algoritmi.

In rari casi, si può riscontrare uno *speed-up* superlineare, cioè $S(p) > p$; questa particolarità viene causata da fattori esterni che non riguardano il codice come l'utilizzo di tecniche di caching o l'eterogeneità dell'hardware.

1.2.2 Efficienza

L'efficienza è una misura che permette di interpretare la scalabilità di un programma parallelo; è un valore spesso compreso tra 0 e 1 che indica se il programma scala con risultati positivi. Esistono due tipi di misura dell'efficienza:

la Strong Scaling Efficiency e la Weak Scaling Efficiency.

La Strong Scaling Efficiency - o scalabilità forte - rappresenta la capacità di mantenere un buon valore di *speed-up* con l'aumentare del numero di processori e tenendo fissa la dimensione totale del problema che si sta valutando. Nello specifico, si indica con $E(p)$ e si calcola come segue:

$$E(p) = \frac{S(p)}{p}$$

Lo scopo è quello di ridurre la porzione di lavoro svolto da ogni core con l'aumentare di questi, in modo da migliorare il tempo totale d'esecuzione mano a mano che si aggiungono processori. Di conseguenza, se un programma presentasse uno *speed-up* pressoché lineare, si potrebbe già prevedere una buona efficienza nella scalabilità forte. Nei grafici 1.3 e 1.4 sono mostrati rispettivamente un esempio di misurazione del tempo di un programma e la relativa Strong Scaling Efficiency.

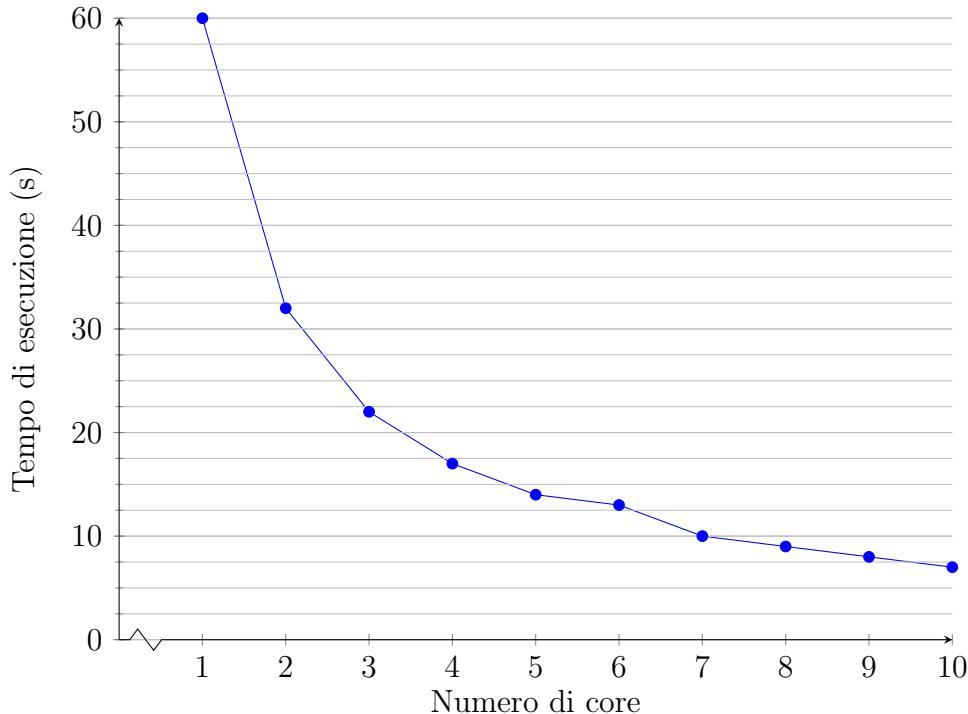


Figura 1.3: esempio di tempi per il calcolo della scalabilità forte

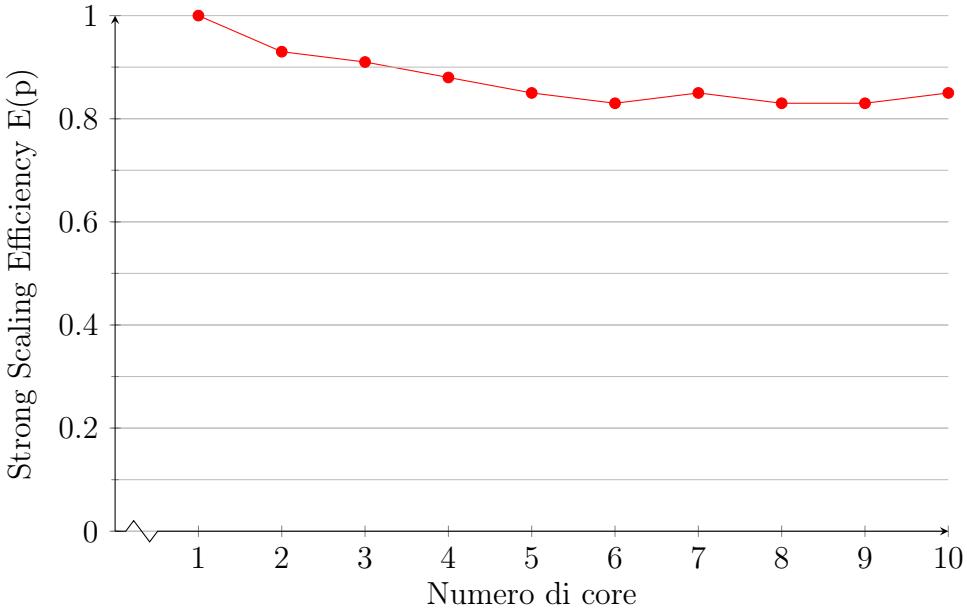


Figura 1.4: esempio di Strong Scaling Efficiency quando lo *speed-up* è quasi lineare

La Weak Scaling Efficiency - o scalabilità debole - interpreta la capacità di scala di un programma mantenendo fisso il carico di lavoro di ogni processore, aumentando di conseguenza la dimensione totale del problema con il crescere del numero p di processori. Si indica con $W(p)$ e si calcola come segue:

$$W(p) = \frac{T_1}{T_p}$$

dove T_1 rappresenta il tempo necessario per completare 1 unità di lavoro con 1 processore, mentre T_p è quello necessario per completare p unità di lavoro con p processori.

L'obiettivo della scalabilità debole è quello di controllare se un problema può essere risolto in dimensioni più grandi nello stesso lasso di tempo; il carico di lavoro totale del problema aumenta proporzionalmente con il crescere del numero p di processori. A differenza della scalabilità forte, quella debole non è strettamente correlata allo *speed-up*; di fatti, esso non compare nella formula poiché la Weak Scaling Efficiency mira al mantenimento dei tempi con il crescere dell'input di un problema, piuttosto che al miglioramento di essi. Nei grafici 1.5 e 1.6 sono mostrati rispettivamente un esempio di misurazione del tempo di un programma e la relativa Weak Scaling Efficiency.

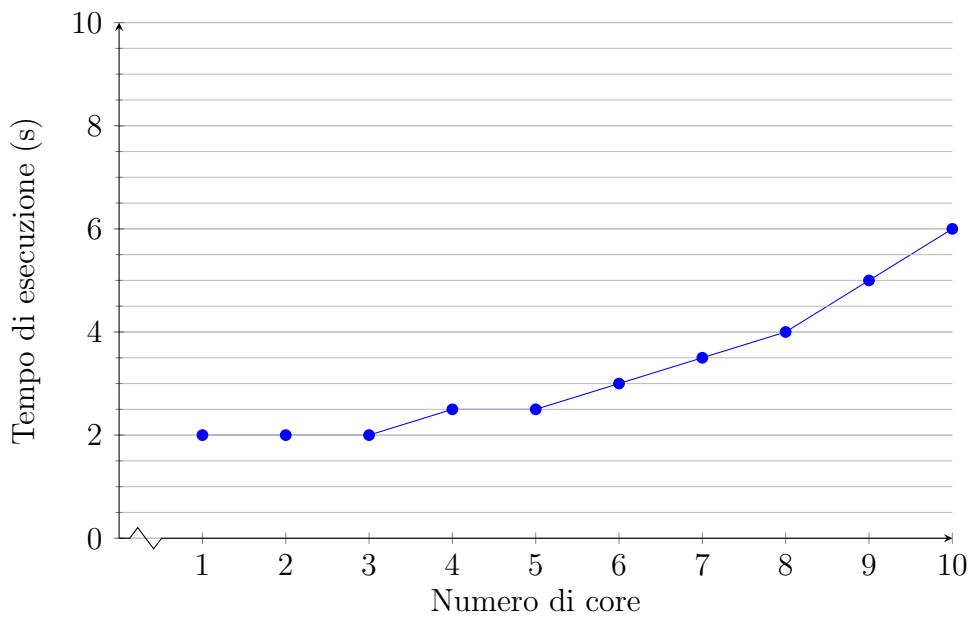


Figura 1.5: esempio di tempi per il calcolo della scalabilità debole

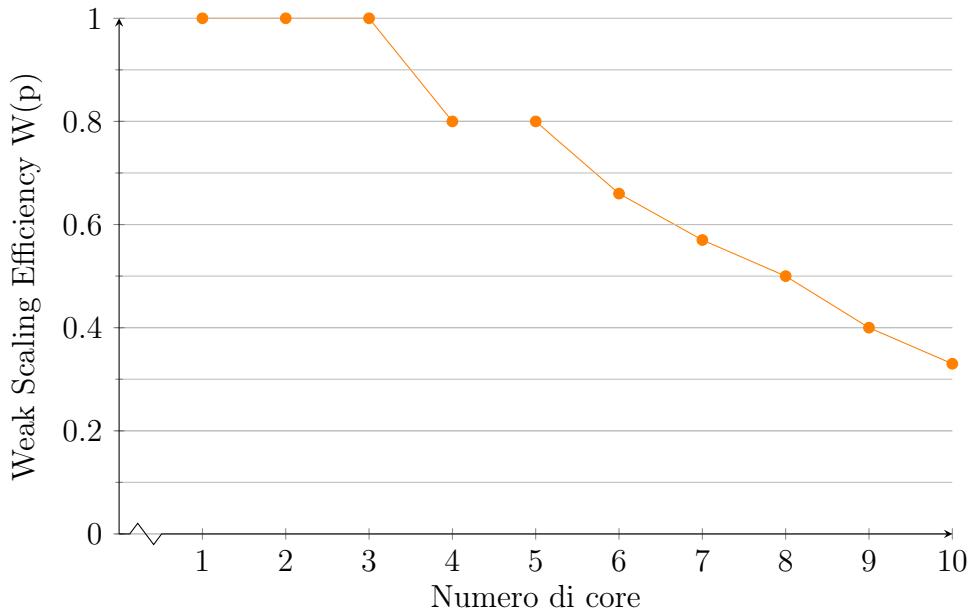


Figura 1.6: Weak Scaling Efficiency relativo ai tempi della figura 1.5

Capitolo 2

Il problema dell'inviluppo convesso

In questo capitolo viene introdotto il problema del calcolo di un inviluppo convesso. Nello specifico, si discute dei campi nei quali può essere utilizzato e si elencano alcune versioni esistenti.

2.1 L'algoritmo

In matematica, si definisce inviluppo convesso (o talvolta involucro convesso) di un sottoinsieme \mathbf{I} di uno spazio vettoriale reale, l'intersezione di tutti gli insiemi convessi che contengono \mathbf{I} [7].

Nello spazio vettoriale \mathbf{R}^2 , l'inviluppo convesso di un insieme di punti \mathbf{P} corrisponde al poligono convesso di area minima che contiene ogni punto di \mathbf{P} ; un esempio è mostrato nella figura 2.1.

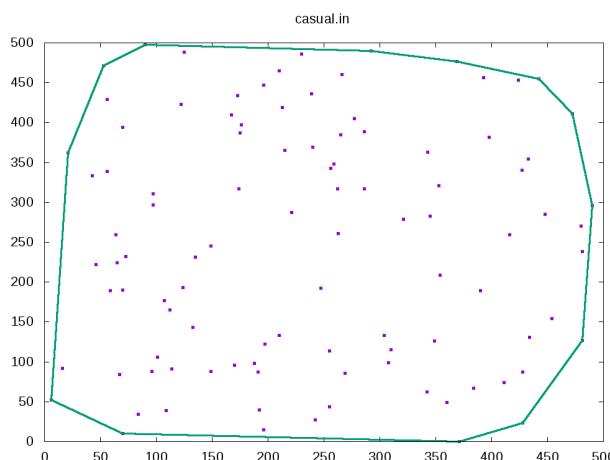


Figura 2.1: Inviluppo convesso di un insieme di 100 punti casuali

Il calcolo dell'inviluppo convesso è sfruttato in diversi ambiti: si applica a discipline come la matematica, la geometria, ma anche la ricerca operativa, la visione artificiale e molto altro; ad esempio, può essere utile nella segmentazione d'istanze di una rete neurale convoluzionale, ovvero evidenziando i bordi di un elemento in un'immagine riconosciuto da un algoritmo di machine learning. Il termine 'inviluppo convesso' (in inglese 'convex hull') è divenuto standard nel 1938, ma l'algoritmo in sé esiste da molto più tempo: l'involucro convesso di un insieme di punti nel piano appare in una lettera di Isaac Newton risalente al 1676 [7].

L'inviluppo convesso di un insieme di punti può essere trovato in diversi modi: infatti esistono diversi algoritmi conosciuti che risolvono tale problema; di seguito, ne vengono elencati e descritti alcuni. L'inviluppo convesso composto da h punti di un insieme di n punti può essere trovato con:

- **Gift Wrapping** (o anche **Jarvis march**) - uno degli algoritmi più semplici anche se meno efficiente, costruisce l'involucro convesso in maniera incrementale con complessità computazionale di $O(nh)$ [4]; è l'algoritmo che è stato implementato in x10 all'interno del progetto di questa tesi, e di conseguenza verrà approfondito successivamente in questo capitolo.
- **Graham scan** - è più sofisticato del Gift Wrapping, ma anche più efficiente: di fatti, ha una complessità computazionale di $O(n \log n)$; tale algoritmo richiede di ordinare i punti di input e, come conseguenza, in caso siano già ordinati, la complessità si riduce a $O(n)$.
- **QuickHull** - ha un approccio di tipo *divide et impera* e ha una complessità computazionale di $O(n \log n)$ che può degenerare a $O(n^2)$ nel caso pessimo.
- **Algoritmo di Chan** - ideato nel 1996, è il più recente degli algoritmi che risolvono il problema dell'inviluppo convesso: combina il Gift Wrapping con l'esecuzione del Graham scan su piccoli sottoinsiemi dell'input; ciò implica un miglioramento della complessità al tempo $O(n \log h)$

Per questa tesi è stato scelto il Gift Wrapping come algoritmo da parallelizzare in quanto si vuole evidenziare l'efficacia del calcolo parallelo su procedimenti relativamente complessi.

2.2 Gift Wrapping

Come anticipato, Gift Wrapping trova gli h vertici dell'inviluppo convesso di un insieme di n punti in maniera incrementale, ovvero esegue h iterazioni in ognuna delle quali determina uno dei punti del risultato finale. Siccome per

ogni iterazione viene letto ognuno degli n punti dell'insieme, la complessità dell'intero procedimento è a $O(nh)$. Nel caso in cui l'insieme di input sia già un involucro convesso, ogni vertice appartiene anche al risultato finale dell'algoritmo: tale situazione costituisce il caso pessimo per il Gift Wrapping, con la quale si nota un cambiamento della complessità che peggiora a $O(n^2)$. L'involucro convesso viene costruito partendo da uno dei punti estremi dell'insieme di punti iniziale, ovvero uno dei vertici con coordinata minima o massima a scelta tra ascisse e ordinate, per poi procedere in senso orario o antiorario; per convenienza, in questa implementazione, consideriamo come punto di partenza il vertice con ascissa minore, cioè quello più a sinistra di tutti, il quale appartiene sicuramente all'involucro convesso, e, come senso di procedimento, quello orario. Sia p l'array monodimensionale contenente l'insieme di punti di input: ad ogni iterazione dell'algoritmo, partendo dall'ultimo vertice inserito $p[\text{cur}]$, viene aggiunto $p[\text{next}]$ il vertice successivo dell'involucro convesso; ciò avviene facendo in modo che la tripletta $p[\text{cur}] - p[\text{next}] - p[j]$ curvi verso destra per ogni j : in figura 2.2 sono illustrati i tipi di triplettre di punti possibili.

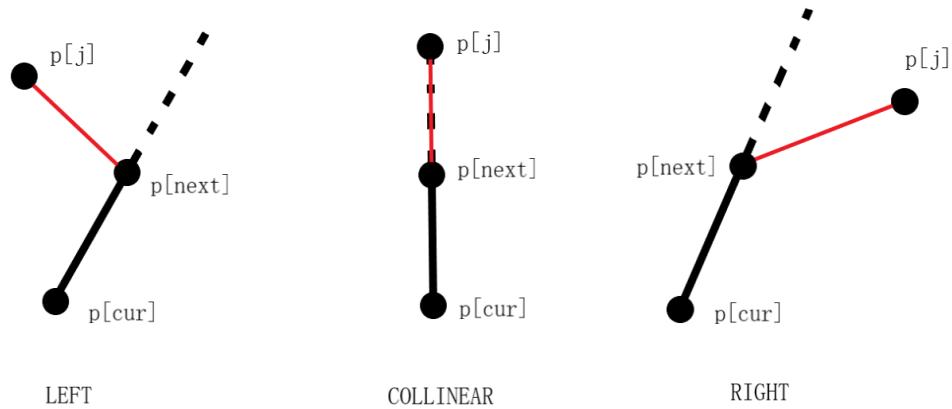


Figura 2.2: Triplettre di punti: curva a sinistra, collineare e destra.

Per determinare l'indice **next** vengono controllati tutti i punti $p[j]$ appartenenti all'insieme di input; nello specifico, si parte dal punto successivo a quello corrente, dopo di che si iterano tutti i punti j : se il vertice j in questione crea una curva a sinistra nella spezzata $p[\text{cur}] - p[\text{next}] - p[j]$, allora diventa il nuovo candidato a punto successivo.

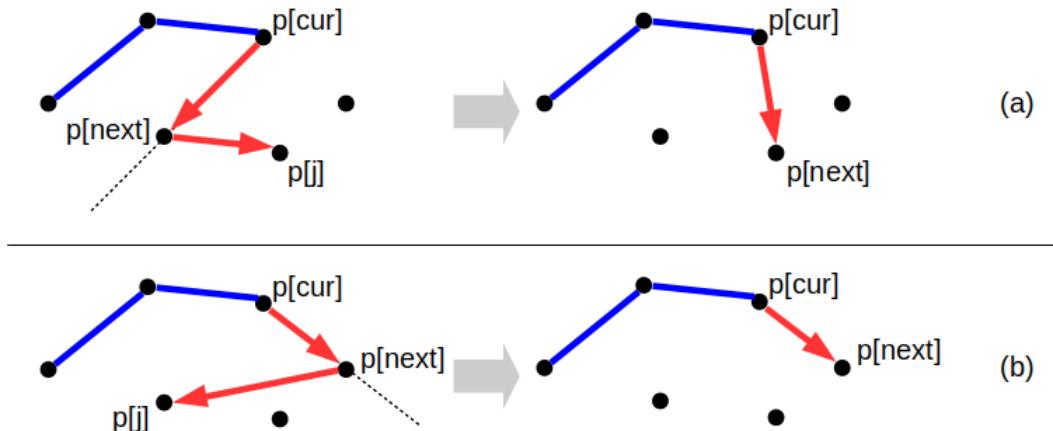


Figura 2.3: Determinazione del punto successivo del poligono convesso [5].

Analizzando la figura 2.3: al passo (a) la tripletta di punti considerata crea una curva a sinistra, di conseguenza il punto `next` viene sostituito da `j`; invece, al passo (b), la spezzata curva verso destra, quindi il vertice `next` non viene cambiato.

Iterando in questa maniera si riesce a costruire il poligono convesso che contiene tutti i punti in input. L'algoritmo termina quando il punto `next` trovato alla fine di un ciclo corrisponde a quello iniziale, ovvero quello più a sinistra. L'intero procedimento può essere descritto con il seguente pseudocodice [5].

Algorithm: Gift Wrapping

Data: $P[]$ is the set of points

Result: $H[]$ is the set of corners of the convex hull of $P[]$

$cur = leftmost = \text{index of leftmost point in } P[0 \dots n - 1]$

do

```

    append  $P[cur]$  to  $H[]$ 
     $next = (cur + 1) \% n$ 
    for  $j = 0$  to  $n - 1$  do
        if  $P[cur]-P[next]-P[j]$  turns left then
             $next = j$ 
        end
    end
     $cur = next$ 
while  $cur \neq leftmost$ 

```

return $H[]$

Capitolo 3

Analisi di x10

In questa parte verrà analizzato il linguaggio x10: in primo luogo, verranno forniti dati storici sulla nascita e sullo sviluppo di x10 definendone le componenti e le potenzialità; in secondo luogo, verrà esposto l'elaborato mostrandone il codice e spiegandone le caratteristiche.

3.1 Nascita e sviluppo del linguaggio x10

La nascita dell'High Performance Computing ha portato molti vantaggi, tra i quali il principale è quello dell'incremento di velocità d'esecuzione dei programmi; tuttavia, non sono mancate delle problematiche: chi lavora in tale ambito deve sapere come dividere il lavoro disponibile in blocchi che possano essere eseguiti simultaneamente, senza incorrere in situazioni che bloccino i computer, come il *deadlock*¹. Un tentativo di soluzione del problema è stato il passaggio al modello a memoria frammentata, in cui più processori si dividono i dati in memorie separate e interagiscono tra di loro grazie ad un sistema di scambio di messaggi: uno dei protocolli più utilizzati che implementa il *message-passing* è MPI (*Message Passing Interface*). Tale modello ha portato ad una perdita di produttività dei programmatore: il formato *message-passing* è integrato su un linguaggio ospite e il programmatore deve gestire l'interazione tra i processori così come lo scambio dei dati; inoltre, grandi strutture dati, come array distribuiti o grafi, devono essere pensate come frammentate in diversi nodi, quando concettualmente sono intese per essere unitarie [6].

La nascita del modello PGAS (*Partitioned Global Address Space*) è stata la soluzione naturale per le problematiche di produttività: il programmatore è portato a pensare ad una computazione eseguita per ciascun processore, ma

¹Deadlock, indica una situazione in cui due o più processi o azioni si bloccano a vicenda, aspettando che uno esegua una certa azione che serve all'altro e viceversa.

con un'unica area di memoria condivisa comune; in più, i processori vengono sincronizzati automaticamente da un insieme di barriere globali. Il linguaggio x10 appartiene a questa classe e la estende grazie alle sue componenti asincrone rientrando quindi nel modello APGAS (*Asynchronous PGAS*).

Il progetto High Productivity Computing Systems portato avanti dal *DARPA*² aveva come scopo quello di sviluppare linguaggi che riescano a conseguire alte performance e produttività: tre linguaggi che si fecero strada all'interno del progetto furono x10, Fortress e Chapel; x10, prodotto da IBM, è quello che ha avuto più successo [1].

3.2 Componenti

X10 è un linguaggio staticamente tipizzato e orientato a oggetti; similmente a Java o altri linguaggi, x10 è un linguaggio basato su classi e interfacce, che consente anche ereditarietà e polimorfismo. Nel listato 3.1 viene fornito un semplice esempio di classe con campi, costruttori e metodi.

```
public class DoublePoint {

    var x:double;
    var y:double;

    def this(p:DoublePoint) { x = p.getX(); y = p.getY(); }
    def this(px:double, py:double) { x = px; y = py; }
    def this() { x = 0; y = 0; }

    public def getX() { return x; }

    public def getY() { return y; }

    public def setX(v:double) { this.x = v; }

    public def setY(v:double) { this.y = v; }

}
```

Listato 3.1: Esempio di classe in x10 che rappresenta un punto nel piano con coordinate di tipo Double.

²Defense Advanced Research Projects Agency, agenzia degli USA responsabile per lo sviluppo di tecnologie per utilizzo nel campo militare.

Nonostante estenda dei linguaggi sequenziali, il punto forte di x10 sono le sue componenti parallele. In termini di design del linguaggio parallelo, l'implementazione di componenti di concorrenza e distribuzione è stata fondamentale.

3.2.1 Activity

I thread di x10 vengono chiamati *Activity*: un qualsiasi programma x10 parte con l'invocazione del metodo `main`, il quale viene lanciato con una *Activity* iniziale, detta *root Activity*. Le computazioni x10 possono avere una o più *Activity* concorrenti che eseguono allo stesso momento; il comando per la loro creazione è `async {statements();}` viene lanciata una *Activity* che esegue il contenuto delle parentesi graffe in maniera indipendente, mentre l'*Activity* che ha lanciato il comando salta il blocco asincrono e riprende la sua esecuzione dalle istruzioni successive. Il corpo di una *Activity* è soggetto ad una restrizione per cui deve essere interpretabile come un metodo di tipo `void`: non deve avere un valore di ritorno e non può modificare delle variabili di tipo `var`. Nei programmi x10, le *Activity* non sono ordinate in alcun modo: non hanno un numero identificativo come i thread OpenMP o i processi MPI, tuttavia vengono aggiunte all'insieme di *Activity* generate.

Ogni *Activity*, ad eccezione di quella *root*, viene generata dinamicamente da un'altra; di conseguenza, nei programmi x10, esse vanno a formare un albero con radice la *main Activity*; per questo, le computazioni x10 vengono dette '*rooted*'. Ogni *Activity* può essere nei seguenti stati: *running*, *blocked* o *terminated*. Una *Activity* termina quando non ha più istruzioni da eseguire. x10 distingue tra terminazione locale e globale: una *Activity* termina localmente quando finisce l'esecuzione della sua ultima istruzione; invece, termina globalmente quando, oltre ad essere terminata localmente, tutte le *Activity* che può aver generato terminano globalmente [6]. I concetti di terminazione locale e globale sono importanti per il programmatore, in quanto deve essere consapevole del numero di *Activity* in esecuzione e del loro stato per poterle gestire e sincronizzare. Prendendo in esempio il seguente codice:

```
async {statement1();}
      async {statement2();}
```

Supponendo di essere all'interno del metodo `main`, la *root Activity* genera due *child Activity* e, subito dopo, termina localmente. Le due *Activity* figlie eseguono le istruzioni all'interno del rispettivo blocco asincrono e possono richiedere più tempo per finire la loro computazione; quando sia `statement1();` sia `statement2();` terminano localmente, la *root Activity* termina globalmente. Per queste caratteristiche, x10 non permette la creazione di thread 'demoni', ovvero flussi di dati che sopravvivono anche dopo la terminazione della *root*

Activity.

La gestione della concorrenza è fondamentale, e x10 fornisce gli strumenti per riuscire a sincronizzare le *Activity* ed evitare situazioni di *race condition*.

I costrutti principali per la sincronizzazione in x10 sono l'istruzione `finish` e la classe `Clock`. L'istruzione `finish {statements();}` permette di creare un blocco all'interno del quale la terminazione globale viene convertita in locale: l'*Activity* che esegue il contenuto del blocco, una volta terminati gli `statements()`, attende che tutte le *Activity* figlie generate all'interno del blocco terminino. Invece, la classe `Clock` permette di generare delle istanze che possono essere associate alle *Activity*, ad eccezione di quella *root*, registrandole tramite l'istruzione `clocked`:

```
val c:Clock = Clock.make();
async clocked(c){s1();}
async clocked(c){s2();}
```

In questa maniera, i blocchi `s1()` e `s2()` eseguono le loro computazioni in parallelo, ma sono in grado di sincronizzarsi tramite l'istanza `c` di `Clock`; ad esempio, con il comando `c.advance()` viene creata una barriera per tutte le *Activity* che sono registrate all'istanza `c`.

Per evitare situazioni di *race condition*, x10 mette a disposizione un comando con il quale si garantisce l'atomicità delle istruzioni, il costrutto `atomic`. Con `atomic {statements();}` viene creato un blocco che viene eseguito sequenzialmente come se fosse un'unica istruzione atomica, nei confronti di altri blocchi `atomic` eseguiti dalle altre *Activity*. Per via di queste caratteristiche i blocchi `atomic` sono soggetti ad una serie di restrizioni; il corpo di un'istruzione `atomic` non può generare *Activity*, non può utilizzare direttive bloccanti come `finish` o `Clock.advance()`, e non può utilizzare l'espressione `at` per cambiare di *Place*.

3.2.2 Place

Per implementare la distribuzione dei dati, x10 utilizza delle partizioni fisiche di memoria che vengono chiamate *Place*: un *Place* può essere considerato come un deposito per dati che introduce il concetto di 'località'. Le *Activity* che eseguono all'interno di un *Place* possono accedere ai dati locali con facilità, mentre sono previsti dei costi di comunicazione per l'accesso a dati remoti, cioè a dati localizzati in altri *Place*. La classe `x10.lang.Place` fornisce una serie di metodi utili per gestirne le istanze: ad esempio, il metodo `Place.places()` ritorna l'insieme di *Place* in esecuzione, come istanza di `PlaceGroup` (un array di `Place`), per poterli iterare.

Ogni programma viene lanciato con un numero `n` di *Place* scelto a priori, settando la variabile d'ambiente `X10_NPLACES` oppure con l'opzione `-np n` da riga di comando. Essi sono numerati a partire da 0 e il rispettivo numero

identificativo è contenuto nel campo `place.id`; il metodo `Place.numPlaces()` ritorna il numero totale di `Place` in esecuzione. Ogni programma inizia con il metodo `main` che esegue all'interno di `Place.FIRST_PLACE`, il quale è il primo della lista `Place.places()`. La variabile `here` è utile per fare riferimento al `Place` corrente: essa punta costantemente al `Place` nel quale sta avvenendo la computazione, similmente a come `this` fa riferimento all'istanza di una classe. Quando un oggetto viene creato la sua istanza viene localizzata nel `Place` corrente e non può cambiare località; tuttavia, le variabili possono essere copiate da un `Place` a un altro tramite il costrutto `at`.

```
at (Expression) {statements();}
```

In questo modo si cambia il `Place` corrente a quello indicato in *Expression*, che deve essere di tipo `Place`; tutte le variabili a cui si fa riferimento in `statements()` vengono copiate in altre con lo stesso nome all'interno del `Place` indicato in *Expression*. Come ogni operazione distribuita, ha un costo elevato: richiede lo scambio di almeno due messaggi tra i `Place` e l'eventuale copia di un insieme di dati di dimensioni variabili. Il costrutto `at(p) {Stmts();}` non è asincrono: non crea una nuova *Activity*, ma deve essere visto come se trasportasse la *Activity* corrente in `p` e vi eseguisse `Stmts()`, per poi riportarla al `Place` di partenza.

La combinazione di `at` e `async` è possibile ed è la base per la programmazione distribuita in x10: sia `at(p) async {Stmts();}` sia `async at(p) {Stmts();}` sono sintatticamente corretti; in generale, è preferibile utilizzare la prima forma in quanto evita di creare una *Activity* solo per valutare `p`. Siccome nella programmazione distribuita è necessaria la presenza di dati persistenti in ogni partizione di memoria, il costrutto `at` non è d'aiuto visto che copia i valori di variabili per poi cancellarli; per questo problema è utile la classe generica `GlobalRef[T]`: essa salva in `here` un riferimento a delle variabili di tipo `T` che permane nella durata di vita del programma. Le variabili di tipo `GlobalRef` non vengono considerate dal costrutto `at` e, infatti, vi si può accedere solamente dal `Place` nel quale vengono create.

Il modo in cui x10 gestisce gli `async` è associandoli ad un gruppo di *thread worker* all'interno di ogni `Place`: il numero di *thread worker* presenti in ciascun `Place` è dato dalla variabile d'ambiente `X10_NTHREADS`. Per avere delle buone prestazioni, sarebbe corretto impostare un *thread worker* per ogni processore disponibile e dividere quest'ultimi equamente: per esempio, supponendo di voler usare un programma con 4 `Place` su una macchina con 12 core, bisognerebbe impostare la variabile `X10_NTHREADS` a 3 (4 `Place` \times 3 *thread worker* per `Place` = 12 core attivi) [2].

3.3 Potenzialità del linguaggio

Grazie a questa struttura, x10 è un linguaggio che permette di parallelizzare codice permettendo al programmatore di costruirsi un'architettura a memoria condivisa oppure a memoria distribuita.

- **Memoria condivisa** - è possibile mantenendo tutti i dati all'interno di un unico *Place* e generando molteplici *Activity*; in questo modo, possono essere implementati diversi protocolli di parallelizzazione. Si può scegliere a priori un numero di *Activity* e generarle dinamicamente, oppure tale numero può essere scelto a run-time in base alle dipendenze algoritmiche. La combinazione di `async` con la classe `Clock` permette di implementare facilmente il protocollo master-slave, nel quale le *Activity* generate si coordinano con la principale.
- **Memoria distribuita** - è possibile distribuendo i dati tra i *Place* messi a disposizione e generando una *Activity* per ciascuno di questi. Come ogni programma a memoria distribuita, tale modello introduce sicuramente dei ritardi per via della comunicazione tra *Place*, ma può avere risultati positivi nel caso di grandi moli di dati o di algoritmi che non necessitano di costante sincronizzazione.

Attualmente, x10 non permette la creazione di *thread* demoni, ovvero *thread* che sopravvivono anche dopo il termine della *root Activity*; tuttavia, rendere l'*Activity* iniziale un demone potrebbe essere un'estensione futura del linguaggio per permettere computazioni che non terminano, come ad esempio quelle di un web server [6].

Infine, un altro punto di forza di x10 è *X10RT*, ovvero una libreria utilizzata dai programmi a memoria distribuita per la comunicazione tra *Place*: al momento, vi sono diverse implementazioni di *X10RT*, tra le quali quella standard è **STANDALONE**, che supporta l'utilizzo di molteplici *Place* su un singolo host; al contrario, l'implementazione **SOCKETS** utilizza il protocollo TCP per consentire molteplici *Place* su host diversi [3].

3.4 Contenuto dell'elaborato

In questa sezione viene illustrato il codice sviluppato ed esposte eventuali considerazioni pensate in fase di progettazione.

Nello specifico, è stato sviluppato un metodo per velocizzare anche la versione seriale del Gift Wrapping: tale algoritmo può eseguire un numero di iterazioni maggiore del necessario poiché aggiunge all'inviluppo convesso anche punti allineati tra di loro; i punti collineari possono essere non considerati, in modo da

ridurre il numero h di vertici dell'inviluppo convesso, e, di conseguenza, anche la complessità $O(nh)$. Questa ottimizzazione è possibile prendendo il punto più lontano in caso di vertici collineari durante la ricerca del punto successivo nell'inviluppo: in questo modo, se parte del poligono convesso è composta da punti allineati, l'algoritmo evita automaticamente quelli intermedi. Nel listato 3.2 viene fornita un'implementazione dell'algoritmo Gift Wrapping in x10 che contiene il miglioramento appena descritto.

```

public static def serialConvexHull(pset:PointSet):PointSet {

    val leftmost:long = leftmost(pset);
    var hull:PointSet = new PointSet(pset.length());
    var cur:long = leftmost;
    var next:long;

    do {
        hull.p(hull.n) = pset.p(cur);
        hull.n++;
        next = (cur + 1) % pset.n;

        for(j in 0..(pset.n-1)) {
            val d:Direction = turn(pset.p(cur), pset.p(next),
                pset.p(j));
            if (d.isCollinear() && distanceBetween(pset.p(cur),
                pset.p(j)) > distanceBetween(pset.p(cur),
                pset.p(next))) {
                next = j;
            }
            if (d.isLeft() ) {
                next = j;
            }
        }
        cur = next;

    } while(leftmost!=cur);

    return new PointSet(hull);
}

```

Listato 3.2: Implementazione seriale dell'algoritmo Gift Wrapping in x10.

Sia per la versione seriale che per quelle parallele, sono stati utilizzati i seguenti metodi:

- `leftmost(pset:PointSet)` - determina l'indice del punto più a sinistra

nell'insieme `pset`;

- `turn(p0:DoublePoint, p1:DoublePoint, p2:DoublePoint)` - determina se la tripletta ordinata di punti $p_0 - p_1 - p_2$ crea una curva verso destra, verso sinistra o dritta;
- `distanceBetween(p0:DoublePoint, p1:DoublePoint)` - calcola la distanza nel piano tra i punti p_0 e p_1 .

Un problema nella parallelizzazione dell'algoritmo Gift Wrapping è la presenza di una dipendenza in ogni sua iterazione: per poter determinare il punto successivo del poligono convesso, Gift Wrapping ha bisogno di conoscere l'ultimo vertice trovato; di conseguenza, non è possibile parallelizzare in maniera semplice il ciclo *do-while* in quanto ogni iterazione dipende dalla precedente. Un modo per parallelizzare tale ciclo consiste nel determinare in anticipo dei punti che appartengono sicuramente all'inviluppo convesso e lanciare una *Activity* per ognuno di questi: purtroppo gli unici vertici con tale caratteristica sono i quattro vertici 'cardinali', cioè quello più in alto, quello più in basso, quello più a destra e quello più a sinistra; tuttavia, dividendo la computazione in questo modo le prestazioni sarebbero limitate in quanto lo *speed-up* rimarrebbe minore o uguale a 4. Sia la versione a memoria condivisa, sia quella a memoria distribuita, sono state implementate parallelizzando ogni iterazione, o meglio il ciclo *for* dell'algoritmo.

3.4.1 Versione a memoria condivisa

Per la compilazione è stato utilizzato il back end Java poiché rende l'esecuzione più veloce rispetto al compilatore C++, il quale ha altri vantaggi. La strategia di parallelizzazione della versione a memoria condivisa consiste nella creazione di p *Activity* ad ogni iterazione, ognuna delle quali opera su una partizione dell'insieme di n punti: all'interno di quest'ultima, ogni *Activity* trova il punto successivo all'ultimo aggiunto nel poligono convesso; successivamente, le p *Activity* terminano per permettere alla *root* di determinare il migliore fra i punti trovati. Il partizionamento dell'insieme di vertici viene fatto a grana grossa: ogni partizione è composta da circa $\frac{n}{p}$ punti; siccome si sta lavorando a memoria condivisa, l'istanza `pset` di `PointSet` non viene divisa, bensì ogni *Activity* opera su un determinato insieme di indici. Nella figura 3.1 viene dato un semplice esempio di partizionamento regolare a grana grossa con 4 core.



Figura 3.1: Partizionamento regolare di un vettore monodimensionale su 4 processori.

Nel caso in cui n non sia un multiplo di p , l'ultima *Activity* si prende carico degli indici rimanenti. Il listato 3.3 mostra come ogni *Activity* trova un candidato per il punto successivo dell'inviluppo convesso e lo aggiunge al rispettivo indice dell'array `candidates`.

```
finish for(id in 0..(activities-1)) {
    async {

        val actualSize:long = id != activities-1 ? localSize :
            localSize + resto;
        var localNext:long = localSize * id;

        for(j in (localSize * id)..(localSize * id + actualSize - 1)) {

            val d:Direction = turn(pset.p(cur), pset.p(localNext),
                pset.p(j));
            if ( d.isCollinear() && distanceBetween(pset.p(cur),
                pset.p(j)) > distanceBetween(pset.p(cur),
                pset.p(localNext)) ) {
                localNext = j;
            }
            if ( d.isLeft() ) {
                localNext = j;
            }
        }
        candidates.raw()(id) = localNext;
    }
}
```

Listato 3.3: Parte dell'implementazione parallela a memoria condivisa dell'algoritmo Gift Wrapping in x10.

Una volta uscite dal blocco asincrono, le *Activity* si sincronizzano grazie alla direttiva `finish`. In seguito, la *root Activity* itera l'array `candidates` per determinare quale dei vertici trovati sia effettivamente quello successivo del poligono convesso.

Un aspetto negativo dell'implementazione consiste nella generazione di `p Activity` ad ogni iterazione del ciclo `do while`: l'istruzione `async` ha un costo, anche

se minimo, che può introdurre dei ritardi nel programma in caso di chiamata ripetuta numerosamente. In x10, la sincronizzazione dei *thread* può avvenire con l'utilizzo della classe **Clock**; tuttavia, nell'implementazione parallela del Gift Wrapping risulta sconveniente poiché bisognerebbe sincronizzare le *Activity* due volte per ciclo: una dopo aver trovato i vertici candidati a successivi, e un'altra dopo che la *root Activity* abbia individuato l'effettivo prossimo vertice, in modo da bloccare l'inizio della iterazione seguente.

3.4.2 Versione a memoria distribuita

Per la compilazione è stato utilizzato il back end di C++ in quanto permette l'ottimizzazione delle comunicazioni remote tra *Place* con l'opzione **-OPTIMIZE_COMMUNICATIONS** da riga di comando: in tale modo, i tempi peggiorano con lo stesso dominio rispetto alla compilazione con back end Java; tuttavia, si ottengono risultati più efficienti nella valutazione delle prestazioni. La strategia di parallelizzazione della versione a memoria distribuita è la stessa utilizzata con architettura a memoria condivisa: ad ogni iterazione vengono generate *p Activity* che operano su una partizione dell'insieme totale di punti, con la differenza che essi vengono distribuiti tra *p Place*.

In x10, esistono diversi modi per accedere alla memoria locale di un *Place* e per distribuire dei dati in remoto: uno di questi consiste nell'utilizzo dei *DistArray*, cioè array distribuiti. Nel listato 3.4 si mostra come viene effettuata la distribuzione dell'insieme di punti iniziale.

```
val blk:Dist = Dist.makeBlock(Region.make(0, pset.n-1));
val dist:DistArray[DoublePoint] = DistArray.make[DoublePoint](blk,
  ([i]:Point) => new DoublePoint(pset.p(i)) );
```

Listato 3.4: Creazione ed inizializzazione di un array distribuito

L'array distribuito è salvato nella variabile **dist** e il partizionamento dei dati viene effettuato a grana grossa e regolare grazie alla funzione **Dist.makeBlock**: la classe **Dist** mette a disposizione una serie di metodi che permettono di gestire la distribuzione dei dati arbitrariamente, e tale metodo rappresenta il caso migliore per il Gift Wrapping.

Il listato 3.5 mostra come ogni *Activity* agisce localmente in un *Place* diverso per trovare un candidato per il punto successivo dell'inviluppo convesso, per poi copiarne le coordinate in un array remoto mantenuto nel *Place* principale. Grazie al campo **home** della classe **GlobalRef** non è necessario tener traccia del *Place* nel quale è salvata l'istanza **candidates**: esso punta direttamente al *Place* remoto nel quale è stato creato l'array.

```

finish for (p in Place.places()) {
    async at(p) {

        val local = dist.getLocalPortion().raw();
        var localNext:long = 0;

        for(j in 0..(local.size-1)) {
            val d:Direction = turn(lastPoint, local(localNext), local(j));
            if ( d.isCollinear() && distanceBetween(lastPoint, local(j))
                > distanceBetween(lastPoint, local(localNext)) ) {
                localNext = j;
            }
            if ( d.isLeft() ) {
                localNext = j;
            }
        }

        val nextPoint = localNext;
        val index:long = here.id;

        at (candidates.home) {
            candidates.getLocalOrCopy().raw()(index) = local(nextPoint);
        }
    }
}

```

Listato 3.5: Parte dell'implementazione parallela a memoria distribuita dell'algoritmo Gift Wrapping in x10.

La versione a memoria distribuita non è tanto diversa da quella a memoria condivisa; invece, implementando l'algoritmo Gift Wrapping in OpenMP e MPI, è necessario cambiare la strategia di parallelizzazione in quanto le architetture sottostanti costituiscono dei limiti: x10 permette di ottenere, più o meno, la stessa implementazione di un algoritmo sia a memoria condivisa, sia a memoria distribuita, cambiandone esclusivamente la struttura.

Capitolo 4

Valutazione delle prestazioni

In questo capitolo viene analizzate le prestazioni delle due implementazioni parallele dell'algoritmo Gift Wrapping: nello specifico, vengono evidenziati lo *speed-up* e l'efficienza misurati. Le misurazioni sono state effettuate su una macchina con sistema operativo Linux che dispone di 12 core logici; perciò, sono stati presi i tempi dei programmi fino ad un massimo di 15 *Activity*, per studiare anche il caso in cui il numero di *thread* supera quello dei processori. Sia per la versione a memoria condivisa, sia per quella a memoria distribuita, l'input utilizzato rappresenta il caso pessimo per il Gift Wrapping, ovvero un insieme di punti che forma già un poligono convesso; in particolare, è stato utilizzato un numero di vertici diverso, a seconda della misurazione, il quale forma una circonferenza nel piano: è riportato un esempio nella figura 4.1.

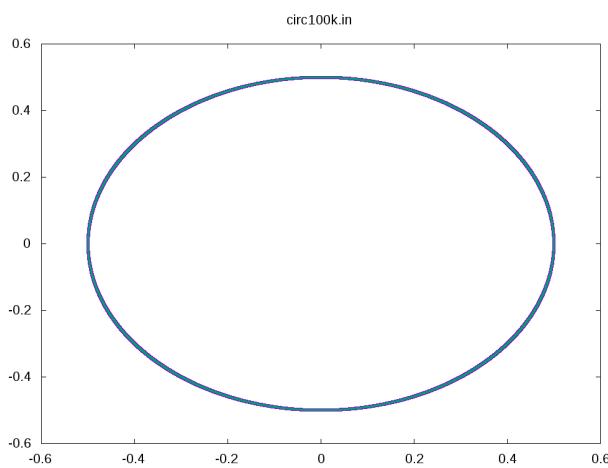


Figura 4.1: Esempio di input del caso pessimo per il Gift Wrapping

Inoltre, la misurazione dei tempi è stata automatizzata grazie all'utilizzo di

script bash: per ogni numero di *thread*, sono state osservate più esecuzioni e ne si è calcolata la media per ottenere un risultato affidabile.

4.1 Versione a memoria condivisa

Per l'implementazione a memoria condivisa sono stati utilizzati degli *script bash* che eseguono il programma con un solo *Place* ed un numero crescente di *Activity*. I risultati ottenuti vengono considerati corretti, in quanto tutti gli inviluppi convessi calcolati presentano lo stesso numero di punti e la stessa area interna. Nel listato 4.1 viene illustrato un esempio dell'output ottenuto, che mostra i risultati ottenuti dall'algoritmo per ogni esecuzione; in particolare, si mostra:

- il numero di *Activity* in esecuzione;
- il nome del file di input utilizzato;
- il numero di vertici appartenenti all'inviluppo convesso;
- l'area dell'inviluppo convesso;
- i tempi d'esecuzione in millisecondi e secondi.

```
Activities = 1
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 112685
Elapsed time (s): 112
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 122143
Elapsed time (s): 122
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 113309
Elapsed time (s): 113
```

```

Activities = 2
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 70845
Elapsed time (s): 70
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 71290
Elapsed time (s): 71
Shared memory: parallel convex hull of input circ100k.in:
Convex hull of 100000 points of 100000
Total volume: 0.7853981599528191
Elapsed time (ms): 67621
Elapsed time (s): 67

```

Listato 4.1: Output del programma a memoria condivisa.

4.1.1 Speed-up

Per calcolare lo *speed-up*, è necessario avere i tempi d'esecuzione del programma con un numero crescente di *thread* e stesso dominio: il grafico 4.2 mostra tale misurazione, la quale è stata effettuata con un numero di *Activity* da 1 a 15 e con un input di 100.000 punti.

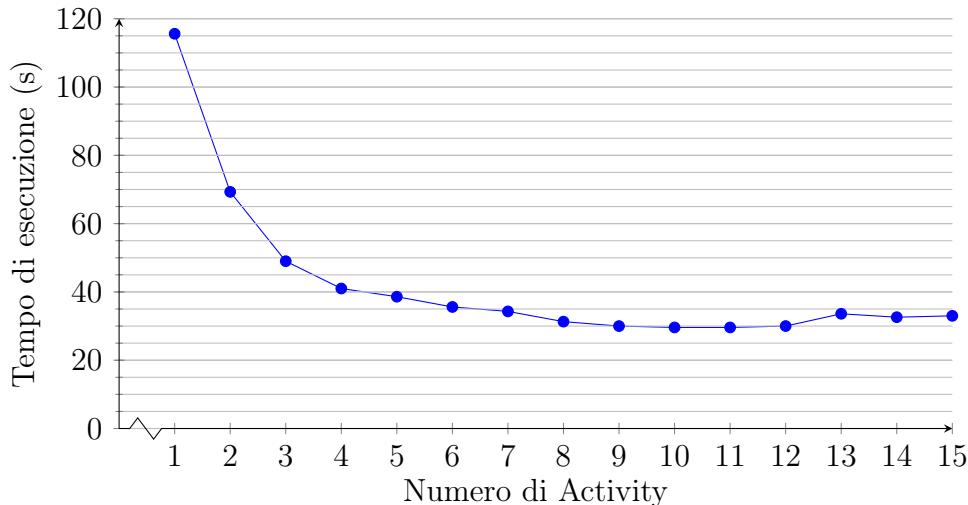


Figura 4.2: Grafico dei tempi dell'implementazione a memoria condivisa

Nel grafico 4.2 si nota un progressivo miglioramento dei tempi che si interrompe con un numero di processori p superiore a 12: questa particolarità è dovuta al superamento del numero di core logici messi a disposizione dalla macchina su cui sono state eseguite le misurazioni; poiché, in questo caso, il massimo di processori logici è 12, con un numero di *thread* maggiore o uguale a 13, quelli 'in più' dovranno attendere che i core abbiano terminato le loro computazioni prima di poter iniziare le proprie.

Con questi tempi viene calcolato lo *speed-up* mostrato nel grafico 4.3.

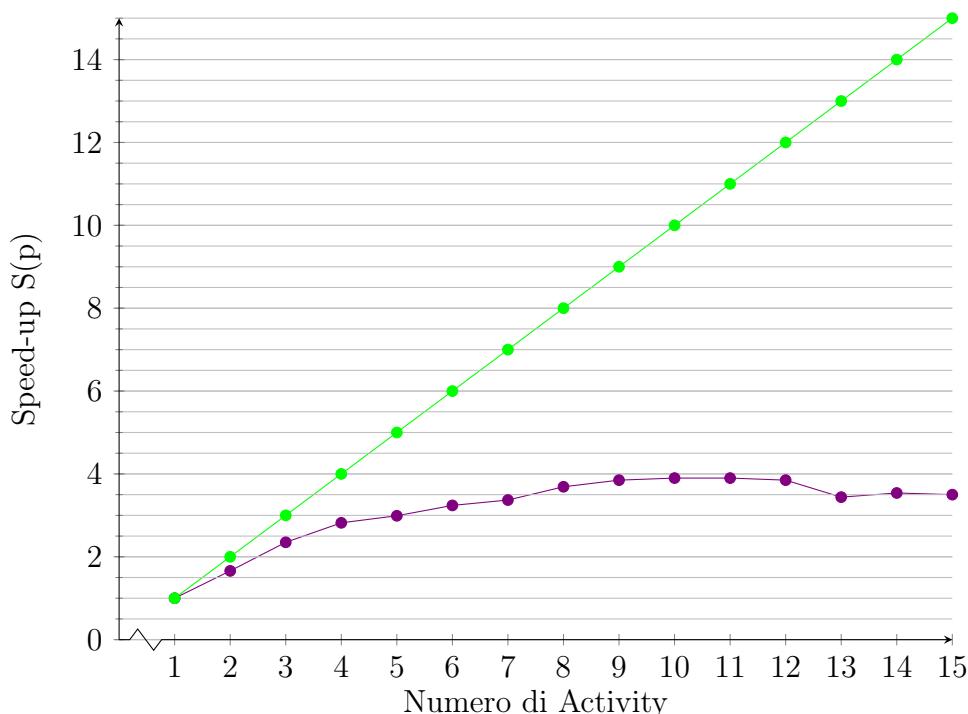


Figura 4.3: *Speed-up* dell'implementazione a memoria condivisa (viola) sui tempi in figura 4.2 a confronto con uno *speed-up* lineare (verde)

Come si può notare, lo *speed-up* ottenuto non è lineare: probabilmente tale fenomeno è dovuto alla sincronizzazione delle *Activity*; sebbene le *Activity* impieghino circa lo stesso tempo per terminare un ciclo, la maggior parte del tempo viene spesa dalle *Activity* più leggermente più veloci alla fine del blocco **finish** in attesa delle altre più lente, e l'attesa aumenta con il crescere delle *Activity*.

Un metodo per migliorare questo ritardo potrebbe essere quello di cambiare il tipo di partizione effettuata: con un partizionamento a grana grossa il tempo totale di esecuzione di una computazione diventa quello del *thread* più lento;

invece, facendo una partizione a grana fine, cioè dividendo l'input in parti più piccole, si potrebbe ridurre il tempo richiesto per la sincronizzazione.

4.1.2 Strong Scaling Efficiency

Osservando lo *speed-up* in figura 4.3, si può già prevedere che la scalabilità forte non sia ottimale. Nel grafico 4.4 viene mostrata la Strong Scaling Efficiency calcolata sullo *speed-up* precedentemente analizzato.

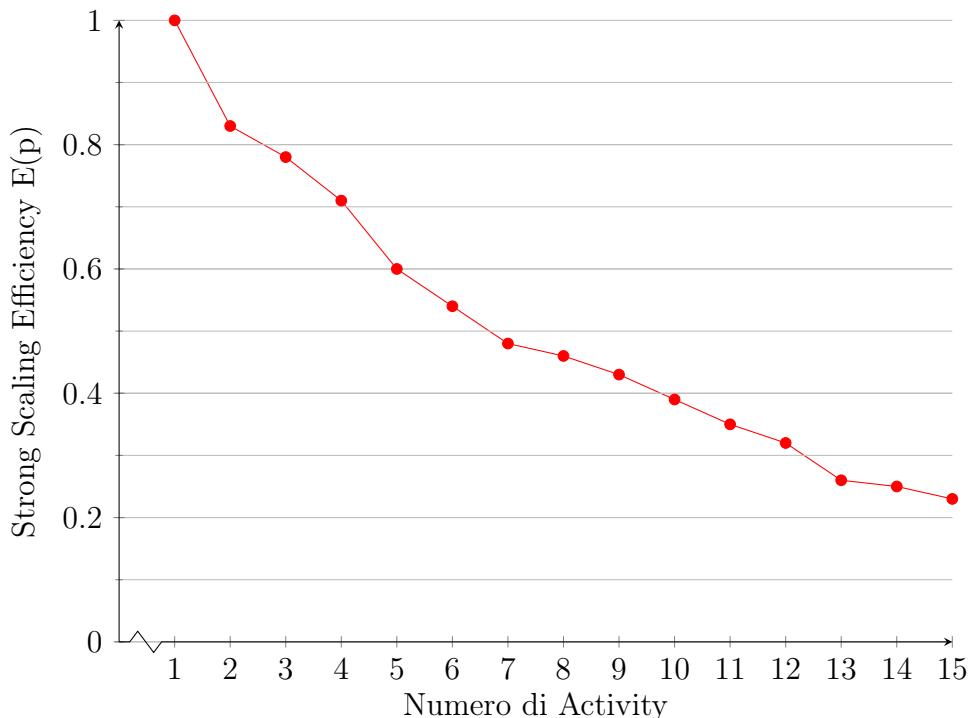


Figura 4.4: Strong Scaling Efficiency dell'implementazione a memoria condivisa

L'efficienza decresce quasi linearmente, mostrando come l'aumentare delle *Activity* rallenti la loro sincronizzazione: i tempi sono migliori rispetto alla versione seriale dell'algoritmo, ma le prestazioni ottenute non soddisfano lo scopo principale della scalabilità forte, cioè ridurre il carico assegnato ad ogni processore con l'aumentare di questi.

4.1.3 Weak Scaling Efficiency

Per calcolare la scalabilità debole è necessario effettuare un'altra valutazione dei tempi: poiché è necessario proporzionare il dominio del problema al numero

di processori, è stata creata una serie di file di input che rappresentano il caso pessimo per Gift Wrapping, ma con un numero crescente di punti. In particolare, con 1 processore l'input è costituito da 10.000 punti e con p processori è costituito da $\sqrt{p} \times 10.000$ punti; questo perchè, dato che si sta considerando il caso pessimo con complessità $O(n^2)$, il lavoro di ogni *Activity* è $\frac{n^2}{p}$, e per mantenerlo costante bisogna utilizzare $\sqrt{p} \times 10.000$ punti ($\frac{(\sqrt{p} \times 10.000)^2}{p} = 10.000^2$). La misurazione è mostrata nel grafico 4.5.

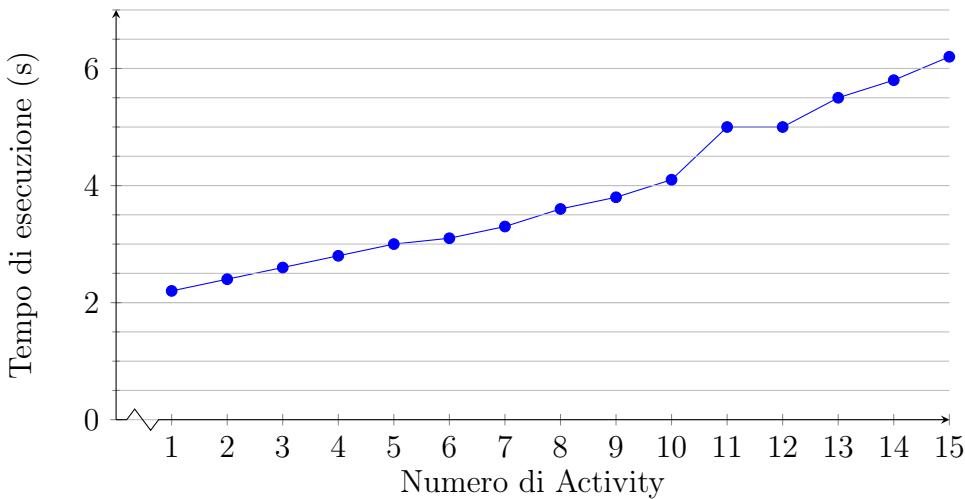


Figura 4.5: Grafico dei tempi dell'implementazione a memoria condivisa per la scalabilità debole

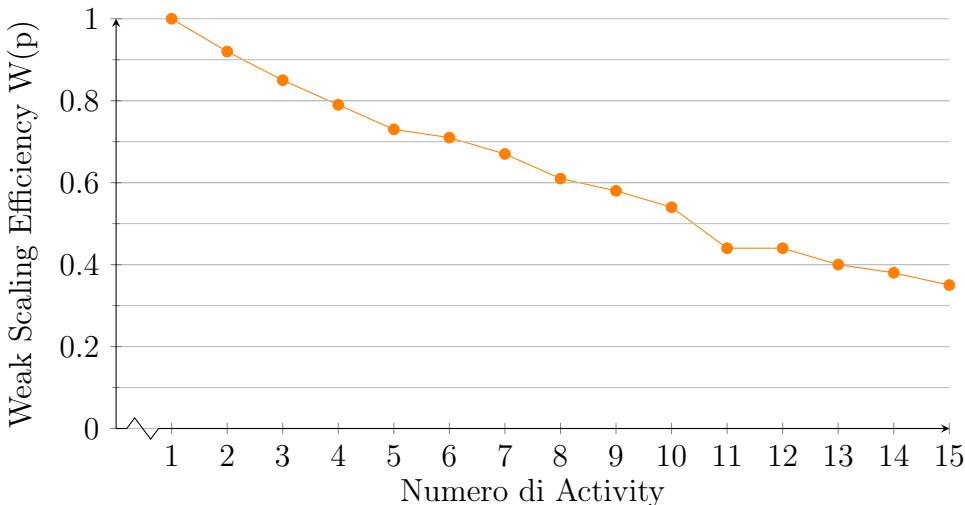


Figura 4.6: Weak Scaling Efficiency dell'implementazione a memoria condivisa

Dal grafico 4.5 si nota che non viene rispettato perfettamente l'obiettivo della scalabilità debole, ovvero eseguire programmi con dominio più grande nello stesso tempo. La misurazione del tempo risulta crescere quasi linearmente, ma non ha una pendenza troppo ripida: la diretta conseguenza è un leggero calo nel calcolo dell'efficienza, la quale viene mostrata nel grafico 4.6.

4.2 Versione a memoria distribuita

Per l'implementazione a memoria distribuita sono stati utilizzati degli *script bash* che eseguono il programma con un numero crescente di *Place*: il programma crea dinamicamente una *Activity* per ogni *Place*, come mostrato nel listato 3.5, di conseguenza non è necessario gestirne il numero da riga di comando. Il tipo di output generato dagli *script* per questa implementazione è il medesimo di quella a memoria condivisa, illustrato nel listato 4.1.

4.2.1 Speed-up

Per la misurazione dei tempi, illustrata nel grafico 4.7, è stato utilizzato un input di 5.000 punti e un numero crescente di *Place* da 1 a 15.

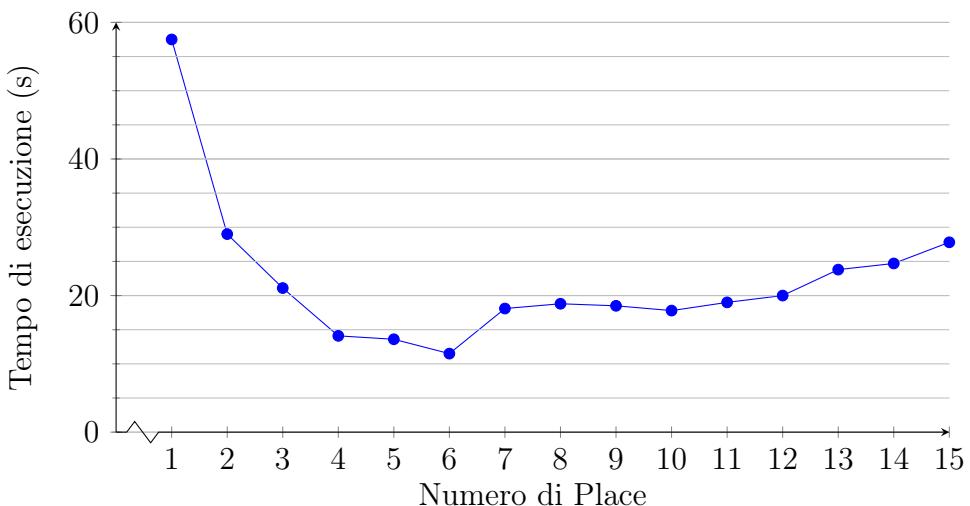


Figura 4.7: Grafico dei tempi dell'implementazione a memoria distribuita

I tempi migliorano fino ad un numero di *Place* uguale 6, per poi peggiorare leggermente: probabilmente, questo fenomeno è dato dall'aumentare dei costi di comunicazione dovuti al crescere dei *Place*; i tempi rimangono migliori della versione sequenziale, ma vengono introdotti dei ritardi mano a mano che si

aggiungono *Place* e *Activity* da sincronizzare.

Con questi tempi viene calcolato lo *speed-up* mostrato nel grafico 4.8.

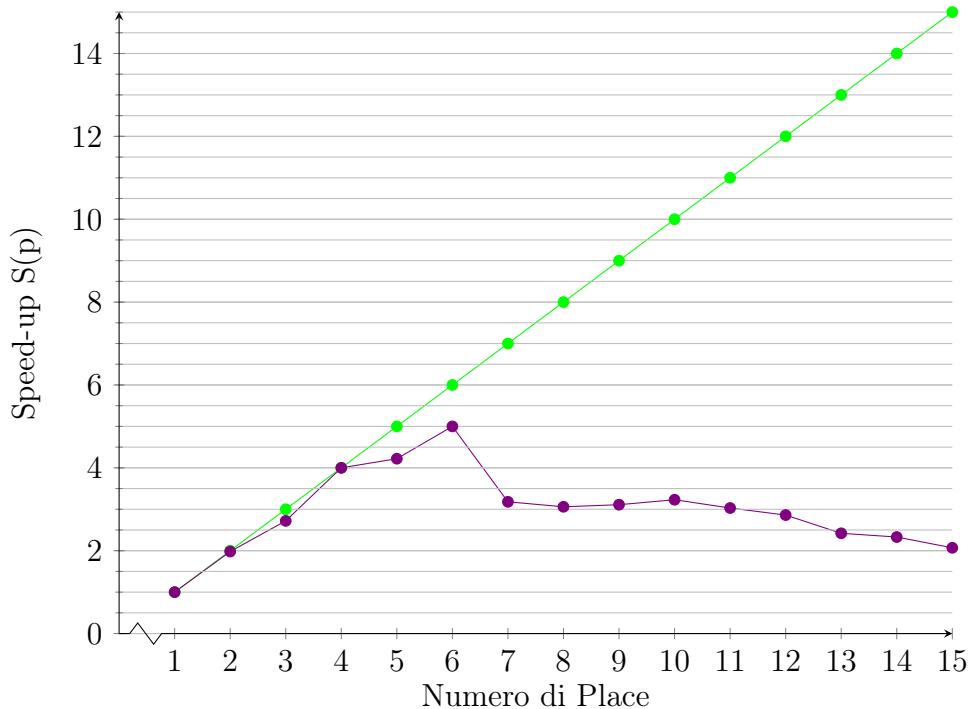


Figura 4.8: *Speed-up* dell’implementazione a memoria distribuita (viola) sui tempi in figura 4.7 a confronto con uno *speed-up* lineare (verde)

Lo *speed-up* ottenuto è quasi lineare per un numero di *Place* p fino a 4 e mantiene un buon livello fino a 6 *Place*. Come si può intuire dal grafico dei tempi, lo *speed-up* decresce con 7 o più *Place*.

4.2.2 Strong Scaling Efficiency

Osservando lo *speed-up* in figura 4.8, si può già prevedere che la scalabilità forte è ottima per un numero di *Place* inferiore a 7. Nel grafico 4.9 viene mostrata la Strong Scaling Efficiency calcolata sullo *speed-up* precedentemente analizzato.

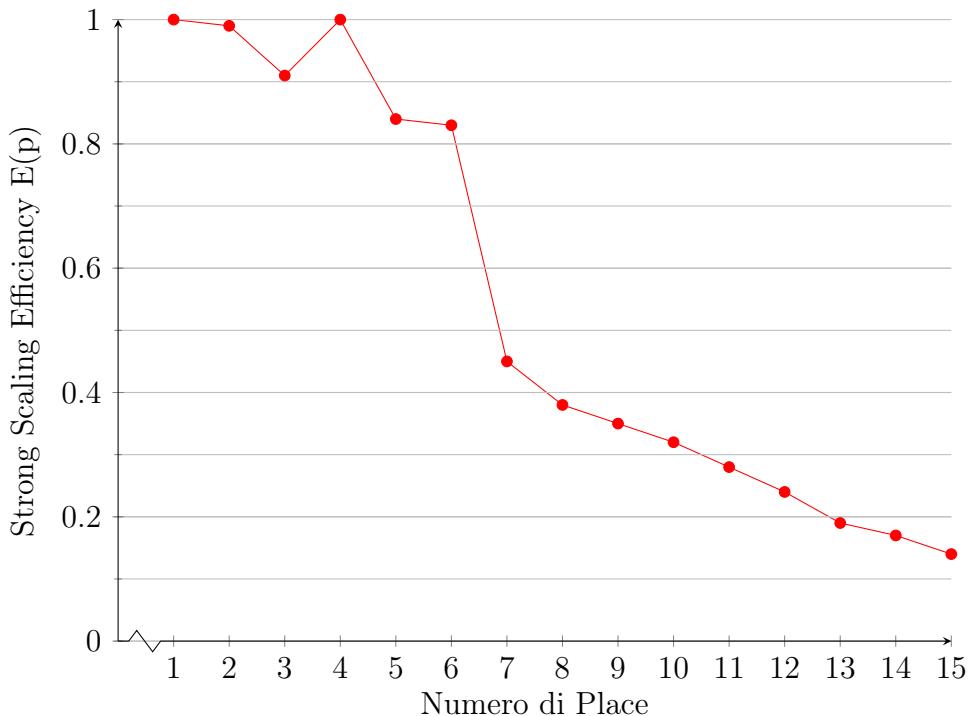


Figura 4.9: Strong Scaling Efficiency dell’implementazione a memoria distribuita

Il grafico della scalabilità forte suggerisce in maniera chiara che il ritardo introdotto dalla sincronizzazione è minimo, ma permette di soddisfare l’obiettivo della Strong Scaling Efficiency solo per un numero limitato di *Place*. Un metodo per ottenere un ulteriore miglioramento utilizzando la distribuzione in x10, potrebbe consistere nell’utilizzo di pochi *Place*, ma con molteplici *Activity* che eseguono localmente all’interno di ognuno; ad esempio, volendo sfruttare 12 core logici a pieno, si potrebbero distribuire dati tra 4 *Place* in ciascuno dei quali operano 3 *Activity*.

4.2.3 Weak Scaling Efficiency

Per calcolare la scalabilità debole è necessario effettuare un’altra valutazione dei tempi, illustrata nel grafico 4.10: la misurazione è stata fatta con un dominio crescente, come per la versione a memoria condivisa; in particolare, con 1 processore l’input è costituito da 1.000 punti e con p processori è costituito da $\sqrt{p} \times 1.000$ punti.

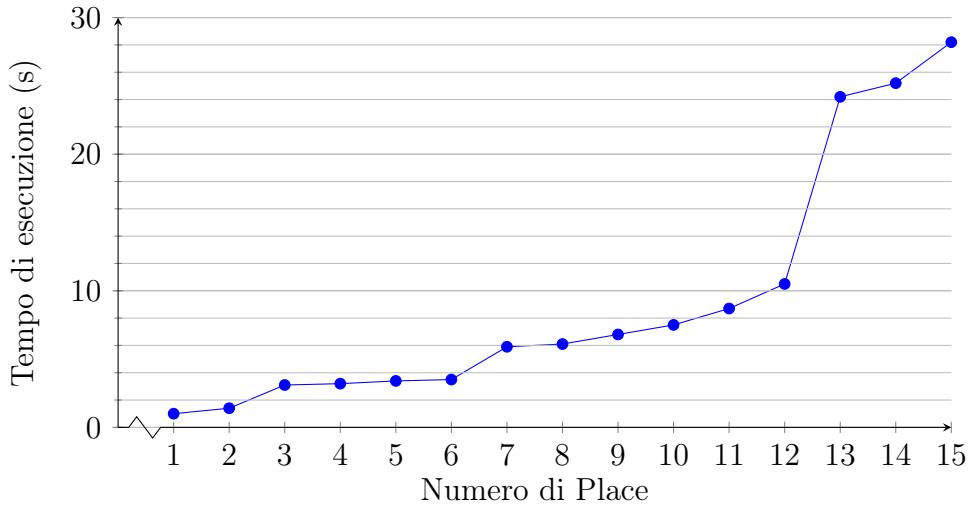


Figura 4.10: Grafico dei tempi dell’implementazione a memoria distribuita per la scalabilità debole

I tempi della scalabilità debole crescono di poco per un numero di *Place* minore o uguale a 13: l’obiettivo della Weak Scaling Efficiency non viene raggiunto neanche in questa implementazione. Per un numero di *Place* superiore a 12, i tempi salgono a picco: motivo per cui l’efficienza mostrata nel grafico 4.11 presenta un netto peggioramento sopra a tale cifra.

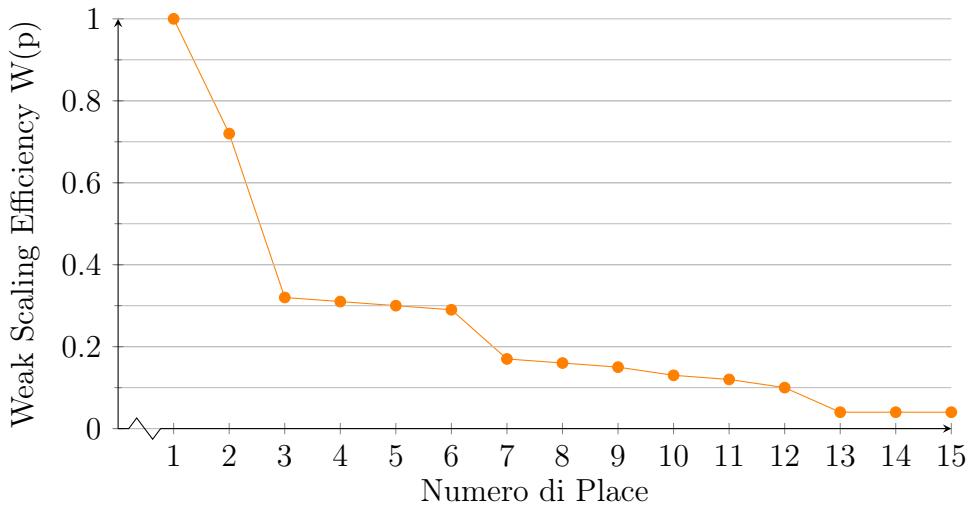


Figura 4.11: Weak Scaling Efficiency dell’implementazione a memoria distribuita

Conclusioni

In questa tesi è stato analizzato il linguaggio x10 evidenziandone le potenzialità e i contributi che può dare all'High Performance Computing grazie alle sue componenti particolari per le architetture parallele; il codice sorgente dell'elaborato è reperibile nel repository pubblico <https://bitbucket.org/SRomagno121/parallelconvexhull/>. Questo lavoro ha preso in esempio un problema e lo ha valutato utilizzando relativamente poche risorse, ovvero 12 processori; pensare di poter applicare una miglioria come la parallelizzazione a problemi di grandi dimensioni in ambiti come l'astrofisica o il machine learning, rende tale disciplina ancora più affascinante di quanto non lo sia già.

X10 è un linguaggio ancora in via di sviluppo; ciò nonostante, presenta molte componenti utili per la parallelizzazione che permettono ai programmatore di operare in maniera produttiva, suddividendo il carico di lavoro in modo veloce e naturale. Inoltre, essendo orientato agli oggetti, può essere la base di partenza per sviluppi futuri di linguaggi o protocolli moderni che consentano di fare ulteriori passi avanti nell'ambito dell'informatica.

È importante ricordare che la scelta dell'architettura sottostante al programma sia un interrogativo fondamentale da porsi prima della progettazione di un elaborato, in quanto memoria condivisa e distribuita apportano vantaggi e svantaggi diversi a seconda del problema che si vuole affrontare: con x10 tale problematica assume un peso molto meno significativo, dato che supporta entrambe le strutture e rende semplice al programmatore l'azione di cambiare dall'una all'altra.

Ringraziamenti

Ringrazio il relatore di questo lavoro, il Prof. Moreno Marzolla, che ha acceso il mio personale interesse nei confronti di questa splendida disciplina e mi ha seguito nella svolgimento dell'elaborato, partendo dal progetto, fino alla stesura della tesi.

Ringrazio i miei genitori, che mi hanno sempre spronato a dare il massimo e mi sono sempre stati vicini durante il mio percorso di studi.

Ringrazio amici e colleghi che hanno contribuito alla stesura di questa tesi e che hanno reso quest'esperienza unica supportandomi durante l'intero percorso.

Ringrazio in particolare Chiara Zammarchi che mi ha aiutato durante i miei studi e che mi ha sostenuto con idee e consigli durante la scrittura di questa tesi.

Bibliografia

- [1] IBM. *The X10 Parallel Programming Language*. 2019. URL: <http://x10-lang.org/>.
- [2] IBM. *X10 Performance Tuning*. 2019. URL: <http://x10-lang.org/documentation/practical-x10-programming/performance-tuning.html>.
- [3] IBM. *X10RT Implementations*. 2019. URL: <http://x10-lang.org/documentation/practical-x10-programming/x10rt-implementations.html>.
- [4] R. A. Jarvis. «On the identification of the convex hull of a finite set of points in the plane». In: *Information Processing Letters* 2 (1973), pp. 18–21.
- [5] Moreno Marzolla. «Progetto di High Performance Computing 2019-2020». In: (2019). URL: <https://www.moreno.marzolla.name/teaching/HPC/convex-hull>.
- [6] Vijay Saraswat et al. *X10 Language Specification. Version 2.6.2*. 2019. URL: <http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf>.
- [7] Wikipedia. *Convex hull*. 2019. URL: https://en.wikipedia.org/wiki/Convex_hull.

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

**ANALISI DELLE PRESTAZIONI
DEL SISTEMA GRAFICO VIDEOCORE IV
APPLICATO AL CALCOLO GENERICO**

Elaborato in
HIGH PERFORMANCE COMPUTING

Relatore
Prof. MORENO MARZOLLA

Presentata da
SIMONE MAGNANI

Anno Accademico 2018 – 2019

Indice

Introduzione	v
1 Il calcolo Parallelo	1
1.1 Classificazione elaboratori	1
1.2 Valutazione di un programma parallelo	2
1.2.1 Speedup	2
1.2.2 Scalabilità forte	3
1.2.3 Scalabilità debole	3
1.3 Sviluppo di programmi paralleli	4
1.3.1 Confronto tra CPU e GPU	4
1.3.2 Esempi di linguaggi per lo sviluppo	4
2 Raspberry Pi	9
2.1 Panoramica	10
2.2 Programmazione GPU	10
2.3 QPULib	11
2.3.1 Confronto tra QPULib e CUDA-C	11
2.3.2 Programmare con QPULib	12
3 Kernel computazionali	15
3.1 Earthquake	15
3.1.1 Modello Earthquake	15
3.1.2 Implementazione QPULib	16
3.2 Moltiplicazione tra matrici	18
3.2.1 Modello Matmul	18
3.2.2 Implementazione QPULib	18
4 Valutazioni delle prestazioni	21
4.1 OpenMP Earthquake	21
4.2 Matmul con Vector data type	22
4.3 Prestazioni QPULib e confronto	23
4.3.1 Scalabilità dato un input costante	23

4.3.2 Scalabilità debole	24
4.3.3 Confronto con altre implementazioni	25
Conclusioni	27
Ringraziamenti	29

Introduzione

Il calcolo parallelo rappresenta una risorsa per ridurre i tempi di esecuzione di un qualsiasi problema numerico.

Storicamente i programmi erano scritti per essere eseguiti sequenzialmente da una sola unità di calcolo, ad esempio una singola CPU. Le singole unità di calcolo però non erano sempre sufficientemente potenti per eseguire, in un tempo consono, una grande quantità di operazioni.

L'idea alla base del calcolo parallelo è di suddividere il lavoro su più unità di calcolo in modo da ridurre il tempo necessario per svolgere la computazione.

Portando un esempio più vicino alla quotidianità di tutti, è come avere un lavoro che necessita di molto tempo per essere svolto, così si decide di suddividere il lavoro in sotto-problemi e assegnare ognuno di questi a una persona.

Il calcolo parallelo mira proprio a una gestione di problemi complessi dividendoli in sotto-problemi, poi facendo in modo che le unità di calcolo designate collaborino ed eseguano la propria parte per trovare la soluzione.

Esistono due macro tipi di unità di calcolo: CPU e GPU; le prime sono più veloci ma con meno possibilità di parallelizzazione, al contrario le GPU sono generalmente più lente ma con un numero di ALU¹ molto maggiore.

In particolare nel capitolo 4 verrà confrontato l'uso della GPU rispetto alla CPU su un Raspberry Pi (capitolo 2): ovvero una scheda *single-board* economica, ma abbastanza potente da essere un buono strumento per un programmatore esperto, e un ottimo strumento per chi vuole avvicinarsi alla programmazione parallela.

¹ Arithmetic-Logic Unit è l'unità di calcolo vera e propria

Capitolo 1

Il calcolo Parallello

La programmazione parallela è una forma ormai ben nota e diffusa di approccio ai problemi numerici, soprattutto in ambiti scientifici dove è necessario lavorare con un'importante mole di dati.

In questo capitolo verranno descritti quali sono le varie classificazioni di elaboratori, come valutare le prestazioni dei programmi paralleli e qualche possibilità per svilupparli.

1.1 Classificazione elaboratori

Per quanto riguarda la classificazione dei calcolatori, non esiste uno standard universalmente riconosciuto, in generale si utilizza la tassonomia di Flynn del 1972. Quest'ultima utilizza due caratteristiche principali per raggruppare i diversi tipi di macchine: il numero di istruzioni computate parallelamente e le sequenze di dati che possono essere elaborate concorrentemente.

Da questa divisione nascono 4 principali tipologie di elaboratori:

- SISD (Single Instruction-Single Data): è il caso classico dell'architettura di Von Neumann che segue il ciclo fetch-decode-execute eseguendo serialmente un'istruzione su un singolo flusso di dati.
- SIMD (Single Instruction-Multiple Data): calcolatori che possono eseguire contemporaneamente la stessa istruzione su più dati, ad esempio le GPU.
- MISD (Multiple Instruction-Single Data): macchine che supportano l'esecuzione di più istruzioni concorrentemente su un singolo dato attraverso più istruzioni concorrentemente; non esistono ancora hardware di importanza rilevante appartenenti a questa categoria.

- MIMD (Multiple Instruction-Multiple Data): sistemi in grado di eseguire più istruzioni differenti su dati differenti, l'architettura sottostante può essere costituita da core autonomi che lavorano indipendentemente su diversi elementi: le CPU appartengono ormai tutte a questa categoria.

1.2 Valutazione di un programma parallelo

Per valutare le prestazioni di un programma parallelo è necessario definire tre concetti principali: lo Speedup, la scalabilità forte e la scalabilità debole.

1.2.1 Speedup

Lo Speedup si definisce come il rapporto tra il tempo impiegato dal programma seriale per risolvere il problema (T_{serial} o $T(1)$) e il tempo del programma parallelo con p processi/core ($T_{parallel}(p)$ o $T(p)$).

$$S(p) = \frac{T_{serial}}{T_{parallel}(p)} \quad (1.1)$$

Nel caso ideale si ipotizza che $T_{parallel}(p) = T_{serial}/p$ quindi avremo $S(p) = p$ (quando questa condizione si avvera, si parla di **Speedup lineare**). È normale pensare che p sia il valore massimo dello Speedup, ma ci sono casi in cui si può ottenere $S(p) > p$, ovvero uno Speedup superlineare: parallelizzando il problema è possibile che ci siano meno controlli da effettuare, quindi il lavoro totale e di conseguenza il tempo di esecuzione diminuiscono.

Normalmente esistono parti del programma che non possono essere parallelizzate, più in particolare definendo α come la frazione di tempo del programma non parallelizzabile avremo che:

$$T_{parallel}(p) = \alpha T_{serial} + \frac{(1 - \alpha)T_{serial}}{p} \quad (1.2)$$

dato $0 \leq \alpha \leq 1$ si evince

$$T_{parallel}(p) \geq T_{serial}/p \quad (1.3)$$

sostituendo l'ultima formula (eq. 1.3) alla definizione di Speedup (eq. 1.1) e raccogliendo T_{serial} si ottiene la legge di Amdahl (eq. 1.4), che definisce il massimo Speedup come

$$S(p) = \frac{1}{\alpha + \frac{(1 - \alpha)}{p}} \quad (1.4)$$

da cui si capisce che $\frac{1}{\alpha}$ è un limite massimo per lo Speedup di programmi non completamente parallelizzabili, infatti considerando $p \rightarrow \infty$

$$S(p) = \lim_{p \rightarrow \infty} \frac{1}{\alpha + \frac{(1-\alpha)}{p}} = \frac{1}{\alpha} \quad (1.5)$$

1.2.2 Scalabilità forte

Per studiare la scalabilità forte si confrontano i tempi di esecuzione mantenendo costante il carico di lavoro totale e modificando solo il numero di processi p ; la scalabilità forte dipende direttamente dallo Speedup:

$$E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)} \quad (1.6)$$

In questo caso vengono confrontati il tempo impiegato da p processori per eseguire l'algoritmo e il tempo di un singolo processore. Se all'aumentare di p il valore di E risulta costante allora si dice che l'algoritmo è scalabile.

Il valore ideale della scalabilità è 1, infatti a questo valore corrisponde il pieno utilizzo delle risorse. Generalmente al crescere di p si ha un incremento dell'overhead dovuto alla sincronizzazione tra i processi, quindi una diminuzione di $E(p)$. Considerando la formula 1.6 e sapendo che esiste il caso di Speedup superlineare, è ovvia la possibilità che $E(p) \geq 1$

1.2.3 Scalabilità debole

La scalabilità debole considera i diversi tempi di esecuzione dei programmi in cui viene mantenuto fisso il carico di lavoro per processo p cambiando sia p che la dimensione del problema.

$$W(p) = \frac{T_1}{T_p} \quad (1.7)$$

Dove T_1 è il tempo di esecuzione di una unità di lavoro svolto da un processore, invece T_p è il tempo di p unità di lavoro svolto da p processori.

Anche in questo caso il valore ideale della scalabilità è 1, nonostante esista la possibilità che questo venga superato.

1.3 Sviluppo di programmi paralleli

Prima di iniziare lo sviluppo di codice parallelo è necessario scegliere le tecnologie hardware e software attentamente. L'esecuzione dei programmi avrà rendimenti diversi in base alle tecnologie su cui si basa: in particolare per alcune computazioni l'uso della CPU o della GPU può portare a importanti differenze nei tempi di calcolo.

1.3.1 Confronto tra CPU e GPU

Parlando di programmi paralleli ad alta velocità possiamo parlare in generale di due unità di calcolo hardware: CPU e GPU.

La CPU è composta oggigiorno da massimo una decina di core, questi lavorano ad alta frequenza (nell'ordine dei 2.5-4 GHz) per effettuare calcoli e coordinare il resto delle periferiche della macchina. Storicamente tutti i programmi erano mandati in esecuzione sulla CPU, questo è il motivo per cui ancora oggi la maggior parte degli eseguibili e dei linguaggi di programmazione sono specifici per CPU.

La GPU presenta un numero di core molto maggiore rispetto alla CPU (da centinaia fino a migliaia) che lavorano a frequenza inferiori (circa 1-2 GHz) e sono specializzati nella grafica digitale, operando SIMD su una grande mole di numeri in virgola mobile. Per questa caratteristica è nata l'idea di usare queste unità di calcolo anche per scopi non strettamente legati alla grafica ed è stato coniato il nome di **GPGPU** (*General-Purpose GPU*).

L'uso della programmazione per CPU piuttosto che per GPU dipende solitamente dal progetto che si deve svolgere: la CPU è più veloce nella gestione di operazioni complesse su pochi dati, al contrario la GPU può gestire una quantità di dati molto maggiore a discapito della complessità del programma e della precisione (alcune GPU consentono di eseguire calcoli solamente con i **float**).

1.3.2 Esempi di linguaggi per lo sviluppo

Ci sono vari linguaggi per lo sviluppo di programmi paralleli, in particolare per la programmazione CPU vengono descritti **OpenMP** e i **Vector data type** che verranno riutilizzati anche nel capitolo 4 per il confronto con altre versioni.

OpenMP

OpenMP [1] è un modello per la programmazione parallela in architetture a memoria condivisa; il modello si basa su direttive del compilatore per permettere al programmatore di dividere le parti seriali da quelle parallele e per gestire le sincronizzazioni dei thread.

Per usare OpenMP non è necessario essere esperti della programmazione parallela: modificando il codice seriale tramite le direttive adeguate si ottiene subito un livello base di parallelizzazione. OpenMp definisce la direttiva `#pragma omp parallel` per delimitare le sezioni di codice parallele: in particolare è possibile parallelizzare dei cicli `for` che rispettano alcune condizioni tramite la `#pragma omp parallel for`.

Ogni direttiva può essere integrata con delle clausole che possono limitare o meno la visibilità delle variabili (`shared`, `private`, `default`), specificano come gestire il carico di lavoro (`schedule`, `collapse`) o effettuano operazioni sui dati (`reduction`).

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    int n = 1024, sum = 0, *v1 = (int*)malloc(n * sizeof(v1[0]));
    //fill the array
    for (int i=0; i<n; i++) {
        v1[i] = i%4;
    }

#pragma omp parallel for reduction (+:sum)
    for (int i=0; i<n; i++) {
        sum += v1[i];
    }

    printf("sum = %d",sum);
    free(v1);
    return EXIT_SUCCESS;
}
```

Listato 1.1: Riduzione di un vettore OpenMP

Come possiamo vedere dal codice (Listato 1.1) il ciclo viene eseguito in modo parallelo, e ogni thread calcola la sua somma provvisoria `sum` che viene

a sua volta usata per calcolare la somma finale dalla `reduction`; infine viene stampato una sola volta il risultato finale.

Programmazione SIMD e GCC Vector data type

La programmazione SIMD sfrutta appositi registri e unita di calcolo per effettuare una singola operazione su più elementi durante lo stesso ciclo di clock. Ogni architettura presenta un suo linguaggio specifico per la programmazione SIMD, questo induce il codice a non essere portabile; per evitare questo si possono usare estensioni basate sul compilatore: nel caso di `GCC`, i `GCC Vector data type` [6].

Come suggerito dal nome i `Vector data type` sono vettori di elementi sui quali si possono applicare le più comuni operazioni aritmetiche (somma, prodotto..).

Il compilatore, quando viene richiamato, emette le istruzioni SIMD compatibili per l'architettura sottostante, oppure il codice viene compilato per l'esecuzione seriale nel caso l'hardware non supporti la programmazione SIMD.

I `Vector data type` possono essere usati come i classici array in C; per usarli però vanno prima definiti attraverso la `typedef`. Esiste una convenzione per i nomi dei vettori: è buona norma che il nome inizi sempre con la lettera `v`, dopodiché vengono inseriti il numero di elementi del vettore e una lettera che rappresenta il tipo degli elementi; ad esempio i `v4i` sono vettori di 4 `int`.

```
/* The following #define is required by posix_memalign() */
#define _XOPEN_SOURCE 600

#include "hpc.h"
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef int v4i __attribute__((vector_size(16)));
#define VLEN (sizeof(v4i)/sizeof(int))

int main(int argc, char* argv[])
{
    int n = 1024, sum = 0, *v;
    v4i vsum = {0,0,0,0}, *pv;

    int ret = posix_memalign((void**)&v, __BIGGEST_ALIGNMENT__, n *
        sizeof(*v));
    assert( 0 == ret );
    //fill the array
```

```
for(int i=0; i<n; i++){
    v[i]=i%4;
}
//calculating partial sum
for(int i=0; i<n-VLEN+1; i+=VLEN) {
    pv = (v4i*)(v+i);
    vsum += *pv;
}
//calculating sum
for(int i=0; i<VLEN; i++){
    sum += vsum[i];
}

printf("sum = %d\n", sum);
free(v);
return EXIT_SUCCESS;
}
```

Listato 1.2: Codice SIMD

Il codice (Listato 1.2) effettua le stesse operazioni di quello precedente implementato in OpenMP (Listato 1.1): ogni elemento del v4i **vsum** calcola la propria somma parziale tramite le operazioni SIMD, dopodiché viene calcolata serialmente la somma finale.

Capitolo 2

Raspberry Pi

La nostra missione è mettere il potere dell'informatica e del calcolo digitale nelle mani delle persone di tutto il mondo [5].

Questo è lo slogan di Raspberry, per presentare *Il computer delle dimensioni di una carta di credito che costa 25 sterline* [5]: Raspberry Pi.

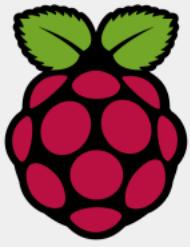


Figura 2.1: Confronto tra le dimensioni di Raspberry Pi e una carta di credito.
(Fonte: [3])

2.1 Panoramica

Raspberry Pi è una classe di calcolatori *Single-Board* che presenta alcune caratteristiche comuni: una CPU ARM, una GPU con innestato il chip VideoCore IV, un supporto per la scheda MicroSD, ingressi USB 2.0 o 3.x e ingressi Ethernet che vanno dai 10/100 Mbit/s fino a 1Gbit/s.

Le CPU ARM installate sono in grado di gestire sistemi operativi tra cui Raspbian e Ubuntu (considerati i più rilevanti). Nonostante nelle prime versioni le CPU non fossero molto performanti (700MHz single-core fino al Raspberry 2), Raspberry Pi presentava GPU con capacità di calcolo molto promettenti, considerando anche il ridotto consumo di energia (massimo $\sim 5\text{W}$).



	Raspberry Pi 3 Model B	Raspberry Pi Zero	Raspberry Pi 2 Model B	Raspberry Pi Model B+
Introduction Date	2/29/2016	11/25/2015	2/2/2015	7/14/2014
SoC	BCM2837	BCM2835	BCM2836	BCM2835
CPU	Quad Cortex A53 @ 1.2GHz	ARM11 @ 1GHz	Quad Cortex A7 @ 900MHz	ARM11 @ 700MHz
Instruction set	ARMv8-A	ARMv6	ARMv7-A	ARMv6
GPU	400MHz VideoCore IV	250MHz VideoCore IV	250MHz VideoCore IV	250MHz VideoCore IV
RAM	1GB SDRAM	512 MB SDRAM	1GB SDRAM	512MB SDRAM
Storage	micro-SD	micro-SD	micro-SD	micro-SD
Ethernet	10/100	none	10/100	10/100

Figura 2.2: Confronto delle specifiche di alcuni modelli Raspberry Pi
(Fonte: [2])

2.2 Programmazione GPU

All'interno della GPU di Raspberry, sono presenti 12 QPU (Quad Processing Units), ossia 12 unità costruite per processare vettori da 16 elementi in 4 cicli di clock, che rendono la GPU potenzialmente in grado di elaborare fino a 24 GFLOPS¹ fin dai primi modelli di Raspberry Pi.

La comunità scientifica si è mostrata interessata a questo vantaggioso rapporto *capacità di calcolo/consumo di energia* da quando Raspberry Pi è stato

¹Un GFLOPS equivale a 10^9 operazioni in virgola mobile al secondo

presentato al pubblico (2012). Per qualche anno non è stato possibile programmare la GPU per scopi diversi dalla sola grafica: i classici metodi di programmazione infatti, non erano disponibili. Nel 28 febbraio 2014 Raspberry e l’azienda costruttrice della scheda video pubblicarono la documentazione completa della VideoCore IV [12]; da quella data sono stati sviluppati diversi esempi di programmazione GPU Raspberry Pi per vari livelli di linguaggio:

- GPU_FFT [7]: un programma scritto direttamente usando il codice assembly della GPU Raspberry, è la prima diretta conseguenza delle pubblicazioni sopra descritte.
- PyVideoCore [11]: una libreria Python che permette una migliore gestione dei dati, ma comunque limita l’uso della GPU alla programmazione attraverso il suo assembly.
- QPULib [10]: implementa tramite una libreria C++ la possibilità di compilare funzioni per la GPU, in modo da poterle invocare per delegare una parte del lavoro.

2.3 QPULib

Come già accennato, QPULib è un linguaggio di programmazione per le QPUs di Raspberry Pi ed è implementato come una libreria in C++ che viene eseguita dalla CPU ARM di Raspberry Pi. Per valutare QPULib è utile compararlo con un linguaggio simile: CUDA-C.

2.3.1 Confronto tra QPULib e CUDA-C

CUDA-C [4] è un linguaggio proprietario NVIDIA nato nel 2006 per rendere *general-purpose* le GPU compatibili; come QPULib genera programmi in esecuzione sulla CPU i quali demandano le computazioni più onerose alla GPU. Sotto questo punto di vista QPULib e CUDA-C sono simili, infatti una volta inizializzati i dati nella memoria della GPU, è permesso compilare e invocare le funzioni scritte per quest’ultima.

CUDA-C può contare sulla storia di NVIDIA e su un maggior numero di anni di evoluzione: questo rende l’ambiente di sviluppo e il linguaggio meno inclini alla presenza di errori non documentati; al contrario di QPULib che non può dare alcuna garanzia vista anche la versione sicuramente meno sviluppata 0.1.0.

Entrambi i linguaggi vengono eseguiti su hardware specifici, però a vantaggio di QPULib, Raspberry è più economico di una qualsiasi GPU NVIDIA compatibile.

Non da tutti classificato come vantaggio o svantaggio, bisogna considerare che QPULib è open-source, e in quanto tale può essere ritenuto più sicuro oltre che aggiornabile e personalizzabile da chiunque; al contrario CUDA-C essendo scritto specificatamente per hardware costruito da NVIDIA potrebbe essere più veloce. Purtroppo non esistono GPU compatibili a entrambi i linguaggi, quindi non è possibile confrontarli direttamente.

2.3.2 Programmare con QPULib

Per programmare usando QPULib bisogna innanzitutto prendere confidenza con i tipi di dato, come gestirli e con le funzioni principali che possono essere utilizzate all'interno del codice che verrà compilato per la GPU.

Le QPUs possono lavorare con solo tre tipi di dato `Int`, `Float` e `Ptr`, i primi due non sono altro che vettori da 16 elementi di 32-bit interpretati come interi o come numeri in virgola mobile, mentre l'ultimo è un puntatore ad un indirizzo di memoria. Parlando di indirizzi di memoria e sapendo che non esiste una memoria condivisa per CPU e GPU, è necessario introdurre la classe degli `SharedArray`: una classe utilizzabile solo dal codice in esecuzione sulla CPU, ma che alloca un vettore in memoria in modo che questa sia accessibile da entrambe le unità di calcolo.

È importante precisare che le QPUs non possono lavorare che con questi tre tipi di dato (quindi non sarà possibile salvare un solo `int`, ma sarà necessario salvare un `Int`) ed è necessario calcolarlo quando si alloca lo spazio per le strutture dati che conterranno questi dati.

La CPU ARM di Raspberry Pi implementa lo standard IEEE 754 [8] che impone, tra le altre specifiche, di approssimare il numero per eccesso o per difetto in base alle cifre meno significative; QPULib al contrario, approssima sempre per difetto. Nonostante questo possa sembrare irrilevante in molti problemi, è importante saperlo e considerarlo nel caso le operazioni da svolgere rielaborino iterativamente i propri output. In particolare, nel nostro caso renderà impossibile il confronto diretto dell'output dei programmi (capitolo 4).

Funzioni principali

Per quanto riguarda i costrutti principali della programmazione C, QPULib implementa delle funzioni nascoste dietro a delle macro, per agevolare la programmazione della GPU:

- **For:** simile al costrutto per il ciclo `for`, deve essere richiamato usando le virgolette al posto dei punti e virgola, inoltre si può inizializzare la variabile su cui iterare ricordandosi che deve essere `Int` o `Float`.

- **Print:** serve principalmente in fase di sviluppo per scrivere sullo stdout delle stringhe, degli `Int` o, modificando a dovere la libreria, dei `Float`.
- **If:** questa funzione implementa l'`if` del C, assegnando all'espressione che viene passata al suo interno un solo valore (vero o falso), in modo da decidere il flusso di istruzioni da seguire.
- **Where:** a differenza dell'`If` questa funzione assegna ad ognuno dei 16 elementi del vettore risultato un valore (vero o falso), quindi solo gli elementi che soddisfano l'espressione accedono all'area di codice all'interno del `Where`.

Gestione dei dati

Il caricamento e il salvataggio dei dati in un vettore (o in una matrice) può essere effettuato usando la sintassi d'accesso del C++ con operazioni bloccanti; per rendere gli accessi alla memoria più veloci QPULib implementa funzioni non bloccanti:

- la `gather` serve per iniziare il caricamento del dato, questa infatti, inserisce in modo asincrono in una coda FIFO (First In First Out) i primi 16 elementi (un `Int` o un `Float`) a partire dall'indirizzo di memoria specificato come argomento della funzione.
- La `receive` recupera il primo dato (`Int` o `Float`) della coda FIFO dedicata e lo inserisce nella variabile passata come parametro alla funzione; questa viene considerata l'operazione complementare della `gather`.
- La `store` prende in input un `Int` o un `Float` e un indirizzo di memoria, dopodiché salva il primo input all'interno del secondo. Attualmente può essere presente solo una `store` in esecuzione, per questo la funzione è considerata ancora in fase di sviluppo.

Se ne vengono richiamate due di fila, la seconda risulta bloccante fino al completamento della prima.

Capitolo 3

Kernel computazionali

L’obiettivo di questa tesi è la valutazione delle implementazioni tramite QPULib di due kernel computazionali¹: Earthquake e Matmul.

3.1 Earthquake

Lo scopo del progetto Earthquake è l’implementazione di un semplice modello matematico di propagazione dei terremoti. Il modello che consideriamo è una estensione in due dimensioni dell’automa cellulare Burridge-Knopoff (BK) [9]. È importante studiare e valutare questo kernel come esempio per un qualsiasi pattern stencil².

3.1.1 Modello Earthquake

Simuleremo una porzione di crosta terrestre di dimensioni $n \times n$ celle; ogni cella contiene il valore dell’energia potenziale relativamente alla propria posizione.

In accordo con il modello delle placche [13], l’energia aumenta ad ogni iterazione a causa del movimento continuo delle zolle; poi quando questa supera una soglia data, si scarica nelle celle adiacenti emulando un terremoto.

Più nello specifico inizializzeremo una matrice quadrata di dimensione $n \times n$ di valori reali casuali (float) per emulare la crosta terrestre. Ogni cella rappresenta una porzione di crosta terrestre e il valore ad essa associato è l’energia potenziale; il valore viene aggiornato ad istanti discreti di tempo $t = 0, 1, 2, \dots$ seguendo due regole: incremento e propagazione.

¹viene chiamato kernel computazionale una piccola funzione ricorrente nell’esecuzione del programma

²La computazione di uno stencil riguarda solitamente una matrice le cui celle sono aggiornate in base a un calcolo fisso che coinvolge anche le celle vicine.

La fase 1 o incremento, aggiorna tutti i valori delle celle aumentandoli di una costante EDELTA.

La propagazione o fase 2 controlla che l'energia E di ogni cella non superi il valore critico EMAX, altrimenti ad E viene sottratta EMAX e ad ogni cella vicina (distanza 1 in metrica City-Block³) si somma EMAX/4.

Per avere dei risultati interessanti riguardo all'energia media, è utile che il dominio non sia ciclico, in modo che le celle di bordo abbiano meno di 4 vicini. Infatti, assumendo un dominio non ciclico durante ogni terremoto delle celle di margine, e sapendo che l'energia trasferita alle celle adiacenti (massimo 3) è comunque EMAX/4, l'energia totale diminuisce.

Il modello BK è un automa cellulare sincrono, ovvero un automa in cui tutte le celle vengono aggiornate contemporaneamente. Questa è la motivazione per cui esisteranno due matrici, `cur` e `next`, contenenti rispettivamente i valori delle energie al passo corrente e al passo successivo.

3.1.2 Implementazione QPULib

Prima di iniziare a programmare è stato inevitabile decidere come dividere la matrice in modo che ogni QPU abbia un carico di lavoro bilanciato, cosicché ho deciso di assegnare ad ogni QPU un numero di righe uguali (± 1); inoltre ho reputato vantaggioso l'uso di ghost cell intorno alle matrici in modo da non dover controllare ogni volta gli accessi alle celle vicine.

Inizializzazione del programma

Ad ogni QPU è stato assegnato un numero di righe pari a $n/QPUs$ con un $+1$ alle prime QPUs se è necessario per computare tutta la matrice. Ho inserito all'interno di uno `SharedArray` l'indice da cui ogni QPU deve iniziare a computare (non considerando le ghost cell), più precisamente la QPU di indice `x` inizia dall'indice `init_i[(x)*16]+1(GC)`. Dopodiché ho creato le due matrici principali `cur` e `next` e due `SharedArray`. Le matrici sono state inizializzate considerando come dimensione $n+ = 2(GC)$ in modo che esista attorno alle matrici un margine di ghost cell. Gli `SharedArray` si dividono due compiti: uno serve per memorizzare quante celle hanno generato la scossa durante l'ultima iterazione (`count`) e l'altro per la somma totale delle energie (`sum`).

Per una migliore gestione delle matrici ho anche usato 3 puntatori a `SharedArray` (`pcur`, `pnext`, `tmp`), in modo da poter scambiare i riferimenti a `cur` e `next`.

³la metrica City-Block considera come vicini le celle a Nord, Est, Sud e Ovest

parte centrale del programma

Una volta entrati nel ciclo principale, è opportuno implementare la funzione di aggiornamento, affinché sia in esecuzione sulle QPUs per ottimizzare il cuore del programma, ovvero la parte in cui viene speso la maggior parte del tempo di esecuzione.

Dopo una prima elaborazione dei dati richiesti in input per adattarli all’ambiente QPU, è stato utile costruire una maschera (`imask`) per estendere il dominio di `n` in input, dai soli multipli di 16 a qualsiasi valore maggiore di 15. Inserendo la maschera come condizione di un `Where` (controllando se è $=0$ o > 0), si possono manipolare solo alcuni elementi invece che tutti e 16 insieme. È comunque necessario tenere presente che durante la `store` verrà salvato un `Float` (Sezione 2.3.2) all’interno della matrice, ma sarà poi, in parte, sovrascritto.

Dopo aver considerato la parte della riga non multipla di 16, ogni QPU potrà continuare a calcolare `Earthquake` senza la necessità di fare controlli sullo stato della cella (ghost cell o no).

Ogni elemento quando deve essere elaborato, come prima cosa viene aumentato, dopodiché si controlla se è maggiore di EMAX per aggiornare `count`, viene propagata l’energia dei vicini che in quell’iterazione superano EMAX, infine viene tolta l’energia dovuta al sisma e aggiornato `sum`. Tutta la gestione dei dati, dove possibile è stata effettuata in modo non bloccante, in modo da rendere il programma più fluido, a discapito della leggibilità.

Quando tutti gli elementi sono stati considerati, ogni QPU deve calcolare il proprio `count` e `sum` e salvarlo nello SharedArray appositamente creato; per fare questo è stata usata la funzione `rotate`, grazie alla quale si ottimizza il numero di somme da effettuare.

È inevitabile una parte seriale per sommare i `count` e `sum` rilevati da ogni QPU, per scrivere e per invertire i puntatori a `cur` e `next`.

3.2 Moltiplicazione tra matrici

La moltiplicazione tra matrici, anche detta prodotto tra matrici o prodotto riga per colonna viene definita così:

Data una matrice reale $A_{m \times n}$ e una matrice reale $B_{n \times p}$, il prodotto matriciale è una matrice reale $C_{m \times p}$, i cui elementi c_{ik} sono la somma dei prodotti degli elementi corrispondenti della riga i di A e della colonna k di B .

È utile studiare le prestazioni di questo kernel in quanto la moltiplicazione tra matrici è un'operazione molto comune all'interno di svariati problemi (la costruzione di oggetti 3D, il render e la modifica di immagini, ..), quindi è opportuno che sia il più veloce possibile.

3.2.1 Modello Matmul

Matmul viene eseguito con l'impiego di 4 matrici di dimensioni $n \times n$; ogni cella delle prime due matrici $C_{i,j}$, dove i e j sono rispettivamente l'indice della riga e l'indice della colonna, sarà inizializzata tramite la formula $C_{i,j} = (i \% 10 + j) / 10$.

La moltiplicazione tra matrici non è un'operazione **cache efficient**, infatti l'accesso a una matrice per colonna (come è necessario fare per la seconda) è sconsigliato; per questo motivo, in situazioni del genere è utile trasporre⁴ la seconda matrice.

3.2.2 Implementazione QPULib

Le QPUs nel recuperare i valori in memoria, non hanno la possibilità di accedere ai dati in modo sparso, in particolare, non potrebbero accedere ai valori della seconda matrice per colonna, quindi la trasposizione è d'obbligo.

Inizializzazione del problema

Anche in questo caso è stata eseguita una divisione del dominio per righe tramite lo `SharedArray init_i`. A differenza di prima, data la mancanza delle ghost cell in questo problema l'indice di partenza è `init_i[(x)*16]`. L'inizializzazione delle matrici risulta più intuitiva per questo esercizio, non essendoci le ghost cell.

⁴dati a_{ij} e a'_{ij} generici elementi delle matrici A e A^T quadrate di dimensione n ;
 A^T è la trasposta di A se $a'_{ij} = a_{ij} \forall i, j < n$

Problema trasposizione

La trasposizione in QPULib non è un problema facile da affrontare, come ho già detto (Sezione 3.2.2) lavorare con dati sparsi nella memoria usando QPULib è dispendioso, in termini di tempo di esecuzione; per completezza ho provato tre diversi approcci per risolvere questo problema:

- il primo tentativo è stato quello di un approccio *totalmente QPULib*, per meglio dire, tutta la trasposizione della matrice è effettuata tramite le QPUs. Questo portava a una 'matrice' trasposta in cui alcune righe (l'ultima riga computata da ogni QPU) erano più lunghe delle altre, questo come conseguenza diretta dell'uso esclusivo dei **Float**. Così per rendere l'output della funzione una vera matrice, è obbligatoria una parte seriale che riorganizza le celle extra.
- Ho provato anche con una soluzione ibrida: ovvero l'utilizzo delle QPUs escludendo l'ultima riga dal calcolo, la quale viene successivamente computata in modo seriale. Logicamente sembra il metodo più veloce, in quanto parallelizza il più possibile senza 'sprecare' calcoli; in pratica però non lo è.
- Per ultimo ho provato, quasi per curiosità ad effettuare la trasposizione in modo seriale e ho notato che è l'opzione più veloce per ogni input gestibile da Raspberry Pi 2.

parte centrale del programma

Al contrario della soluzione di Earthquake (sezione 3.1.2), si è deciso di limitare l'input **n** a un valore maggiore di 15 e multiplo di 16; così facendo sono necessari meno controlli e il programma risulta più veloce.

Sviluppato già il problema della trasposizione, la funzione che effettua la moltiplicazione prende in ingresso la seconda matrice già trasposta. L'implementazione QPULib di **Matmul** per la sua risoluzione invoca 3 cicli **For** innestati; i primi due agiscono sull'indice riga **i** e sull'indice **j** colonna della matrice risultato. Il ciclo più interno modifica l'indice **k** per spostarsi lungo la riga di entrambe le matrici (una volta trasposta la seconda matrice, la moltiplicazione avviene riga per riga). Considerando che il dominio delle matrici è obbligatoriamente multiplo di 16, non è utile inserire **Where** in quanto rallenterebbero solo il flusso del programma.

Completato il ciclo più interno si ha come risultato un **Float** i cui elementi vanno sommati per trovare il valore risultato della cella **i,j**; anche in questo caso si utilizza la **rotate** per risparmiare qualche ciclo di clock.

Ricordando che QPULib non può salvare in memoria meno dati di un `Float`; alla fine del calcolo avremo (come nel caso della trasposizione totalmente QPULib) una 'matrice' che presenta nell'ultima riga calcolata da ogni QPU 15 elementi extra. Anche in questo caso è necessaria una parte seriale che ricontralli l'output e lo renda una matrice.

Capitolo 4

Valutazioni delle prestazioni

In questo capitolo valuteremo le prestazioni delle implementazioni QPU-Lib appena descritte. Per farlo ogni programma verrà confrontato anche con una versione parallela su CPU: per farlo sono state implementate una versione OpenMP di Earthquake, e una SIMD (basata sui `Vector data type`) di Matmul.

4.1 OpenMP Earthquake

Per l'implementazione di Earthquake è stato utile condensare le quattro funzioni del codice seriale in una sola, in modo che la sezione da parallelizzare sia unica e ben definita, come si può vedere dal listato 4.1.

```
for (int i = 1; i<n-1; i++) {
    for (int j = 1; j<n-1; j++) {
        float F = *IDX(grid, i, j, n);
        /* update cells */
        if (F > EMAX) { count++; }
        *IDX(next, i, j, n) = F;
        sum += F;
    }
}
```

Listato 4.1: Ciclo di aggiornamento da parallelizzare

Modificata a dovere la versione seriale del programma, una prima versione parallela OpenMP è stata pressoché immediata: infatti è bastato aggiungere una direttiva `#pragma omp parallel for` prima del ciclo più esterno.

La prima versione parallela è stata poi studiata e sottoposta a diverse considerazioni: in particolare sono sorte delle domande riguardo la clausola **reduction**¹ e sulla clausola **collapse**².

La clausola **reduction** può essere utilizzata per effettuare le due somme che si vedono nel ciclo: quella di **count** e di **sum**. In particolare, è sembrata utile in quanto è risultata più efficiente di altre due implementazioni: la prima era realizzata tramite un array nel quale ogni thread OpenMP inseriva la sua somma parziale e alla fine del ciclo tutti gli elementi venivano sommati in modo seriale (dal **master**); un'altra soluzione era proteggere le variabili risultato con una direttiva **atomic**, in modo che gli accessi in memoria non si potessero sovrapporre. Nonostante la clausola **reduction** aggiunga un po' di overhead è comunque la soluzione migliore per il calcolo di **count** e di **sum**.

La clausola **collapse** può aumentare le prestazioni in quanto rende unico lo spazio di iterazione, rendendo più bilanciato il carico di lavoro per thread, però considerando il carico di ogni iterazione circa costante notiamo che: dato un numero di iterazioni sufficientemente alto rispetto al numero dei thread OpenMP, lo sbilanciamento di carico è minimo, quindi l'overhead generato dalla clausola **collapse** è maggiore del guadagno che porterebbe usarla.

4.2 Matmul con Vector data type

Programmando con i **Vector data type** è fondamentale definire il tipo di vettore con cui si andrà a lavorare; in questo caso è stato definito il tipo **v4f** (vettore di 4 **float**) per effettuare l'accesso per riga alle matrici. È necessario usare un tipo **v4f** anche per sommare i valori dei vari prodotti e per poter calcolare il valore finale da assegnare a ogni cella risultato.

Considerando che l'hardware potrebbe non supportare le istruzioni SIMD (come detto nella sezione 1.3.2 il codice compilato potrebbe essere seriale) è utile mantenere una variabile o una macro per sapere di quanti elementi è composto il vettore: in questo caso ho definito la macro **VLEN** usata come addendo nel ciclo più interno.

¹la clausola **reduction** prende in input un'operazione e una lista di variabili; dopodiché esegue l'operazione su ogni elemento della lista

²la clausola **collapse** specifica quanti cicli, in una serie di cicli innestati, devono essere collassati in un unico spazio di iterazione

4.3 Prestazioni QPULib e confronto

In questa sezione valuteremo le prestazioni delle implementazioni QPULib usando i criteri di scalabilità introdotti nella sezione 1.2, infine confronteremo i tempi d'esecuzione con le versioni operanti su CPU appena descritte.

Per tutti i calcoli delle prestazioni è stato usato un Raspberry Pi 2. Sono state effettuate 11 rilevazioni dei tempi dalle quali è stato tolto il valore che più si discostava dal valor medio, dopodiché è stata ricalcolata la media e considerata come valore effettivo.

Calcolare i tempi di esecuzione dato un carico di lavoro totale costante non ha portato alcun problema: una volta deciso l'input iniziale è bastato modificare il numero di QPUs attive.

Riferendosi invece al carico di lavoro per thread costante, essendo una matrice quadrata l'oggetto di entrambi i problemi, il carico di lavoro scala con la radice quadrata del lato. In quanto ovviamente non si possono avere numeri di celle non interi, per calcolare l'input del lato della matrice è stato necessario moltiplicare l'input di partenza per la radice quadrata del fattore di parallelizzazione e arrotondare all'intero più vicino. Questo porta una leggera imprecisione nel calcolo di alcune scalabilità deboli (in particolare nel caso di Earthquake QPULib in cui l'input deve essere anche multiplo di 16).

4.3.1 Scalabilità dato un input costante

Per valutare le prestazioni dei due kernel bisogna prima di tutto decidere su quale input si effettueranno le rilevazioni: per Earthquake sono sembrate opportune 30.000 iterazioni su una matrice 150×150 ; invece per Matmul le matrici sono di dimensione $n = 512$.

Come si può vedere nel grafico 4.1, dato un input costante Earthquake non scala in modo adeguato dopo le 3 QPUs attive; possiamo individuare il problema controllando il grafico 4.2 dei tempi relativo alle stesse prove: la durata dell'esecuzione tende ad un valore asintotico di circa 33 secondi. Probabilmente quel tempo è dato dal calcolo seriale di `count` e `sum` che limita lo Speedup massimo e di conseguenza il tempo d'esecuzione per la legge di Amdahl (formula 1.4). Al contrario Matmul, nonostante contenga una parte seriale, mantiene la scalabilità forte oltre il 50%.

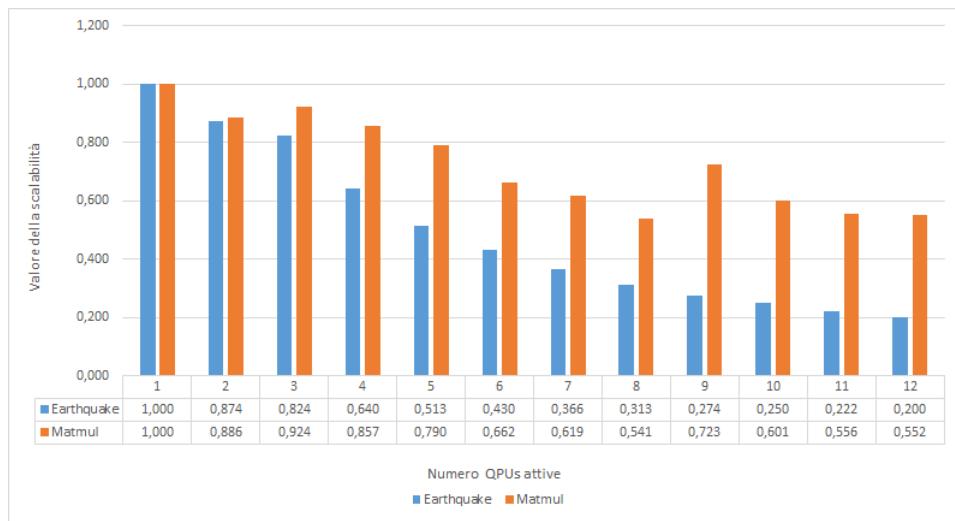


Figura 4.1: Grafico della scalabilità forte delle implementazioni QPULib

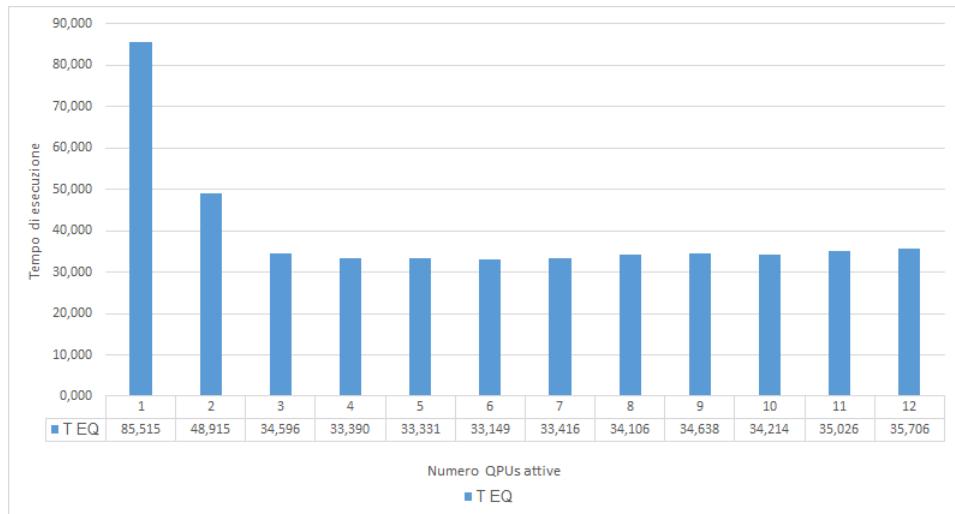


Figura 4.2: Grafico dei tempi dato un input costante di QPULib Earthquake

4.3.2 Scalabilità debole

In entrambi i kernel è stato necessario diminuire l'input iniziale su cui si sono basati i precedenti grafici. Per Matmul è stato ridotta la dimensione iniziale della matrice a $n_1 = 256$ in quanto al crescere del numero di QPUs la memoria di Raspberry Pi non avrebbe contenuto tutte le matrici. L'esecuzione di Earthquake è stata effettuata con 20.000 iterazioni fisse su una matrice a dimensione variabile (in base alle QPUs), a partire da $n = 128$;

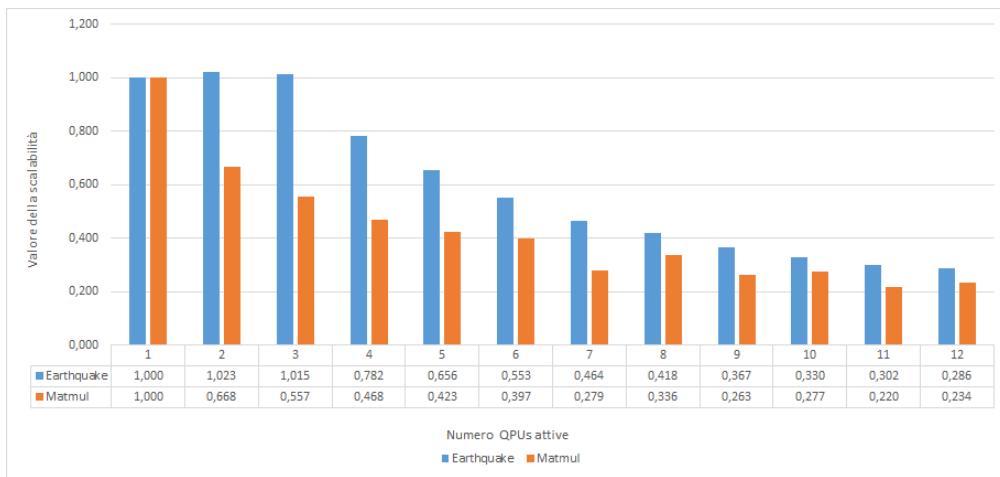


Figura 4.3: Grafico della scalabilità debole delle implementazioni QPULib

Come si può vedere dal grafico 4.3 in entrambe le implementazioni abbiamo un notevole abbassamento della scalabilità debole. Questo può essere dovuto dalle caratteristiche generali di Raspberry Pi nella gestione dei dati condivisi, oppure alla velocità del bus nella gestione delle QPUs.

4.3.3 Confronto con altre implementazioni

Come si vede dal grafico 4.4 per valutare più implementazioni di un programma è necessario anche tenere conto anche dei tempi di esecuzione seriali: considerando $T_{serial} = T(1)$ tutti i concetti descritti nella sezione 1.2 riguardano l'esecuzione di una stessa implementazione.

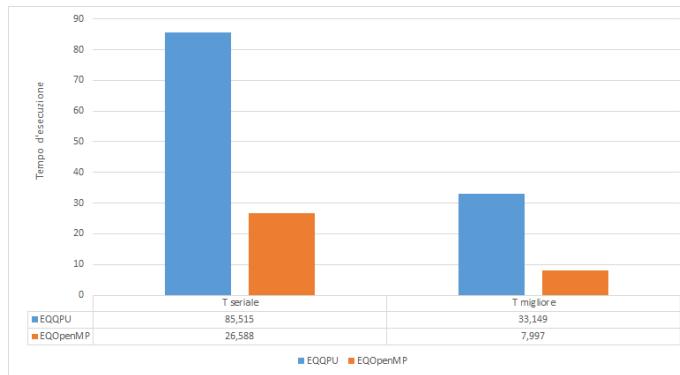


Figura 4.4: tempi seriali e migliori per le implementazioni di Earthquake

La prima colonna del grafico 4.4 sottolinea che l'implementazione QPULib necessita di molti più calcoli (o in generale di più gestione dei dati) della

versione seriale e nonostante la parallelizzazione questa differenza non riesce ad essere colmata.

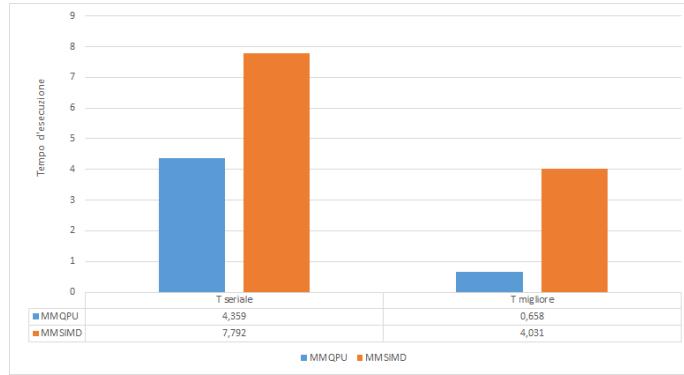


Figura 4.5: tempi seriali e migliori per le implementazioni di Matmul

Al contrario l’implementazione QPULib di Matmul (grafico 4.5) risulta essere più efficiente già con una QPU attiva, a maggior ragione utilizzandone di più dato che scala meglio della versione SIMD.

Conclusioni

In questo lavoro abbiamo mostrato come è possibile programmare le GPU di Raspberry Pi per scopi diversi dal solo calcolo grafico. Per ogni chiarimento sul codice, questo è reperibile nel repository pubblico all’indirizzo <https://bitbucket.org/SimoneMagnani/videocore-iv-e-il-calcolo-generico/>.

Nonostante QPULib sia un linguaggio ancora in fase di sviluppo, in alcune implementazioni il miglioramento delle prestazioni è evidente. Come si può vedere dai tempi di esecuzione, soprattutto nella gestione di grandi quantità di dati, è importante saper programmare la GPU per scopi non strettamente legati alla grafica.

È anche importante ricordare che, nonostante in QPULib attraverso gli **SharedArray** la gestione della memoria condivisa sia del tutto trasparente, questo porta un overhead non trascurabile soprattutto finché la **store** non possiede una coda di esecuzione (sezione 2.3.2). Per questo motivo è corretto analizzare il problema e la quantità di computazioni da svolgere prima di decidere qual è l’unità di calcolo più adatta: per ottenere un effettivo miglioramento delle prestazioni sfruttando il calcolo parallelo sulla GPU del Raspberry Pi, occorre che la quantità di dati da elaborare sia significativa e che le operazioni di calcolo siano principalmente SIMD.

Al fine di ottenere un programma ad alte prestazioni la scelta migliore risulta essere un compromesso tra le tipologie di calcolo, mantenendo su GPU le operazioni SIMD per oltre una soglia minima di valori e su CPU i calcoli sequenziali o su piccole quantità di dati.

Ringraziamenti

Ringrazio prima di tutto il prof. Moreno Marzolla per la prontezza e la disponibilità che ha avuto nel seguirmi.

Ringrazio inoltre lo Spaceteam, l'>#Einstein, gli amici e tutti i compagni di corso con i quali ho condiviso molte ore di studio e di svago.

Ringrazio in particolare Mattia Magnani e Jessica Biondi per avermi supportato e sopportato durante la stesura della tesi ed infine, senza il cui supporto non sarei riuscito a raggiungere questo risultato, ringrazio la mia famiglia per tutto ciò che mi hanno permesso di fare.

Bibliografia

- [1] Blaise Barney. Lawrence livermore national laboratory. <https://computing.llnl.gov/tutorials/openMP/>.
- [2] Brian Benchoff. Introducing the raspberry pi 3. <https://hackaday.com/2016/02/28/introducing-the-raspberry-pi-3/>.
- [3] Ritesh Bendre. Raspberry pi 3: Everything you wanted to know about the credit card sized computer. <https://www.bgr.in/news/raspberry-pi-3-everything-you-wanted-to-know-about-the-credit-card-sized-computer/>.
- [4] NVIDIA Corporation. Cuda zone, 2006. <https://developer.nvidia.com/cuda-zone>.
- [5] Raspberry Pi Foundation. Raspberry pi, 2015. <https://www.raspberrypi.org>.
- [6] GCC Official Guide. Using vector instructions through built-in functions. <https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html>.
- [7] Andrew Holme. Fast fourier transform library for the raspberry pi, 2014. http://www.aholme.co.uk/GPU_FFT/Main.htm.
- [8] IEEE. Ieee 754-2019 - ieee standard for floating-point arithmetic. <https://standards.ieee.org/content/ieee-standards/en/standard/754-2019.html>.
- [9] R. Burridge; L. Knopoff. Model and theoretical seismicity. *Seismological Society of America*, 1967. <https://pubs.geoscienceworld.org/ssa/bssa/article-abstract/57/3/341/116471/model-and-theoretical-seismicity>.
- [10] mn416. A language and compiler for the raspberry pi gpu, 2016. <https://github.com/mn416/QPULib>.
- [11] nineties (Koichi NAKAMURA). Python library for gpgpu on raspberry pi, 2015. <https://github.com/nineties/py-videocore>.
- [12] Eben Upton. A birthday present from broadcom. <https://www.raspberrypi.org/blog/a-birthday-present-from-broadcom> (Official Blog Raspberry Pi).
- [13] Philip Kearey; Frederick J. Vine. *Tettonica globale*. Zanichelli, 1994.

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI SCIENZE

Corso di Laurea in
Ingegneria e Scienze Informatiche

Implementazione ottimizzata dell'operatore di
Dirac su GPGPU

TESI IN

High Performance Computing

RELATORE:
Prof. Moreno Marzolla

CANDIDATO:
Nicolas Farabegoli

CORRELATORE:
Dr. Enrico Calore

ANNO ACCADEMICO 2019/2020

Alla mia famiglia

Indice

Introduzione	1
1 Calcolo parallelo e GPU	2
1.1 Importanza del calcolo parallelo	2
1.1.1 I sistemi paralleli	2
1.1.2 Il vantaggio dei programmi paralleli	4
1.2 Il ruolo delle GPU	5
1.2.1 Storia delle GPU	5
1.2.2 La rivoluzione del 3D	5
1.2.3 General Purpose GPU (GPGPU)	6
1.2.4 L'utilizzo delle GPGPU	6
1.2.5 La fine della legge di Moore e l'inizio dell'HPC	8
2 CUDA e Tensor Core	10
2.1 Architettura di una CUDA-capable GPU	10
2.1.1 Griglie e blocchi in CUDA	15
2.1.2 Tensor Core	16
2.1.3 Architettura dei Tensor Core	18
2.1.4 Studio prestazioni Tensor Core	23
3 Operatore di Dirac e Tensor Core	26
3.1 Funzionamento operatore di Dirac	26
3.1.1 Complessità	28
3.2 Strutture dati	29
3.2.1 Tensor Core	30
3.2.2 Analisi delle prestazioni	39
Conclusioni e sviluppi futuri	41

Introduzione

L'obiettivo di questa tesi consiste nello studio dell'architettura dei Tensor Core, ovvero moduli che consentono di eseguire moltiplicazioni di matrici in hardware e come possono essere utilizzati per aumentare le prestazioni del calcolo dell'operatore di Dirac. Questo operatore viene utilizzato nelle simulazioni lattice QCD e rappresenta, nel complesso, l'operazione che impiega il maggior tempo per la computazione; ottimizzarne i tempi di esecuzione si riflette in un incremento generale delle prestazioni dell'intero algoritmo lattice QCD.

L'operatore di Dirac svolge come operazione principale la moltiplicazione tra matrici 3×3 e vettori 3×1 ; questa operazione può essere ottimizzata mediante l'uso dei Tensor Core, ovvero moduli hardware presenti nelle più recenti GPU NVIDIA che effettuano direttamente in hardware l'operazione chiamata **MMA** (Matrix Multiply and Accumulate) che effettua una moltiplicazione di matrici e somma di una terza.

NVIDIA consente di sfruttare i Tensor Core mediante diverse librerie: cuBlas, cuDNN e WMMA API. Si è deciso di utilizzare le WMMA API in quanto garantiscono un controllo a più basso livello dei Tensor Core affinché se ne possa capire meglio il funzionamento. Durante il tirocinio mi sono concentrato sullo studio dell'architettura dei Tensor Core, analizzando i possibili scenari di utilizzo realizzando piccoli programmi di esempio che li sfruttano e misurandone le prestazioni. Sono poi passato a studiare come è possibile adattare il calcolo dell'operatore di Dirac su Tensor Core analizzando l'algoritmo e la struttura dati preesistente. Infine sono state testate alcune soluzioni per incrementare le prestazioni dell'operatore di Dirac con Tensor Core.

Questa tesi si suddivide in tre capitoli: nel primo si illustrano le motivazioni storico-tecnologiche che hanno portato a sviluppare architetture parallele per velocizzare sempre più algoritmi di simulazione e non solo, analizzando le prime architetture fino ad arrivare alle moderne GPU che consentono di sfruttare un parallelismo massivo. Nel secondo capitolo si studia il funzionamento dell'architettura CUDA concentrando particolare attenzione al funzionamento dei Tensor Core e come questi ultimi possono garantire un vantaggio in termini di prestazioni nella moltiplicazione tra matrici. Infine nel terzo capitolo si introduce l'operatore di Dirac mostrando la struttura dati con cui opera l'algoritmo e le soluzioni sviluppate per aumentarne le prestazioni con i Tensor Core. Vengono analizzate nel dettaglio le tre soluzioni proposte, evidenziandone vantaggi e svantaggi di ognuna oltre a fornire nel dettaglio la misura delle prestazioni ottenute da ognuna di esse.

Capitolo 1

Calcolo parallelo e GPU

1.1 Importanza del calcolo parallelo

Dal 1986 al 2002, le prestazioni dei microprocessori aumentavano di circa il 50% ogni anno [1]. Dal 2002 l'incremento delle prestazioni di un microprocessore single-core ha subito un drastico calo: dal 50% al 20%. Quindi se con un aumento del 50% in 10 anni si incrementavano le prestazioni di 60 volte, attualmente incrementandole del 20%, in 10 anni abbiamo un aumento di solo 6 volte le prestazioni iniziali.

Questa differenza nell'aumento delle prestazioni è associata a un drastico cambiamento nella progettazione delle CPU: dal 2005 i maggiori produttori di microprocessori hanno deciso di virare in direzione del parallelismo, cercando quindi di inserire all'interno di un singolo chip più microprocessori piuttosto che incrementare le prestazioni di un singolo microprocessore monolitico.

Questa scelta ha non poche conseguenze per gli sviluppatori di software: aggiungendo semplicemente più core all'interno di un chip non si migliorano le prestazioni di programmi seriali. Il programmatore si deve quindi prendere carico di riscrivere il software (o parti di esso) affinché si sfrutti il parallelismo messo a disposizione dall'hardware: in quanto il software non è a conoscenza della presenza degli altri core. Quindi, lo stesso software eseguito su un processore multi-core ha le stesse prestazioni dello stesso software eseguito su un processore single-core.

1.1.1 I sistemi paralleli

L'incremento delle prestazioni su processori single-core ha portato ad aumentare sempre più la densità di transistor all'interno del chip: al decrescere della dimensione di un singolo transistor, la sua velocità aumenta e quindi l'intero circuito integrato aumenta di prestazioni. Oltre all'aumentare delle prestazioni, riducendo la dimensione dei transistor aumentano anche i consumi: gran parte di questi consumi vengono dissipati sotto forma di calore e il surriscaldamento del chip degrada velocemente le sue prestazioni. Nel primo decennio degli anni duemila i circuiti integrati hanno raggiunto i loro limiti fisici per dissipare il calore con raffreddamento ad aria [2]. Per questo motivo è impossibile continuare a cercare di aumentare la velocità di un circuito integrato, ciononostante l'incremento della densità dei transistor continuerà ancora per un po'.

Analizzando la storia dell'High Performance Computing in termini di evoluzioni tecnologiche si osserva un notevole incremento nello sviluppo di nuovi processori e l'impatto che hanno nella comunità scientifica. Vengono suddivisi in tre grandi epoche questi sviluppi tecnologici:

- **Epoca 1:** CRAY-1, il primo super computer che disponeva di un'architettura vettoriale e raggiungeva una potenza computazionale pari a 160 MegaFLOP.
- **Epoca 2:** la barriera dei MegaFLOP è stata infranta muovendosi da processori single-core verso processori multi-core. Il CRAY-2 disponeva di 4 core vettoriali che gli consentivano di raggiungere un potenza computazionale di 2 GigaFLOPs.
- **Epoca 3:** Per oltrepassare la barriera del GigaFLOP occorreva costruire nodi computazionali interconnessi tra loro mediante una rete di comunicazione ad alte prestazioni. CRAY-T3D è stato il primo a raggiungere una potenza computazionale di 1 TeraFLOP. La rete di interconnessione era un toroide a tre dimensioni che consentiva di ottenere una banda di trasmissione fino a 300 MB/Sec.

Dopo queste epoche per almeno 20 anni, non si sono rilevate innovazioni rilevanti. Le innovazioni tecnologiche si sono focalizzate principalmente su tre aspetti:

- Passare da un set di istruzioni a 8-bit, poi a 16-bit, poi a 32-bit ed infine a 64-bit.
- ILP (Instruction Level Parallelism) ovvero istruzioni che possono essere eseguite in parallelo.
- Incrementare il numero di core.

In figura 1.1 si può osservare uno schema che riassume le principali evoluzioni architetturali dei sistemi High Performance. Inizialmente si sviluppavano sistemi single-core dove la memoria centrale comunicava con una sola CPU; si è passati poi ad avere sistemi multi-core dove esiste una sola memoria centrale che però viene contesa dalle varie CPU. Infine si è passati ad una architettura a memoria distribuita, dove ogni CPU ha una sua memoria dedicata ed esiste un bus di interconnessione tra le varie CPU che consente il trasferimento dei dati e la comunicazione tra le varie CPU.

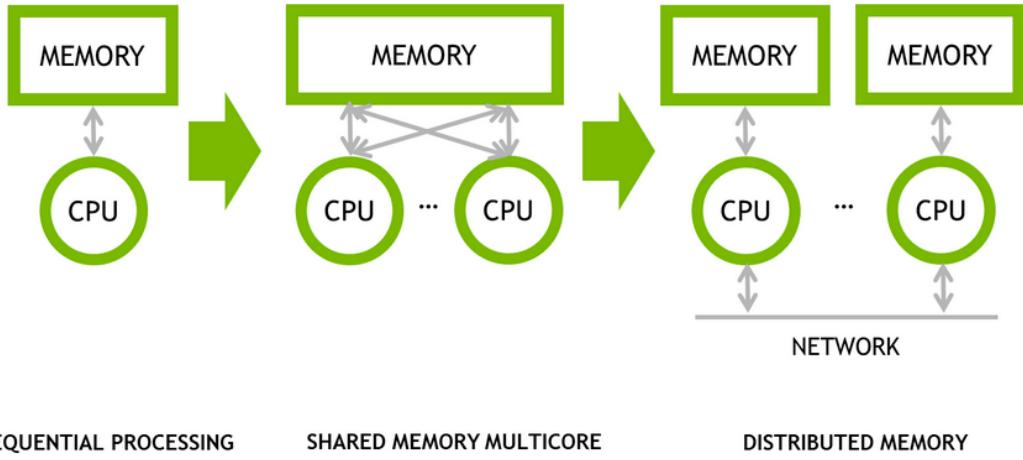


Figura 1.1: Evoluzione architetturale dei sistemi HPC

1.1.2 Il vantaggio dei programmi paralleli

Molti dei programmi scritti per sistemi a singolo processore non riescono a sfruttare appieno la presenza di più core. L'esecuzione di più istanze di un programma, ma non sarebbe l'obiettivo desiderato: se per esempio vengono eseguite più istanze di un videogioco, non si ottiene una maggiore fluidità oppure un maggiore dettaglio della grafica, si desidera invece che il videogioco sfrutti al massimo la presenza dei core del processore per avere una grafica più realistica, ad esempio. Talvolta per ottenere questo risultato occorre riscrivere il programma seriale affinché diventi un programma parallelo che sfrutti il parallelismo dell'architettura.

Non esistono strumenti o programmi che convertono programmi seriali in programmi paralleli, attualmente non esiste nulla di tutto ciò. Esistono diversi limiti che impediscono di effettuare questa conversione: dal momento che si possono scrivere programmi che riconoscono strutture comuni in programmi seriali e automaticamente traducono queste strutture in strutture parallele, la sequenza di costrutti paralleli può essere piuttosto inefficiente. Per questo motivo risulta molto difficile o talvolta impossibile "automatizzare" il processo di conversione di un programma da seriale a parallelo.

1.2 Il ruolo delle GPU

Ad oggi le GPU sono uno dei componenti cruciali in un computer. Inizialmente lo scopo di una scheda video (GPU) era quello di prendere un flusso di dati binari dalla CPU e renderizzarli su un display. Le moderne GPU sono in grado di effettuare calcoli complessi, come ricerche su enormi quantità di dati (big data research), machine learning ed Intelligenza artificiale. Le schede video si sono evolute nel tempo, passando da essere schede video single-core con una funzione ben specifica, ad essere schede programmabili con un numero molto elevato di core in grado di operare in parallelo.

1.2.1 Storia delle GPU

All'inizio del 1951, il MIT costruì *Whirlwind*, un simulatore per la marina militare statunitense. Viene considerato il primo sistema grafico 3D, ad oggi rappresenta la base di tutte le GPU. Nel 1976 RCA costruì il chip video *Pixie*, il quale era in grado di fornire in uscita un segnale video ad una risoluzione di 62x128 pixel. Hardware in grado di supportare la codifica RGB, sprite multicolor e tilemap background risalgono al 1979.

Nel 1981 IBM iniziò ad usare un adattatore monocromatico e a colori (MDA/CDA) nei suoi computer. Non è ancora una GPU moderna ma è una componente progettata per un solo scopo: visualizzare video. All'inizio era in grado di gestire 80 colonne e 25 righe di caratteri o simboli. *ISBX 275* creata da Intel, rappresenta la successiva rivoluzione: era in grado di gestire una profondità di colore pari a 8-bit con una risoluzione di 256x256 pixel, oppure 512x512 in gamma monocromatica.

Nel 1985 venne fondata la *ATI Technologies* questa azienda fu leader di mercato per anni con la sua linea di prodotti *Wonder*. Dal 1995 tutti i maggiori produttori di schede video introdussero un acceleratore 2D nelle proprie GPU. Il livello di integrazione delle schede video fu significativamente incrementato con la possibilità di sfruttare le Application Programming Interface (API) in modo da consentire agli sviluppatori di sfruttare a pieno le potenzialità dell'hardware.

Gli anni novanta hanno rappresentato l'era delle GPU: molte compagnie vengono fondate o acquistate. Tra le maggiori aziende leader del mercato in quegli anni troviamo NVIDIA la quale alla fine del 1997 deteneva circa il 25% del mercato mondiale per le schede video.

1.2.2 La rivoluzione del 3D

La storia delle moderne GPU inizia nel 1995 con l'introduzione del primi acceleratore 3D all'interno di una scheda video. In precedenza l'industria si era focalizzata sull'accelerazione 2D e le schede video avevano costi talvolta inaccessibili ai più.

La scheda video 3DFx's *Voodoo*, lanciata nel mercato a fine 1996, ebbe un successo tale che prese circa l'85% del mercato. Schede video che erano in grado di supportare solo l'accelerazione 2D divennero obsolete in breve tempo; *Voodoo1* non implementò nessun acceleratore 2D all'interno della scheda video, tanto è vero che era necessario affiancare una scheda video separata che si occupasse della parte 2D. Ciononostante fu un grande passo avanti per i videogiocatori. *Voodoo2* disponeva di tre chip a bordo e fu una delle prime GPU a supportare in parallelo il lavoro di due schede video in un singolo computer.

Con il progresso tecnologico nella costruzione di chip video, gli acceleratori 2D e 3D vennero integrati nello stesso chip. Nel 1999 apparse la prima vera GPU: NVIDIA lanciò sul mercato la *GeForce 256*. NVIDIA definì il termine **Graphics Processing Unit** come “*a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.* [3]”

1.2.3 General Purpose GPU (GPGPU)

L’era delle GPU general purpose inizia nel 2007 quando NVIDIA e ATI iniziano ad aggiungere alle proprie schede video molte più funzioni. Le due aziende però intraprendono due strade differenti per quanto concerne le GPGPU: nel 2007 NVIDIA rilascia l’ambiente di sviluppo *CUDA* [4], il primo (e maggiormente adottato) modello di programmazione per le computazioni con GPU. Due anni più tardi *OpenCL* [5] divenne largamente supportato nelle varie schede video. Questo framework consente di sviluppare codice sia per GPU che CPU con una particolare enfasi sulla portabilità. Così le GPU sono diventate dei dispositivi su cui si possono effettuare computazioni più generali.

Ad oggi le GPU non sono solo utili a fini grafici ma hanno trovato larga applicazione in diversi ambiti quali: machine learning, elaborazione di immagini scientifiche, algebra lineare, ricostruzioni 3D, ricerche mediche e molto altro. La tecnologia delle GPU tende ad aggiungere sempre più programmabilità e parallelismo ad una architettura che evolve sempre più verso a quella delle CPU.

1.2.4 L’utilizzo delle GPGPU

Sono molte le applicazioni, soprattutto in ambito scientifico, che richiedono di effettuare moltissime operazioni identiche su molti dati diversi. Per questo motivo le GPU possono venire in aiuto per migliorare le prestazioni: dal momento che le schede video sono state progettate per operare su migliaia di pixel di un’immagine o video in parallelo, la loro architettura si è evoluta negli anni aggiungendo un numero sempre più elevato di core che potessero operare concorrentemente in modo da velocizzare le computazioni dei nuovi pixel dell’immagine. Oggi non è difficile trovare schede video che dispongono di centinaia se non migliaia di questi core; questi ultimi però differiscono non poco dai core presenti nelle CPU, infatti i core delle GPU sono meno complessi e anche meno prestazionali rispetto a quelli presenti in un processore. Il vantaggio nell’utilizzare i core della GPU sta nel loro parallelismo massivo. Infatti per molte applicazioni avere a disposizione un numero molto elevato di core con ridotte prestazioni, ma ottimizzati per lavorare in parallelo, si rivela più vincente che avere a disposizione un numero ridotto di core ad alte prestazioni, ottimizzati per eseguire sequenzialmente le operazioni. Questa affermazione non è sempre vera: tipicamente per problemi di ridotte dimensioni la CPU riesce a produrre risultati utili in un tempo inferiore, ma dal momento in cui si deve lavorare a un problema di grandi dimensioni, sfruttare il parallelismo massivo delle GPU può portare a un notevole vantaggio.

Con gli anni i produttori di schede video stanno sviluppando prodotti studiati appositamente per essere molto più efficienti nell’effettuare calcoli a scopo generale piuttosto che essere ottimizzate per il rendering, ray tracing ecc. Questi prodotti si collocano in

una fascia differente di mercato: quella riservata ai datacenter ad esempio. L'architettura di queste schede video si discosta da quelle tradizionali, in quanto si predilige la velocità in certi tipi di calcoli oltre ad avere una velocità di banda di memoria molto elevata, in quanto è richiesto trasferire moltissimi dati dalla GPU alla CPU (e viceversa) in tempi molto brevi.

In figura 1.2 possiamo osservare la differenza architettonica tra una CPU, che dispone di un numero limitato di core e una GPU che invece dispone di un numero molto più elevato di core.

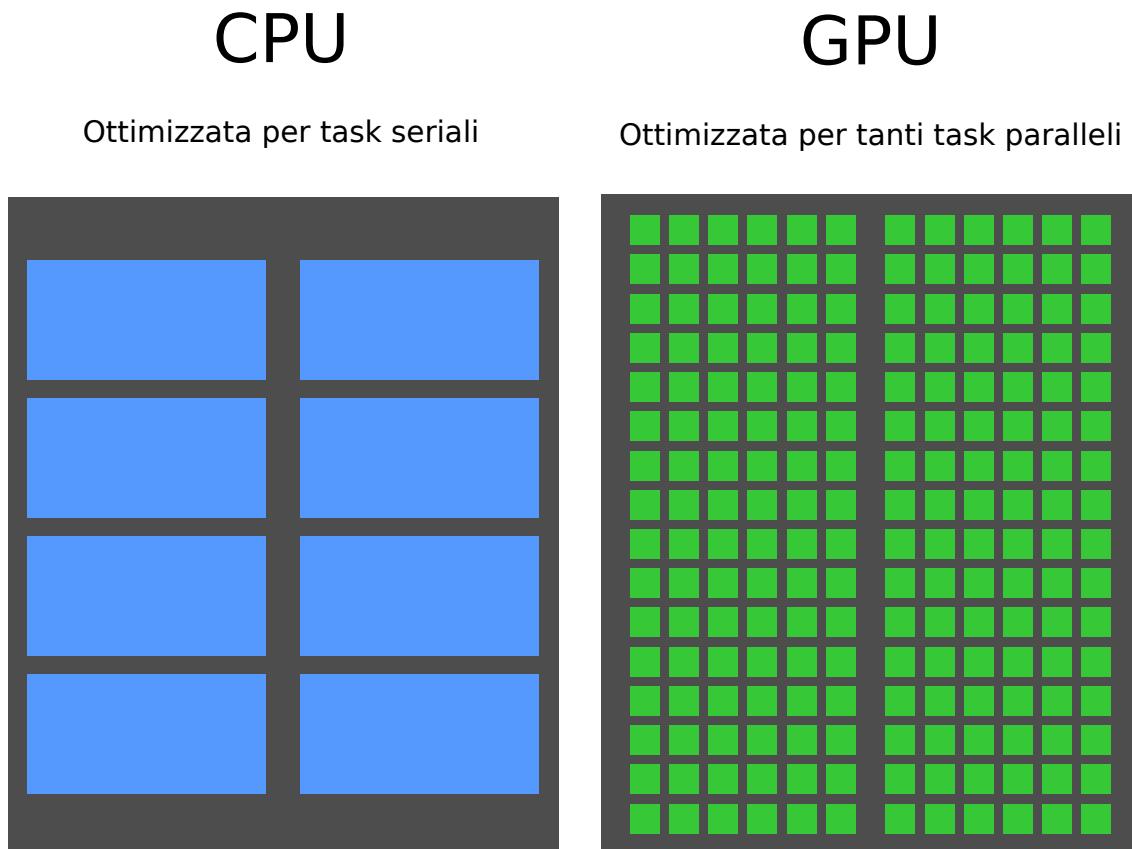


Figura 1.2: Confronto architettura CPU e GPU. I rettangoli di colore blu rappresentano i core di una CPU, i quadrati di colore giallo rappresentano i core di una GPU.

1.2.5 La fine della legge di Moore e l'inizio dell'HPC

La società moderna dipende sempre più fortemente dalla rapida crescita delle prestazioni dei calcolatori in grado di migliorare vari aspetti della nostra vita quotidiana e lavorativa. La crescita delle capacità computazionali è regolata principalmente dalla legge di Moore: si tratta di un modello tecnico-economico che ha predetto (sin dal 1968, anno in cui è stata pubblicata) che l'industria informatica avrebbe raddoppiato le prestazioni e le funzionalità di un prodotto (nel caso specifico si parla di microchip) ogni 18 mesi circa, tenendo fissi il costo, l'energia consumata e la superficie di ingombro [6]. Per quanto riguarda ingombro e consumi non è più così: con l'avvento dei dispositivi mobili come smartphone, laptop ecc. i consumi e le dimensioni hanno iniziato ad avere un ruolo estremamente rilevante, a tal punto che le aziende hanno deciso di investire molto su questi aspetti ottenendo risultati soddisfacenti. Possiamo quindi semplificare la legge di Moore dicendo che ogni 18 mesi la densità dei transistor all'interno di un chip raddoppiava (di conseguenza anche le prestazioni tendevano a raddoppiare) ad un costo che non variava. In figura 1.3 viene mostrato il livello di integrazione di transistor per ogni anno.

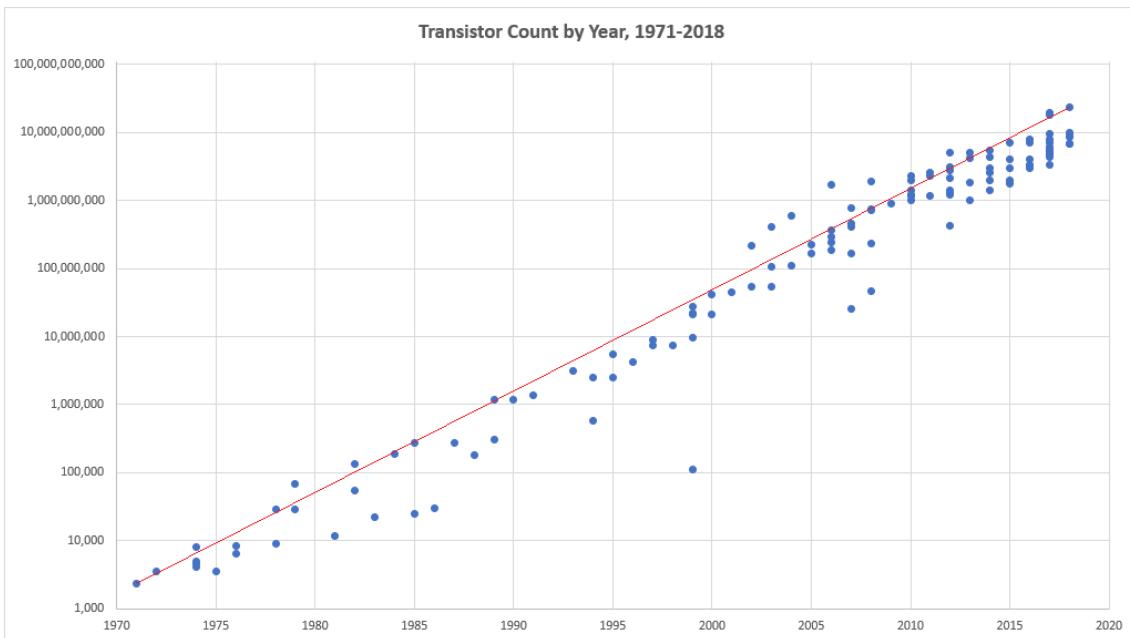


Figura 1.3: Andamento legge di Moore dal 1971 fino al 2018.

Fonte: <https://medium.com/predict/moores-law-is-alive-and-well-eaa450188>

Questa legge ha portato una sorta di stabilità nell'ecosistema tecnologico: compilatori, simulatori, emulatori ecc. vennero costruiti per processori ad uso generale come x86, ARM e Power PC. Ciononostante però, da circa un decennio il valore di questo modello inizia sempre più a perdere valore: l'aumento dell'integrazione dei transistor nei microchip, riducendone sempre più le dimensioni, ha portato a raggiungere la dimensione atomica per i transistor[7]. Questo significa che risulta molto difficile creare litografie in grado di produrre dispositivi con transistor di dimensioni pari a 3-5 nm, poiché questa dimensione corrisponde a qualche dozzina (o anche meno) di atomi di silicio rendendo molto difficile

controllare le cariche che fluiscono tra gli atomi, processo che sta alla base del funzionamento del transistor. Il classico processo tecnologico osservato fino ad oggi e che ha sempre seguito la legge di Moore per i 50 anni passati, sembra non rispecchiare più la realtà dei fatti a tal punto che nel 2025 si pensa che questo modello non sarà più valido.

Senza una nuova tecnologia di transistor in grado di soppiantare l'attuale **CMOS** (Complementary Metal-Oxide Semiconductor), l'opportunità principale per continuare ad incrementare le prestazioni per i dispositivi elettronici, sistemi HPC ecc. è quella di sfruttare in maniera più efficiente le architetture che abbiamo a disposizione. Per questo motivo si rende necessario ottimizzare i programmi affinché sfruttino al massimo le potenzialità offerte dall'architettura: si può pensare di programmare sistemi che meglio sfruttano il trasferimento dei dati in memoria, in quanto ad oggi la memoria centrale rappresenta un collo di bottiglia che penalizza fortemente le prestazioni delle applicazioni.

Alcune delle cause che portano a dover ottimizzare i programmi invece che aspettare architetture più prestazionali sono elencate di seguito:

- La fine della crescita delle frequenze di clock: questo limite (raggiunto ormai da parecchi anni) ha contribuito a sfruttare in maniera massiva il parallelismo offerto dalle architetture che lo supportano.
- La fine della riduzione della dimensione dei transistor mediante il processo fotolitografico: questo ha portato ad un grande aumento delle architetture eterogenee per poter incrementare le prestazioni (un esempio è l'uso di GPU per eseguire calcoli a scopo generale).
- Gestione aggressiva della potenza e contesa delle risorse con conseguenti tassi di esecuzione eterogenei: questo produce eterogeneità delle prestazioni che è in contrapposizione ai modelli di programmazione attuali).

In conclusione il progresso tecnologico nel campo della costruzione dei microchip mediante fotolitografia non è ai livelli di qualche decennio fa. Questo porta gli sviluppatori di software a dover sviluppare programmi che sfruttino al massimo le potenzialità delle architetture di cui dispongono per raggiungere prestazioni sempre più elevate; in tal senso stanno nascendo sempre più architetture eterogenee che consentono di migliorare le prestazioni delle applicazioni, ma il cui utilizzo richiede uno sforzo notevolmente più elevato rispetto a quello a cui si era abituati per architetture omogenee.

Fino a quando non verranno scoperte nuove tecniche per la costruzione di microchip in grado di superare le prestazioni dei chip attuali, l'unico modo di migliorare le prestazioni delle applicazioni è quello di ottimizzare al massimo le risorse a disposizione.

Capitolo 2

CUDA e Tensor Core

In questo capitolo introduciamo CUDA e l'architettura tipica di una GPU. Vedremo quindi il ruolo dei Tensor Core all'interno della scheda video e quali campi di utilizzo possono avere. Infine mostreremo qualche esempio di utilizzo dei Tensor Core e le prestazioni che riescono a raggiungere. Si precisa che in questo contesto i termini GPU, GPGPU e scheda video rappresentano la medesima cosa.

2.1 Architettura di una CUDA-capable GPU

CUDA è l'acronimo di *Compute Unified Device Architecture*, una piattaforma per il calcolo parallelo sviluppata da NVIDIA, la quale mette a disposizione del programmatore API che consentono la programmazione di GPU compatibili con CUDA per realizzare programmi ad uso generale e non specifici per la grafica. La piattaforma CUDA è uno strato che consente di accedere direttamente al set virtuale di istruzioni della GPU e agli elementi computazionali paralleli, per l'esecuzione di kernel.

L'architettura CUDA è stata progettata per lavorare con linguaggi come *C*, *C++* e *Fortran* rendendo più semplice agli esperti in programmazione parallela l'uso delle risorse della GPU. L'uso di un dialetto dei linguaggi sopracitati consente di sviluppare codice più agevolmente, piuttosto che apprendere un nuovo linguaggio dedicato per la programmazione delle GPU. Le GPU NVIDIA predisposte per CUDA si possono programmare anche in altri modi: famosi framework come OpenACC [8] e OpenGL sono comunque supportati dalle GPU che prevedono CUDA. Questa compatibilità con altri framework non limita le GPU NVIDIA ad essere programmate solo con CUDA.

In figura 2.1 è schematizzato il collegamento tra CPU e GPU; tutte le comunicazioni avvengono mediante il bus PCI, il che consente di ottenere una discreta banda di trasmissione dei dati.

Un esempio di flusso di un programma CUDA è il seguente:

1. Si copiano i dati dalla memoria centrale (RAM) alla memoria della GPU.
2. La CPU si occupa di inizializzare i kernel che devono essere eseguiti sulla GPU.
3. I CUDA core eseguono in parallelo il kernel.
4. Viene copiato il risultato dalla memoria della GPU alla memoria centrale (RAM).

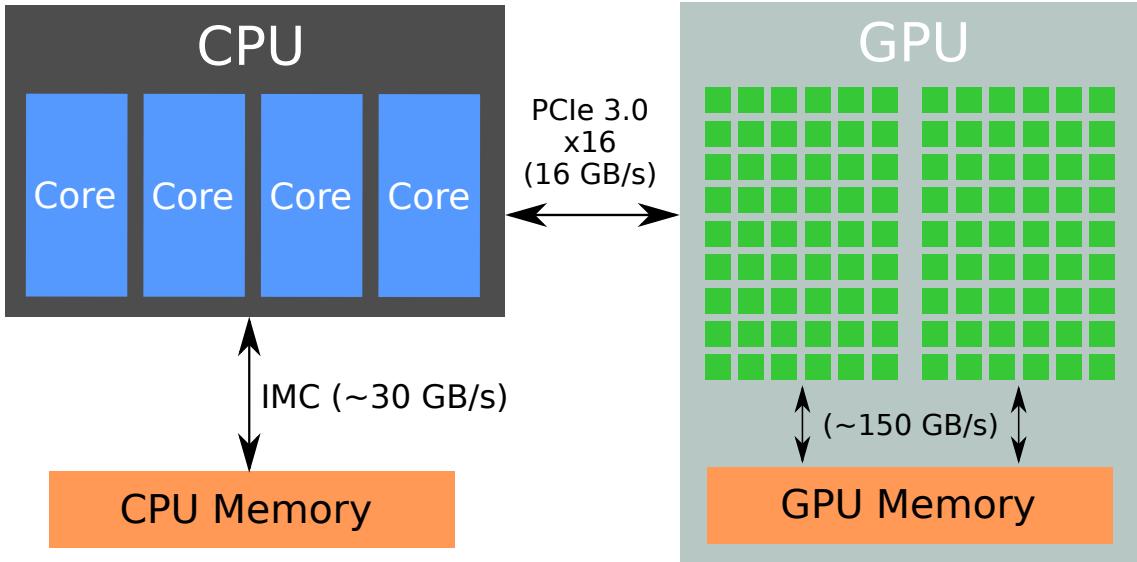


Figura 2.1: Flusso dati da CPU a GPU e viceversa

Architetturalmente una GPU CUDA è composta da uno o più *Streaming Multiprocessor* (SMs), ovvero un’astrazione dell’hardware: ogni SM è a sua volta formato da più *Streaming Processor* anche chiamati **CUDA Core**. Ogni CUDA Core ha accesso ad una cache chiamata anche memoria condivisa (shared memory), questa memoria è più veloce della memoria globale (global memory); la memoria condivisa impone alcune limitazioni: ad esempio può essere acceduta solo dai thread dello stesso *Streaming Multiprocessor* e quindi il suo contenuto non può essere visto da altri thread che non appartengono allo stesso SM.

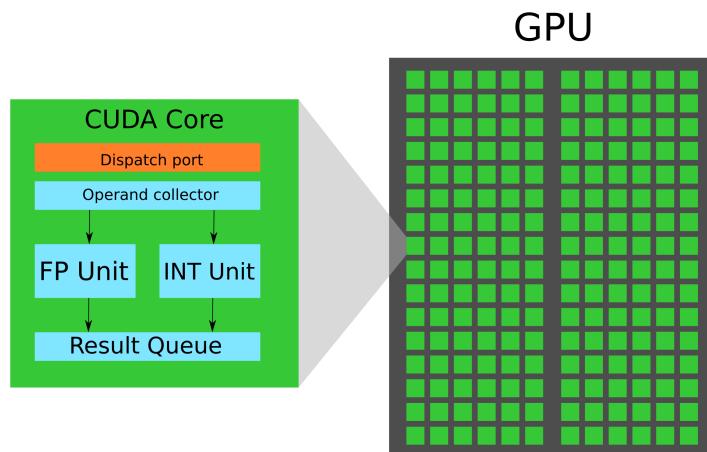


Figura 2.2: Architettura interna di un singolo CUDA core.

Il numero di SP/CUDA Core (in figura 2.2 viene mostrata l'architettura interna di un CUDA Core) dipende strettamente dal dispositivo utilizzato, è però importante ricordare che esistono modelli con qualche migliaia di core all'interno di una GPU. Ad esempio una scheda video NVIDIA Tesla V100 dispone di 5120 CUDA Core, una banda di memoria fino a 900 GB/s e una memoria globale di 16 GB. Ogni CUDA Core dispone di una ALU e una FPU, utilizza lo standard *IEEE 754-2008* per la rappresentazione dei numeri in virgola mobile (quindi pienamente compatibile con la rappresentazione adottata dalle CPU moderne), dispone di un modulo chiamato FMA (Fuse Multiply Add) in grado di operare su numeri in virgola mobile sia a singola che a doppia precisione. Il modulo FMA migliora l'operazione chiamata Multiply/Add (MAD) in quanto effettua la moltiplicazione e a seguito la somma con un solo arrotondamento finale, questo evita di perdere precisione nella somma; quindi il modulo FMA è più accurato ed è in grado di operare separatamente i calcoli. L'ALU nelle prime CUDA-capable GPU erano limitate a 24-bit per le moltiplica-

cazioni ma, a partire dall’architettura *Fermi*, venne ridisegnata l’ALU e resa a 32-bit per tutte le istruzioni aderendo di conseguenza agli standard dei linguaggi di programmazione. L’ALU è stata poi ulteriormente migliorata prevedendo l’estensione a 64-bit per le operazioni aritmetiche.

Sono inoltre supportate tutte le operazioni booleane, shifting, confronti, conversioni di tipo, bit-field extract, bit-reverse insert e population count.

Ogni SM dispone di 16 load/store unit, consentendo di calcolare gli indirizzi sorgente/destinazione per sedici thread ad ogni ciclo di clock. Queste unità sono in grado di operare sia sui dati della DRAM che sulla cache. Ogni scheda video dispone anche di almeno quattro *Special Function Unit* (SFUs) le quali eseguono istruzioni trascendenti come seno, coseno, reciproco e radice quadrata. Ogni SFU esegue un’istruzione per thread per ciclo di clock; la pipeline della SFU è disaccoppiata dalla dispatch unit consentendo a quest’ultima di servire altre unità mentre la SFU è occupata. La *dispatch unit* è quella unità responsabile di verificare che i thread siano mandati in esecuzione: prende l’istruzione da eseguire, il warp ID e la thread mask, quindi calcola l’ID del thread su cui mandare in esecuzione l’istruzione. Terminata l’esecuzione dell’istruzione, il dispatcher invia ogni istruzione del thread ad una unità funzionale libera; nel caso in cui non ci siano unità funzionali libere, allora la dispatch unit mette in coda l’istruzione del thread per il ciclo successivo. Esistono almeno 4 unità funzionali: **Load/Store memory unit**, **Double precision floating-point unit**, **Special function unit** (utilizzata per approssimare le funzioni trascendenti) e unità **”cores”** che si occupa di gestire tutte le restanti operazioni.

Ogni SM organizza i thread in gruppi di 32 thread paralleli chiamati **warp**. Ogni SM dispone di almeno due scheduler per i warp e almeno due instruction dispatcher unit, consentendo quindi ad almeno due warp di essere preparati e mandati in esecuzione parallelamente. Dal momento che i warp eseguono le operazioni indipendentemente tra loro, lo scheduler non si deve preoccupare di controllare dipendenze all’interno del flusso di istruzioni. L’utilizzo di questa tecnica consente all’hardware di raggiungere prestazioni pressoché massime.

Una delle più grandi innovazioni architettoniche è la memoria condivisa: si tratta di una memoria che solo i thread di uno stesso thread block possono vedere in modo da facilitare il riuso dei dati in quanto non è necessario ogni volta rileggerli dalla memoria centrale, pratica che creerebbe un elevato traffico nel bus della memoria. In termini pratici questa memoria (shared memory) è una sorta di cache abbastanza grande e quindi con velocità di lettura/scrittura più elevate rispetto alla memoria centrale; quello che solitamente si fa è: copiare una volta sola all’inizio del programma i dati che servono dalla memoria centrale alla memoria condivisa, quindi i thread eseguono i calcoli andando a leggere i dati dalla shared memory (anche più volte se necessario) memorizzano i dati sempre nella shared memory. Infine i risultati vengono copiati in massa dalla memoria condivisa alla memoria centrale. La shared memory è molte volte la chiave per il successo in molte applicazioni HPC.

L’uso della shared memory può risultare vantaggiosa per molte applicazioni, ma non per tutte. Alcuni algoritmi si adattano bene alla presenza della memoria condivisa, altri richiedono la cache, mentre alcuni necessitano una combinazione di entrambi. La struttura di memoria (schematizzata in figura 2.3) ottimale dovrebbe offrire i benefici sia della shared memory che della cache e consentire al programmatore di scegliere quale usare e come.

Con le nuove generazioni di GPU tutto ciò è possibile: l'architettura Fermi implementa un unico percorso per le load/store nella memoria con una cache L1 per SM e una cache L2 unificata per servire tutte le operazioni (load, store e texture). La cache L1 presente in ogni SM può essere configurata per supportare sia la shared memory che la cache. I 64 KB di memoria possono essere configurati per essere distribuiti in 48 KB di memoria condivisa e 16 KB per la cache L1; viceversa si può configurare affinché 16 KB siano destinati alla memoria condivisa e 48 KB alla cache L1.

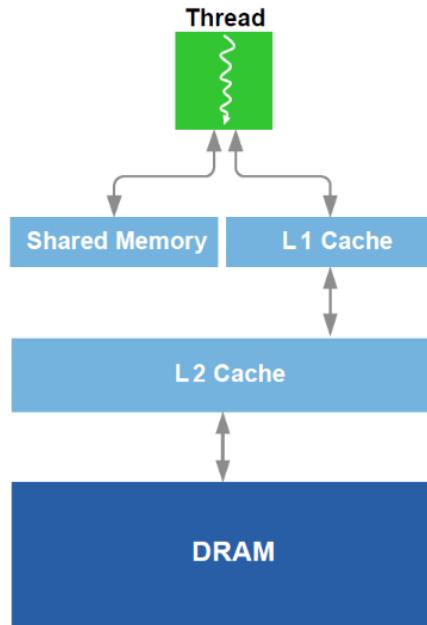


Figura 2.3: Gerarchia delle memorie all'interno di una GPU NVIDIA.

Come le CPU, le GPU supportano il multitasking attraverso l'uso del cambio di contesto, ovvero viene riservato alle applicazioni un intervallo di tempo detto *time slice* dove possono usufruire delle risorse. Con il tempo le pipeline delle GPU si sono evolute in modo da ridurre il tempo che occorre per ripristinare il contesto qualora si renda necessario cambiarlo (*cambio di contesto*); questo tempo è al di sotto dei 20 microsecondi, che però, comparato con i tempi impiegati dalla CPU (circa 1.5 microsecondi), è ben più lungo. Questo divario viene mitigato dal fatto che, come dicevamo in precedenza, coesistono più scheduler all'interno di uno SM ed essendo indipendenti tra loro aumentano il parallelismo migliorando le prestazioni.

A partire dall'architettura Fermi è possibile mandare in esecuzione kernel in parallelo, ovvero kernel diversi della stessa applicazione possono essere eseguiti parallelamente. L'esecuzione concorrente di più kernel consente al programma di sfruttare al massimo la potenzialità della GPU nel caso in cui siano implementati kernel di piccole dimensioni o che effettuano calcoli molto semplici. Se per esempio prendiamo un simulatore fisico che deve invocare un kernel per risolvere il comportamento di un fluido su un corpo rigido, se si eseguissero i kernel sequenzialmente si utilizzerebbe solo la metà dei thread disponibili nel processore; con questa tecnica quindi è possibile eseguire in parallelo kernel differenti

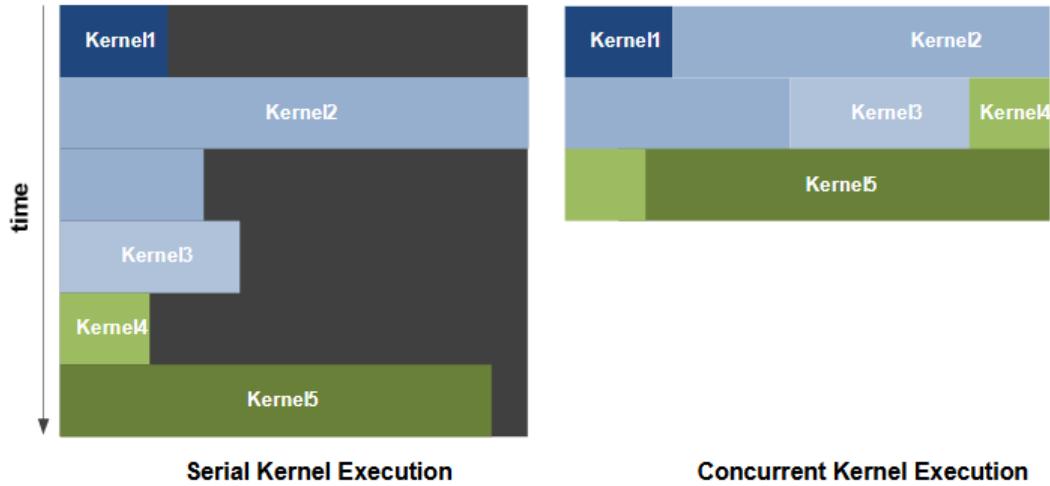


Figura 2.4: Confronto tempi di esecuzione di uno stesso programma in cui a destra viene mostrata l'esecuzione seriale dei kernel, mentre a sinistra l'esecuzione parallela.

per il calcolo del risultato finale. In figura 2.4 è mostrato lo schema che riassume questo concetto.

2.1.1 Griglie e blocchi in CUDA

Il framework CUDA prevede tre astrazioni fondamentali [9]:

- Una gerarchia di gruppi di thread.
- Memoria condivisa (Shared memory)
- Barriere per la sincronizzazione

Queste tre astrazioni sono disponibili al programmatore sotto forma di estensioni del linguaggio di programmazione.

CUDA prevede un parallelismo a grana fine a livello di dati e thread, il tutto incapsulato in un parallelismo a grana grossa per quanto riguarda il parallelismo dei task. Con questo metodo si spinge il programmatore a suddividere il problema in "grandi" sotto problemi che possono essere risolti indipendentemente in parallelo da blocchi di thread e ogni sotto problema può essere risolto in modo cooperativo in parallelo da tutti i thread dello stesso blocco.

Un kernel è eseguito in parallelo da un array di thread:

- Tutti i thread eseguono lo stesso identico codice.
- Ogni thread ha un identificativo che si usa per l'indirizzamento di memoria e prendere decisioni.

I thread sono organizzati come una griglia di blocchi (figura 2.5):

- Kernel diversi possono avere una configurazione diversa di griglie/blocchi.
- Thread dello stesso blocco hanno accesso alla stessa memoria condivisa e la loro esecuzione può essere sincronizzata.

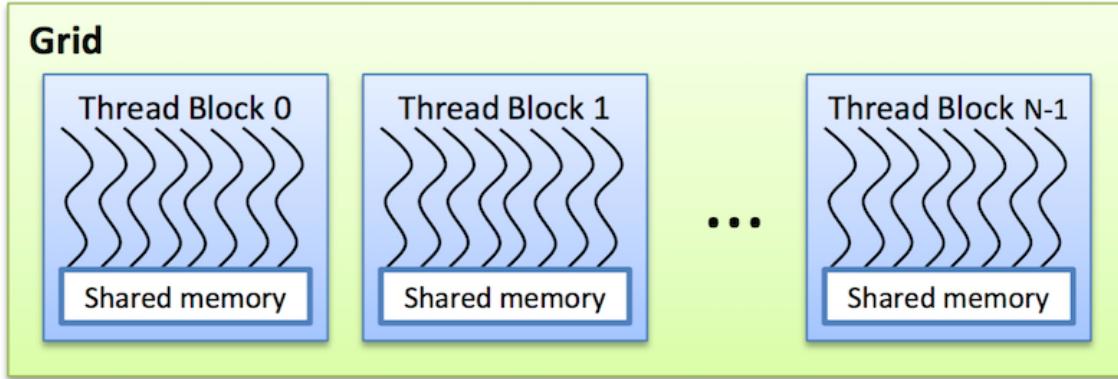


Figura 2.5: Organizzazione dei thread in blocchi e griglie.

2.1.2 Tensor Core

Come visto nelle sezione precedente, i CUDA Core effettuano una singola istruzione ben precisa per ciclo di clock della GPU: questo significa che la frequenza del clock gioca un ruolo fondamentale nelle prestazioni di CUDA (ovviamente anche il numero di CUDA Core influenza le prestazioni). I **Tensor Core**, dall’altro lato, possono effettuare una moltiplicazione di una matrice 4×4 in un singolo ciclo di clock con dati in virgola mobile a 16-bit di precisione (fp16) e sommare una terza matrice (sempre di dimensioni 4×4) con precisione in virgola mobile pari a 32-bit o 16-bit (vedi figura 2.6), in quanto l’accumulatore opera in *”mixed-precision”*, il che significa che può prendere in input un numero a 16-bit e restituire come risultato un numero a 32-bit. Mettendo quindi a confronto CUDA Core con Tensor Core possiamo affermare che i CUDA Core sono più lenti ma offrono una precisione maggiore; al contrario i Tensor Core sono sensibilmente più veloci ma hanno una precisione sensibilmente inferiore (con precisione in questo caso ci riferiamo all’ampiezza dei calcoli in virgola mobile che sono in grado di effettuare).

TENSOR CORE 4X4X4 MATRIX-MULTIPLY ACC

$$\mathbf{D} = \left(\begin{array}{cccc} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \mathbf{A}_{0,2} & \mathbf{A}_{0,3} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} \\ \mathbf{A}_{2,0} & \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} \\ \mathbf{A}_{3,0} & \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} \end{array} \right) \text{FP16 or FP32} \quad \left(\begin{array}{cccc} \mathbf{B}_{0,0} & \mathbf{B}_{0,1} & \mathbf{B}_{0,2} & \mathbf{B}_{0,3} \\ \mathbf{B}_{1,0} & \mathbf{B}_{1,1} & \mathbf{B}_{1,2} & \mathbf{B}_{1,3} \\ \mathbf{B}_{2,0} & \mathbf{B}_{2,1} & \mathbf{B}_{2,2} & \mathbf{B}_{2,3} \\ \mathbf{B}_{3,0} & \mathbf{B}_{3,1} & \mathbf{B}_{3,2} & \mathbf{B}_{3,3} \end{array} \right) \text{FP16} + \left(\begin{array}{cccc} \mathbf{C}_{0,0} & \mathbf{C}_{0,1} & \mathbf{C}_{0,2} & \mathbf{C}_{0,3} \\ \mathbf{C}_{1,0} & \mathbf{C}_{1,1} & \mathbf{C}_{1,2} & \mathbf{C}_{1,3} \\ \mathbf{C}_{2,0} & \mathbf{C}_{2,1} & \mathbf{C}_{2,2} & \mathbf{C}_{2,3} \\ \mathbf{C}_{3,0} & \mathbf{C}_{3,1} & \mathbf{C}_{3,2} & \mathbf{C}_{3,3} \end{array} \right) \text{FP16 or FP32}$$

8 NVIDIA.

Figura 2.6: Funzionamento architettura Tensor Core.

Fonte: NVIDIA.

L'architettura *Turing* ha introdotto un miglioramento ai Tensor Core: è prevista l'operabilità con dati di tipo INT4 e INT8 per operazioni di inferenza che possono tollerare errori di quantizzazione. L'introduzione di questa novità consente di accelerare ulteriormente le applicazioni basate su *Intelligenza Artificiale*. Quindi, anziché usare molti CUDA Core e molti cicli di clock, per ottenere lo stesso task con Tensor Core occorre solamente un ciclo di clock e questo causa un drammatico incremento delle prestazioni. NVIDIA stima che lo stesso calcolo effettuato con Tensor Core risulta essere più veloce di circa 12 volte rispetto ai CUDA Core. Le applicazioni di Machine Learning hanno tratto un enorme vantaggio da questa nuova architettura ma, se ci spostiamo nell'ambito dell'High Performance Computing, questa architettura può avere alcune limitazioni. NVIDIA rende accessibile al programmatore delle API per poter sfruttare i Tensor Core:

- **WMMA API:** API di basso livello che consentono di accedere ai Tensor Core.
- **cuBLAS:** API blas-compliant che sfruttano i Tensor Core per operazioni di algebra lineare.
- **cuDNN:** API che sfruttano i Tensor Core per calcoli come la convoluzione ecc. utili per reti neurali.

Per l'HPC le prime due rappresentano una soluzione pratica; la prima libreria è stata progettata appositamente per poter realizzare algoritmi che sfruttano la moltiplicazione di matrici come operazione base, mentre la seconda (cuBLAS) è stata progettata ad un livello di astrazione più alto e specifica per operazioni di algebra lineare come moltiplicazioni vettore/vettore, matrice/vettore, matrice/matrice ecc prestando particolare attenzione all'efficienza e alle prestazioni. Una comparativa sulle prestazioni della moltiplicazione

tra matrici con uso di Tensor Core e con diverse librerie la si può trovare nell'articolo *"NVIDIA Tensor Core Programmability, Performance & Precision"*[10].

2.1.3 Architettura dei Tensor Core

L'uso dei Tensor Core è determinante in tutte quelle applicazioni in cui si devono moltiplicare matrici di grandi dimensioni; se invece occorre moltiplicare matrici di piccole dimensioni allora questa architettura potrebbe avere prestazioni non ottimali [11].

Vediamo ora in pratica come operano i Tensor Core: come accennato prima, effettuano una sola operazione di moltiplicazione tra due matrici 4×4 e accumulo di una terza. In pratica però i Tensor Core lavorano su matrici 16×16 in cui le operazioni vengono gestite su due Tensor Core alla volta, questa particolarità sembra essere dovuta alla nuova architettura *Volta* e più specificatamente come questi Tensor Core siano collocati in uno SM (Streaming Multiprocessor). Per l'architettura *Volta* gli SM sono stati suddivisi in quattro blocchi di elaborazione o sub-core (vedi figura 2.7):

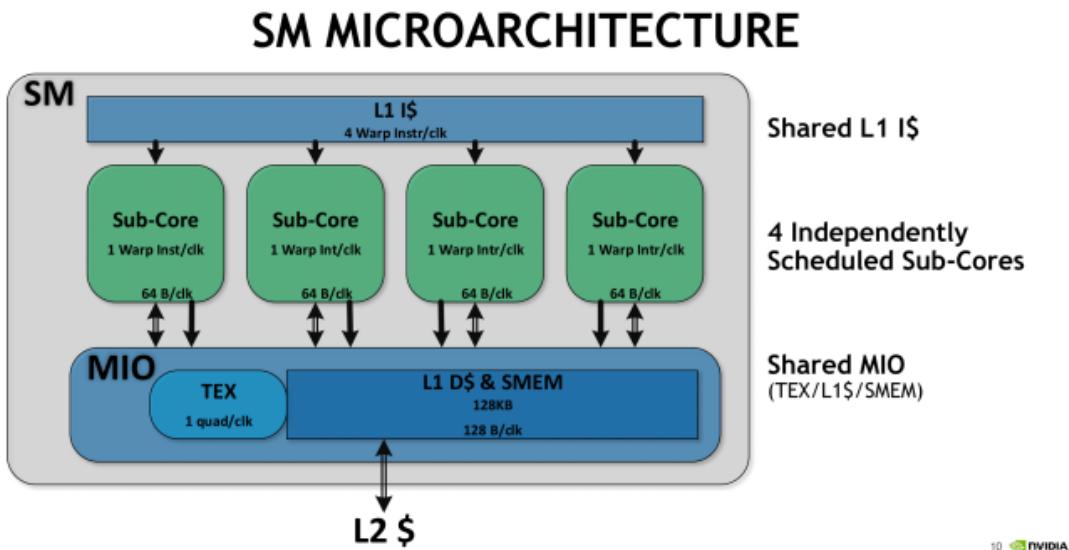


Figura 2.7: Architettura di uno Streaming Multiprocessor nell'architettura Volta.

per ogni sub-core lo scheduler invia un istruzione di un warp alla **Local Branch Unit** per ciclo di clock, all'array di **Tensor Core**, alla **Math Dispatcher Unit** o alla unità **MIO** condivisa. Se avessimo un solo Tensor Core per sub-core, ci precluderemo la possibilità di eseguire l'operazione su Tensor Core e simultaneamente altre operazioni matematiche. Utilizzando due Tensor Core il warp-scheduler invia l'operazione di moltiplicazione di matrici e, dopo aver ricevuto le matrici in input dal registro, effettua la moltiplicazione. Una volta terminata la moltiplicazione il Tensor Core scrive il risultato nel registro. In figura 2.8 viene mostrata l'architettura interna di un sub-core in cui si evidenziano i moduli che lo compongono e la loro interazione con gli altri.

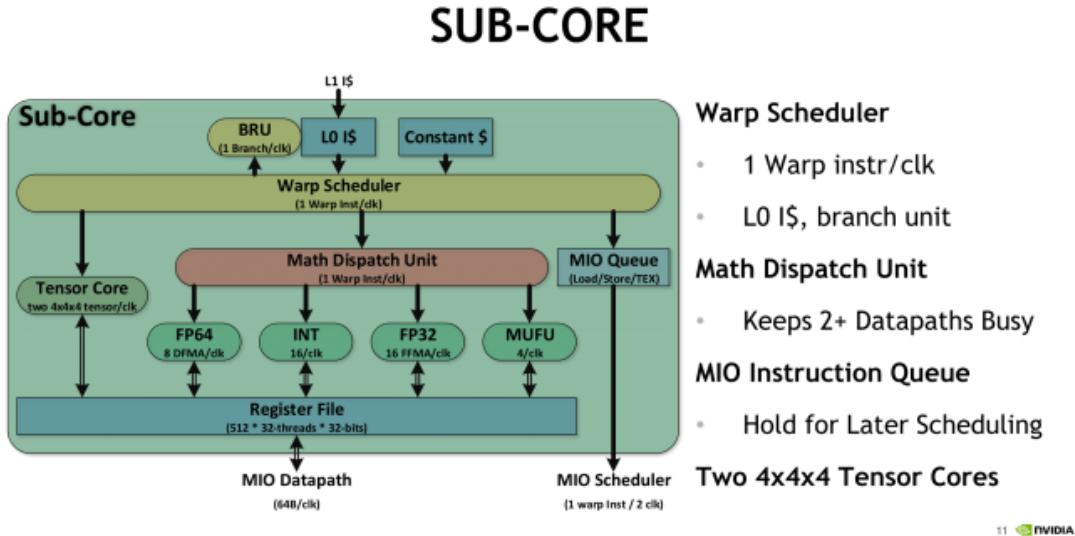


Figura 2.8: Struttura interna di un sub-core nell’architettura Volta in cui si mostrano le unità fondamentali che lo compongono.

Un analisi delle prestazioni a basso livello [12] effettuata da un team della Citadel LLC ha rivelato un numero di dettagli sull’architettura Volta, incluse le operazioni dei Tensor Core, i fragment, le locazioni dei registri e l’identità dei warp che sono coinvolti nelle matrici di input. Attualmente i programmati CUDA possono usare delle primitive warp-level per effettuare la moltiplicazione del tipo 16x16x16 in half-precision con i Tensor Core. Andremo quindi ad analizzare in dettaglio il meccanismo di computazione che i Tensor Core adottano; in questi esempi consideriamo il caso in cui le matrici siano memorizzate in column-major order. Nell’esempio consideriamo le matrici di dimensione 16x16. Gli elementi delle matrici A e B sono float in half-precision (FP16), mentre in C abbiamo un float in single-precision (FP32).

Prima di invocare la moltiplicazione, il programmatore deve caricare i dati dalla memoria (DRAM) ai registri con la primitiva `wmma::load_matrix_sync`. Il compilatore NVCC traduce questa primitiva in una serie di istruzioni per il caricamento dei dati dalla memoria. A tempo di esecuzione ogni thread carica 16 elementi dalla matrice A e 16 elementi dalla matrice B .

Nell’articolo redatto dal team della Citadel LLC viene descritta la relazione tra le posizioni interne di ciascuna matrice; hanno inoltre scoperto che gli indici dei thread che caricano i corrispondenti dati nei registri sono fissi. In figura 2.9 e in figura 2.10 sono riportati gli indici dei thread che operano rispettivamente sulle matrici A e B . A tempo di esecuzione due thread differenti caricano i dati da una posizione in A e in B all’interno dei propri registri.

0 (0,8)	32 (1,9)	64 (2,10)	96 (3,11)	128 (0,8)	160 (1,9)	192 (2,10)	224 (3,11)	256 (0,8)	288 (1,9)	320 (2,10)	352 (3,11)	384 (0,8)	416 (1,9)	448 (2,10)	480 (3,11)
2 (0,8)	34 (1,9)	66 (2,10)	98 (3,11)	130 (0,8)	162 (1,9)	194 (2,10)	226 (3,11)	258 (0,8)	290 (1,9)	322 (2,10)	354 (3,11)	386 (0,8)	418 (1,9)	450 (2,10)	482 (3,11)
4 (0,8)	36 (1,9)	68 (2,10)	100 (3,11)	132 (0,8)	164 (1,9)	196 (2,10)	228 (3,11)	260 (0,8)	292 (1,9)	324 (2,10)	356 (3,11)	388 (0,8)	420 (1,9)	452 (2,10)	484 (3,11)
6 (0,8)	38 (1,9)	70 (2,10)	102 (3,11)	134 (0,8)	166 (1,9)	198 (2,10)	230 (3,11)	262 (0,8)	294 (1,9)	326 (2,10)	358 (3,11)	390 (0,8)	422 (1,9)	454 (2,10)	486 (3,11)
8 (16,24)	40 (17,25)	72 (18,26)	104 (19,27)	136 (16,24)	168 (17,25)	200 (18,26)	232 (19,27)	264 (16,24)	296 (17,25)	328 (18,26)	360 (19,27)	392 (16,24)	424 (17,25)	456 (18,26)	488 (19,27)
10 (16,24)	42 (17,25)	74 (18,26)	106 (19,27)	138 (16,24)	170 (17,25)	202 (18,26)	234 (19,27)	266 (16,24)	298 (17,25)	330 (18,26)	362 (19,27)	394 (16,24)	426 (17,25)	458 (18,26)	490 (19,27)
12 (16,24)	44 (17,25)	76 (18,26)	108 (19,27)	140 (16,24)	172 (17,25)	204 (18,26)	236 (19,27)	268 (16,24)	300 (17,25)	332 (18,26)	364 (19,27)	396 (16,24)	428 (17,25)	460 (18,26)	492 (19,27)
14 (16,24)	46 (17,25)	78 (18,26)	110 (19,27)	142 (16,24)	174 (17,25)	206 (18,26)	238 (19,27)	270 (16,24)	302 (17,25)	334 (18,26)	366 (19,27)	398 (16,24)	430 (17,25)	462 (18,26)	494 (19,27)
16 (4,12)	48 (5,13)	80 (6,14)	112 (7,15)	144 (4,12)	176 (5,13)	208 (6,14)	240 (7,15)	272 (4,12)	304 (5,13)	336 (6,14)	368 (7,15)	400 (4,12)	432 (5,13)	464 (6,14)	496 (7,15)
18 (4,12)	50 (5,13)	82 (6,14)	114 (7,15)	146 (4,12)	178 (5,13)	210 (6,14)	242 (7,15)	274 (4,12)	306 (5,13)	338 (6,14)	370 (7,15)	402 (4,12)	434 (5,13)	466 (6,14)	498 (7,15)
20 (4,12)	52 (5,13)	84 (6,14)	116 (7,15)	148 (4,12)	180 (5,13)	212 (6,14)	244 (7,15)	276 (4,12)	308 (5,13)	340 (6,14)	372 (7,15)	404 (4,12)	436 (5,13)	468 (6,14)	500 (7,15)
22 (4,12)	54 (5,13)	86 (6,14)	118 (7,15)	150 (4,12)	182 (5,13)	214 (6,14)	246 (7,15)	278 (4,12)	310 (5,13)	342 (6,14)	374 (7,15)	406 (4,12)	438 (5,13)	470 (6,14)	502 (7,15)
24 (20,28)	56 (21,29)	88 (22,30)	120 (23,31)	152 (20,28)	184 (21,29)	216 (22,30)	248 (23,31)	280 (20,28)	312 (21,29)	344 (22,30)	376 (23,31)	408 (20,28)	440 (21,29)	472 (22,30)	504 (23,31)
26 (20,28)	58 (21,29)	90 (22,30)	122 (23,31)	154 (20,28)	186 (21,29)	218 (22,30)	250 (23,31)	282 (20,28)	314 (21,29)	346 (22,30)	378 (23,31)	410 (20,28)	442 (21,29)	474 (22,30)	506 (23,31)
28 (20,28)	60 (21,29)	92 (22,30)	124 (23,31)	156 (20,28)	188 (21,29)	220 (22,30)	252 (23,31)	284 (20,28)	316 (21,29)	348 (22,30)	380 (23,31)	412 (20,28)	444 (21,29)	476 (22,30)	508 (23,31)
30 (20,28)	62 (21,29)	94 (22,30)	126 (23,31)	158 (20,28)	190 (21,29)	222 (22,30)	254 (23,31)	286 (20,28)	318 (21,29)	350 (22,30)	382 (23,31)	414 (20,28)	446 (21,29)	478 (22,30)	510 (23,31)

Figura 2.9: Relazione tra le posizioni nella matrice A (column major) e gli indici dei thread nel caricamento dei dati. Ogni numero prima della parentesi è relativo all'indirizzo nella matrice A (espresso in byte), ogni numero dentro la parentesi è l'indice del thread che carica il dato dalla corrispondente posizione.

0 (0,8)	32 (1,9)	64 (2,10)	96 (3,11)	128 (0,8)	160 (1,9)	192 (2,10)	224 (3,11)	256 (0,8)	288 (1,9)	320 (2,10)	352 (3,11)	384 (0,8)	416 (1,9)	448 (2,10)	480 (3,11)
2 (0,8)	34 (1,9)	66 (2,10)	98 (3,11)	130 (0,8)	162 (1,9)	194 (2,10)	226 (3,11)	258 (0,8)	290 (1,9)	322 (2,10)	354 (3,11)	386 (0,8)	418 (1,9)	450 (2,10)	482 (3,11)
4 (0,8)	36 (1,9)	68 (2,10)	100 (3,11)	132 (0,8)	164 (1,9)	196 (2,10)	228 (3,11)	260 (0,8)	292 (1,9)	324 (2,10)	356 (3,11)	388 (0,8)	420 (1,9)	452 (2,10)	484 (3,11)
6 (0,8)	38 (1,9)	70 (2,10)	102 (3,11)	134 (0,8)	166 (1,9)	198 (2,10)	230 (3,11)	262 (0,8)	294 (1,9)	326 (2,10)	358 (3,11)	390 (0,8)	422 (1,9)	454 (2,10)	486 (3,11)
8 (16,24)	40 (17,25)	72 (18,26)	104 (19,27)	136 (16,24)	168 (17,25)	200 (18,26)	232 (19,27)	264 (16,24)	296 (17,25)	328 (18,26)	360 (19,27)	392 (16,24)	424 (17,25)	456 (18,26)	488 (19,27)
10 (16,24)	42 (17,25)	74 (18,26)	106 (19,27)	138 (16,24)	170 (17,25)	202 (18,26)	234 (19,27)	266 (16,24)	298 (17,25)	330 (18,26)	362 (19,27)	394 (16,24)	426 (17,25)	458 (18,26)	490 (19,27)
12 (16,24)	44 (17,25)	76 (18,26)	108 (19,27)	140 (16,24)	172 (17,25)	204 (18,26)	236 (19,27)	268 (16,24)	300 (17,25)	332 (18,26)	364 (19,27)	396 (16,24)	428 (17,25)	460 (18,26)	492 (19,27)
14 (16,24)	46 (17,25)	78 (18,26)	110 (19,27)	142 (16,24)	174 (17,25)	206 (18,26)	238 (19,27)	270 (16,24)	302 (17,25)	334 (18,26)	366 (19,27)	398 (16,24)	430 (17,25)	462 (18,26)	494 (19,27)
16 (4,12)	48 (5,13)	80 (6,14)	112 (7,15)	144 (4,12)	176 (5,13)	208 (6,14)	240 (7,15)	272 (4,12)	304 (5,13)	336 (6,14)	368 (7,15)	400 (4,12)	432 (5,13)	464 (6,14)	496 (7,15)
18 (4,12)	50 (5,13)	82 (6,14)	114 (7,15)	146 (4,12)	178 (5,13)	210 (6,14)	242 (7,15)	274 (4,12)	306 (5,13)	338 (6,14)	370 (7,15)	402 (4,12)	434 (5,13)	466 (6,14)	498 (7,15)
20 (4,12)	52 (5,13)	84 (6,14)	116 (7,15)	148 (4,12)	180 (5,13)	212 (6,14)	244 (7,15)	276 (4,12)	308 (5,13)	340 (6,14)	372 (7,15)	404 (4,12)	436 (5,13)	468 (6,14)	500 (7,15)
22 (4,12)	54 (5,13)	86 (6,14)	118 (7,15)	150 (4,12)	182 (5,13)	214 (6,14)	246 (7,15)	278 (4,12)	310 (5,13)	342 (6,14)	374 (7,15)	406 (4,12)	438 (5,13)	470 (6,14)	502 (7,15)
24 (20,28)	56 (21,29)	88 (22,30)	120 (23,31)	152 (20,28)	184 (21,29)	216 (22,30)	248 (23,31)	280 (20,28)	312 (21,29)	344 (22,30)	376 (23,31)	408 (20,28)	440 (21,29)	472 (22,30)	504 (23,31)
26 (20,28)	58 (21,29)	90 (22,30)	122 (23,31)	154 (20,28)	186 (21,29)	218 (22,30)	250 (23,31)	282 (20,28)	314 (21,29)	346 (22,30)	378 (23,31)	410 (20,28)	442 (21,29)	474 (22,30)	506 (23,31)
28 (20,28)	60 (21,29)	92 (22,30)	124 (23,31)	156 (20,28)	188 (21,29)	220 (22,30)	252 (23,31)	284 (20,28)	316 (21,29)	348 (22,30)	380 (23,31)	412 (20,28)	444 (21,29)	476 (22,30)	508 (23,31)
30 (20,28)	62 (21,29)	94 (22,30)	126 (23,31)	158 (20,28)	190 (21,29)	222 (22,30)	254 (23,31)	286 (20,28)	318 (21,29)	350 (22,30)	382 (23,31)	414 (20,28)	446 (21,29)	478 (22,30)	510 (23,31)

Figura 2.10: Relazione tra le posizioni nella matrice B (column major) e gli indici dei thread nel caricamento dei dati. Ogni numero prima della parentesi è relativo all'indirizzo nella matrice B (espresso in byte), ogni numero dentro la parentesi è l'indice del thread che carica il dato dalla corrispondente posizione.

```

1 ; set 0:
2 HMMA.884.F32.F32.STEP0 R8, R26.reuse.T, R16.reuse.T, R8;
3 HMMA.884.F32.F32.STEP1 R10, R26.reuse.T, R16.reuse.T, R10;
4 HMMA.884.F32.F32.STEP2 R4, R26.reuse.T, R16.reuse.T, R4;
5 HMMA.884.F32.F32.STEP3 R6, R26.T, R16.T, R6;
6
7 ; set 1:
8 HMMA.884.F32.F32.STEP0 R8, R20.reuse.T, R18.reuse.T, R8;
9 HMMA.884.F32.F32.STEP1 R10, R20.reuse.T, R18.reuse.T, R10;
10 HMMA.884.F32.F32.STEP2 R4, R20.reuse.T, R18.reuse.T, R4;
11 HMMA.884.F32.F32.STEP3 R6, R20.T, R18.T, R6;
12
13 ; set 2:
14 HMMA.884.F32.F32.STEP0 R8, R22.reuse.T, R12.reuse.T, R8;
15 HMMA.884.F32.F32.STEP1 R10, R22.reuse.T, R12.reuse.T, R10;
16 HMMA.884.F32.F32.STEP2 R4, R22.reuse.T, R12.reuse.T, R4;
17 HMMA.884.F32.F32.STEP3 R6, R22.T, R12.T, R6;
18
19 ; set 3:
20 HMMA.884.F32.F32.STEP0 R8, R2.reuse.T, R14.reuse.T, R8;
21 HMMA.884.F32.F32.STEP1 R10, R2.reuse.T, R14.reuse.T, R10;
22 HMMA.884.F32.F32.STEP2 R4, R2.reuse.T, R14.reuse.T, R4;
23 HMMA.884.F32.F32.STEP3 R6, R2.T, R14.T, R6;

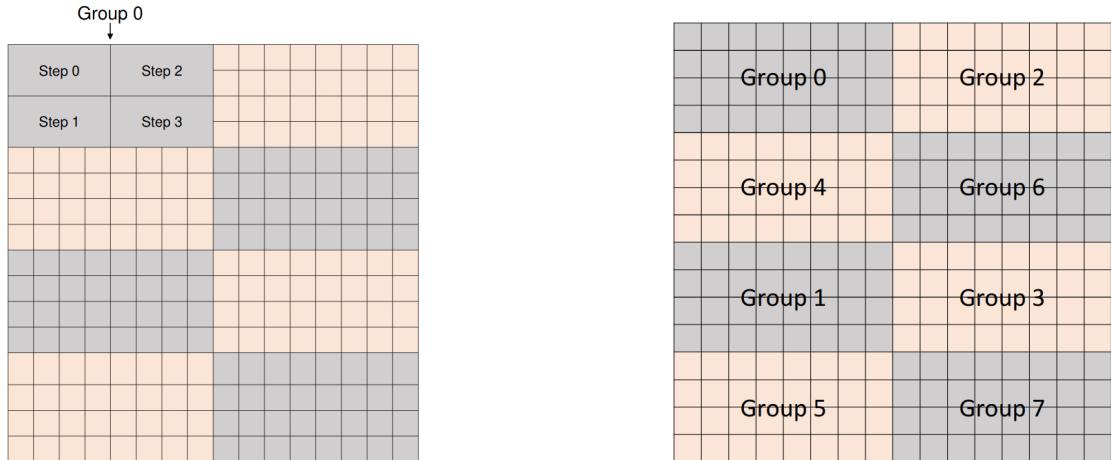
```

Listato 2.1: Codice assembly generato per l'istruzione `wmma::mma_sync` dal compilatore NVCC

Dopo il caricamento dei dati dalla memoria ai registri, il programmatore deve usare la primitiva `wmma::mma_sync` per operare la moltiplicazione con i Tensor Core. Il compilatore NVCC traduce la singola primitiva in 4 set di istruzioni HMMA, come mostrato nel listato 2.1.

Ogni set è costituito da quattro istruzioni HMMA con flag che va dallo "STEP0" allo "STEP3". Il registro di destinazione è lo stesso per tutte e quattro le istruzioni dei set; i flag da "STEP0" fino a "STEP3" corrispondono a diverse posizione della matrice C (vedi figura 2.11a).

A tempo di esecuzione, l'architettura Volta divide i 32 thread di un warp in 8 gruppi ($group_id = thread_id/4$) e condivide i valori del registro tra tutti i thread dello stesso gruppo e tra i gruppi. Ogni gruppo di thread computa $4 \times 8 = 32$ elementi nella matrice C . La figura 2.11b mostra la relazione tra la posizione della matrice C e gli indici dei gruppi. Per un gruppo di thread tutte le istruzioni HMMA contribuiscono al calcolo della stessa posizione nella matrice C e ognuno di essi accumula e moltiplica gli elementi da parti differenti delle matrici A e B (vedi figura 2.12).



(a) Quattro step di una istruzione HMMA per la computazione di elementi differenti nella matrice C .

(b) Mapping tra la posizione nella matrice C e gli indici dei gruppi di thread.

Figura 2.11: Schema operazione istruzione HMMA e posizioni degli indici nella matrice C .

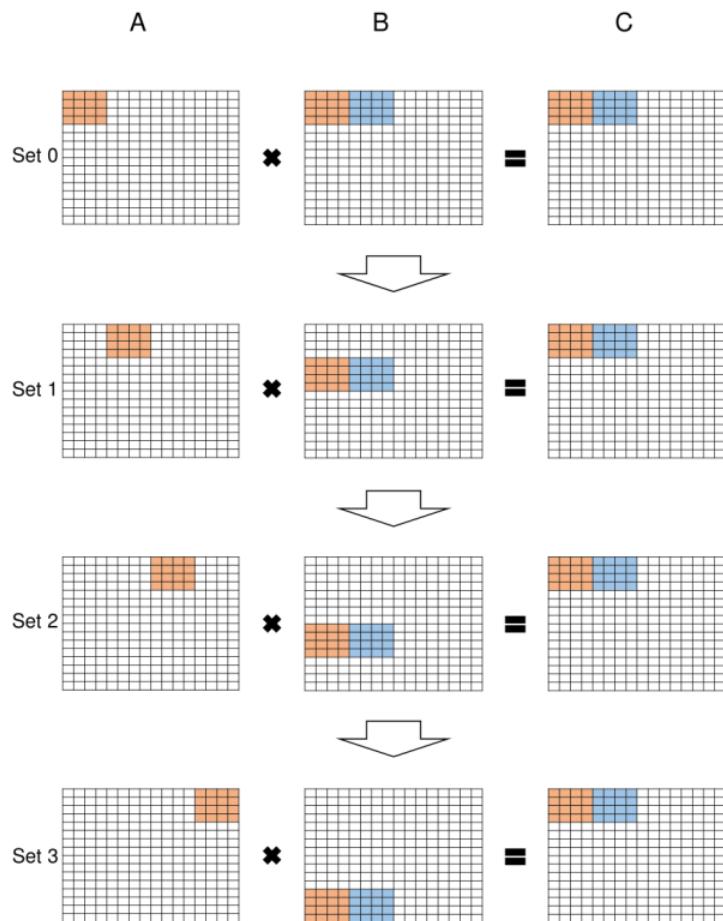


Figura 2.12: Quattro set di istruzioni HMMA completano il risultato parziale 4×8 nella matrice C del gruppo 0. Set differenti usano elementi differenti in A e B . L'istruzione nel set 0 viene eseguita per prima, poi il set 1, il set 2 e infine il set 3. In questo modo l'istruzione HMMA può eseguire correttamente i 4×8 elementi della matrice C .

2.1.4 Studio prestazioni Tensor Core

In questa sezione analizzeremo le prestazioni di varie librerie che fanno uso dei Tensor Core; le librerie prese in considerazione sono: **WMMA API** e **cuBLAS**.

Partendo ad analizzare la prima libreria, possiamo osservare l'uso dei fragment cioè istruzioni che consentono di partizionare la matrice in blocchi 16×16 . In questo caso la funzione presente nel listato 2.2 moltiplica una matrice 16×16 , quindi non è stato necessario andare a scomporre la matrice in quanto era già delle dimensioni supportate dal fragment. Con le operazioni `wmma::load_matrix_sync` si carica dalla memoria nei registri i dati delle matrici. Con `wmma::mma_sync` effettuiamo la moltiplicazione $A \times B + C$ e infine con `wmma::store_matrix_sync` copiamo il risultato della moltiplicazione dal registro alla memoria centrale.

```

1  __global__ void dot_wmma16x16(half *a, half *b, half *c)
2  {
3      wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
4      wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
5      wmma::fragment<wmma::accumulator, 16, 16, 16, half> c_frag;
6      wmma::load_matrix_sync(a_frag, a, 16);
7      wmma::load_matrix_sync(b_frag, b, 16);
8      wmma::fill_fragment(c_frag, 0.0f);
9      wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
10     wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
11 }
```

Listato 2.2: Uso delle WMMA API per la moltiplicazione di due matrici 16×16 .

Se si volesse moltiplicare una matrice di dimensioni maggiori occorre tener presente che la dimensione della matrice (numero colonne e numero righe) deve essere un multiplo di 16, in caso contrario la moltiplicazione produce risultati errati. Nel listato 2.3 è indicata una possibile implementazione di un kernel che fa uso delle **WMMA API** per moltiplicare due matrici di qualunque dimensione purché valgano le condizioni sopra citate.

```

1  __global__ void simple_wmma_gemm(half *a, half *b, float *c, float *d,
2                                   int m_ld, int n_ld, int k_ld,
3                                   float alpha, float beta) {
4     // Leading dimensions. Packed with no transpositions.
5     int lda = m_ld;
6     int ldb = k_ld;
7     int ldc = n_ld;
8
9     // Tile using a 2D grid
10    int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
11    int warpN = (blockIdx.y * blockDim.y + threadIdx.y);
12
13    // Declare the fragments
14    wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half,
15                  wmma::row_major> a_frag;
16    wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half,
17                  wmma::col_major> b_frag;
18    wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float>
19                  acc_frag;
```

```

20     wmma::fragment<wmma::accumulator, WMMA_M WMMA_N WMMA_K float>
21             c_frag;
22     wmma::fill_fragment(acc_frag, 0.0f);
23
24     // Loop over k
25     for (int i = 0; i < k_ld; i += WMMA_K) {
26         int aCol = i;
27         int aRow = warpM * WMMA_M;
28         int bCol = i;
29         int bRow = warpN * WMMA_N;
30
31         // Bounds checking
32         if (aRow < m_ld && aCol < k_ld && bRow < k_ld && bCol < n_ld) {
33             // Load the inputs
34             wmma::load_matrix_sync(a_frag, a + aCol + aRow * lda, lda);
35             wmma::load_matrix_sync(b_frag, b + bCol + bRow * ldb, ldb);
36
37             // Perform the matrix multiplication
38             wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
39         }
40     }
41
42     // Load in the current value of c, scale it by beta,
43     // and add this our result
44     // scaled by alpha
45     int cCol = warpN * WMMA_N;
46     int cRow = warpM * WMMA_M;
47
48     if (cRow < m_ld && cCol < n_ld) {
49         wmma::load_matrix_sync(c_frag, c + cCol + cRow * ldc, ldc,
50                               wmma::mem_row_major);
51
52         for (int i = 0; i < c_frag.num_elements; i++) {
53             c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
54         }
55
56         // Store the output
57         wmma::store_matrix_sync(d + cCol + cRow * ldc, c_frag, ldc,
58                               wmma::mem_row_major);
59     }
60 }
```

Listato 2.3: Implementazione di un kernel che fa uso delle WMMA API per moltiplicare matrici di qualsiasi dimensione.

Il test condotto è il seguente: si sono scelte diverse implementazioni del calcolo della moltiplicazione tra matrici e si sono misurate i tempi di esecuzione prodotti da ogni singola implementazione andando a variare la dimensione delle matrici; i test sono stati eseguiti con valori piuttosto elevati in termini di dimensioni delle matrici in quanto l'architettura che fa uso di Tensor Core raggiunge le prestazioni migliori quando gran parte, o meglio tutti i Tensor Core sono attivati. Quello che possiamo osservare dalla figura 2.13 è che, prendendo come riferimento la versione CUDA, la libreria **cuBLAS** è quella che ha ottenuto le prestazioni migliori in qualsiasi circostanza; questo non ci sorprende dal

momento che è una libreria studiata e ottimizzata per le GPU NVIDIA, per far uso dei Tensor Core e per eseguire calcoli di algebra lineare tra cui la moltiplicazione tra matrici. Potrebbe sorprendere invece la libreria che fa uso delle **WMMA API** in quanto il degrado delle loro prestazioni rispetto a **cuBLAS** è quasi sempre di 8 volte inferiore e sapendo che anche questa implementazione fa uso dei Tensor Core potrebbe stupire. Se pensiamo però che l'esempio proposto non fa uso di Shared Memory, né di tecniche particolari per l'ottimizzazione dei calcoli o meglio ancora tecniche studiate appositamente per eseguire i calcoli più velocemente sulle GPU, questo risultato è quantomeno plausibile. Se considerassimo soltanto l'uso della Shared Memory (che possiamo osservare nell'implementazione **CUDA (SHM)** senza Tensor Core), allora possiamo vedere che utilizzandola si ottiene uno speedup di quasi due volte.

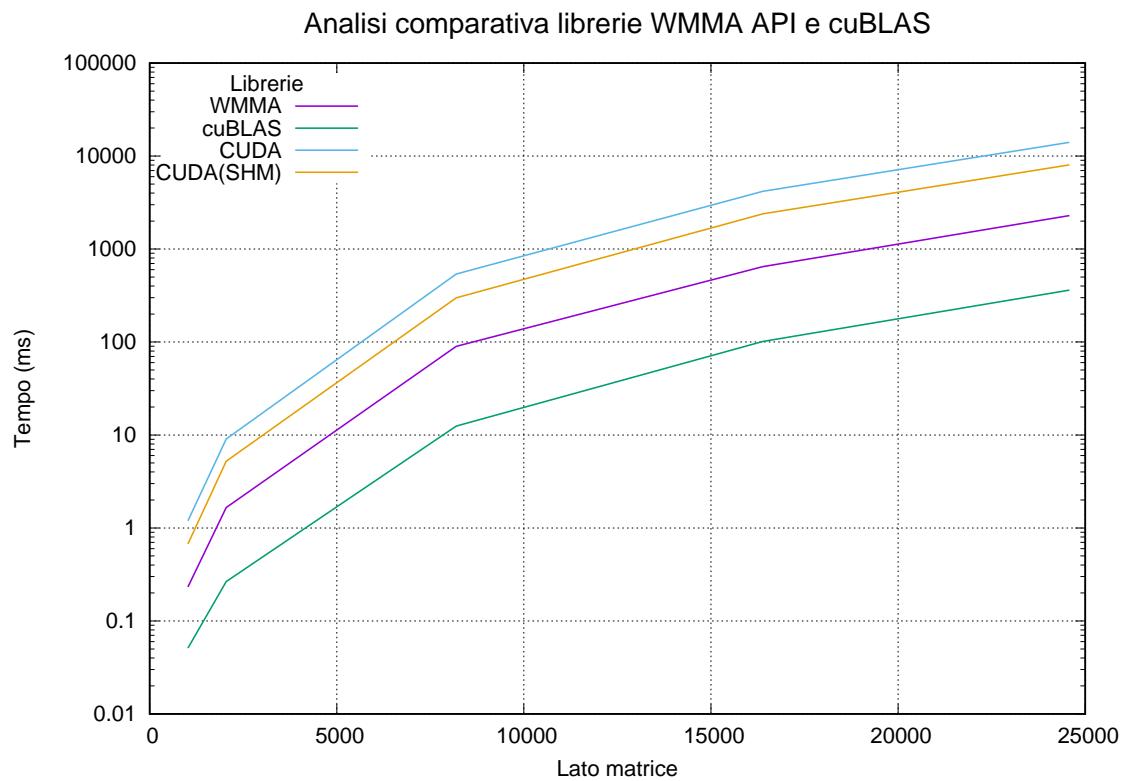


Figura 2.13: Confronto tra librerie per la moltiplicazione tra matrici.

In definitiva possiamo affermare che qualora sia necessario moltiplicare matrici di grandi dimensioni, la soluzione migliore risulta essere la libreria **cuBLAS**. Dal momento che questa libreria viene sviluppata e distribuita da NVIDIA, non è possibile conoscerne i dettagli implementativi: infatti NVIDIA non rilascia i relativi codici sorgenti, quindi risulta molto difficile capire in che modo questa libreria operi e che ottimizzazioni siano state eseguite.

Prendendo quindi come base di partenza per la misurazione delle prestazioni la versione CUDA base, possiamo osservare che **cuBLAS** ottiene le prestazioni migliori avendo uno speedup che varia dalle 30 alle 40 volte rispetto alla versione CUDA.

Capitolo 3

Operatore di Dirac e Tensor Core

In questo capitolo viene analizzato il funzionamento dell'algoritmo per il calcolo dell'operatore di Dirac. Questo operatore viene utilizzato in simulazioni Lattice QCD e rappresenta l'operazione principale poiché il tempo impiegato per il suo calcolo va dal 40% all'80% del tempo totale impiegato dalla simulazione [13]. Ottimizzarne l'efficienza permette di velocizzare le prestazioni dell'intero algoritmo di simulazione.

3.1 Funzionamento operatore di Dirac

L'operatore di Dirac effettua una serie di moltiplicazioni tra matrici 3×3 e vettori 3×1 , entrambi composti da numeri complessi. Le matrici e i vettori (anche detti *fermioni*) rappresentano uno schema ben preciso identificato da un ipercubo a quattro dimensioni, o *lattice*, le cui dimensioni sono: $nx \times ny \times nz \times nt$.

Questo ipercubo presenta due componenti fondamentali: i siti e i collegamenti tra essi. Ogni sito è collegato a tutti i suoi 8 vicini (due per dimensione). Un sito è rappresentato da un fermione (ovvero un vettore 3×1), mentre ogni collegamento è rappresentato dalla matrice 3×3 . In figura 3.1 viene mostrato un esempio di lattice in tre dimensioni poiché un lattice quadridimensionale è difficile da rendere graficamente. I siti che risiedono sul bordo sono vicini ai siti presenti sul bordo opposto, definendo quindi un dominio ciclico (vedi figura 3.2). Le due operazioni principali che l'operatore di Dirac effettua sono Deo (Dirac even odd) e Doe (Dirac odd even). Tali operazioni definiscono come devono essere letti i valori dei siti: Deo legge i valori dei siti la cui somma delle coordinate è un numero pari e scrive il risultato sui siti la cui somma delle coordinate è un numero dispari; Doe effettua l'operazione opposta.

Le due operazioni appena citate agiscono sulla metà dei dati del dominio. Nel listato 3.1 possiamo osservare una possibile implementazione di come possano essere iterati i siti.

```
1  for (unsigned t=0; t<nt; t++)
2      for (unsigned z=0; z<nz; z++)
3          for (unsigned y=0; y<ny; y++)
4              for (unsigned hx=0; hx<nxh; hx++) {
5                  //Compute output vector of site idxh
6              }
```

Listato 3.1: Codice per l'iterazione dei siti del lattice.

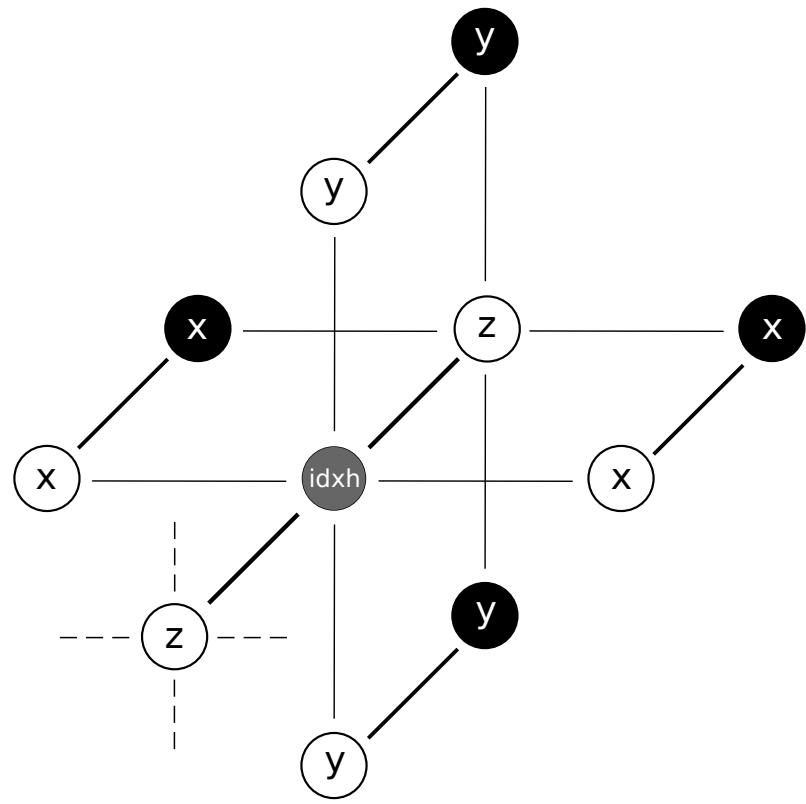


Figura 3.1: Esempio di un lattice in tre dimensioni. idxh identifica il fermione che si sta valutando.

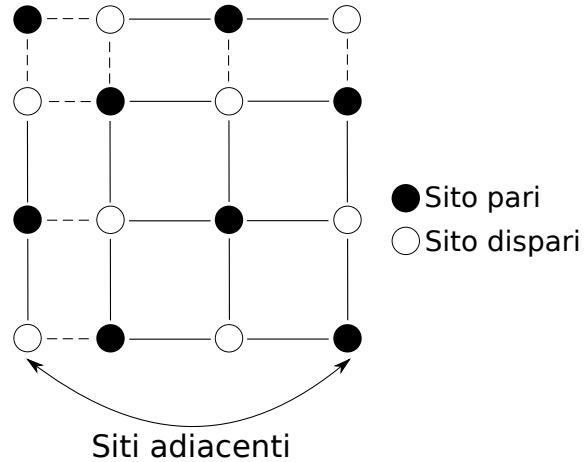


Figura 3.2: Esempio di un lattice in due dimensioni in cui si evidenzia l'adiacenza dei siti sul bordo.

La coordinata x , come appena detto, scorre solamente metà dei dati in quanto solo i siti pari e i siti dispari vengono valutati rispettivamente dalle funzioni `Deo` e `Doe`. Per questo motivo il ciclo for più interno va da zero fino a nxh , che rappresenta la metà del dominio di x .

Il ciclo più interno presente in `Deo` e `Doe` serve a selezionare i vicini di un sito. Per ogni vicino v_i si moltiplicare la matrice rappresentata dal collegamento di v_i al corrente con il fermione rappresentato da v_i (vedi listati 3.2 e 3.3).

```

1  x = 2*hx
2
3  if (((y+z+t) % 2) != 0) x++
4
5  aux = A0[x,y,z,t] x V[x+1,y,z,t]
6  aux += A2[x,y,z,t] x V[x,y+1,z,t]
7  aux += A4[x,y,z,t] x V[x,y,z+1,t]
8  aux += A6[x,y,z,t] x V[x,y,z,t+1]
9
10 aux -= CA1[x-1,y,z,t] x V[x-1,y,z,t]
11 aux -= CA3[x,y-1,z,t] x V[x,y-1,z,t]
12 aux -= CA5[x,y,z-1,t] x V[x,y,z-1,t]
13 aux -= CA7[x,y,z,t-1] x V[x,y,z,t-1]
14
15 endFremion[x,y,z,t] = aux/2

```

Listato 3.2: Pseudo codice ciclo interno di `Deo`.

```

1  x = 2*hx
2
3  if (((y+z+t) % 2) == 0) x++
4
5  aux = A1[x,y,z,t] x V[x+1,y,z,t]
6  aux += A3[x,y,z,t] x V[x,y+1,z,t]
7  aux += A5[x,y,z,t] x V[x,y,z+1,t]
8  aux += A7[x,y,z,t] x V[x,y,z,t+1]
9
10 aux -= CA0[x-1,y,z,t] x V[x-1,y,z,t]
11 aux -= CA2[x,y-1,z,t] x V[x,y-1,z,t]
12 aux -= CA4[x,y,z-1,t] x V[x,y,z-1,t]
13 aux -= CA6[x,y,z,t-1] x V[x,y,z,t-1]
14
15 endFremion[x,y,z,t] = aux/2

```

Listato 3.3: Pseudo codice ciclo interno di `Doe`.

Sia in `Deo` che in `Doe` vengono svolte le stesse operazioni, la principale differenza sta negli indici delle matrici e dei vettori da moltiplicare. Le variabili `Ax` rappresentano le matrici 3×3 , mentre le variabili `V` rappresentano i vettori 3×1 . La variabile `aux` è anch'essa un vettore 3×1 in cui si memorizza il risultato. La variabile `CAx` indica il complesso coniugato della matrice 3×3 .

L'operazione che viene eseguita maggiormente è la moltiplicazione tra matrice 3×3 e vettore 3×1 , quindi se si ottimizzasse questa operazione si ridurrebbero i tempi per il calcolo del fermione risultante.

3.1.1 Complessità

La complessità dell'algoritmo è data dai quattro cicli innestati nelle funzioni `Deo` e `Doe`. Le operazioni all'interno dei cicli hanno tutte complessità costante, quindi possiamo definire la complessità dell'algoritmo come:

$$\Theta(\text{NITER} \times nx \times ny \times nz \times nt)$$

Dove `NITER` indica in numero di chiamate a `Deo` e `Doe`, nx, ny, nz e nt rappresentano le dimensioni del lattice. In una simulazione completa `NITER` è dell'ordine delle migliaia.

3.2 Strutture dati

Analizziamo ora la struttura dati in cui sono memorizzate le informazioni. Sia le matrici che i vettori sono memorizzati in Strutture di Array (SoA). Per quanto riguarda le matrici, queste sono suddivise in un array di 8 elementi, ognuno dei quali memorizza tutte le matrici di tutti i sizeh siti. Ad esempio definiamo u il vettore di 8 elementi, allora $u[0]$ è una struttura che contiene la matrice A_0 dei siti $0, 1, 2, \dots, sizeh - 1$ (vedi figura 3.3).

A0	A1	A2	A3	A4	A5	A6	A7
----	----	----	----	----	----	----	----

Figura 3.3: Array di 8 elementi contenenti le matrici dei siti.

All'interno di ogni elemento di u sono allocati tre vettori: r_0 , r_1 ed r_2 , questi rappresentano le 3 righe di tutte le sizeh matrici A_i , dove i rappresenta l'indice dell'elemento di u . Nel listato 3.4 si riporta la dichiarazione della struttura dati su3_soa.

```

1  typedef struct vec3_soa_t {
2      d_complex c0[sizeh];
3      d_complex c1[sizeh];
4      d_complex c2[sizeh];
5  } vec3_soa;
6
7  typedef struct su3_soa_t {
8      vec3_soa r0;
9      vec3_soa r1;
10     vec3_soa r2;
11 } su3_soa;

```

Listato 3.4: Dichiarazione struttura dati su3_soa

Ad esempio $u[0] \rightarrow r_0$ è una struttura dati che ospita la prima riga (composta da tre numeri complessi) delle matrici dei sizeh siti. In figura 3.4 viene mostrata la rappresentazione di una matrice; nella figura r_0 , r_1 e r_2 sono indicati uno sotto all'altro, ma è bene precisare che in memoria le righe sono adiacenti.

r0	c0[12]	c1[12]	c2[12]
r1	c0[12]	c1[12]	c2[12]
r2	c0[12]	c1[12]	c2[12]

Figura 3.4: Struttura dati contenuta in ogni elemento di u . In questo esempio abbiamo riportato la matrice di indice 12. Le righe sono adiacenti in memoria.

Ad esempio, accedendo ad $u[2] \rightarrow r_0.c2[34]$ viene letto il valore della terza colonna ($c2$), prima riga (r_0) della matrice A_2 del sito 34.

	0	1	...	sizeh-1
c0	cuDoubleComplex
c1
c2

Figura 3.5: Rappresentazione grafica della struttura dati vec3_soa.

Per quanto riguarda i vettori invece, la struttura dati che viene usata è simile a quella delle matrici: si utilizza sempre una struttura di tipo SoA (Struttura di Array) in cui sono presenti 3 componenti rappresentate da un vettore lungo sizeh. La struttura vec3_soa, analizzata in precedenza nel listato 3.4 è la stessa struttura che viene utilizzata per rappresentare i vettori. L'utilizzo dei vettori nella struttura delle matrici è dovuto al fatto che una matrice 3×3 può essere rappresentata come una serie di 3 vettori colonna 3×1 adiacenti, formando quindi una matrice 3×3 . In figura 3.5 viene mostrata la struttura dati vec3_soa.

3.2.1 Tensor Core

Vediamo come poter utilizzare i Tensor Core per migliorare le prestazioni dell'operatore di Dirac. Come visto nella sezione precedente l'algoritmo si occupa principalmente di effettuare delle moltiplicazioni matrice-vettore; questo tipo di operazione differisce leggermente dall'operazione che i Tensor Core sono in grado di effettuare, ovvero moltiplicazioni tra matrici. Possiamo però vedere una moltiplicazione matrice-vettore come una moltiplicazione tra matrici supponendo di avere il vettore disposto lungo la prima colonna della seconda matrice e le celle restanti della seconda matrice poste a 0. In questo modo gli zeri che compaiono nella seconda matrice non alterano il risultato della moltiplicazione e, sapendo che una moltiplicazione matrice-vettore produce un vettore, esso sarà presente nella prima colonna della matrice risultato. In figura 3.6 è espresso graficamente il concetto appena illustrato.

$$\begin{array}{|c|c|c|} \hline C_{0,0} & C_{0,1} & C_{0,2} \\ \hline C_{1,0} & C_{1,1} & C_{1,2} \\ \hline C_{2,0} & C_{2,1} & C_{2,2} \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline C_0 & 0 & 0 \\ \hline C_1 & 0 & 0 \\ \hline C_2 & 0 & 0 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline R_0 & 0 & 0 \\ \hline R_1 & 0 & 0 \\ \hline R_2 & 0 & 0 \\ \hline \end{array}$$

Figura 3.6: Esempio di moltiplicazione matrice-vettore utilizzando una matrice come secondo operando.

Ricordiamo che i Tensor Core possono moltiplicare matrici con dimensione minima

pari a 16×16 , quindi occorre inserire le matrici 3×3 e i vettori 3×1 nella matrice più grande 16×16 ; inoltre i Tensor Core effettuano calcoli su dati di tipo half, ovvero numeri in virgola mobile a 16-bit invece i calcoli di nostro interesse sono con numeri complessi.

Come accennato in precedenza, l'operatore di Dirac prevede l'uso di numeri complessi, quindi le matrici e i vettori con cui abbiamo a che fare sono di natura complessa; questo rende più complicati i calcoli: la moltiplicazione di due matrici di numeri complessi avviene scomponendo la matrice (o il vettore) in due matrici della stessa dimensione, in cui una contiene solo la parte reale e l'altra la parte immaginaria.

$$\mathbf{M}_c = \mathbf{M}_r + i\mathbf{M}_i$$

Dove \mathbf{M}_r rappresenta la matrice della parte reale e \mathbf{M}_i rappresenta la matrice della parte immaginaria. Quindi per moltiplicare due matrici di numeri complessi occorre tener presente che il calcolo del prodotto tra due numeri complessi si effettua come segue:

$$\begin{aligned} z_1 &= a + ib \\ z_2 &= c + id \\ z_1 \cdot z_2 &= (ac - bd) + i(ad + bc) \end{aligned} \tag{3.1}$$

Portiamo questo esempio con matrici. Consideriamo m_1 e m_2 due matrici di numeri complessi, quindi operiamo i calcoli definendo t_1 , t_2 , t_3 e t_4 i risultati parziali dei soli prodotti:

$$\begin{aligned} \mathbf{T}_1 &= \mathbf{M}_{1r} \cdot \mathbf{M}_{2r} \\ \mathbf{T}_2 &= \mathbf{M}_{1i} \cdot \mathbf{M}_{2i} \\ \mathbf{T}_3 &= \mathbf{M}_{1r} \cdot \mathbf{M}_{2i} \\ \mathbf{T}_4 &= \mathbf{M}_{1i} \cdot \mathbf{M}_{2r} \end{aligned} \tag{3.2}$$

Ottenuti i prodotti ora andiamo a svolgere le somme e le differenze:

$$\mathbf{M}_1 \cdot \mathbf{M}_2 = (\mathbf{T}_1 - \mathbf{T}_2) + i(\mathbf{T}_3 + \mathbf{T}_4) \tag{3.3}$$

In conclusione, otteniamo due matrici (una per la parte reale e una per la parte immaginaria) che rappresentano il risultato finale della moltiplicazione. In questo esempio si è fatto uso di due matrici invece che matrice e vettore, ma il concetto rimane il medesimo, ovvero si trasforma il vettore in una matrice (come spiegato in precedenza), si effettua la moltiplicazione e infine si prende solo il vettore colonna della matrice risultante.

Un possibile modo di operare questo calcolo con i Tensor Core è quello di inserire la matrice in alto a sinistra e riempire il resto della matrice con zeri (vedi figura 3.7). Questo ragionamento lo si applica ai numeri complessi scomponendo la parte reale e quella immaginaria in due matrici distinte. In questo modo si spreca molta memoria oltre a far eseguire calcoli non utili ai fini del risultato.

$$\begin{array}{|c|c|c|c|} \hline M & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}
 \times
 \begin{array}{|c|c|c|c|} \hline V & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}
 =
 \begin{array}{|c|c|c|c|} \hline R & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

Figura 3.7: Prima soluzione per utilizzare i Tensor Core per moltiplicazione matrice-vettore.

Un altro possibile modo di operare è quello di inserire le matrici 3×3 (e anche i relativi vettori 3×1) lungo la diagonale della matrice 16×16 (vedi figura 3.8). In questo modo gli sprechi in termini di memoria vengono leggermente ridotti e con una sola moltiplicazione di una matrice (16×16) otteniamo 4 vettori come risultato sempre disposti lungo la diagonale della matrice. In figura 3.8 viene illustrata la soluzione appena descritta.

$$\begin{array}{|c|c|c|c|} \hline M_0 & & & 0 \\ \hline & M_1 & & \\ \hline & & M_2 & \\ \hline & & & M_3 \\ \hline 0 & & & M_4 \\ \hline \end{array}
 \times
 \begin{array}{|c|c|c|c|} \hline V_0 & & & 0 \\ \hline & V_1 & & \\ \hline & & V_2 & \\ \hline & & & V_3 \\ \hline & & & V_4 \\ \hline \end{array}
 =
 \begin{array}{|c|c|c|c|} \hline R_0 & & & 0 \\ \hline & R_1 & & \\ \hline & & R_2 & \\ \hline & & & R_3 \\ \hline & & & R_4 \\ \hline \end{array}$$

Figura 3.8: Seconda soluzione per utilizzare i Tensor Core per moltiplicazione matrice-vettore.

Seppur l'ultima soluzione proposta riduca lo spreco di memoria, il numero di matrici rappresentate all'interno di matrici 16×16 usate dai Tensor Core è ancora piuttosto basso. Nella sezione 3.2.2 analizzeremo nel dettaglio le prestazioni di ogni soluzione, evidenziando i punti favorevoli e sfavorevoli di ognuna.

Le soluzioni appena proposte, oltre ad usare molta memoria, necessitano di quattro moltiplicazioni tra matrici, una somma e una sottrazione e questo potrebbe influire negativamente sulle prestazioni dell'algoritmo. Occorre trovare un modo in cui con una sola moltiplicazione effettuata dai Tensor Core venga prodotto già un risultato completo, ovvero comprendente sia parte reale che parte immaginaria.

L'ultimo modo che analizziamo ha uno spreco minimo in termini di memoria, ma con una sola moltiplicazione si ottengono 8 vettori come risultato. Partiamo analizzando solo la matrice che contiene le matrici 3×3 : queste ultime possiamo inserirle in una matrice 4×4 allineandole in alto a sinistra (vedi figura 3.9)

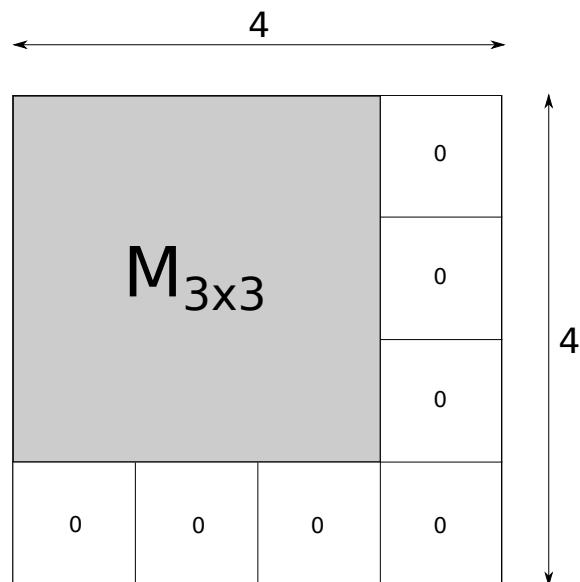


Figura 3.9: Matrice 3×3 inserita in una matrice 4×4 con padding di zeri per l'allineamento.

A questo punto avendo una matrice 4×4 possiamo inserire esattamente 16 matrici in quella 16×16 , avendo uno spreco di memoria irrisorio. Lavorando con i numeri complessi possiamo disporre le matrici come rappresentato in figura 3.10 in modo da poter effettuare 8 prodotti matrice-vettore e ottenere direttamente il risultato sotto forma di numero complesso.

16

16

A_R			0	A_i			0	B_R			0	B_i			0
			0				0				0				0
			0				0				0				0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_R			0	C_i			0	D_R			0	D_i			0
			0				0				0				0
			0				0				0				0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E_R			0	E_i			0	F_R			0	F_i			0
			0				0				0				0
			0				0				0				0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G_R			0	G_i			0	H_R			0	H_i			0
			0				0				0				0
			0				0				0				0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Parte Reale
Parte Immaginaria

Figura 3.10: Matrice 16×16 in cui sono presenti 16 matrici 3×3 disposte per il calcolo con Tensor Core.

Analizzando la matrice illustrata in figura 3.10 possiamo osservare che lungo le righe della matrice sono presenti quattro matrici 3×3 che rappresentano la parte reale e immaginaria. In questo caso non serve scomporre parte reale e parte immaginaria in due matrici separate, ma la separazione coesiste nella stessa matrice. Da qui possiamo ottenere il risultato direttamente in un'unica matrice senza dover "comporre" le varie matrici come illustrato nei precedenti modi.

Analizziamo ora la disposizione dei vettori 3×1 nella matrice 16×16 . La loro disposizione segue uno schema che consente di effettuare le quattro moltiplicazioni matrice-vettore, la somma e la sottrazione sfruttando le proprietà della moltiplicazione tra matrici.

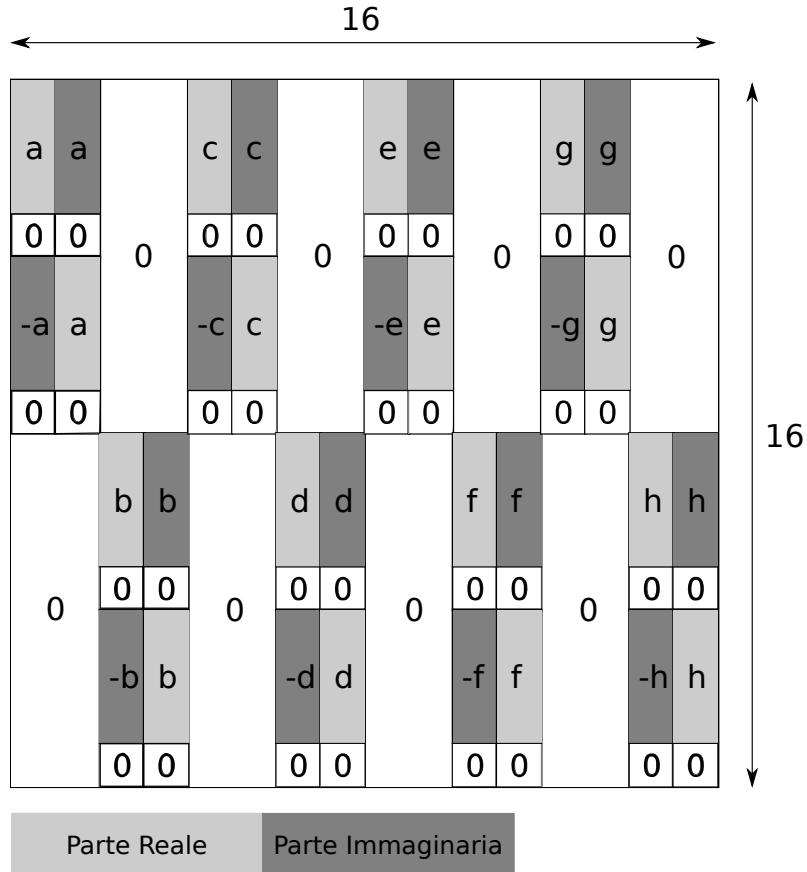


Figura 3.11: Disposizione dei vettori 3×1 nella matrice 16×16 .

Moltiplicando le due matrici appena illustrate otteniamo come risultato una matrice in cui lungo la diagonale sono presenti i vettori ottenuti come risultato dalla moltiplicazione già suddivisi in parte reale e parte immaginaria. In figura 3.12 viene illustrato il risultato della moltiplicazione matrice-vettore.

Con questa particolare disposizione delle matrici e dei vettori all'interno della matrice 16×16 è possibile effettuare otto moltiplicazioni matrice-vettore in una sola moltiplicazione tra matrici effettuata dai Tensor Core. Questo modo semplifica di molto la gestione dei dati in memoria in quanto non è necessario combinare risultati parziali come succede con i metodi illustrati precedentemente; infine la memoria non utilizzata risulta essere molto inferiore rispetto alle soluzioni illustrate in precedenza.

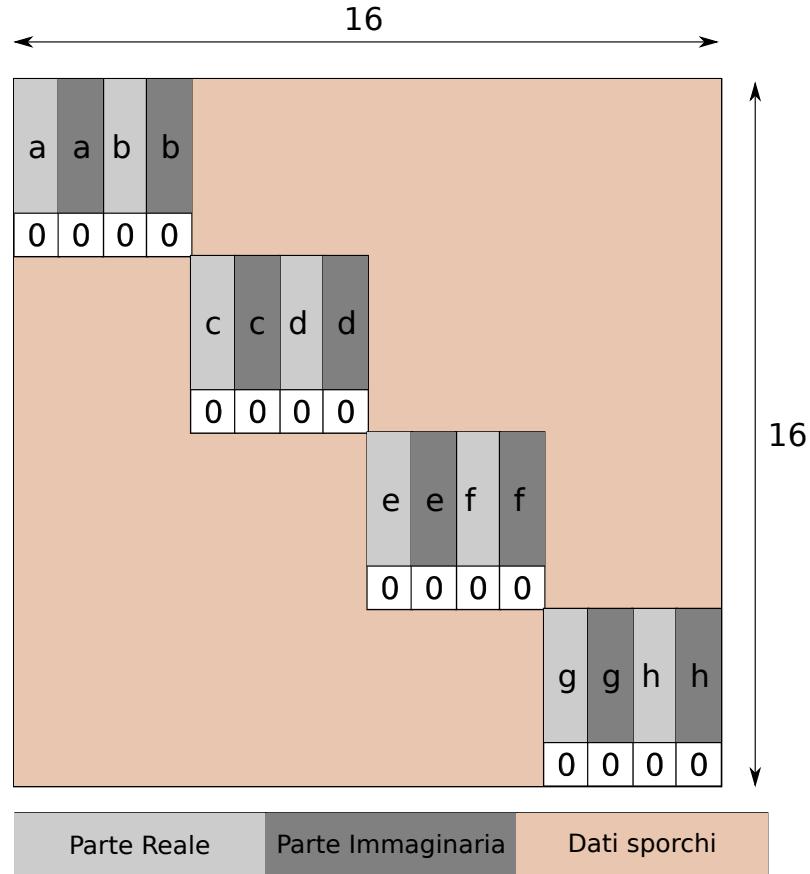


Figura 3.12: Matrice rappresentante il risultato contenente gli otto vettori complessi.

Nel listato 3.5 possiamo osservare la moltiplicazione degli otto vicini al sito corrente. Data una coordinata (x,y,z,t) le variabili xp , yp , zp e tp rappresentano la coordinata di indice $+1$, ovvero la coordinata che rappresenta il sito adiacente; le coordinate xm , ym , zm e tm indicano la coordinata di indice -1 . Queste otto coordinate servono per il calcolo della moltiplicazione matrice-vettore con gli otto siti adiacenti lungo le quattro dimensioni del lattice.

```

1 ...
2 mat_vec_mul( &u[1] , idxh , eta , in , snum(xp,y,z,t) , &aux_tmp );
3 aux = aux_tmp;
4
5 eta = ETA_UPDATE(x);
6 mat_vec_mul( &u[3] , idxh , eta , in , snum(x,yp,z,t) , &aux_tmp );
7 aux = sumResult(aux, aux_tmp);
8
9 eta = ETA_UPDATE(x+y);
10 mat_vec_mul( &u[5] , idxh , eta , in , snum(x,y,zp,t) , &aux_tmp );
11 aux = sumResult(aux, aux_tmp);
12
13 eta = ETA_UPDATE(x+y+z);
14 mat_vec_mul( &u[7] , idxh , eta , in , snum(x,y,z,tp) , &aux_tmp );

```

```

15 aux = sumResult(aux, aux_tmp);
16 ///////////////////////////////////////////////////////////////////
17 eta = 1;
18 conjmat_vec_mul( &u[0] , snum(xm,y,z,t) , eta , in , snum(xm,y,z,t) , &aux_tmp );
19 aux = subResult(aux, aux_tmp);
20
21 eta = ETA_UPDATE(x);
22 conjmat_vec_mul( &u[2] , snum(x,ym,z,t) , eta , in , snum(x,ym,z,t) , &aux_tmp );
23 aux = subResult(aux, aux_tmp);
24
25 eta = ETA_UPDATE(x+y);
26 conjmat_vec_mul( &u[4] , snum(x,y,zm,t) , eta , in , snum(x,y,zm,t) , &aux_tmp );
27 aux = subResult(aux, aux_tmp);
28
29 eta = ETA_UPDATE(x+y+z);
30 conjmat_vec_mul( &u[6] , snum(x,y,z,tm) , eta , in , snum(x,y,z,tm) , &aux_tmp );
31 aux = subResult(aux, aux_tmp)
32 ...

```

Listato 3.5: Moltiplicazione siti adiacenti.

Possiamo sfruttare i Tensor Core per effettuare otto moltiplicazioni matrice-vettore per ogni sito adiacente: per ognuna delle otto moltiplicazioni, invece di eseguire solo una moltiplicazione matrice-vettore ne eseguiamo otto facendo uso dei Tensor Core; in questo modo garantiamo il fatto che effettuiamo moltiplicazioni matrice-vettore di siti adiacenti sia sul lattice che in memoria, aspetto fondamentale che necessitano i Tensor Core per un corretto funzionamento.

Il listato 3.6 mostra lo pseudo-codice di come possono essere usati i Tensor Core per effettuare la moltiplicazione di matrici e vettori adiacenti.

```

1 ...
2
3 t = (blockIdx.z * blockDim.z + threadIdx.z) / nz;
4 z = (blockIdx.z * blockDim.z + threadIdx.z) % nz;
5 y = (blockIdx.y * blockDim.y + threadIdx.y);
6 x = 2*(blockIdx.x * blockDim.x + threadIdx.x) + ((y+z+t+1) & 0x1)
7
8 for (unsigned i=0; i<NUM*2; i+=2)
9     idxh[i/2] = snum(x+i,y,z,t);
10
11 // Calcolo xp, yp, zp, tp
12 // Calcolo xm, ym, zm, tm
13
14 tensor_mat_vec_mul( &u[1] , idxh , eta , in , snum(xp,y,z,t) , &aux_tmp );
15 aux = aux_tmp;
16
17 eta = ETA_UPDATE(x);
18 tensor_mat_vec_mul( &u[3] , idxh , eta , in , snum(x,yp,z,t) , &aux_tmp );
19 aux = sumResult(aux, aux_tmp);
20
21 eta = ETA_UPDATE(x+y);
22 tensor_mat_vec_mul( &u[5] , idxh , eta , in , snum(x,y,zp,t) , &aux_tmp );
23 aux = sumResult(aux, aux_tmp);

```

```

24
25 // Altre moltiplicazioni...
26 ...

```

Listato 3.6: Pseudo-codice funzionamento operatore di Dirac con Tensor Core.

Il calcolo degli indici **y**, **z** e **t** viene effettuato nello stesso modo della versione che non fa uso dei Tensor Core ad eccezione di **x** che assume valori multipli di 8 in quanto i Tensor Core eseguono otto moltiplicazioni matrice-vettore alla volta. Allo stesso modo **idxh** assume valori multipli di 8 per la stessa ragione.

In figura 3.13 viene mostrato graficamente come i Tensor Core elaborano i siti del lattice lungo una coordinata. Nell'immagine il sito evidenziato di rosso scuro indica il sito corrente, mentre i siti in rosso chiaro indicano i siti adiacenti che i Tensor Core elaborano in una sola moltiplicazione. In questo caso è stato rappresentato un lattice in due dimensioni, ma il concetto è espandibile anche per il lattice di quattro dimensioni.

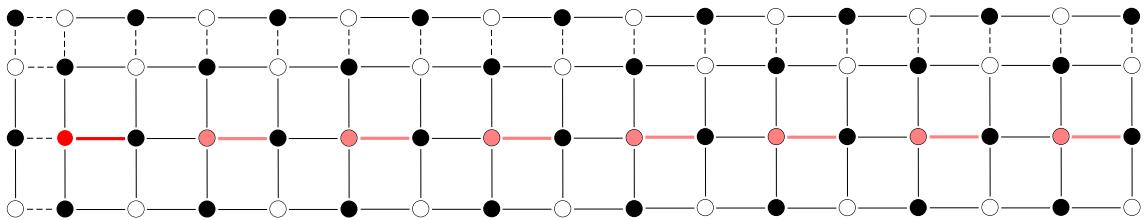


Figura 3.13: Esempio di elaborazione di 8 siti adiacenti lungo una coordinata con Tensor Core.

Possiamo osservare che i risultati parziali di ogni iterazione del lattice vengono accumulati nella variabile aux. In questa variabile, al termine della computazione, vi è il vettore risultato che verrà poi salvato in memoria. Con i Tensor Core è possibile ottimizzare questa operazione sfruttando l'operazione di accumulo prevista nella moltiplicazione tra le matrici. Si ricorda che l'operazione che i Tensor Core eseguono è la seguente:

$$\mathbf{C} = \mathbf{A} \times \alpha \mathbf{B} + \beta \mathbf{C}$$

Fino a questo momento abbiamo analizzato come sfruttare la moltiplicazione con i Tensor Core, vediamo ora come è possibile sfruttare l'accumulo (somma di **C**). Dal momento che dobbiamo sommare o sottrarre i risultati parziali delle moltiplicazioni possiamo sfruttare l'accumulo in **C** nei vari passaggi delle moltiplicazioni. Nel listato 3.5 vengono eseguite le prime quattro moltiplicazioni in cui i risultati vengono sommati ad aux, mentre nelle restanti quattro moltiplicazioni i risultati parziali vengono sottratti ad aux; possiamo quindi sfruttare la somma della matrice **C** nei primi quattro casi ponendo $\beta = 1$ in modo che il risultato della moltiplicazione $\mathbf{A} \times \mathbf{B}$ venga sommato a **C**. Per le restanti quattro operazioni, in cui deve essere sottratto il risultato parziale, si pone $\beta = -1$ in modo da sottrarre i risultati. Al termine della computazione otterremmo nella matrice **C** gli otto vettori ottenuti dalla moltiplicazione con Tensor Core.

Così facendo sfruttiamo la somma nel calcolo con Tensor Core senza eseguirla esplicitamente, riducendo le operazioni. Dal momento che i Tensor Core, indipendentemente

dal valore di β , eseguono anche l'operazione di accumulo, risulta vantaggioso sfruttare questa operazione piuttosto che implementare una procedura che effettui solo la somma (o sottrazione) dei risultati parziali. Sfruttare questa operazione può portare ad un ulteriore incremento delle prestazioni generali dell'algoritmo.

3.2.2 Analisi delle prestazioni

Esaminiamo le prestazioni delle soluzioni proposte nella sezione precedente, analizzandone efficienza e tempi stimati di esecuzione. Si prende come riferimento il tempo per moltiplicare una matrice 3×3 con un vettore 3×1 , nonché l'operazione che viene effettuata dall'operatore di Dirac e questo tempo è circa di $10ns$.

Analisi soluzione 1

Questa soluzione prevede di inserire una singola matrice 3×3 e il vettore 3×1 rispettivamente nelle matrici 16×16 allineate in alto a sinistra. Il kernel è stato eseguito con un blocco di dimensioni pari a 1024 e una griglia di dimensioni pari a 60. Le prestazioni ottenute con questa soluzione per moltiplicare 1920 matrici sono:

- Versione CUDA: $23.6\mu s$, oppure $\sim 11ns$ /matrice.
- Versione Tensor Core: $130ms$, oppure $\sim 68\mu s$ /matrice.

I risultati ottenuti per questa soluzione sono alquanto deludenti: esiste un divario di più di un ordine di grandezza tra la versione con Tensor Core e CUDA, rendendo impraticabile questa soluzione.

Analisi Soluzione 2

Con questa soluzione disponiamo cinque matrici 3×3 e quattro vettori 3×1 lungo la diagonale delle matrici 16×16 . Il kernel è stato eseguito con un blocco di dimensioni pari a 1920 e una griglia di dimensioni pari a 32. Le prestazioni ottenute da questa soluzione per moltiplicare 1920 matrici sono:

- Versione CUDA: $27.5\mu s$, oppure $\sim 12ns$ /matrice.
- Versione Tensor Core: $67.86\mu s$, oppure $\sim 35.3ns$ /matrice.

Come si evince dai tempi il kernel che fa uso dei Tensor Core non produce nessun miglioramento ma peggiora di più del doppio le prestazioni rispetto alla versione CUDA.

Questo peggioramento è dovuto al fatto che nella matrice 16×16 disponiamo le 5 matrici 3×3 lungo la diagonale, occupando solamente il 17.57% del totale della matrice, mentre occupiamo solo il 5.86% dell'intera matrice per i vettori. Questo, oltre a produrre uno spreco di memoria in fase di allocazione, fa sì che molti Tensor Core lavorino su dati che non sono utili ai fini del calcolo dell'algoritmo.

Nella matrice 16×16 agiscono 16 Tensor Core Unit: questo significa che solamente 4 dei Tensor Core Unit effettuano calcoli utili per il problema (ovvero i quattro che operano lungo la diagonale) mentre i restanti 12 non effettuano calcoli utili, quindi l'utilizzo di Tensor Core Unit utile è inferiore al 25%. Con questa analisi si spiega come questo modo di operare non risulti efficiente.

Analisi soluzione 3

Quest'ultima soluzione rappresenta quella che meglio si comporta sia in termini di occupazione di memoria che di prestazioni. Il kernel è stato eseguito con un blocco di dimensioni pari a 1024 e una griglia di dimensioni pari a 10. Le prestazioni ottenute con questo metodo moltiplicando 2560 matrici sono le seguenti:

- Versione CUDA: $23.86\mu s$, oppure $\sim 9.92 ns$ /matrice.
- Versione Tensor Core: $13.57\mu s$, oppure $\sim 5.48 ns$ /matrice.

Come si osserva dai tempi ottenuti la versione con Tensor Core produce un significativo miglioramento delle prestazioni. Il vantaggio di questo metodo è dovuto al fatto che con una sola moltiplicazione di due matrici 16×16 si ottengono 8 vettori in forma complessa. In questo caso l'uso di Tensor Core produce uno speedup di circa il doppio.

In figura 3.14 vengono mostrati i tempi impiegati dalle soluzioni appena descritte; per una migliore leggibilità del grafico si è esclusa la soluzione 1 in quanto non significativa sia in termini di efficienza che di prestazioni.

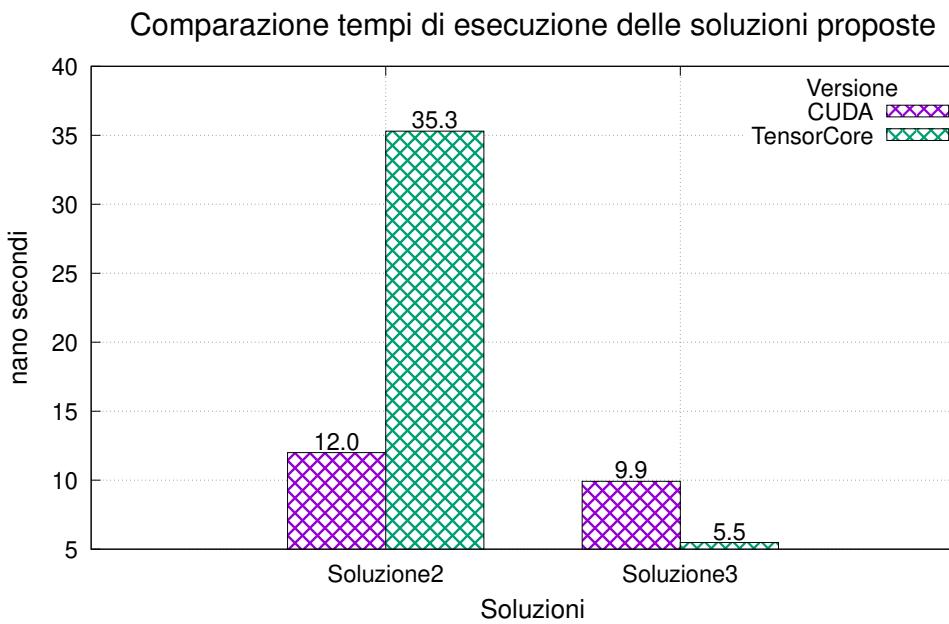


Figura 3.14: Analisi comparativa dei tempi di esecuzione delle soluzioni proposte per l'ottimizzazione dell'operatore di Dirac.

Conclusioni e sviluppi futuri

In questa tesi si è analizzata l’architettura dei Tensor Core e come questi ultimi possono essere utilizzati nell’operatore di Dirac. Nella prima parte del lavoro si è analizzato come le GPU siano diventate uno strumento rilevante per il calcolo parallelo. Si è quindi analizzata nel dettaglio l’architettura interna della GPU in particolare ai Tensor Core, studiando il loro funzionamento e i vantaggi che possono offrire nella moltiplicazione di matrici. Nell’ultima parte, descritta nel capitolo tre, si è studiato il funzionamento dell’operatore di Dirac contestualizzandone un possibile utilizzo nelle simulazioni LQCD; si è poi descritta la struttura dati utilizzata per rappresentare i collegamenti e i fermioni, enfatizzandone i vantaggi che possono offrire in termini di efficienza di calcolo dell’operatore.

Come contributo per un possibile miglioramento delle prestazioni dell’operatore di Dirac sono state formulate alcune soluzioni che prevedono l’uso dei Tensor Core. Di queste soluzioni sono state analizzate nel dettaglio le prestazioni che offrono e si è identificata in particolare una soluzione che, applicata all’operatore di Dirac, migliora di circa il doppio le prestazioni misurate con i soli CUDA Core.

Se da un lato i Tensor Core possono offrire un modo per aumentare le prestazioni dell’algoritmo di Dirac, il fatto che operino con tipi di dato a mezza precisione (half/FP16) può portare ad una eccessiva approssimazione dei valori ottenuti.

A fronte del lavoro svolto si è appreso che i Tensor Core nascono per accelerare algoritmi di machine learning e non principalmente per l’High Performance Computing. Questo rende l’architettura molto specifica per un tipo di operazione con vincoli molto rigidi: il lato delle matrici deve avere come dimensione un multiplo di otto, oltre a rappresentare i dati a mezza precisione. Questo rende molto difficile adattare i problemi di High Performance Computing a questi vincoli, infatti in questo caso è stato necessario disporre i dati nelle matrici in modo piuttosto complesso, che però ha portato a dei miglioramenti nelle prestazioni dell’operatore di Dirac.

Un possibile sviluppo del progetto è quello di modificare leggermente la struttura dati per consentire la creazione di un’area attorno al dominio in modo da ottenere un contorno ciclico affinché non sia necessario controllare che il sito corrente sia di bordo, potendo leggere direttamente i dati in memoria, aumentando ulteriormente le prestazioni dell’algoritmo. Un altro possibile sviluppo consiste nell’analizzare l’impatto che la rappresentazione a mezza precisione dei dati ha nella simulazione, ad esempio calcolando la massima variazione percentuale tra i dati ottenuti con rappresentazione a doppia precisione e i dati ottenuti con rappresentazione a mezza precisione.

Bibliografia

- [1] Maurice Herlihy e Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011.
- [2] John L Hennessy e David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
- [3] NVIDIA GeForce 256. <https://www.nvidia.co.uk/page/ geforce256>.
- [4] CUDA Toolkit. <https://developer.nvidia.com/cuda-zone>.
- [5] OpenCL: the open standard for parallel programming of heterogeneous systems. <https://www.khronos.org/opencl/>.
- [6] Gordon E. Moore. «Cramming more components onto integrated circuits». In: *Electronics* 38.8 (apr. 1965).
- [7] J. M. Shalf e R. Leland. «Computing beyond Moore's Law». In: *Computer* 48.12 (dic. 2015), pp. 14–23. ISSN: 1558-0814. DOI: 10.1109/MC.2015.374.
- [8] OpenACC: more science. Less programming. <https://www.openacc.org/>.
- [9] Introduction to GPUs: CUDA. <https://nyu-cds.github.io/python-gpu/02-cuda/>.
- [10] S. Markidis et al. «NVIDIA Tensor Core Programmability, Performance Precision». In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091.
- [11] A. Abdelfattah, S. Tomov e J. Dongarra. «Fast Batched Matrix Multiplication for Small Sizes Using Half-Precision Arithmetic on GPUs». In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 111–122. DOI: 10.1109/IPDPS.2019.00022.
- [12] Zhe Jia et al. «Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking». In: *CoRR* abs/1804.06826 (2018). arXiv: 1804.06826. URL: <http://arxiv.org/abs/1804.06826>.
- [13] Claudio Bonati et al. «Portable LQCD Monte Carlo code using OpenACC». In: *EPJ Web of Conferences* 175 (gen. 2018), p. 09008. DOI: 10.1051/epjconf/201817509008.

Ringraziamenti

Ringrazio la mia famiglia per il sostegno e il supporto offertomi durante il percorso di studi. Desidero ringraziare il professore Moreno Marzolla per la disponibilità, la professionalità e per avermi seguito durante lo sviluppo della tesi. Un grazie è rivolto a Enrico Calore e Fabio Schifano che mi hanno concesso l'opportunità di poter lavorare a questa tesi fornendo strumenti, consigli e suggerimenti. Ringrazio tutti coloro che hanno contribuito alla stesura di questa tesi offrendo consigli, osservazioni e critiche. Inoltre ringrazio amici e colleghi che hanno reso questa esperienza più piacevole oltre a sostenermi nei momenti più difficili. Desidero ringraziare l'amica e collega Martina Cavallucci per il sostegno e l'aiuto offerto durante tutto il percorso affrontato. Infine ringrazio i colleghi del gruppo *CeSeNA Security* che hanno contribuito a fornirmi conoscenze e metodologie che si sono rivelate fondamentali per gli studi, in particolare ringrazio Fabrizio Margotta il quale mi ha supportato e aiutato in ogni momento del percorso universitario.

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

**STUDIO E Sperimentazione di Soluzioni allo Stato
dell'Arte per il Rilevamento di Mascherina
Sanitaria Facciale con Micro-Controllori**

Elaborato in
Programmazione Di Applicazioni Data Intensive

Relatore
Prof. Gianluca Moro

Presentata da
Alessandro Marcantoni

Terza Sessione di Laurea
Anno Accademico 2019 – 2020

PAROLE CHIAVE

Machine Learning

Deep Learning

Convolutional Neural Network

Micro-Controllore

Python

*A chiunque mi sia stato vicino,
e mi abbia aiutato a raggiungere questo traguardo.*

Introduzione

L'intelligenza artificiale (AI) è una disciplina vastissima che trova origine nel 1950 quando Alan Turing ideò il famoso test (che prese successivamente il suo nome) per stabilire se una macchina fosse intelligente.

Nell'ambito del settore informatico, potremmo identificare l'AI - Artificial Intelligence - come la disciplina che si occupa di realizzare macchina (hardware e software) in grado di "agire" autonomamente (risolvere problemi, compiere azioni, ecc.).

Recentemente è diventata uno dei principali trend tecnologici strategici grazie ai progressi nella ricerca scientifica del settore ed all'enorme sviluppo della capacità di calcolo dei sistemi hardware. Invero, solo negli ultimi 10 anni i risultati della ricerca scientifica in questo campo hanno di gran lunga superato le conoscenze raggiunte in precedenza e si sono tradotti in tecnologie software di pubblico dominio. Ciò ne ha comportato l'applicazione in quasi ogni ambito delle discipline scientifiche.

I metodi di AI più recenti si distinguono da quelli classici precedenti in quanto basati sull'estrazione di conoscenza da masse di dati: maggiore è la disponibilità di dati, maggiore risulterà l'accuratezza della conoscenza estratta. L'introduzione di dispositivi elettronici e sensori in diversi ambienti accresce le dimensioni della massa di dati e di conseguenza cresce la necessità di ricercare nuovi metodi e tecniche per la loro gestione ed elaborazione.

Macchine intelligenti che "apprendono" e migliorano le proprie funzionalità: è questo il concetto alla base del **machine learning**, macro area di tecniche oggi molto popolari i cui recenti successi sono trasversali a numerosi settori (tra cui speech recognition e visione artificiale).

Il successo di queste tecniche è dovuto alla facilità che dimostrano nella gestione di ingenti quantità di dati, fondando la loro potenza su grandi insiemi di informazioni. Si stanno diffondendo molto velocemente in molti campi in quanto i modelli di sviluppo tradizionali non riescono più a garantire risultati veloci, precisi ed accurati.

L'utilizzo del machine learning presenta però dei limiti e può portare a risultati non soddisfacenti quanto si devono gestire immagini o testi in forma naturale poiché l'attività di preparazione dei dati e di selezione delle variabili

decisive per l'obiettivo finale richiede l'intervento di esperti e risulta dispendiosa. In questi casi si ottengono risultati generalmente più accurati e precisi con tecniche diverse, appartenenti alla macro area di ricerca del **deep learning**, che ha tra le sue prerogative quella di riuscire ad estrarre autonomamente le variabili migliori direttamente dai dati grezzi. Il deep learning è una tecnica relativamente giovane, le cui potenzialità sono ancora tutte da esplorare in grado di elaborare in maniera ancora più approfondita i dati e sarà affrontata con maggiore dettaglio successivamente all'interno del presente elaborato.

Il deep learning ha quindi permesso di migliorare drasticamente i risultati raggiunti in passato in numerosi settori, consentendo, ad esempio, lo sviluppo di auto a guida autonoma, assistenti virtuali in grado di comprendere una conversazione e di fornire risposte coerenti alle nostre domande o macchinari medicali capaci di identificare masse tumorali con una precisione maggiore rispetto a quella umana.

All'interno di questo elaborato verranno analizzati e sperimentati diversi approcci recenti in ambito *visione artificiale* (CV) e deep learning (DL), allo scopo di identificare persone all'interno di immagini e video e stabilire se queste indossano la mascherina oppure no.

Il lavoro di tesi è stato suddiviso nei seguenti capitoli:

- **Capitolo 1** - Analisi preliminare del problema;
- **Capitolo 2** - Panoramica sulle varie tecnologie esistenti in letteratura utilizzabili per il problema posto;
- **Capitolo 3** - Introduzione all'ambiente di lavoro e agli strumenti utilizzati;
- **Capitolo 4** - Descrizione delle soluzioni intermedie sperimentate;
- **Capitolo 5** - Descrizione della soluzione definitiva individuata;
- **Capitolo 6** - Descrizione dell'integrazione della soluzione;
- **Capitolo 7** - Il lavoro realizzato e il codice impiegato.

Indice

1	Analisi preliminare	1
1.1	Contesto applicativo	1
1.2	Immagini	2
1.3	Dataset utilizzati	3
1.3.1	RMFD	3
1.3.2	FaceMaskDataset	3
1.3.3	HardDataset	4
1.4	Obiettivi	4
2	Le tecnologie disponibili	7
2.1	Machine Learning	7
2.1.1	Sviluppo di un modello	9
2.1.2	Apprendimento supervisionato	9
2.1.3	Apprendimento non supervisionato	10
2.1.4	Apprendimento per rinforzo	11
2.1.5	Discesa del gradiente	12
2.1.6	Validazione del modello	13
2.2	Deep Learning	14
2.2.1	Struttura	15
2.2.2	Funzione di attivazione	17
2.2.3	Addestramento	19
2.3	Visione Artificiale	19
2.3.1	Reti Neurali Convoluzionali	20
3	Ambiente di lavoro e strumenti utilizzati	25
3.1	Micro-controllori	25
3.1.1	Nvidia Jetson Nano Developer Kit	26
3.1.2	Raspberry Pi 4 Model B	27
3.2	Python	28
3.2.1	Librerie Python	29
3.3	Google Colaboratory	29

4 Panoramica delle soluzioni testate	31
4.1 Addestrare una nuova rete	31
4.1.1 Transfer Learning	31
4.1.2 Xception	32
4.1.3 Data Augmentation	33
4.1.4 Fase di addestramento	34
4.1.5 Risultati ottenuti	35
4.2 Modello di AIZOOTech	35
4.2.1 Architettura della rete	36
4.2.2 Funzionamento della soluzione	37
4.2.3 Efficacia della soluzione	37
4.2.4 Limiti della soluzione	39
4.2.5 Possibili soluzioni	39
5 Analisi della soluzione definitiva	41
5.1 Funzionamento della soluzione	41
5.2 Rilevamento dei volti	42
5.3 Classificazione dei volti	43
5.4 Efficacia della soluzione	44
5.4.1 Efficacia per numero di persone	45
6 Deep Learning su micro-controllori	49
6.1 Tensorflow Lite	49
6.1.1 Vantaggi di Tensorflow Lite	49
6.1.2 Funzionamento di Tensorflow Lite	50
6.2 Descrizione dell'applicazione	50
6.2.1 Funzionamento dell'applicazione	51
6.3 Test su micro-controllori	51
6.3.1 Raspberry Pi 4 Model B 8GB	51
6.3.2 Nvidia Jetson Nano	52
6.3.3 Prestazioni	53
6.3.4 Nvidia Face Mask Detection	55
7 Il codice prodotto	57
7.1 Il codice della soluzione	57
Conclusioni e sviluppi futuri	61
Ringraziamenti	63
Bibliografia	65

Elenco delle figure

1.1	Come un computer vede le immagini	2
1.2	Esempi di immagini contenute nel dataset <i>RMFD</i>	3
1.3	Esempio di immagine contenuta nel dataset <i>FaceMaskDataset</i> . .	4
1.4	Esempio di immagine contenuta nel dataset <i>HardDataset</i>	4
1.5	Esempio di soluzione da ricercare.	5
2.1	Esempio di un sistema di suggerimenti guidato dal machine learning.	8
2.2	Rappresentazione grafica di un problema di clustering.	11
2.3	Modello di base nell'apprendimento di rinforzo.	11
2.4	Discesa del gradiente dal punto A al punto B visualizzabile graficamente.	12
2.5	Esempi di <i>Underfitting</i> , <i>Apprendimento bilanciato</i> e <i>Overfitting</i> . .	13
2.6	Gerarchia delle tecnologie di intelligenza artificiale.	14
2.7	Diverse astrazioni trattate nelle reti neurali.	15
2.8	Struttura tipica di una rete neurale.	16
2.9	Struttura di un neurone.	16
2.10	Grafico della funzione sigmoide	17
2.11	Grafico della funzione ReLU	18
2.12	Grafico della funzione Softplus	18
2.13	Campo recettivo di un neurone in una rete neurale semplice (sinistra) e in una rete neurale convoluzionale (destra).	20
2.14	Strati convoluzionali con campi recettivi locali rettangolari. . . .	21
2.15	Esempio di applicazione di filtri diversi e generazione delle mappe delle caratteristiche corrispondenti.	22
2.16	Rappresentazione in 3D di uno strato convoluzionale.	22
2.17	Rappresentazione di uno strato MaxPooling.	23
2.18	Architettura tipica di una rete neurale convoluzionale	24
3.1	Schema Nvidia Jetson Nano Developer Kit	27
3.2	Schema Raspberry Pi 4 Model B	28
4.1	Architettura della rete Xception.	32

4.2	Differenza tra una normale rete (sinistra) e una rete con connessione residua (destra).	33
4.3	Rappresentazione di uno strato convoluzionale separabile.	33
4.4	Esempio di data augmentation.	34
4.5	Dimostrazione del funzionamento della rete AIZOO.	35
4.6	Architettura della rete AIZOO.	36
4.7	Applicazione della <i>non-maximum suppression</i>	37
5.1	Struttura della soluzione finale.	42
5.2	Architettura della rete Dual Shot Face Detector.	42
5.3	Singolo blocco di strati della rete MobileNet.	43
6.1	Funzionamento del framework Tensorflow Lite.	50

Capitolo 1

Analisi preliminare

In questo capitolo vengono introdotti il contesto applicativo e i dati utilizzati durante il lavoro. Viene inoltre illustrato l'obiettivo del presente elaborato di tesi.

1.1 Contesto applicativo

Nel corso delle prime settimane di gennaio 2020, alcuni scienziati hanno riscontrato polmoniti anomale in alcuni lavoratori del mercato umido di Wuhan, la cui causa si è rivelata essere un nuovo *coronavirus*, identificato come *SARS-CoV-2*.

Alla fine del gennaio 2020, non si era ancora riusciti ad accettare le caratteristiche del virus, tanto da nutrire incertezze sulle esatte modalità di trasmissione e sulla patogenicità (la capacità di creare danno), sebbene si fosse riscontrata la facilità di trasmissione da persona a persona. La malattia associata è stata poi riconosciuta con il nome di **COVID-19**.

I pazienti accusavano sintomi simili all'influenza come dermatiti, febbre, tosse secca, stanchezza, difficoltà di respiro. Nei casi più gravi, spesso riscontrati in soggetti già gravati da precedenti patologie, si sviluppavano polmoniti ed insufficienza renale acuta talvolta così gravi da condurre al decesso.

Una volta individuate le principali modalità di diffusione del SARS-CoV-2 nelle goccioline di saliva derivanti da starnuti e tosse e nei contatti diretti personali e contatti tra le mani contaminate e le mucose, l'Organizzazione Mondiale della Sanità (OMS) emanava raccomandazioni dirette a limitare la diffusione del virus. Le linee guida prevedono anche oggi il lavaggio regolare delle mani per più di 20 secondi, la copertura di bocca e naso in caso di tosse o starnuti ed il distanziamento da chiunque mostri sintomi di malattie respiratorie (come appunto tosse e starnuti).

L'aumento esponenziale dei malati, con conseguente sovraffollamento delle strutture ospedaliere, ha indotto numerosi paesi in tutto il mondo, allo scopo di diminuire il più possibile i contatti interpersonali, a ricorrere a periodi di "lockdown" totale.

Altra misura adottata per contenere il numero di contagi è l'utilizzo delle mascherine. Da alcune ricerche [1] è emerso invero che le mascherine chirurgiche possono ridurre efficacemente l'emissione di particelle virali nell'ambiente e, di conseguenza, diminuire il rischio di contagio. Non sorprende quindi il fatto che, in vari stati tra cui l'Italia, l'utilizzo delle mascherine sia stato reso obbligatorio ovunque, tranne rare eccezioni. L'inadempimento a tale normativa, così come l'inesatto adempimento (vedi il non corretto utilizzo) comporta l'applicazione di sanzioni pecuniarie. Sarà sempre più importante, anche per il momento in cui stiamo vivendo, accertare l'utilizzo da parte delle persone e/o il corretto utilizzo di mascherine.

L'obiettivo principale di questa tesi è quello di trovare un metodo in grado di stabilire se un soggetto inquadrato da una fotocamera indossi correttamente la mascherina, con lo stesso grado di certezza con cui lo fa una persona.

1.2 Immagini

Le immagini, su cui opererà la soluzione, sono un tipo di dato destrutturato. Ciò significa che sono sprovviste di un modello in grado di spiegarne la semantica: aspetto che rende il processo di estrazione dell'informazione molto più complesso rispetto ai dati strutturati. Basti pensare che, di fatto, le immagini sono costituite da griglie (tre sovrapposte nel caso di foto a colori codificate in formato *RGB* o una soltanto per foto in bianco e nero) contenenti valori numerici (solitamente tra 0 e 255). Risulta perciò molto difficile per la macchina interpretare il loro contenuto e apprendere da esse.

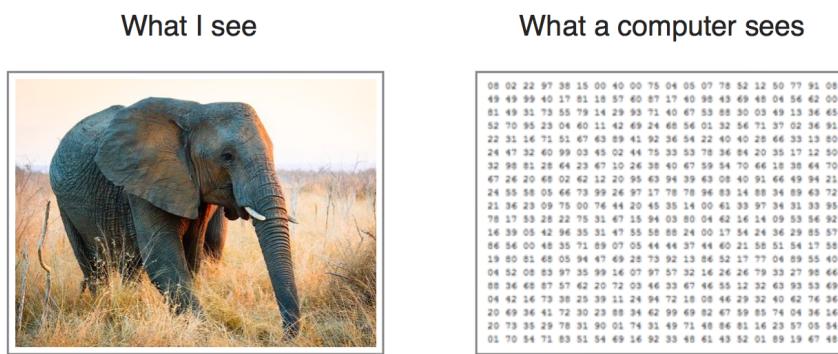


Figura 1.1: Come un computer vede le immagini.

1.3 Dataset utilizzati

È possibile reperire dal web un numero considerevole di immagini contenenti persone munite di mascherina facciale e, ovviamente, di altre che non lo sono. È particolarmente utile raccogliere molte immagini (esempi) in modo da favorire il processo di addestramento. Inoltre, è opportuno che il dataset risulti bilanciato nelle sue classi: il numero di immagini di persone con mascherina dovrebbe equivalere quello di immagini di persone senza mascherina. Questi accorgimenti permettono al modello di apprendere in modo più efficace. In particolare, sono stati utilizzati diversi dataset per raggiungere l'obiettivo prefissato.

1.3.1 RMFD

Questo dataset [2] contiene 95000 immagini, di cui 90000 raffigurano volti di persone senza mascherina e 5000 con mascherina. Le immagini non sono etichettate, perciò è stato necessario automatizzare il processo di assegnazione delle etichette.



Figura 1.2: Esempi di immagini contenute nel dataset *RMFD*.

1.3.2 FaceMaskDataset

Sono qui contenute circa 8000 immagini già etichettate di persone con e senza mascherina. Queste sono divise in *training set* e *validation set*, che corrispondono rispettivamente al 77% e al 23% del totale. L'aspetto interessante di questo dataset è che contiene immagini raffiguranti persone per intero (talvolta anche più di una) e non solamente il volto, perciò può essere un ottimo modo per testare la soluzione completa [3].



Figura 1.3: Esempio di immagine contenuta nel dataset *FaceMaskDataset*.

1.3.3 HardDataset

Il seguente è invece un dataset creato opportunamente per testare la rete in situazioni "difficili". Le circa 100 immagini al suo interno contengono infatti persone lontane dall'inquadratura o girate di profilo.



Figura 1.4: Esempio di immagine contenuta nel dataset *HardDataset*.

1.4 Obiettivi

L'obiettivo principale di questa tesi, come già sottolineato, sarà sviluppare un sistema capace di etichettare le persone inquadrate all'interno di un'immagine o un video in modo da stabilire se esse indossano la mascherina sanitaria facciale o meno.

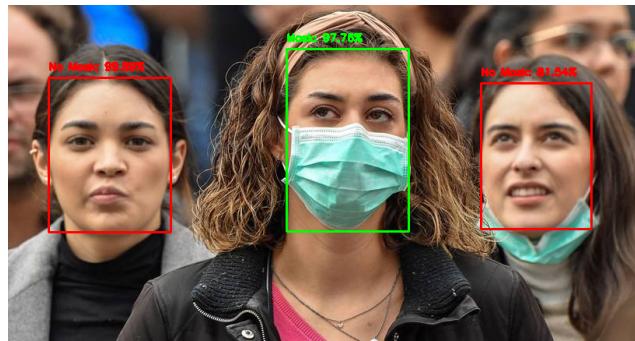


Figura 1.5: Esempio di soluzione da ricercare.

Verranno considerati e presi in considerazione diversi approcci esistenti al problema, usando metodi di **deep learning** differenti, per testare quale si adatti meglio al problema in causa.

Inoltre, la soluzione dovrà essere pensata per essere sfruttata da micro-controllori, i quali hanno risorse limitate.

Capitolo 2

Le tecnologie disponibili

In questa parte verranno descritte più nello specifico tutte le tecnologie delle quali si è già accennato nell'introduzione, per rendere più chiari gli obiettivi e le metodologie di lavoro impiegate di cui ai capitoli successivi.

2.1 Machine Learning

Il machine learning è una branca dell'intelligenza artificiale sviluppatasi recentemente (negli ultimi decenni del 1900 circa) che fornisce un nuovo approccio ai problemi il cui obiettivo è stimare, predire o ipotizzare "*qualcosa*".

Una delle definizioni più citate quando si parla di questa tecnologia, è quella proposta dal professor Tom Mitchell [4]:

Si dice che un programma apprende dall'esperienza E con riferimento ad alcune classi di compiti T e con misurazione della performance P , se le sue performance nel compito T , come misurato da P , migliorano con l'esperienza E .

Cioè, riuscire a far eseguire un'attività a una macchina, che riesce in maniera autonoma a migliorare il modo in cui la esegue, usando l'esperienza che acquisisce nel farlo. La macchina quindi *impara ad eseguire un compito* in maniera sempre più accurata e precisa, senza essere stata programmata in maniera specifica a farlo.

Questo approccio rappresenta una completa innovazione nel modo in cui gli algoritmi vengono utilizzati per risolvere problemi, in quanto generalmente l'approccio classico ne prevedeva lo sviluppo sfruttando la conoscenza plessa dell'autore o autori, mentre al contrario in questo modo la macchina impara da esempi e dati.

La circostanza che non esista un unico campo dove poterne sfruttare le potenzialità, consente di adattarsi benissimo ad un insieme molto vasto di problematiche. Per citarne qualcuna:

- Predizione di consumi;
- Categorizzazione di elementi;
- Previsione andamento meteorologico;
- Stima di costi.

E non solo. Al giorno d'oggi, infatti, questa tecnologia è impiegata negli ambiti e nei settori più disparati: sanità, sicurezza, management aziendale e previsione di fenomeni o eventi di carattere generale. Anche gli smartphone si affidano all'intelligenza artificiale per portare a termine i compiti più disparati (la capacità di predizione del testo durante la digitazione o le fotocamere in grado di applicare filtri personalizzati a ogni immagine).

Il progredire di questi metodi è stato sostenuto anche dalla crescente disponibilità di dati da analizzare che, come già accennato in precedenza, negli anni si sono letteralmente moltiplicati. Unitamente all'aumento della disponibilità di risorse computazionali si sono create le condizioni ideali per lo sviluppo di nuovi studi e ricerche.

Per fare un esempio di quanto questa tecnologia abbia avuto impieghi in casi del tutto reali e concreti, si prendano i **suggerimenti di prodotti simili** a quelli che si stanno visualizzando (ad esempio durante una sessione di shopping online). Non sono altro che dei sistemi che utilizzano il machine learning per trovare legami tra prodotti in maniera automatica, per poi successivamente consigliarne l'acquisto ai visitatori.



Figura 2.1: Esempio di un sistema di suggerimenti guidato dal machine learning.

Sono sistemi che ormai sono entrati nella quotidianità di tutti, e che silenziosamente garantiscono un'esperienza d'uso più gradevole e in grado di invogliare a passare più tempo sul sito o l'applicazione.

E per farsi un’idea di quanto siano importanti per le aziende questi meccanismi, basti pensare che nel caso di Amazon il 35% delle vendite viene generato proprio dai suggerimenti visualizzati all’interno delle pagine web.

2.1.1 Sviluppo di un modello

All’atto pratico, alla base del machine learning c’è un **modello**, il vero e proprio nucleo che compone il sistema che si vuole costruire.

Il modello deve riuscire ad approssimare nel modo migliore possibile la realtà descritta dai dati, trovando correlazioni non banali tra loro.

In base alla tipologia e all’organizzazione dei dati a disposizione, i problemi che si possono riscontrare durante lo sviluppo di un sistema di machine learning possono rientrare in una delle seguenti tipologie:

1. Apprendimento supervisionato;
2. Apprendimento non supervisionato;
3. Apprendimento per rinforzo.

2.1.2 Apprendimento supervisionato

Nell’**apprendimento supervisionato** i dati disponibili sono composti da due parti principali: una parte di input da affidare al modello che contiene uno o più campi descrittivi, che definiscono le caratteristiche di ogni elemento, e una parte in cui è specificato il valore in output reale.

Si prenda come esempio il seguente dataset (di tipo supervisionato), utilizzato per identificare la specie di iris partendo dai dati generici del fiore:

Lung. sepalo	Larg. sepalo	Lung. petalo	Larg. petalo	Specie
5.7	4.4	1.5	0.4	<i>Setosa</i>
7.7	3.8	6.7	2.2	<i>Virginica</i>
5.4	3.7	1.5	0.2	<i>Setosa</i>
7.2	3.6	6.1	2.5	<i>Virginica</i>
6.0	3.4	4.5	1.6	<i>Versicolor</i>
6.2	3.4	5.4	2.3	<i>Virginica</i>
5.0	3.3	1.4	0.2	<i>Setosa</i>
6.3	3.3	4.7	1.6	<i>Versicolor</i>
6.7	3.3	5.7	2.1	<i>Virginica</i>

Tabella 2.1: Esempio di dati in ingresso con le caratteristiche di diversi fiori.

Le prime 4 colonne compongono l'insieme di dati da passare al modello per far sì che riesca ad elaborare tutte le correlazioni esistenti, mentre l'ultima colonna contiene il valore che ci si aspetta in output dopo l'elaborazione.

Il modello, avendo disponibile il risultato corretto da predire in uscita, può effettuare una predizione, e comportarsi di conseguenza in base alla differenza tra risultato atteso e risultato predetto.

Quindi ciclicamente, un sistema di machine learning supervisionato segue questo schema:

- Predizione in base ai dati di input;
- Controllo della differenza tra risultati attesi e predetti (**errore del modello**);
- Miglioramento del modello in base agli errori commessi.

Questo processo è definito **addestramento** del modello, e viene eseguito finché l'errore ottenuto non raggiunge un livello abbastanza basso da poter considerare il modello valido. Verrà affrontato nella sezione 2.1.5 il modo in cui il modello riesce a migliorare le proprie predizioni.

I modelli supervisionati sono generalmente usati per 2 diversi tipi di problemi:

- **Classificazione** - Come nell'esempio della tabella 2.1 relativa alla specie corretta dell'iris, abbinare l'elemento alla categoria corretta;
- **Regressione** - Predire un valore numerico, al posto di una categoria (come ad esempio il prezzo di un immobile), partendo dalle caratteristiche dell'elemento.

2.1.3 Apprendimento non supervisionato

Nell'**apprendimento non supervisionato** i dati disponibili **non** contengono alcun tipo di indicazione sulla bontà della previsione. Questo tipo di informazione è detta anche *non etichettata*,

Ovviamente i dati con il valore richiesto in output sono molto più facili da reperire, poiché non è necessaria l'analisi di una persona umana (il dataset d'esempio precedente ha richiesto l'identificazione della specie corretta per ogni fiore presente).

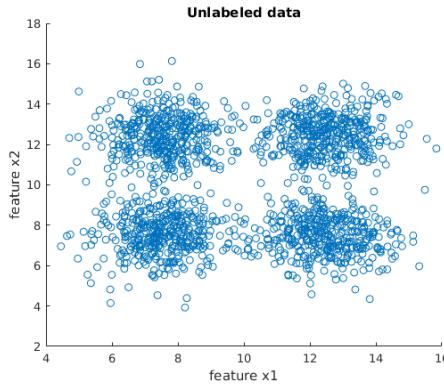


Figura 2.2: Rappresentazione grafica di un problema di clustering.

Fonte: https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/unsupervised_learning.html

Non avendo alcun tipo di *indicazione* sul valore che si aspetta in uscita, **non esiste un metodo generico per controllare l'errore** sviluppato dal modello.

Generalmente questo approccio viene utilizzato nei problemi di **clustering**, quando cioè è necessario classificare un insieme di elementi senza però a priori avere le possibili categorie.

Il fatto di non sapere in partenza le diverse categorie, risulta però essere in molti casi vantaggioso, poiché il modello riesce ad analizzare e cogliere legami spesso *invisibili* alla mente umana.

2.1.4 Apprendimento per rinforzo

L'apprendimento per rinforzo è una tipologia utilizzata quando è necessario che il nostro modello impari relazionandosi con l'ambiente in cui si trova: non sfrutta dataset pregressi, ma le mutazioni che riesce a captare.

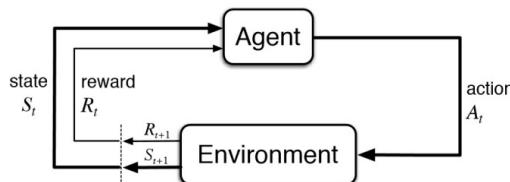


Figura 2.3: Modello di base nell'apprendimento di rinforzo.

Fonte: <https://www.kdnuggets.com/images/reinforcement-learning-fig1-700.jpg>

Viene utilizzato in casi molto particolari, generalmente facendo eseguire un compito a un modello in maniera ciclica. Dopo tantissimi tentativi (anche

svariati milioni) il sistema risulta esser in grado di svolgere operazioni complesse con un'accuratezza maggiore di quella umana.

Ad esempio, l'apprendimento per rinforzo è stato utilizzato in maniera soddisfacente per insegnare a un modello senza alcuna conoscenza di base a giocare a scacchi, affinando la propria tecnica partita dopo partita, arrivando a competere (e vincere) anche contro campioni mondiali.

2.1.5 Discesa del gradiente

Il metodo della discesa del gradiente è la base del machine learning, ciò che permette al modello di migliorarsi iterazione dopo iterazione.

In matematica viene utilizzato per trovare i punti di minimo e di massimo all'interno di **una funzione**, procedendo a step graduali.

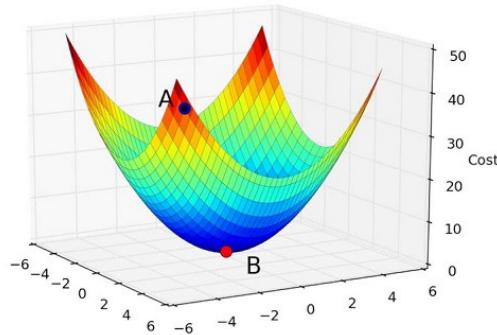


Figura 2.4: Discesa del gradiente dal punto A al punto B visualizzabile graficamente.

Fonte: <https://medium.com/abdullah-al-imran/intuition-of-gradient-descent-for-machine-learning-49e1b6b89c8b>

Nello stesso modo, quando nel processo di addestramento del modello vengono generate le predizioni, è possibile sviluppare una funzione che associa ad ognuna l'errore generato.

Si aggiunge quindi *una dimensione* alla funzione del modello, così da poter applicare la discesa del gradiente e minimizzare l'errore generato dalle predizioni future.

Nel grafico in figura 2.4 ad esempio, le dimensioni x e y che compongono il piano sono quelle relative ai dati di input, mentre la terza dimensione, la z , viene aggiunta per associare l'errore prodotto dal modello.

Per ridurre l'errore quindi, è necessario:

- Calcolare la funzione d'errore e il suo valore attuale;
- Calcolare il gradiente della funzione nel punto, quindi *la direzione* da seguire per raggiungere il punto di minimo;

- Sottrarre al punto iniziale un vettore proporzionale al gradiente;
- Ricominciare da capo finché non viene raggiunto il punto obiettivo.

Questo approccio affonda le proprie radici nel campo della statistica, dove viene chiamato **regressione**. Si occupa di analizzare un insieme di dati che possono essere rappresentati da una funzione (lineare, polinomiale, o di altri gradi) e cercare di approssimare al meglio le variabili dipendenti e indipendenti.

2.1.6 Validazione del modello

Avendo un insieme di dati su cui addestrare il modello, non vengono utilizzati tutti per la fase di training, ma al contrario si suddivide in due (o tre) sottogruppi differenti e non sovrapposti.

Questo modo di approcciare i problemi è detto **hold-out**, e prevede la divisione dei dati in due insiemi secondo una proporzione nota (generalmente 30%-70%). Sull'insieme più grande (*training set*) verrà addestrato il modello, mentre il restante più piccolo (*validation set*) verrà utilizzato per controllare la validità dell'addestramento.

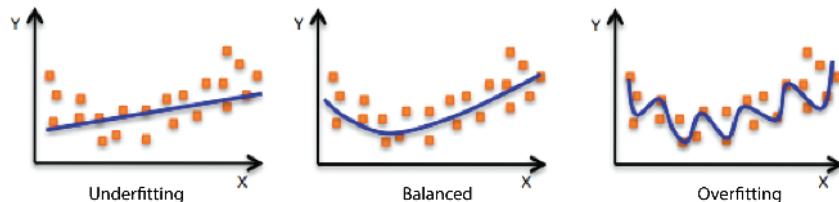


Figura 2.5: Esempi di *Underfitting*, *Apprendimento bilanciato* e *Overfitting*.

Il rischio, infatti, è quello di addestrare **troppo** un modello, in grado di produrre risultati accurati al 100% solo sui dati su cui viene sviluppato.

Al contrario, un buon sistema di machine learning dovrebbe essere il più generale possibile, ed è per questo che questa metodologia utilizza il gruppo di dati più piccolo, detto **validation set**, per controllare il comportamento del modello dopo l'addestramento:

- Se i risultati ottenuti nelle predizioni sono molto buoni solo sul training set ma non sul validation, allora c'è overfitting;
- Se su entrambi gli insiemi il modello si comporta *circa* nello stesso modo, allora l'addestramento risulta valido.
- Se il modello risulta poco accurato su entrambi gli insiemi, allora c'è underfitting.

2.2 Deep Learning

Il machine learning, però, mostra alcune limitazioni nella precisione delle previsioni sviluppate, in particolar modo se:

- Il tipo di dato che deve gestire è di tipo immagine o testuale (in particolar modo con testi naturali);
- La quantità di dati disponibili per il training non è così abbondante.

Inoltre, per poter utilizzare il machine learning è necessario **selezionare le feature manualmente**, e ciò richiede delle competenze non banali.

Al fine di affrontare queste problematiche, negli ultimi anni una branca del machine learning ha guadagnato sempre più popolarità nel campo dell'intelligenza artificiale: il **deep learning**.

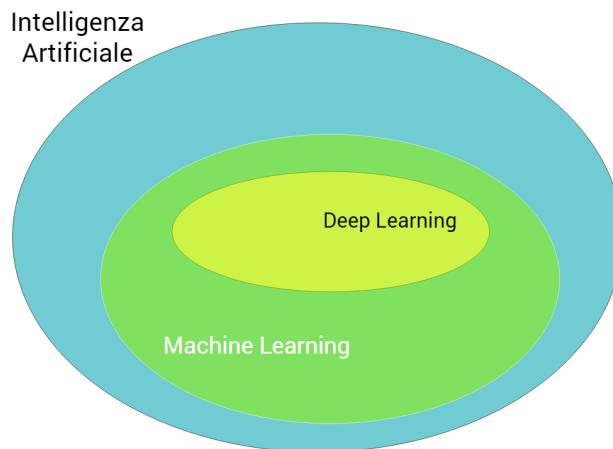


Figura 2.6: Gerarchia delle tecnologie di intelligenza artificiale.

Questa disciplina trova fondamento e ispirazione nella struttura dei neuroni all'interno del cervello umano, dove non esiste un unico punto di ingresso nel quale elaborare i dati in entrata, ma al contrario esistono molti più nodi, suddivisi in diversi strati (da qui il motivo della parola *deep*) che collaborano tra loro per raggiungere risultati estremamente accurati.

Nello stesso modo, all'interno delle **reti neurali** sviluppate tramite deep learning non si ritrova *un solo modello* da addestrare e da gestire, ma una fitta rete di *neuroni* raggruppati all'interno di strati.

Dopo aver eseguito l'addestramento sui dati di training, queste riescono ad eseguire i compiti più disparati, con una **precisione maggiore** di quella raggiungibile tramite machine learning: categorizzare immagini, riconoscere

lettere e numeri scritti a mano, e nei casi più avanzati anche a generare immagini realistiche o testi di senso compiuto.

La divisione in strati dei neuroni consente di gestire in maniera progressiva l'astrazione dei problemi da affrontare: nei livelli più bassi si riescono a riconoscere aspetti e pattern semplici, mentre negli strati più avanzati si gestiscono gli elementi di più alto livello, così da comprendere e individuare caratteristiche sempre più complesse.

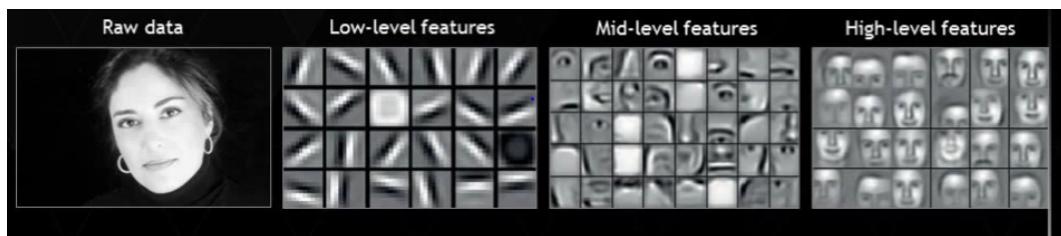


Figura 2.7: Diverse astrazioni trattate nelle reti neurali.

Fonte: <https://www.analyticsvidhya.com/blog/2017/04/comparison-between-deep-learning-machine-learning/>

Questo nuovo approccio ha consentito nel tempo di conseguire risultati impressionanti per precisione e tempi richiesti per il calcolo.

L'esempio più cristallino di quanto una rete neurale sia molto più accurata rispetto a metodi tradizionali, è l'applicazione a problemi riguardanti le immagini (uno su tutti riconoscimento e la categorizzazione dei soggetti) o il riconoscimento della grafia. Le reti neurali più efficaci per le immagini sono quelle convoluzionali e recentemente è stata dimostrata la loro efficacia anche per il trattamento di dati testuali [5].

Prendendo come esempio il popolare dataset MNIST (utilizzato generalmente come punto di riferimento in questo ambito), contenente svariate migliaia di immagini raffiguranti cifre, il tasso di errore migliore mai fatto registrare è attualmente dello **0.23%** [6], ed è stato raggiunto proprio da una rete neurale.

2.2.1 Struttura

Come detto, un sistema di deep learning si sviluppa in una struttura molto più complessa e articolata rispetto a un modello di machine learning.

Si creano degli strati di neuroni, elementi fondamentali che altro non sono che dei semplici algoritmi ai quali, dato in input un vettore di n elementi, generano un risultato in output generalmente compreso tra 0 e 1.

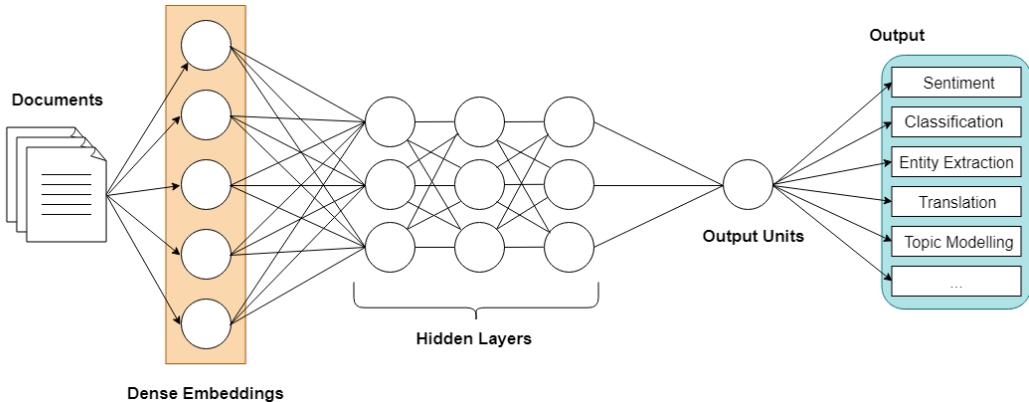


Figura 2.8: Struttura tipica di una rete neurale.

Gli strati sono tra loro collegati, e le informazioni in uscita di uno sono quelle in entrata del successivo. Ogni livello rappresenta un grado di astrazione differente, nel quale vengono gestiti aspetti differenti del dato in input, come in figura 2.7. Nel livello di output viene invece elaborata l'informazione di alto livello richiesta.

Ogni singolo neurone, riceve in input un insieme di valori all'interno di un vettore (detto in certi casi *tensore*) ed elabora internamente il valore y da ritornare in output tale che:

$$y = \sigma\left(\sum_{i=1}^d w_i x_i + b\right)$$

Viene quindi calcolata la combinazione lineare degli input x con i pesi w del singolo neurone, viene sommata l'intercetta b e successivamente viene applicata una funzione di attivazione σ .

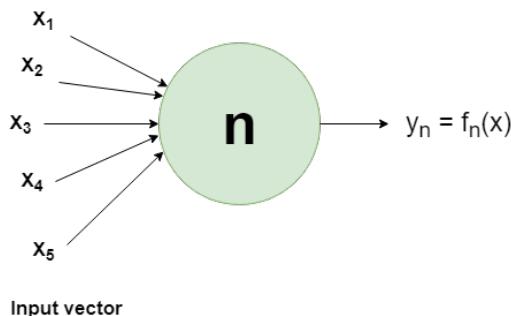


Figura 2.9: Struttura di un neurone.

2.2.2 Funzione di attivazione

Come accennato in precedenza, la **funzione di attivazione** di un neurone definisce l'output che quel nodo restituisce. Ne esistono di diversi tipi ma si possono classificare in:

- **Funzioni di attivazione lineari o di identità:** l'output è proporzionale all'input. Solitamente poco utilizzate nelle reti neurali poiché non permettono di creare mappature complesse dei dati.
- **Funzioni di attivazione non lineari:** sono le più diffuse all'interno delle reti neurali.

Le funzioni di attivazione non lineari permettono quindi di modellare dati complessi come immagini, video, audio e set di dati con correlazioni non lineari o con elevata dimensionalità. Ve ne sono svariate, tuttavia le più utilizzate sono le seguenti:

- **Sigmoide:** è un'approssimazione della funzione "gradino". È caratterizzata da una curva a forma di "S". Viene spesso utilizzata per mappare valori reali in probabilità.

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$$

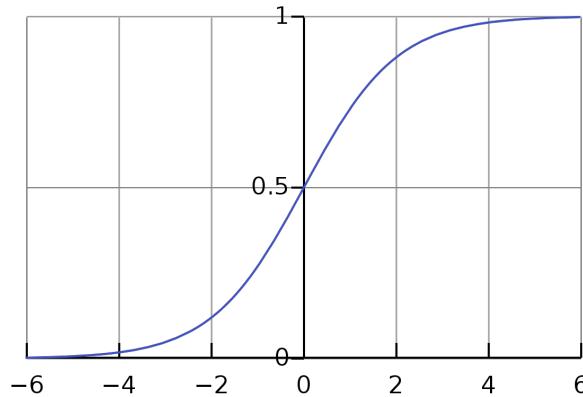


Figura 2.10: Grafico della funzione sigmoide

- **ReLU:** è stata dimostrata essere la migliore da utilizzare all'interno di reti neurali profonde poiché ne favorisce un migliore addestramento

rispetto alle altre funzioni di attivazione [7]. Di fatto, restituisce la parte positiva dell'argomento.

$$\sigma(x) = x^+ = \max(0, x)$$

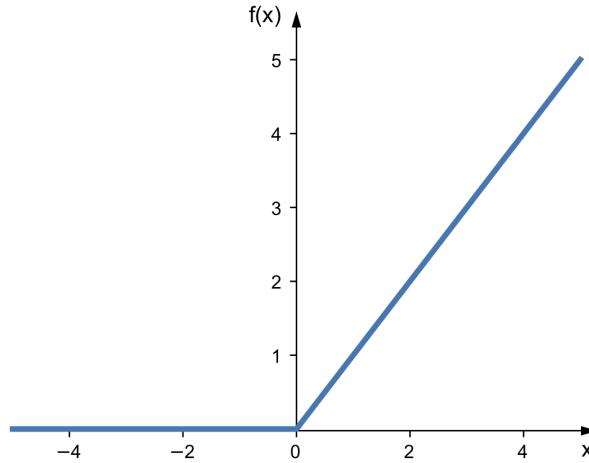


Figura 2.11: Grafico della funzione ReLU

- **Softplus:** è un'approssimazione della funzione \max .

$$\sigma(x) = \ln(1 + e^x)$$

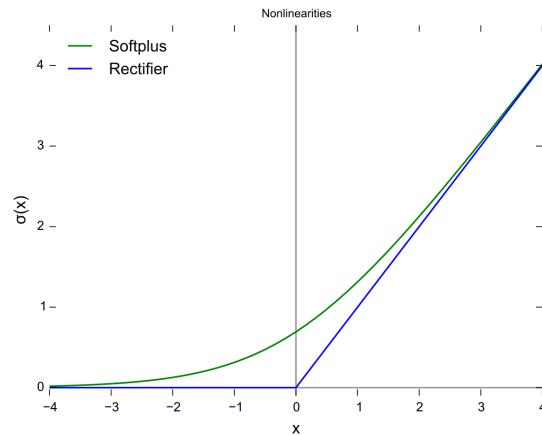


Figura 2.12: Grafico della funzione Softplus

2.2.3 Addestramento

Per anni i ricercatori hanno ricercato un metodo per addestrare reti neurali. Nella seconda metà degli anni '80, è stato introdotto l'algoritmo di *retro-propagazione* [8]. Quest'ultimo è in grado di calcolare il gradiente dell'errore commesso dalla rete considerando tutti i parametri della stessa. Ovvero, può scoprire come ogni peso della rete debba essere modificato in modo da ridurre l'errore: una volta calcolato il gradiente, semplicemente effettua un passo di discesa del gradiente. Il processo è ripetuto finché la rete converge alla soluzione.

L'algoritmo di retro-propagazione gestisce un gruppo di istanze (*batch*) per volta e scorre il training set diverse volte. Ogni batch attraversa quindi la rete, proprio come per ottenere una predizione: l'unica differenza è che i risultati degli strati intermedi vengono conservati. Successivamente, viene calcolato l'errore commesso dalla rete confrontando l'output con i risultati desiderati e quanto ogni connessione della rete (ogni peso) abbia contribuito all'errore. Quest'ultimo passaggio viene eseguito a ritroso, ovvero partendo dallo strato di output e arretrando fino ad arrivare allo strato di input. Infine, l'algoritmo effettua un passo di discesa del gradiente per modificare i pesi sfruttando il gradiente appena calcolato.

2.3 Visione Artificiale

La disciplina che si occupa dei problemi che hanno immagini come dati è detta **visione artificiale**. Il suo scopo principale è quello di riprodurre la vista umana. Vedere è inteso non solo come l'acquisizione di una fotografia bidimensionale di un'area ma soprattutto come l'interpretazione del contenuto di quell'area.

Alcuni classici problemi nella visione artificiale sono infatti:

- **Riconoscimento:** uno o più oggetti prespecificati o memorizzati possono essere ricondotti a classi generiche usualmente insieme alla loro posizione nella scena.
- **Identificazione:** viene individuata un'istanza specifica di una classe. Es. Identificazione di un volto.
- **Rilevamento:** l'immagine è scandita fino all'individuazione di una condizione specifica. Es. Individuazione di possibili cellule anormali o tessuti nelle immagini mediche.

Per svolgere questi compiti, l'immagine deve essere memorizzata in formato digitale, ovvero viene creata una rappresentazione numerica corrispondente

della stessa. L'immagine viene quindi trasformata da punti ottici (pixel) a valori digitali in memoria, con corrispondenza in matrice. A questo punto, l'immagine verrà analizzata da algoritmi in grado di estrarre da essa varie informazioni. Queste possono collocarsi, come già visto prima, su più livelli:

- basso livello: come le statistiche sulla presenza dei vari toni di grigio o colori, sui bruschi cambiamenti di luminosità, ecc.
- livello intermedio: caratteristiche relative a regioni dell'immagine e a relazioni tra regioni.
- alto livello: determinazione di oggetti con valenza semantica.

2.3.1 Reti Neurali Convoluzionali

Le **Reti Neurali Convoluzionali** (CNN) sono nate negli anni '80 in seguito a diversi studi della corteccia visiva del cervello [9] [10]. In particolare, è stato scoperto che i neuroni della corteccia visiva hanno un piccolo *campo recettivo locale*: ciò significa che reagiscono solo agli stimoli visivi che provengono da una regione limitata del campo visivo. I campi recettivi di neuroni diversi possono sovrapporsi, e insieme compongono l'intero campo visivo.

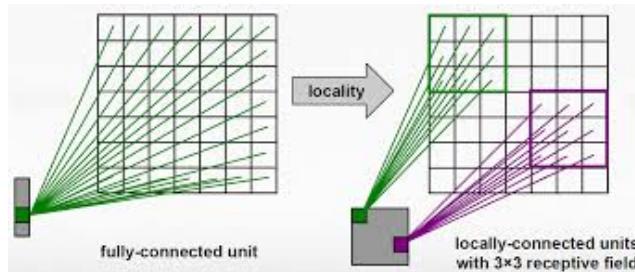


Figura 2.13: Campo recettivo di un neurone in una rete neurale semplice (sinistra) e in una rete neurale convoluzionale (destra).

È stato scoperto inoltre che alcuni neuroni reagiscono solo a immagini di linee orizzontali, mentre altri reagiscono solo a linee con diverse orientazioni (ciò significa che due neuroni potrebbero avere lo stesso campo recettivo ma reagiscono a linee con orientazioni diverse). Si è inoltre notato come alcuni neuroni abbiano campi recettivi più ampi e reagiscano a pattern più complessi (che sono combinazioni di pattern più semplici).

Queste osservazioni hanno suggerito che vi potesse essere una struttura gerarchica tra i neuroni: i neuroni di alto livello si basano sui risultati dei neuroni di livello inferiore a loro vicini. Questa complessa e potente architettura

è in grado di identificare anche i pattern più complessi in qualsiasi area del campo visivo.

Le Reti Neurali Convoluzionali presentano molti elementi in comune con le reti neurali più semplici; tuttavia, introducono anche alcune novità:

- Strato Convoluzionale
- Strato di Pooling

Strato Convoluzionale

L'elemento costitutivo più importante per una Rete Neurale Convoluzionale, come si può intuire dalla denominazione, è sicuramente lo strato convoluzionale. I neuroni di questo strato non sono connessi ad ogni input dello strato precedente, ma solo agli input che fanno parte del loro campo recettivo. Questa architettura permette alla rete di concentrarsi su piccole caratteristiche di basso livello nei primi strati, per poi assemblarle in caratteristiche più grandi e di alto livello negli strati successivi. Questa struttura gerarchica è comune nelle immagini reali: questo è uno dei motivi per cui le CNN si comportano così bene con il riconoscimento di immagini.

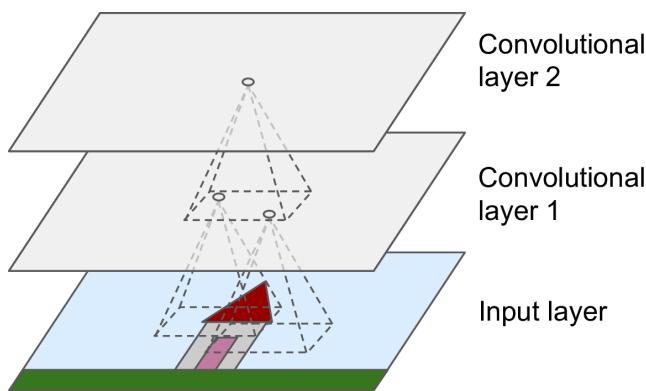


Figura 2.14: Strati convoluzionali con campi recettivi locali rettangolari.

Ogni neurone all'interno di uno strato presenta dei "pesi". Questi possono essere rappresentati come piccole immagini della dimensione del campo recettivo e prendono il nome di *filtri*. L'applicazione di un filtro ad un'immagine comporta la creazione di una *mappa delle caratteristiche*, che evidenzia le zone dell'immagine che attivano maggiormente il filtro.

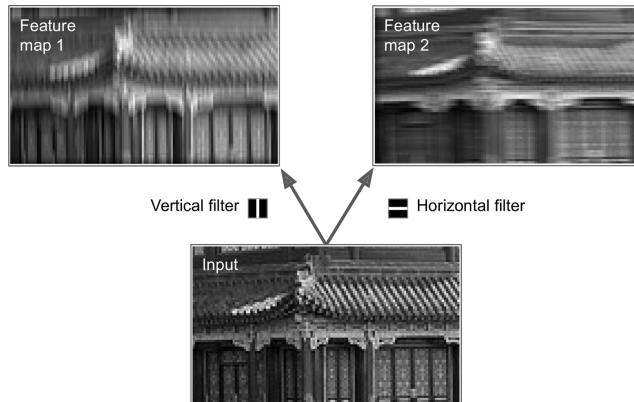


Figura 2.15: Esempio di applicazione di filtri diversi e generazione delle mappe delle caratteristiche corrispondenti.

Ovviamente, i filtri non devono essere definiti a mano ma la rete, durante l’addestramento, apprenderà da sola quali sono i filtri più utili per il compito da svolgere e come combinarli per ottenere pattern più complessi.

Ogni strato convoluzionale presenta più filtri e, di conseguenza, genererà altrettante mappe delle caratteristiche, perciò la rappresentazione più corretta dello strato sarebbe la seguente:

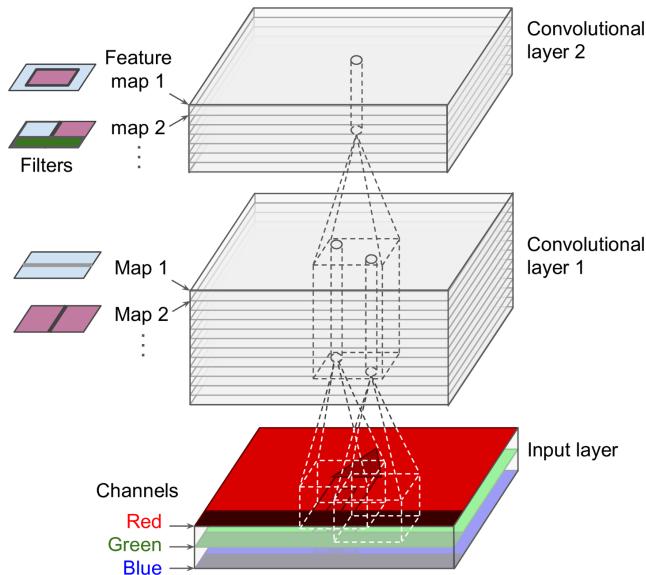


Figura 2.16: Rappresentazione in 3D di uno strato convoluzionale.

Strato di Pooling

Lo scopo dello strato di pooling è quello di restringere l'immagine di input in modo da ridurre il costo computazionale, l'occupazione di memoria e il numero di parametri (permettendo di limitare l'overfitting).

Proprio come negli strati convoluzionali, ogni neurone è connesso solo ad un numero limitato di neuroni dello strato precedente, tipicamente situati in un piccolo campo recettivo rettangolare. Tuttavia, i neuroni di uno strato di pooling non hanno pesi; tutto ciò che fanno è aggregare gli input utilizzando una funzione di aggregazione. I più utilizzati sono:

- **MaxPooling:** l'output è rappresentato dal valore massimo della matrice di partenza.
- **AveragePooling:** l'output è costituito dalla media dei valori della matrice di partenza.

Perciò, solo il valore restituito dalla funzione di aggregazione viene mantenuto, gli altri input vengono scartati.

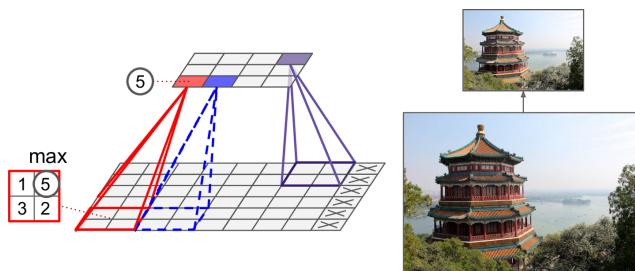


Figura 2.17: Rappresentazione di uno strato MaxPooling.

Architettura delle Reti Neurali Convoluzionali

Tipicamente, l'architettura di una rete neurale convoluzionale è costituita dall'alternarsi di strati convoluzionali e di pooling [11]. Il risultato di questa struttura è che l'immagine diventa sempre più piccola ma sempre più "profonda", cioè sarà costituita da numerose mappe di caratteristiche.

In cima alla pila di strati convoluzionali e di pooling viene poi aggiunta una rete neurale semplice composta da strati densamente connessi e lo strato finale restituirà il risultato della predizione.

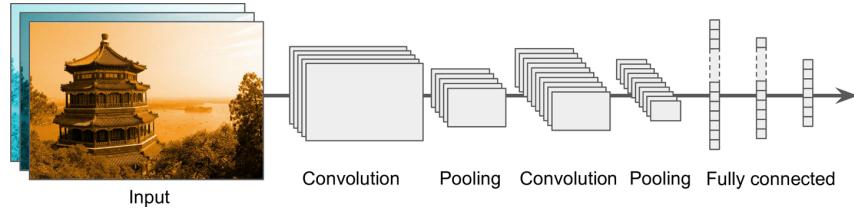


Figura 2.18: Architettura tipica di una rete neurale convoluzionale

L'introduzione degli strati convoluzionali e di pooling permette alla rete di lavorare anche con immagini di grandi dimensioni: una rete neurale profonda, con strati densamente connessi, richiederebbe infatti troppi parametri. Basti pensare che un'immagine di 100x100 pixel (che risulta comunque di piccole dimensioni) contiene 10000 pixel: se considerassimo un primo strato con solo 1000 neuroni, in una rete tradizionale otterremmo già 10 milioni di connessioni (soltanto per questo primo strato!). Le reti neurali convoluzionali risolvono questo problema usando strati parzialmente connessi e condivisione dei pesi.

Capitolo 3

Ambiente di lavoro e strumenti utilizzati

In questo capitolo verranno illustrati l'ambiente di lavoro e gli strumenti utilizzati per sviluppare le varie soluzioni testate.

3.1 Micro-controllori

Un micro-controllore è un dispositivo elettronico integrato su singolo circuito elettronico utilizzato generalmente in sistemi embedded, ovvero per applicazioni specifiche di controllo digitale.

Ad oggi, uno dei paradigmi di erogazione di servizi più utilizzati è il *cloud computing*. Questo si basa su diversi tipi di nodi:

- **nodi sensori**: sono sistemi che dispongono di risorse molto limitate e il loro compito è quello di raccogliere dati che verranno poi elaborati in cloud.
- **nodi edge**: si tratta di sistemi che hanno una disponibilità di risorse solitamente superiore a quella dei sensori e fungono da ponte di comunicazione tra questi ultimi e i nodi cloud. Fanno parte di questa categoria anche i micro-controllori.
- **nodi cloud**: sono sistemi con grande disponibilità di risorse ed eseguono computazioni sui dati ricevuti.

Questo paradigma presenta numerosi vantaggi, tra cui elevate prestazioni poiché i nodi cloud forniscono grande potenza di calcolo e spazio di archiviazione. Tuttavia, in contesti in cui si deve operare con bassa latenza, o addirittura in tempo reale, questa architettura presenta alcuni lati negativi. Infatti, il

passaggio di informazioni dai sensori, ai nodi edge ed infine ai nodi cloud richiede una grande capacità di rete e, soprattutto, è caratterizzato da una latenza introdotta dalla rete stessa.

Tutto ciò ha fatto sì che, in parallelo, si sviluppasse un nuovo paradigma di calcolo distribuito denominato **edge computing** caratterizzato dalla decentralizzazione delle risorse di calcolo. Questo dislocamento del calcolo dei dati prevede quindi l'utilizzo per la computazione dei nodi edge e si adatta perfettamente al contesto *Internet of Things*. Dato che la loro capacità computazionale è notevolmente aumentata e il loro prezzo è ad oggi molto accessibile, è possibile sfruttarli per eseguire anche modelli di intelligenza artificiale.

3.1.1 Nvidia Jetson Nano Developer Kit

La Nvidia Jetson Nano Developer Kit è una scheda di sviluppo di intelligenza artificiale che integra unità di calcolo volte all'accelerazione di algoritmi di machine learning e visione artificiale. Sebbene tale prodotto si rivolga prevalentemente al mercato della prototipazione, i suoi costi contenuti e la grande capacità computazionale, unite all'ottima efficienza energetica, la rendono appetibile anche in parecchie applicazioni commerciali che si basano su reti neurali profonde, specialmente nel settore dell'Internet of Things, come classificazione di immagini, riconoscimento di oggetti, processing multimediale, GPU computing e deep learning.

La Jetson Nano si basa su un modulo GPU Nvidia Maxwell a 128 core affiancato da una CPU quad-core ARM Cortex-A57 con parallelismo a 64 bit e frequenza di clock a 1,43 GHz, supportate da una RAM LPDDR4 da 4GB con bus a 64 bit a 1600 MHz. Il punto di forza di questa configurazione hardware, oltre al prezzo estremamente competitivo, è il consumo energetico stimato compreso tra i 5 e i 10 W grazie all'efficienza dei componenti scelti.

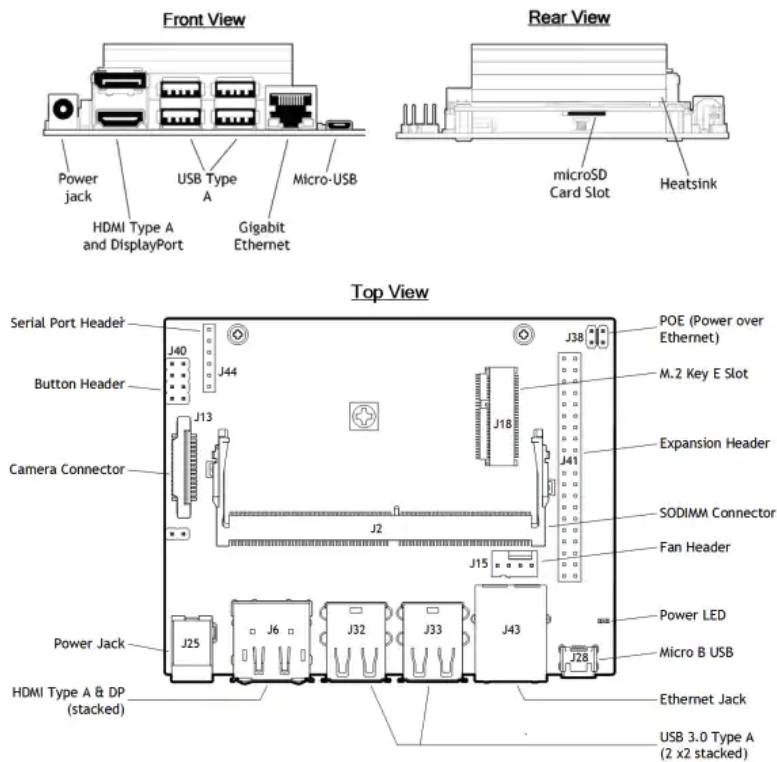


Figura 3.1: Schema Nvidia Jetson Nano Developer Kit

La Jetson Nano è supportata dal Nvidia JetPack SDK, un Software Development Kit sviluppato da Nvidia per tutti i suoi moduli della serie Jetson, che permette di gestire in maniera ottimale i prodotti dell'azienda, fornendo un software studiato per garantire le massime prestazioni nelle applicazioni di intelligenza artificiale sulla base dell'hardware adottato. Jetpack, oltre a fornire un sistema operativo, mette a disposizione librerie, esempi, strumenti di sviluppo e documentazione. Il sistema operativo è un derivato di Linux Ubuntu 18.04 in versione 64 bit con kernel 4.9, che fornisce automaticamente numerose librerie e driver, sufficienti per la maggior parte delle applicazioni.

3.1.2 Raspberry Pi 4 Model B

La Raspberry Pi 4 Model B è una scheda finalizzata all'esecuzione di sistemi operativi basati su kernel Linux e dal costo molto contenuto, risultando una delle soluzioni più diffuse nella prototipazione e nelle applicazioni a ridotto costo computazionale: caratteristiche che la rendono una scelta molto valida in svariati campi, incluso quello dell'Internet of Things. Nonostante non vi siano

acceleratori specificamente finalizzati all'incremento delle prestazioni di reti neurali profonde, l'impiego di questa scheda per svolgere compiti come la classificazione di immagini e il deep learning risulta essere comunque un'interessante ed economica soluzione.

Essendo la versione più aggiornata della scheda, la Raspberry Pi 4 Model B beneficia di un hardware attuale ed efficiente basato su una CPU Broadcom BCM2711 quad-core Cortex-A72 a 64 bit con una frequenza di clock di 1,5 GHz. La scheda è disponibile nei tagli di RAM da 1, 2, 4 e 8 GB LPDDR4 con bus a 64 bit e frequenza di 3200 MHz. Verrà utilizzata in questo caso la scheda con il massimo ammontare di memoria RAM pari a 8 GB. Pur non avendo una GPU dedicata, la scheda è provvista di acceleratori per la codifica e decodifica video (VPU) e supporto alle librerie grafiche OpenGL ES 3.0, garantendo una buona velocità ed efficienza nelle applicazioni multimediali.

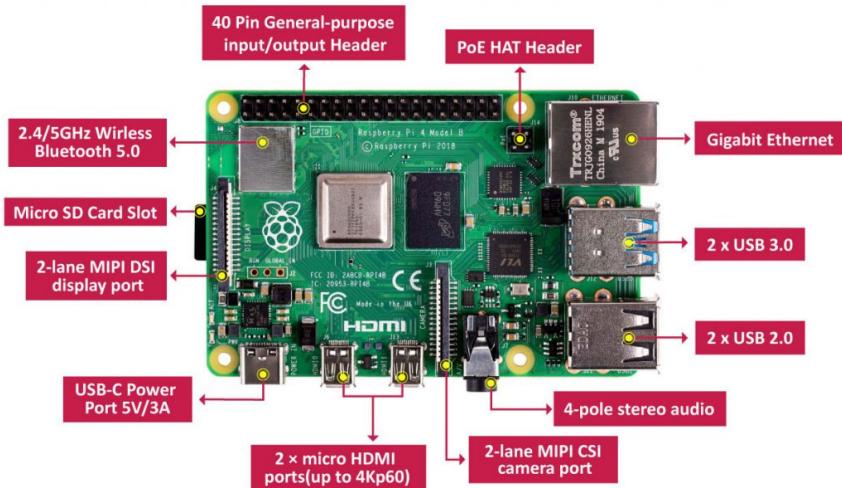


Figura 3.2: Schema Raspberry Pi 4 Model B

Grazie alla sua ampia diffusione e ai suoi diversi contesti di applicazione, la Raspberry Pi 4 dispone di diversi sistemi operativi di terze parti realizzati e ottimizzati appositamente per la scheda. Tuttavia il sistema operativo ufficiale e più conosciuto, nonché lo stesso diffuso da Raspberry Pi Foundation, è Raspberry Pi OS (chiamato precedentemente Raspbian). Questo deriva dalla distribuzione Linux Debian 10 Buster ottimizzato per architetture ARM.

3.2 Python

Il linguaggio utilizzato per lo sviluppo è Python. È un linguaggio interpretato ed estremamente flessibile. Inoltre, presenta numerose librerie che permettono

di gestire al meglio i dati e di interagire con i modelli di machine learning.

3.2.1 Librerie Python

Le librerie Python più utilizzate sono le seguenti:

- **Tensorflow**: è una libreria software open source per il machine learning che fornisce moduli sperimentali e ottimizzati, utili nella realizzazione di algoritmi per diversi tipi di compiti percettivi e di comprensione del linguaggio. La sua variante **Tensorflow Lite** permette di eseguire dei modelli di intelligenza artificiale estremamente leggeri e poco onerosi di risorse, mantenendo comunque una buona efficacia: questi sono perfetti per operare su dispositivi mobili e micro-controllori.
- **OpenCV**: è una libreria software multi-piattaforma nell'ambito della visione artificiale. Permette la lettura, elaborazione, conversione e visualizzazione delle immagini [12].
- **Numpy**: è una libreria open source che aggiunge supporto a grandi matrici e array multidimensionali insieme ad una vasta collezione di funzioni matematiche di alto livello per poter operare efficientemente su queste strutture dati [13].

3.3 Google Colaboratory

Colaboratory o, in breve, **Colab** è un prodotto di *Google Research*. Colab permette a chiunque di scrivere ed eseguire codice Python arbitrario tramite il browser ed è particolarmente adatto per machine learning, analisi dei dati e formazione. Più tecnicamente, è un servizio di blocchi note Jupyter ospitato, il cui utilizzo non richiede alcuna configurazione, che fornisce al contempo l'accesso gratuito alle risorse di calcolo, comprese le GPU e TPU.

Inizialmente, le soluzioni e i modelli sono stati testati su questa piattaforma per comodità e per facilità di condivisione.

Capitolo 4

Panoramica delle soluzioni testate

Di seguito sono descritte e discusse le varie soluzioni testate che hanno condotto infine alla soluzione finale del problema.

4.1 Addestrare una nuova rete

Il primo approccio al problema è stato quello di addestrare una rete neurale per svolgere il compito desiderato. Tuttavia, la fase di addestramento da zero di una rete neurale è un procedimento estremamente oneroso di risorse computazionali e di tempo. Inoltre, non è assolutamente garantito il raggiungimento di un risultato soddisfacente.

Per questo motivo si è scelto di iniziare l'addestramento basandosi su una rete neurale già esistente.

4.1.1 Transfer Learning

Il transfer learning è una tecnica che permette di addestrare una rete neurale sfruttando le conoscenze già acquisite da una rete esistente. Questo è possibile poiché, come già detto, la conoscenza di una rete neurale è strutturata in modo gerarchico: i primi strati estraggono delle caratteristiche di basso livello e più generali mentre gli ultimi strati acquisiscono delle informazioni più specifiche e dettagliate. È quindi possibile e consigliato riutilizzare per una nuova rete neurale i pesi già acquisiti da una rete che svolge un compito simile.

Per eseguire questa tecnica è necessario recuperare i primi strati della rete già esistente e renderli *non addestrabili*. In questo modo durante la fase di addestramento verranno modificati solo i parametri degli ultimi strati. Successivamente, è utile eseguire un ulteriore addestramento della rete rendendo però questa volta modificabili tutti i parametri. Questo permetterà di rendere la rete ancora più precisa.

4.1.2 Xception

La rete scelta per eseguire il transfer learning è la **Xception** [14]: si tratta di una rete convoluzionale già addestrata su uno dei più popolari dataset che è **ImageNet** [15] il quale contiene milioni di immagini di migliaia di categorie di oggetti. La rete è infatti in grado di riconoscere persone, animali, automobili, edifici e migliaia di altre categorie.

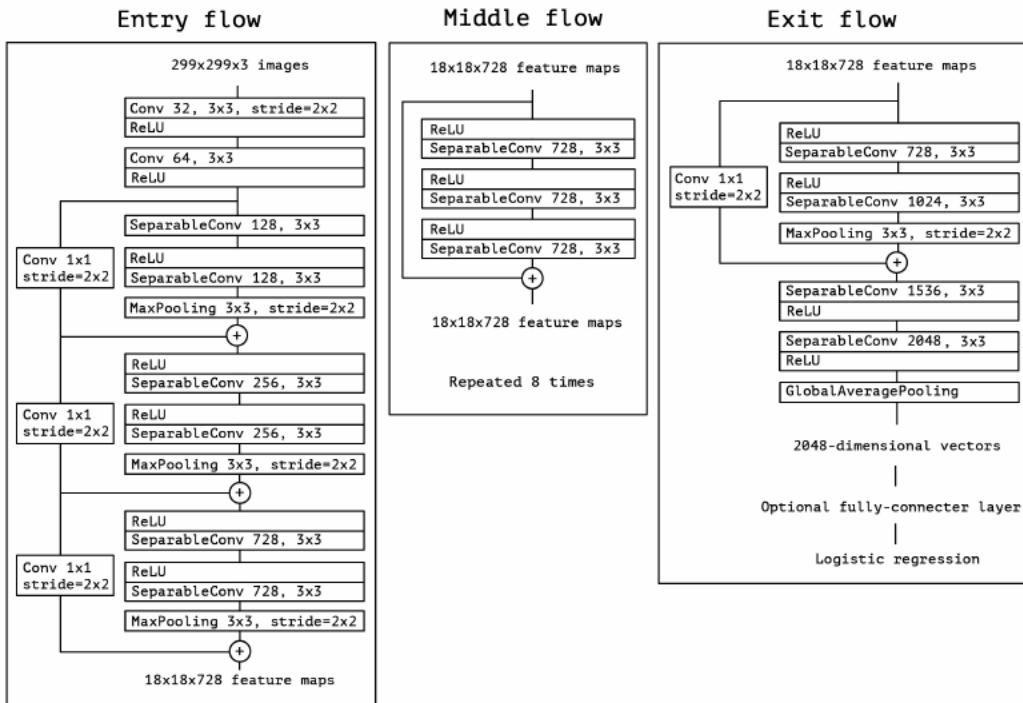


Figura 4.1: Architettura della rete Xception.

Come mostrato in figura 4.1, la sua architettura è caratterizzata prevalentemente da strati convoluzionali per l'estrazione di caratteristiche e strati di pooling per ridurre le dimensioni delle immagini. Sono presenti anche *connessioni residue* originariamente introdotte dalla rete **ResNet** [16]. Si tratta di una tecnica che consente di velocizzare notevolmente l'addestramento della rete e si esegue replicando il segnale in entrata di uno strato anche all'output di uno strato che si trova più avanti nella rete.

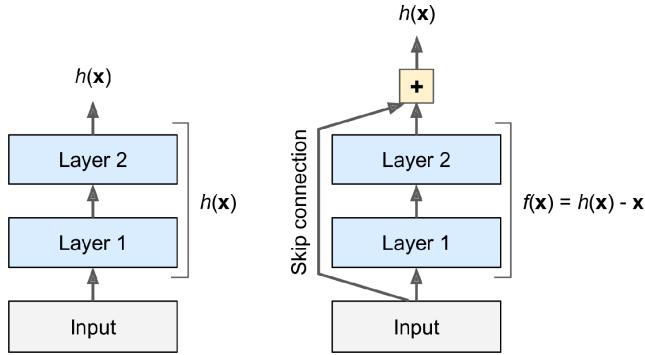


Figura 4.2: Differenza tra una normale rete (sinistra) e una rete con connessione residua (destra).

La rete Xception presenta inoltre un particolare tipo di strati convoluzionali: gli *strati convoluzionali separabili*. Mentre gli strati convoluzionali semplici utilizzano filtri che cercano di individuare allo stesso tempo dei pattern sia spaziali che su canali diversi, in quelli separabili si assume che questi due tipi di pattern possano essere modellati separatamente. Sono perciò costituiti da due parti: la prima applica un singolo filtro spaziale per ogni mappa delle caratteristiche in ingresso, poi la seconda cerca esclusivamente dei pattern tra canali diversi.

Questo tipo di strato richiede meno parametri, memoria e capacità computazionale ed è generalmente migliore dello strato tradizionale.

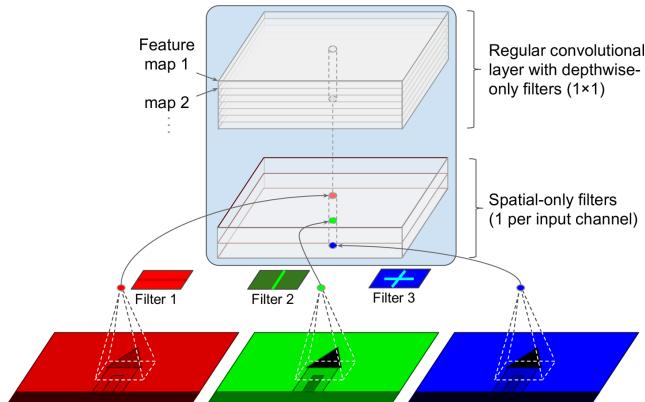


Figura 4.3: Rappresentazione di uno strato convoluzionale separabile.

4.1.3 Data Augmentation

Per l'addestramento della rete è stato utilizzato il dataset **RMFD** descritto in precedenza. Tuttavia, questo risultava fortemente sbilanciato poiché conteneva un numero di immagini di persone senza mascherina significativamente

maggiori di quelle di persone con mascherina. Tale caratteristica rende impossibile l'addestramento di una rete in grado di generalizzare al meglio i dati: basti pensare che la rete raggiungerebbe un'accuratezza molto alta semplicemente classificando tutte le immagini come appartenenti alla classe più numerosa.

Si è deciso quindi di eseguire un processo di **data augmentation**. Questo consiste nell'aumentare il numero di istanze della classe meno numerosa sfruttando quelle già a nostra disposizione: nel nostro caso sfrutteremo le immagini di persone con mascherina per crearne di nuove.

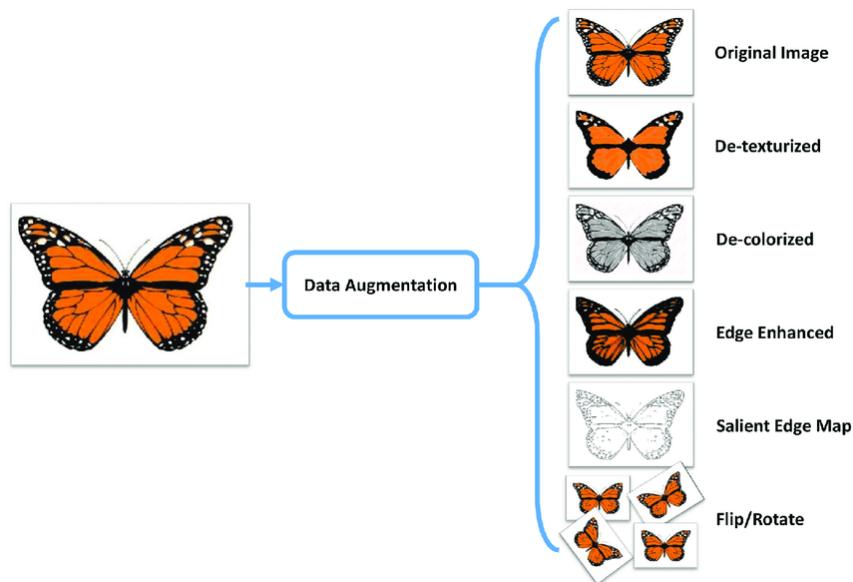


Figura 4.4: Esempio di data augmentation.

Come si può notare dalla figura 4.4, la data augmentation consiste nell'effettuare diverse elaborazioni sulla stessa immagine. Per citarne solo alcune:

- Rotazione dell'immagine
- Inversione dell'immagine sui piani orizzontale e/o verticale
- Ritaglio dell'immagine
- Applicazione di saturazione
- Modifica della luminosità

4.1.4 Fase di addestramento

L'addestramento è stato quindi effettuato utilizzando la tecnica del transfer learning a partire dalla rete Xception sfruttando il 70% del dataset RMFD (il

restante 30% è stato utilizzato come validation set) su cui era stata eseguita la data augmentation. Sono state utilizzate anche funzioni che hanno permesso di individuare gli iperparametri di addestramento migliori, ovvero configurazioni particolari che rendono l'apprendimento più efficace.

In totale, questa fase ha richiesto svariati giorni sulla piattaforma Colab.

4.1.5 Risultati ottenuti

I risultati ottenuti dalla rete al termine della fase di addestramento si sono rivelati insoddisfacenti poiché l'accuratezza raggiunta è stata intorno al 50%, ovvero la stessa che si otterrebbe da un classificatore casuale.

Questo tipo di approccio si è perciò dimostrato non efficace.

4.2 Modello di AIZOOTech

Avendo constatato che l'addestramento di una nuova rete neurale con transfer learning non ha prodotto i risultati sperati, si è deciso di cambiare totalmente approccio. È stata perciò individuata una rete preaddestrata [3] appositamente con lo scopo di individuare persone all'interno delle immagini e di stabilire se esse indossano o meno la mascherina medica.

La rete è stata implementata in diversi framework, tra cui Tensorflow e Tensorflow Lite. Il suo addestramento è stato effettuato sul dataset **FaceMask-Dataset** contenente un totale di circa 8000 immagini (il dataset è disponibile al link GitHub) già suddiviso in training set e validation set.

Rispetto alla soluzione precedente, è ora possibile individuare più persone all'interno dell'immagine e stabilirne le coordinate. Inoltre, sono già implementate alcune funzioni in grado di contornare i volti con colori diversi a seconda della classificazione eseguita.

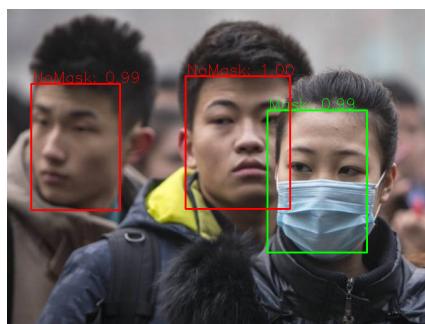


Figura 4.5: Dimostrazione del funzionamento della rete AIZOO.

4.2.1 Architettura della rete

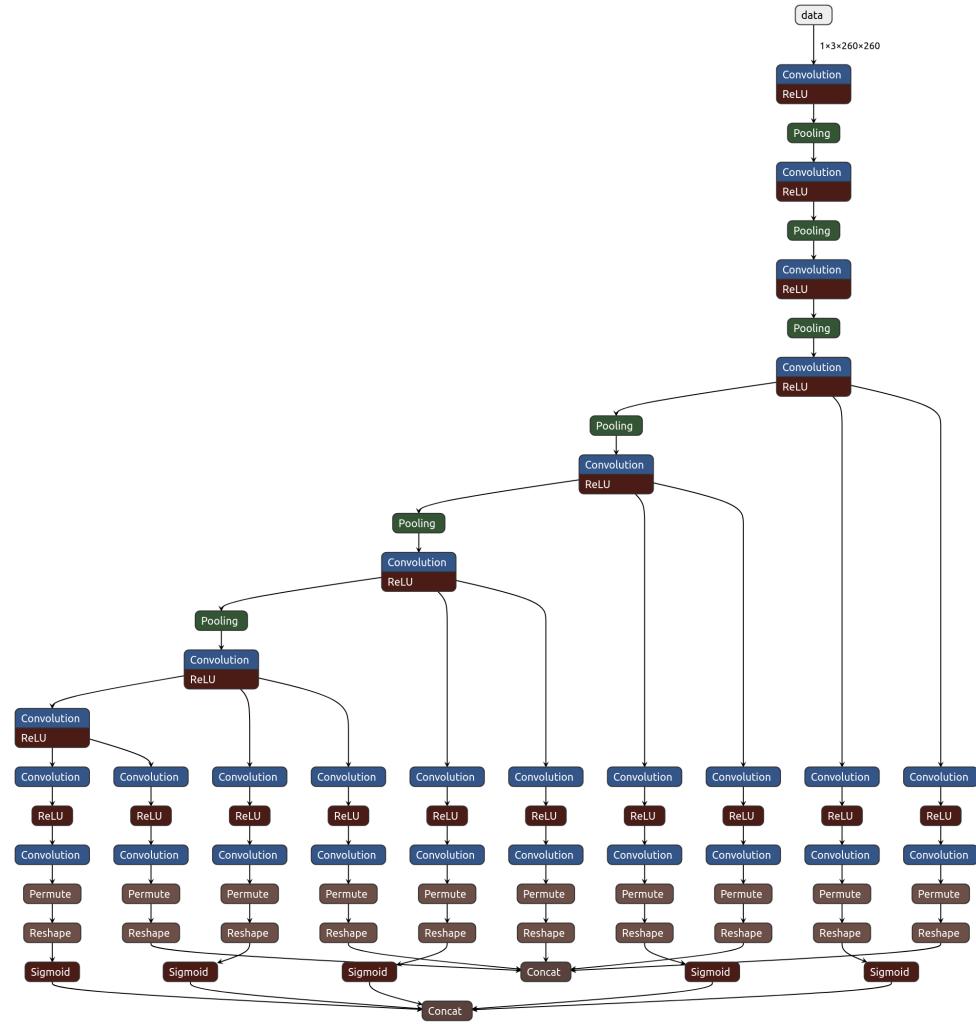


Figura 4.6: Architettura della rete AIZOO.

L’architettura della rete è costituita principalmente da strati convoluzionali per estrarre le caratteristiche dalle immagini e strati di pooling per ridurne le dimensioni. Come si può notare dalla figura 4.6 utilizza numerosi flussi separati per poi concatenarne nella parte finale gli output.

La rete lavora con immagini di dimensione 260x260 pixel e raggiunge un'accuratezza maggiore con codifica BGR al posto della classica RGB.

4.2.2 Funzionamento della soluzione

Le immagini che vogliamo classificare devono subire un pre-processamento per essere adattate all'input accettato dalla rete neurale. Questa fase consiste in:

- Conversione dell'immagine da codifica RGB a codifica BGR.
- Ridimensionamento dell'immagine a 260x260 pixel.
- Normalizzazione del valore dei pixel dal range 0-255 al range 0-1.

A questo punto l'immagine pre-processata sarà passata in input alla rete neurale che restituirà dei valori matriciali.

Infine, viene eseguito un post-processamento dei risultati ottenuti applicando alcune funzioni tipiche in visione artificiale tra le quali la **non-maximum suppression (NMS)** che permette di eliminare tutti i rilevamenti superflui mantenendo solo quelli più significativi.

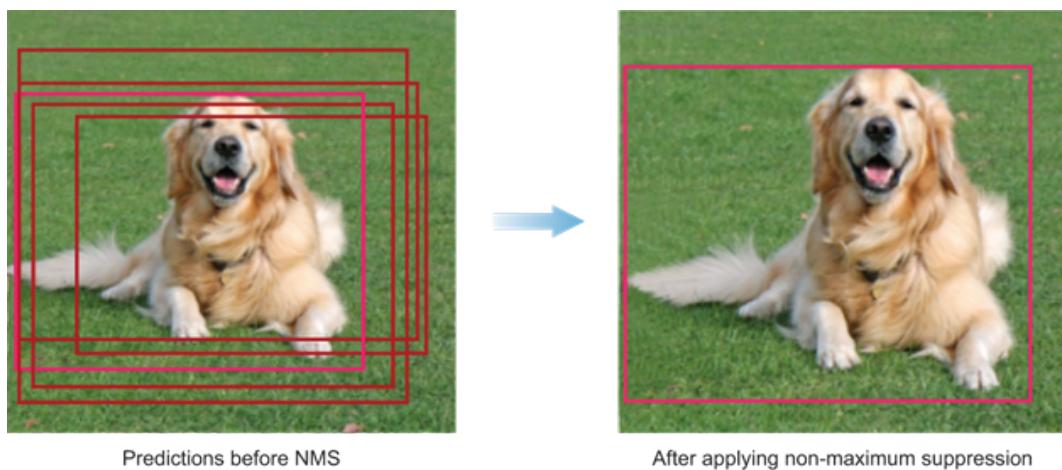


Figura 4.7: Applicazione della *non-maximum suppression*.

Il risultato finale contiene quindi le coordinate dei volti di tutte le persone individuate, la classe di appartenenza ("con mascherina" o "senza mascherina") e con quale probabilità è stata classificata tale (vedi figura 4.5).

4.2.3 Efficacia della soluzione

Il modello è stato testato sul validation set ottenuto dal dataset **FaceMaskDataset**, ovvero su immagini che la rete non aveva mai visto prima. Le metriche utilizzate per misurare l'efficacia sono:

- **Accuratezza:** indica la percentuale di predizioni corrette sul numero di predizioni totali. Il valore ottenuto è 94%.
- **Precisione:** indica la probabilità che l'avvenuta predizione di una classe eseguita dalla rete sia corretta. Sono stati ottenuti i seguenti risultati:

	Senza mascherina	Con mascherina
Precisione	96%	93%
Recall	92%	93%

Tabella 4.1: Precisione su FaceMaskDataset in Colab.

Si nota quindi che il modello è leggermente più preciso nel riconoscimento di persone senza mascherina.

- **Recall:** al contrario della precisione, indica la percentuale di elementi effettivamente appartenenti ad una classe che vengono riconosciuti come tali. I valori ottenuti sono:

	Senza mascherina	Con mascherina
Recall	92%	93%
F ₁ Score	94%	93%

Tabella 4.2: Recall su FaceMaskDataset in Colab.

Notiamo qui che la differenza tra le due classi è esigua.

- **F₁ Score:** è una misura di accuratezza che viene calcolata come media armonica della precisione e della recall. Si ottengono quindi:

	Senza mascherina	Con mascherina
F ₁ Score	94%	93%
Macro F ₁ Average	93.5%	

Tabella 4.3: F₁ Score su FaceMaskDataset in Colab.

Il modello è stato inoltre testato sul dataset **RMFD** e i risultati sono stati molto simili a quelli appena descritti.

4.2.4 Limiti della soluzione

La soluzione individuata risulta quindi efficace ma presenta dei limiti: il modello si comporta egregiamente su immagini relativamente semplici. Quando il sistema sarà utilizzato, dovrà essere in grado di riconoscere e classificare correttamente delle persone in situazioni più complesse: persone posizionate di profilo, lontane dall'inquadratura, condizioni di luce non ottimali e altre situazioni che non vengono correttamente gestite da questa soluzione.

4.2.5 Possibili soluzioni

Per migliorare la rete neurale si potrebbe procedere modificandone la sua struttura e/o riaddestrandola utilizzando anche immagini che contengono situazioni difficili. Tuttavia, per fare ciò sono necessarie buone conoscenze nell'ambito della progettazione delle reti e, soprattutto, il procedimento richiederebbe una significativa quantità di tempo. Inoltre, il conseguimento di un risultato soddisfacente non è garantito.

Capitolo 5

Analisi della soluzione definitiva

Il questo capitolo verrà analizzata la soluzione definitiva individuata per questo elaborato di tesi. La soluzione è disponibile su GitHub ed è pronta per essere eseguita sia sui calcolatori ordinari che su micro-controllori sfruttando una videocamera.

5.1 Funzionamento della soluzione

Il funzionamento della soluzione si basa su due fasi:

- **Rilevamento dei volti:** una rete neurale addestrata per il rilevamento dei volti individua le coordinate di questi all'interno dell'immagine e poi invia i riquadri che li contengono al successivo modulo del progetto.
- **Classificazione dei volti:** una volta ottenuti i riquadri con i volti, una seconda rete neurale addestrata per la classificazione in "con mascherina" e "senza mascherina" eseguirà l'inferenza per ognuno di essi e restituirà la classe di appartenenza.

La soluzione utilizza quindi due reti neurali allo stato dell'arte: ciò permette sia un migliore rilevamento dei volti che una migliore accuratezza nella classificazione delle istanze rilevate.

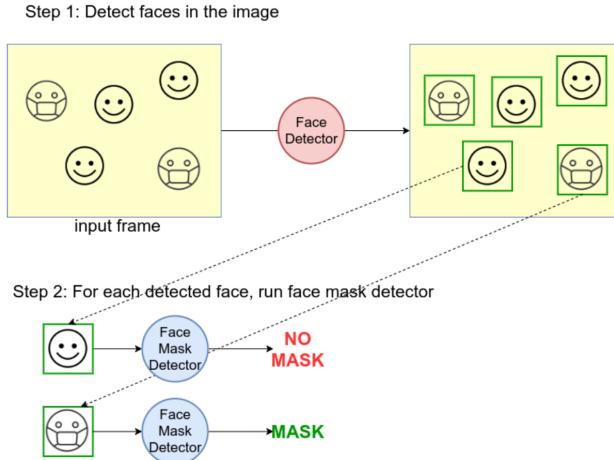


Figura 5.1: Struttura della soluzione finale.

5.2 Rilevamento dei volti

La rete utilizzata per il rilevamento dei volti è **Dual Shot Face Detector (DSFD)** [17] [18]. Si tratta di una rete allo stato dell’arte che presenta un’architettura particolare: è infatti costituita, come suggerisce il nome, da due flussi separati. In particolare, l’immagine viene all’inizio analizzata dalla prima catena di rilevamento costituita da strati convoluzionali; le mappe delle caratteristiche generate dagli strati vengono poi trasferite alla seconda catena mediante un modulo, denominato "intensificatore di caratteristiche" dove verranno nuovamente analizzate. Vi sono quindi una fase di analisi delle caratteristiche originali ed una di analisi delle caratteristiche intensificate come mostrato in figura 5.2.

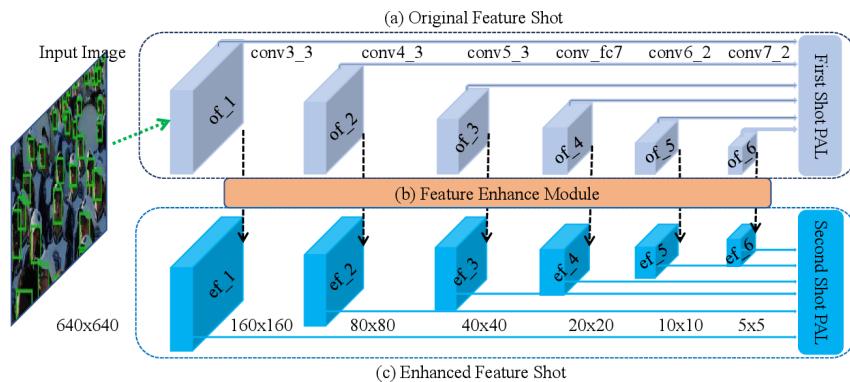


Figura 5.2: Architettura della rete Dual Shot Face Detector.

Tale rete è stata addestrata sfruttando un training set proveniente da **Wider Face** [19], ovvero un popolare dataset per il rilevamento dei volti, e successivamente testata su un test set ricavato dallo stesso dataset (ovviamente su immagini diverse da quelle utilizzate in fase di addestramento). I risultati ottenuti hanno confermato la validità della soluzione e la sua superiorità rispetto alle altre proposte allo stato dell'arte.

5.3 Classificazione dei volti

Per la classificazione dei volti nelle due classi "con mascherina" e "senza mascherina", è stata utilizzata una particolare rete [20] creata con transfer learning sfruttando l'architettura **MobileNet V2** [21]. Si tratta di un modello pensato per essere eseguito su dispositivi con limitate risorse computazionali grazie alla presenza di un numero ridotto di parametri.

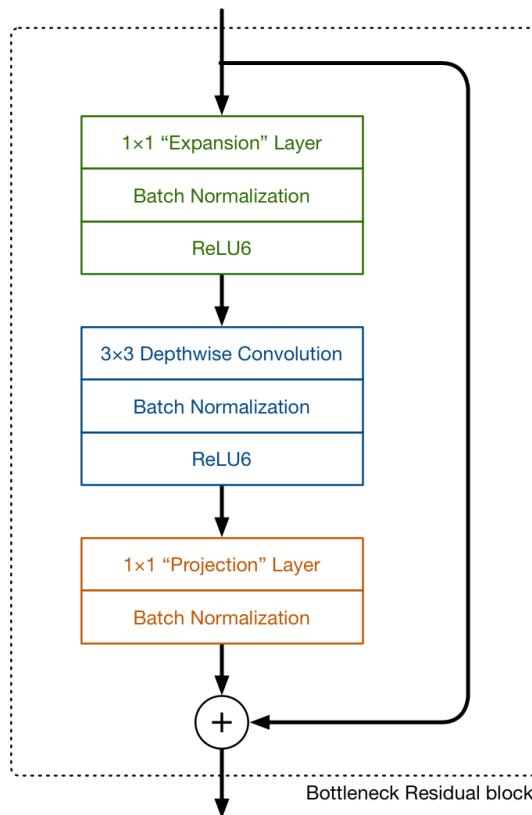


Figura 5.3: Singolo blocco di strati della rete MobileNet.

Come mostra la figura 5.3, i principali blocchi costitutivi della rete sono caratterizzati da tre strati convoluzionali:

- **Strato di espansione:** strato convoluzionale che permette di aumentare il numero di canali dei dati in ingresso prima che arrivino alla convoluzione separabile.
- **Strato di convoluzione separabile:** strato che, come già descritto in precedenza, ricerca prima dei pattern su ogni canale in ingresso e successivamente pattern tra canali diversi.
- **Strato di proiezione:** strato convoluzionale che riduce considerevolmente il numero di canali dei dati in ingresso.

Questa particolare struttura fa sì che i dati in ingresso e uscita di ogni blocco siano tensori con poche dimensioni mentre la fase di estrazione delle caratteristiche, che avviene al suo interno, sia eseguita su tensori ad alta dimensionalità.

Altri componenti del blocco sono:

- **Batch Normalization** [22]: esegue una normalizzazione dei dati in ingresso. Viene utilizzata per rendere più stabile l'apprendimento della rete ed evitare lo svanire e l'esplosione del gradiente durante la retro-propagazione.
- **ReLU6**: si tratta di una particolare versione della funzione di attivazione ReLu. Nel dettaglio, limita il valore dell'attivazione a 6.

$$\sigma(x) = \min(\max(0, x), 6)$$

Da notare inoltre l'utilizzo nel blocco di una connessione residua.

5.4 Efficacia della soluzione

La soluzione è stata testata sullo stesso dataset utilizzato per la precedente soluzione, ovvero **FaceMaskDataset**.

Sono state inoltre introdotte alcune misure di efficacia che aderiscono a standard internazionali:

- **Hamming Loss:** è la percentuale di istanze che vengono classificate in modo *non* corretto (in questo caso $1 - Accuratezza$).
- **Macro F_1 Average:** è la media aritmetica dell' F_1 Score per le singole classi.
- **Exact Match:** è la percentuale di immagini in cui tutte le persone vengono rilevate e correttamente classificate.

	Senza mascherina	Con mascherina
Precisione	95%	88%
Recall	93%	91%
F_1 Score	94%	90%
Accuratezza	92%	
Hamming Loss	8%	
Macro F_1 Average	92%	
Exact Match	73%	

Tabella 5.1: Misure di accuratezza su FaceMaskDataset in Colab.

A primo impatto pare che la nuova soluzione non porti dei tangibili miglioramenti. Tuttavia, possiamo apprezzare le capacità di questo modello se applicato al dataset **HardDataset**, ovvero contenente immagini con condizioni non ottimali:

	Senza mascherina	Con mascherina
Precisione	77% (vs 47%)	90% (vs 89%)
Recall	83% (vs 88%)	86% (vs 49%)
F_1 Score	80% (vs 62%)	88% (vs 63%)
Accuratezza	85% (vs 62%)	
Hamming Loss	15%	
Macro F_1 Average	84%	
Exact Match	23%	

Tabella 5.2: Misure di accuratezza su HardDaset in Colab.

5.4.1 Efficacia per numero di persone

Di seguito è riportata l'efficacia della soluzione all'aumentare del numero di persone all'interno delle immagini.

1 persona

	Senza mascherina	Con mascherina
Precisione	94%	92%
Recall	92%	94%
F_1 Score	93%	92%
Accuratezza	93%	
Hamming Loss		7%
Macro F_1 Average		92.5%
Exact Match		84%

Tabella 5.3: Misure di accuratezza con una persona per immagine.

2 persone

	Senza mascherina	Con mascherina
Precisione	93%	83%
Recall	92%	84%
F_1 Score	92%	84%
Accuratezza	89%	
Hamming Loss		11%
Macro F_1 Average		88%
Exact Match		56%

Tabella 5.4: Misure di accuratezza con due persone per immagine.

3 persone

	Senza mascherina	Con mascherina
Precisione	93%	76%
Recall	91%	81%
F_1 Score	92%	78%
Accuratezza	89%	
Hamming Loss	11%	
Macro F_1 Average	85%	
Exact Match	28%	

Tabella 5.5: Misure di accuratezza con tre persone per immagine.

4 persone

	Senza mascherina	Con mascherina
Precisione	97%	85%
Recall	94%	93%
F_1 Score	96%	89%
Accuratezza	94%	
Hamming Loss	6%	
Macro F_1 Average	92.5%	
Exact Match	33%	

Tabella 5.6: Misure di accuratezza con quattro persone per immagine.

Non sono qui riportate le misure ricavate da immagini contenenti più persone poiché le classi non venivano adeguatamente rappresentate: vi sono infatti troppe poche immagini contenenti più di 5 persone in cui ne sia presente almeno una con mascherina. Ciò renderebbe qualsiasi misura non significativa.

Capitolo 6

Deep Learning su micro-controllori

Si procede con la descrizione del framework utilizzato per l'implementazione della soluzione su micro-controllori. Difatti, con l'aumento di prestazioni di tali dispositivi e la riduzione dei costi, il loro impiego per progetti di deep learning risulta sempre più appetibile. Invero, sono andati sviluppandosi svariati framework e librerie con l'obiettivo di facilitare questi compiti.

6.1 Tensorflow Lite

Tensorflow Lite [23] è un framework per deep learning open-source e multi-piattaforma che converte modelli Tensorflow pre-addestrati in uno speciale formato (**.tflite**) ottimizzato in termini di prestazioni e spazio occupato. I modelli .tflite possono essere eseguiti su nodi edge come dispositivi mobili o micro-controllori.

6.1.1 Vantaggi di Tensorflow Lite

- **Leggerezza:** i nodi edge dispongono di risorse limitate in termini di spazio di archiviazione e capacità computazionale perciò i modelli devono essere leggeri.
- **Bassa latenza:** dato che le inferenze vengono effettuate direttamente sul dispositivo, non è necessario contattare un server per cui si eliminano le latenze derivanti dalla rete.
- **Sicurezza:** di nuovo, le inferenze avvengono sul dispositivo perciò non vi sono dati che fuoriescono da esso. Ciò rende il framework sicuro.
- **Bassi consumi:** i nodi edge potrebbero avere un'autonomia limitata quindi l'utilizzo dei modelli deve richiedere poca energia.

6.1.2 Funzionamento di Tensorflow Lite

L'utilizzo del framework Tensorflow Lite prevede i seguenti passaggi:

- Scelta di un modello: è possibile utilizzare un modello preaddestrato tra quelli resi disponibili dalla piattaforma, riaddestrarne uno utilizzando transfer learning, addestrare un modello personalizzato o addirittura convertire una rete in formato .tflite.
- Esecuzione dell'inferenza: Tensorflow Lite utilizza una particolare libreria, denominata **interprete**, per eseguire le predizioni.

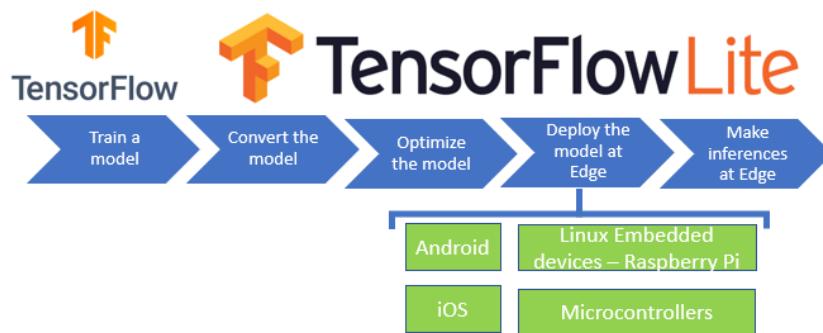


Figura 6.1: Funzionamento del framework Tensorflow Lite.

6.2 Descrizione dell'applicazione

L'applicazione, sviluppata in codice Python e adatta al funzionamento su micro-controllori (ma anche su macchine ordinarie), consente di rilevare e contornare i volti delle persone inquadrati dalla videocamera e stabilire se queste indossano la mascherina oppure no: in caso positivo, il riquadro sarà di colore verde, altrimenti rosso. La soluzione è ovviamente stata progettata per funzionare in tempo reale e ciò la rende adatta a situazioni di monitoraggio. Potrebbe essere sfruttata per automatizzare il controllo all'ingresso di un edificio o la situazione al suo interno e quindi tutelare chi prima svolgeva tale compito di persona. L'interfaccia consente inoltre di ottenere informazioni come il tempo di inferenza e il numero di fotogrammi analizzati al secondo.

6.2.1 Funzionamento dell'applicazione

Per il corretto funzionamento dell'applicazione è necessario disporre di una videocamera collegata alla propria macchina (micro-controllore o altro) e delle librerie Python necessarie: la lista è riportata in seguito. È sufficiente poi avviare l'applicazione utilizzando l'interprete Python nella versione 3 (*python3*) e inquadrare le persone di interesse. Possono ovviamente essere inquadrati più persone contemporaneamente, ferme o in movimento. Di ognuna sarà contornato il viso in verde nel caso questa indossi la mascherina, altrimenti in rosso.

Inoltre, viene eseguita anche una stima della distanza del volto dalla videocamera. Tale funzionalità è ottenuta considerando la dimensione della regione di interesse (il rettangolo che contiene il volto). Infatti, vi è un rapporto di proporzionalità inversa tra la distanza dall'inquadratura e l'area di tale regione: all'aumentare della distanza del volto dalla videocamera diminuisce la dimensione del rettangolo, e viceversa. Fornendo quindi alcuni valori di riferimento (coppie distanza-area), è possibile eseguire tale stima di distanza, non esatta ma comunque accettabile, a partire dall'area rilevata. Il valore ottenuto è mostrato al di sotto del contorno del volto.

6.3 Test su micro-controllori

La soluzione è quindi stata testata sui micro-controllori precedentemente descritti e di seguito sono riportati i valori di accuratezza e le prestazioni.

6.3.1 Raspberry Pi 4 Model B 8GB

I valori di accuratezza ottenuti risultano del tutto in linea con quelli già ricavati dai precedenti test su piattaforma Colab.

Accuratezza FaceMaskDataset

	Senza mascherina	Con mascherina
Precisione	95%	88%
Recall	93%	91%
F_1 Score	94%	90%
Accuratezza		92%
Hamming Loss		8%
Macro F_1 Average		92%
Exact Match		73%

Tabella 6.1: Misure di accuratezza su FaceMaskDataset con Raspberry Pi.

Accuratezza HardDataset

	Senza mascherina	Con mascherina
Precisione	76%	90%
Recall	83%	86%
F_1 Score	79%	88%
Accuratezza		85%
Hamming Loss		15%
Macro F_1 Average		83.5%
Exact Match		23%

Tabella 6.2: Misure di accuratezza su HardDataset con Raspberry Pi.

6.3.2 Nvidia Jetson Nano

Anche in questo caso i risultati ottenuti sono simili ai precedenti; possiamo apprezzare addirittura un lieve miglioramento.

Accuratezza FaceMaskDataset

	Senza mascherina	Con mascherina
Precisione	94%	92%
Recall	92%	94%
F_1 Score	93%	93%
Accuratezza	93%	
Hamming Loss		7%
Macro F_1 Average		93%
Exact Match		73%

Tabella 6.3: Misure di accuratezza su FaceMaskDataset con Jetson Nano.

Accuratezza HardDataset

	Senza mascherina	Con mascherina
Precisione	77%	90%
Recall	83%	86%
F_1 Score	80%	88%
Accuratezza	85%	
Hamming Loss		15%
Macro F_1 Average		84%
Exact Match		23%

Tabella 6.4: Misure di accuratezza su HardDataset con Jetson Nano.

6.3.3 Prestazioni

Di seguito sono riportate le prestazioni riscontrate per la soluzione:

	Jetson Nano	Raspberry Pi
Tempo inferenza	0.150	0.270
Frame al secondo	5.5	3

Tabella 6.5: Confronto prestazioni Jetson Nano e Raspberry Pi

Notiamo subito che Jetson Nano raggiunge delle prestazioni significativamente migliori: quasi il doppio rispetto alla sua concorrente, che comunque rimangono rispettabili dato l'utilizzo di due reti neurali. Tale risultato si può sicuramente ricondurre alle componenti migliori che la casa Nvidia ha riservato per il proprio micro-controllore.

Tuttavia, il framework Tensorflow Lite non è in grado di utilizzare la GPU Nvidia di cui è dotata la scheda Jetson Nano. Ciò significa che, nonostante la già affermata supremazia, non viene ancora sfruttato l'intero potenziale di tale dispositivo.

Si è perciò deciso di implementare la soluzione avvalendosi del framework Tensorflow. Quest'ultimo permetterà infatti di demandare del carico computazionale alla GPU in dotazione, a discapito però di una minore ottimizzazione dei modelli. Sono di seguito riportate le prestazioni ottenute da Jetson Nano con framework Tensorflow:

	TF Lite	TF + GPU	TF No GPU
Tempo inferenza	0.150	0.250	0.300
Frame al secondo	5.5	3.5	2

Tabella 6.6: Confronto prestazioni su Jetson Nano con framwork Tensorflow e Tensorflow Lite

Purtroppo l'utilizzo della GPU non permette di ottenere un miglioramento delle prestazioni. La spiegazione di questo fenomeno è da ricercare nella pesantezza e minore ottimizzazione dei modelli che il framework Tensorflow utilizza. Ciò fa sì che, sebbene si abbia a disposizione una maggiore capacità computazionale, le prestazioni comunque ne risentano. Di seguito sono riportate le dimensioni dei modelli utilizzati:

	Tensorflow Lite	Tensorflow
Rilevamento volti	580KB	6.1MB
Classificatore (Face-Mask)	10.2MB	11.5MB

Tabella 6.7: Confronto dimensioni modelli Tensorflow e Tensorflow Lite

È facile notare come, a parità di modelli utilizzati e numero di parametri dei modelli stessi, l'ottimizzazione del framework Tensorflow Lite permetta di ridurre la dimensione dei modelli (che diventa estremamente rilevante soprattutto per quanto riguarda la rete neurale per il rilevamento dei volti).

6.3.4 Nvidia Face Mask Detection

Una soluzione degna di nota è sicuramente quella fornita dall'azienda Nvidia stessa [24]. Durante la fase di test della soluzione definitiva, Nvidia ha infatti rilasciato la sua versione del sistema per il rilevamento della mascherina sanitaria facciale. Questa risulta particolarmente ottimizzata per i suoi micro-controllori, tra cui proprio Jetson Nano. Tale soluzione non fornisce un modello pre-addestrato. Tuttavia, fornisce le istruzioni per addestrare autonomamente una rete neurale *DetectNet V2* basata su *ResNet18*. Viene perciò applicata la tecnica del Transfer Learning già descritta in precedenza. In questo modo, vengono sfruttate le conoscenze già acquisite dalla rete *ResNet18* per addestrare il nuovo modello *DetectNet V2*. Per effettuare l'addestramento, Nvidia mette a disposizione diversi dataset.

I risultati riportati sono i seguenti:

Pruned	mAP (Mask/No-Mask) (%)	FPS (GPU)
No	86.12 (87.59, 84.65)	6.5
Yes (12%)	85.50 (86.72, 84.27)	21.25

Tabella 6.8: Risultati e prestazioni della soluzione Nvidia.

Capitolo 7

Il codice prodotto

In questo capitolo viene infine mostrato e discusso il codice dell'elaborato di tesi. Il codice è stato scritto in linguaggio Python.

7.1 Il codice della soluzione

Dapprima vengono importate le librerie necessarie:

```
import cv2
import tensorflow._runtime.interpreter as tflite
import numpy as np
import time
```

In particolare:

- **cv2**: libreria OpenCV per l'elaborazione delle immagini.
- **tflite _ runtime.interpreter**: interprete Tensorflow Lite per eseguire le inferenze.
- **numpy**: per gestire in modo più agevole vettori e matrici.
- **time**: per eseguire la misurazione delle prestazioni.

Successivamente, la soluzione è stata organizzata sotto forma di classe. Sono riportati di seguito la dichiarazione della stessa e il metodo costruttore:

```
class FaceMaskDetector:
    """
    Class representing the face mask detector.
    It contains all the fields and the methods that make it possible
        to make the detections.
    """

    def __init__(self):
        self.color_map = {
            0: (0, 255, 0), # Green
            1: (255, 0, 0) # Red
        }
        self.class_map = {
            0: "Mask",
            1: "No Mask"
        }

        # Definition of the Face Detection interpreter.
        self.face_target_shape = (320, 320)
        self.face_interpreter =
            tflite.Interpreter(model_path="models/detector.tflite")
        self.face_interpreter.allocate_tensors()
        self.face_input_details =
            self.face_interpreter.get_input_details()
        self.face_output_details =
            self.face_interpreter.get_output_details()

        # Definition of the Mask Detection Interpreter.
        self.mask_target_shape = (224, 224)
        self.mask_interpreter =
            tflite.Interpreter(model_path="models/mask_detector.tflite")
        self.mask_interpreter.allocate_tensors()
        self.mask_input_details =
            self.mask_interpreter.get_input_details()
        self.mask_output_details =
            self.mask_interpreter.get_output_details()
```

In questo modo vengono definite alcune proprietà della classe:

- **color_map**: dizionario che associa alla classe predetta il colore corrispondente (verde o rosso).

- **class_map**: dizionario che associa alla classe predetta l'etichetta corrispondente.
- Proprietà dei due interpreti (uno per il rilevamento dei volti e l'altro per la loro classificazione):
 - Dimensione dell'immagine in input.
 - Percorso del modello.
 - Definizione della struttura dei valori di ingresso e di uscita del modello.

Seguono poi alcune funzioni di utilità:

- **preprocess**: adatta le immagini ricavate dalla fotocamera in modo che possano essere utilizzate dal modello per il rilevamento dei volti.
- **py_nms**: applica la non-maximum suppression in modo da eliminare i rilevamenti in eccesso.

Una volta applicate le trasformazioni sulle immagini in ingresso viene invocato l'interprete per eseguire le inferenze (rilevamento volti).

```
self.face_interpreter.set_tensor(self.face_input_details[0] ["index"],
    image_for_net)
self.face_interpreter.invoke()
bboxes = self.face_interpreter
    .get_tensor(self.face_output_details[0] ["index"])
```

In questo modo vengono restituite le coordinate dei riquadri contenenti i volti rilevati.

Successivamente, per ognuno di questi viene invocato in modo analogo l'interprete per eseguire la classificazione.

```
self.mask_interpreter.set_tensor(self.mask_input_details[0] ["index"],
    image_tensor)
self.mask_interpreter.invoke()
probabilities = self.mask_interpreter
    .get_tensor(self.mask_output_details[0] ["index"])
```

Vengono così restituite le classi predette con relative probabilità.

Infine, è riportato di seguito il corpo principale della soluzione: il ciclo che permette di ottenere immagini dalla fotocamera ed eseguire le inferenze.

```
while rval:  
    # Lettura di un frame dalla videocamera  
    rval, frame = vc.read()  
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)  
  
    inference_start = time.time()  
    # Rilevamento volti  
    boxes = face_mask_detector.detect_faces(frame)  
    # Classificazione volti  
    frame = face_mask_detector.detect_masks(frame, boxes)  
    inference_stop = time.time()
```

Conclusioni e sviluppi futuri

L'obiettivo iniziale consisteva nello sviluppo di un sistema in grado di determinare se le persone indossassero o meno una mascherina sanitaria facciale.

Dopo aver sperimentato diversi approcci e confrontato i rispettivi risultati, si è giunti alla conclusione che la soluzione costituita dalle reti *Dual Shot Face Detector* (per il rilevamento dei volti) e *MobileNet V2* (per la loro classificazione) si dimostra essere la migliore in termini di accuratezza. Questa soluzione garantisce delle prestazioni tali da poter essere sfruttata anche nel campo dei micro-controllori. Inoltre, l'analisi delle prestazioni, come prevedibile, ha evidenziato la superiorità del micro-controllore Jetson Nano, che tuttavia non ha soddisfatto al pieno le aspettative, posto che la presenza di una GPU avrebbe dovuto ulteriormente migliorarne le prestazioni.

La soluzione individuata è stata adattata al funzionamento su pc Desktop e su micro-controllori e permette, sfruttando una tradizionale webcam, di eseguire delle predizioni in tempo reale. Ciò la rende adatta ad essere utilizzata in vari contesti sostituendosi all'uomo.

Questo lavoro si presta a svariati sviluppi e tra i tanti ad un sistema per il controllo del distanziamento sociale. Ciò permetterebbe di stimare il rischio di contagio basandosi sulla distanza tra le persone e l'utilizzo della mascherina.

Un futuro in cui la macchina si potrà affiancare o addirittura sostituire all'uomo, diventando di fatto "i suoi occhi", rappresenta uno scenario suggestivo. E diventa tanto più stimolante contribuire, in un periodo come quello che ci occupa, ad addivenire a soluzioni con fini socialmente utili. Ciò rende questa disciplina ancor più affascinante di quanto lo sia già.

Ringraziamenti

Il primo ringraziamento va al relatore di questo lavoro, il Prof. Gianluca Moro, che ha reso possibile tutto ciò e ha acceso il mio personale interesse nei confronti di questa splendida disciplina.

Grazie alla mia famiglia, che mi ha sostenuto, spinto a dare il massimo e mi è sempre stata vicina.

Ringrazio inoltre amici e colleghi che mi hanno accompagnato durante il percorso di studi e hanno reso questa esperienza unica.

Bibliografia

- [1] Nancy Leung, Daniel Chu, Eunice Shiu, Kwok-Hung Chan, James Mcdevitt, Ben Hau, Hui-Ling Yen, Yuguo Li, Dennis Ip, Joseph S Peiris, Wing-Hong Seto, Gabriel Leung, Donald Milton, and Benjamin Cowling. Respiratory virus shedding in exhaled breath and efficacy of face masks. *Nature Medicine*, 26, 05 2020.
- [2] Real world masked face dataset. <https://github.com/X-zhangyang/Real-World-Masked-Face-Dataset>.
- [3] Aizoootech - face mask detection. <https://github.com/AIZOOTech/FaceMaskDetection>.
- [4] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.
- [5] Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 649–657, 2015.
- [6] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, June 16-21, 2012*, pages 3642–3649, 2012.
- [7] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. JMLR Workshop and Conference Proceedings.
- [8] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart and James L. McClelland, editors, *Parallel Distributed Processing: Explorations*

- in the Microstructure of Cognition, Volume 1: Foundations*, pages 318–362. MIT Press, Cambridge, MA, 1986.
- [9] David H. Hubel and Torsten N. Wiesel. Receptive fields of single neurons in the cat’s striate cortex. *Journal of Physiology*, 148:574–591, 1959.
 - [10] David H. Hubel and Torsten N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology (London)*, 195:215–243, 1968.
 - [11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
 - [12] G. Bradski. The OpenCV Library. *Dr. Dobb’s Journal of Software Tools*, 2000.
 - [13] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Comput. Sci. Eng.*, 13(2):22–30, 2011.
 - [14] François Chollet. Xception: Deep learning with depthwise separable convolutions, 2016. cite arxiv:1610.02357.
 - [15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.
 - [16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. cite arxiv:1512.03385Comment: Tech report.
 - [17] Jian Li, Yabiao Wang, Changan Wang, Ying Tai, Jianjun Qian, Jian Yang, Chengjie Wang, Ji-Lin Li, and Feiyue Huang. Dsfd: Dual shot face detector. *CoRR*, abs/1810.10220, 2018.
 - [18] Dual shot face detection - tensorflow. <https://github.com/610265158/DSFD-tensorflow/tree/tf2>.
 - [19] Shuo Yang, Ping Luo, Chen Change Loy, and Xiaoou Tang. Wider face: A face detection benchmark. *CoRR*, abs/1511.06523, 2015.
 - [20] Face mask classifier. <https://github.com/estebanuri/facemaskdetector/tree/master/android>.

- [21] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, June 2018.
- [22] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015. cite arxiv:1502.03167.
- [23] Tensorflow lite. <https://www.tensorflow.org/lite>.
- [24] Nvidia-ai-iot - face mask detection. <https://github.com/NVIDIA-AI-IOT/face-mask-detection>.

X10 Language Specification

Version 2.6.2

Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove

Please send comments to x10-core@lists.sourceforge.net

January 4, 2019

This report provides a description of the programming language X10. X10 is a class-based object-oriented programming language designed for high-performance, high-productivity computing on high-end computers supporting $\approx 10^5$ hardware threads and $\approx 10^{15}$ operations per second.

X10 is based on state-of-the-art object-oriented programming languages and deviates from them only as necessary to support its design goals. The language is intended to have a simple and clear semantics and be readily accessible to mainstream OO programmers. It is intended to support a wide variety of concurrent programming idioms.

The X10 design team consists of David Grove, Ben Herta, Louis Mandel, Josh Milthorpe, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, Olivier Tardieu.

Past members include Shivali Agarwal, Bowen Alpern, David Bacon, Raj Barik, Ganesh Bikshandi, Bob Blainey, Bard Bloom, Philippe Charles, Perry Cheng, David Cunningham, Christopher Donawa, Julian Dolby, Kemal Ebcioğlu, Stephen Fink, Robert Fuhrer, Patrick Gallop, Christian Grothoff, Hiroshi Horii, Kiyokuni Kawachiya, Allan Kielstra, Sreedhar Kodali, Sriram Krishnamoorthy, Yan Li, Bruce Lucas, Yuki Makino, Nathaniel Nystrom, Igor Peshansky, Vivek Sarkar, Armando Solar-Lezama, S. Alexander Spoon, Toshio Suganuma, Sayantan Sur, Toyotaro Suzumura, Christoph von Praun, Leena Unnikrishnan, Pradeep Varma, Krishna Nandivada Venkata, Jan Vitek, Hai Chuan Wang, Tong Wen, Salikh Zakirov, and Yoav Zibin.

For extended discussions and support we would like to thank: Gheorghe Almasi, Robert Blackmore, Rob O'Callahan, Calin Cascaval, Norman Cohen, Elmootaz Elnozahy, John Field, Kevin Gildea, Sara Salem Hamouda, Michihiro Horie, Arun Iyer, Chulho Kim, Orren Krieger, Doug Lea, John McCalpin, Paul McKenney, Hiroki Murata, Andrew Myers, Filip Pizlo, Ram Rajamony, R. K. Shyamasundar, V. T. Rajan, Frank Tip, Mandana Vaziri, and Hanhong Xue.

We thank Jonathan Rhees and William Clinger with help in obtaining the L^AT_EX style file and macros used in producing the Scheme report, on which this document is based. We acknowledge the influence of the JavaTM Language Specification [5], the Scala language specification [10], and ZPL [4].

This document specifies the language corresponding to Version 2.6.2 of the implementation. The redesign and reimplementation of arrays and rails was done by Dave Grove and Olivier Tardieu. Version 1.7 of the report was co-authored by Nathaniel Nystrom. The design of structs in X10 was led by Olivier Tardieu and Nathaniel Nystrom.

Earlier implementations benefited from significant contributions by Raj Barik, Philippe Charles, David Cunningham, Christopher Donawa, Robert Fuhrer, Christian Grothoff, Nathaniel Nystrom, Igor Peshansky, Vijay Saraswat, Vivek Sarkar, Olivier Tardieu, Pradeep Varma, Krishna Nandivada Venkata, and Christoph von Praun. Tong Wen has written many application programs in X10. Guojing Cong has helped in the development of many applications. The implementation of generics in X10 was influenced by the implementation of PolyJ [2] by Andrew Myers and Michael Clarkson.

Contents

1	Introduction	12
2	Overview of X10	14
2.1	Object-oriented features	14
2.2	The sequential core of X10	17
2.3	Places and activities	18
2.4	Distributed heap management	19
2.5	Clocks	20
2.6	Arrays, regions and distributions	20
2.7	Annotations	20
2.8	Translating MPI programs to X10	20
2.9	Summary and future work	21
2.9.1	Design for scalability	21
2.9.2	Design for productivity	21
2.9.3	Conclusion	22
3	Lexical and Grammatical structure	23
3.1	Whitespace	23
3.2	Comments	23
3.3	Identifiers	23
3.4	Keywords	24
3.5	Literals	24
3.6	Separators	26
3.7	Operators	26
3.8	Grammatical Notation	27
4	Types	29
4.1	Type System	30
4.2	Unit Types: Classes, Struct Types, and Interfaces	32
4.2.1	Class types	33
4.2.2	Struct Types	33
4.2.3	Interface types	34
4.2.4	Properties	34
4.3	Type Parameters and Generic Types	35
4.4	Type definitions	36

4.4.1	Motivation and use	37
4.5	Constrained types	39
4.5.1	Examples of Constraints	40
4.5.2	Syntax of constraints	41
4.5.3	Constraint solver: incompleteness and approximation	44
4.5.4	Acylicity of Properties	45
4.5.5	Limitation: Generics and Constraints at Runtime	45
4.6	Function types	47
4.7	Default Values	49
4.8	Annotated types	50
4.9	Subtyping and type equivalence	50
4.10	Common ancestors of types	51
4.11	Fundamental types	53
4.11.1	The interface Any	53
4.12	Type inference	54
4.12.1	Variable declarations	54
4.12.2	Return types	54
4.12.3	Inferring Type Arguments	56
4.13	Type Dependencies	60
4.14	Typing of Variables and Expressions	60
4.15	Limitations of Strict Typing	62
5	Variables	63
5.1	Immutable variables	64
5.2	Initial values of variables	65
5.3	Destructuring syntax	66
5.4	Formal parameters	67
5.5	Local variables and Type Inference	68
5.6	Fields	69
6	Names and packages	70
6.1	Names	70
6.1.1	Shadowing	70
6.1.2	Hiding	71
6.1.3	Obscuring	71
6.1.4	Ambiguity and Disambiguation	72
6.2	Access Control	73
6.2.1	Details of <code>protected</code>	73
6.3	Packages	74
6.3.1	Name Collisions	75
6.4	<code>import</code> Declarations	75
6.4.1	Single-Type Import	75
6.4.2	Automatic Import	76
6.4.3	Implicit Imports	76
6.5	Conventions on Type Names	76

7 Interfaces	77
7.1 Interface Syntax	79
7.2 Access to Members	79
7.3 Member Specification	79
7.4 Property Methods	80
7.5 Field Definitions	80
7.5.1 Fine Points of Fields	80
7.6 Generic Interfaces	81
7.7 Interface Inheritance	82
7.8 Members of an Interface	82
8 Classes	83
8.1 Principles of X10 Objects	83
8.1.1 Basic Design	83
8.1.2 Class Declaration Syntax	84
8.2 Fields	85
8.2.1 Field Initialization	85
8.2.2 Field hiding	85
8.2.3 Field qualifiers	86
8.3 Properties	87
8.3.1 Properties and Field Initialization	88
8.3.2 Properties and Fields	89
8.3.3 Acyclicity of Properties	89
8.4 Methods	89
8.4.1 Forms of Method Definition	91
8.4.2 Method Return Types	91
8.4.3 Throws Clause	91
8.4.4 Final Methods	92
8.4.5 Generic Instance Methods	92
8.4.6 Method Guards	92
8.4.7 Property methods	93
8.4.8 Method overloading, overriding, hiding, shadowing and obscuring	95
8.5 Constructors	98
8.5.1 Automatic Generation of Constructors	98
8.5.2 Calling Other Constructors	99
8.5.3 Return Type of Constructor	100
8.6 Static initialization	100
8.6.1 Compatability with Prior Versions of X10	101
8.7 User-Defined Operators	102
8.7.1 Binary Operators	104
8.7.2 Unary Operators	105
8.7.3 Type Conversions	106
8.7.4 Implicit Type Coercions	106
8.7.5 Assignment and Application Operators	107
8.8 User-Defined Control Structures	108

8.8.1	User-Defined for	110
8.8.2	User-Defined if	112
8.8.3	User-Defined try	112
8.8.4	User-Defined throw	113
8.8.5	User-Defined async	113
8.8.6	User-Defined atomic	114
8.8.7	User-Defined when	115
8.8.8	User-Defined finish	115
8.8.9	User-Defined at	116
8.8.10	User-Defined ateach	117
8.8.11	User-Defined while and do	117
8.8.12	User-Defined continue	118
8.8.13	User-Defined break	119
8.9	Class Guards and Invariants	120
8.9.1	Invariants for implements and extends clauses	121
8.9.2	Timing of Invariant Checks	121
8.9.3	Invariants and constructor definitions	121
8.10	Generic Classes	123
8.10.1	Use of Generics	123
8.11	Object Initialization	124
8.11.1	Constructors and Non-Escaping Methods	126
8.11.2	Fine Structure of Constructors	129
8.11.3	Definite Initialization in Constructors	131
8.11.4	Summary of Restrictions on Classes and Constructors	131
8.12	Method Resolution	133
8.12.1	Space of Methods	135
8.12.2	Possible Methods	137
8.12.3	Field Resolution	139
8.12.4	Other Disambiguations	140
8.13	Static Nested Classes	141
8.14	Inner Classes	141
8.14.1	Constructors and Inner Classes	143
8.15	Local Classes	144
8.16	Anonymous Classes	145
9	Structs	147
9.1	Struct declaration	148
9.2	Boxing of structs	149
9.3	Optional Implementation of Any methods	149
9.4	Primitive Types	150
9.4.1	Signed and Unsigned Integers	150
9.5	Example structs	150
9.6	Nested Structs	151
9.7	Default Values of Structs	151
9.8	Converting Between Classes And Structs	151

10 Functions	153
10.1 Overview	153
10.2 Function Application	154
10.3 Function Literals	155
10.3.1 Outer variable access	156
10.4 Functions as objects of type Any	157
11 Expressions	158
11.1 Literals	158
11.2 <code>this</code>	158
11.3 Local variables	159
11.4 Field access	159
11.5 Function Literals	161
11.6 Calls	161
11.6.1 <code>super</code> calls	162
11.7 Assignment	163
11.8 Increment and decrement	164
11.9 Numeric Operations	164
11.9.1 Conversions and coercions	165
11.9.2 Unary plus and unary minus	165
11.10 Bitwise complement	165
11.11 Binary arithmetic operations	165
11.12 Binary shift operations	166
11.13 Binary bitwise operations	166
11.14 String concatenation	166
11.15 Logical negation	166
11.16 Boolean logical operations	167
11.17 Boolean conditional operations	167
11.18 Relational operations	167
11.19 Conditional expressions	167
11.20 Stable equality	168
11.20.1 No Implicit Coercions for <code>==</code>	169
11.20.2 Non-Disjointness Requirement	170
11.21 Allocation	171
11.22 Casts and Conversions	172
11.22.1 Casts	172
11.22.2 Explicit Conversions	174
11.22.3 Resolving Ambiguity	174
11.23 Coercions and conversions	175
11.23.1 Coercions	175
11.23.2 Conversions	178
11.24 <code>instanceof</code>	179
11.24.1 Nulls in Constraints in <code>as</code> and <code>instanceof</code>	180
11.25 Subtyping expressions	180
11.26 Rail Constructors	181
11.27 Parenthesized Expressions	181

12 Statements	183
12.1 Empty statement	183
12.2 Local variable declaration	184
12.3 Block statement	185
12.4 Expression statement	186
12.5 Labeled statement	186
12.6 Break statement	187
12.7 Continue statement	188
12.8 If statement	188
12.9 Switch statement	189
12.10 While statement	189
12.11 Do-while statement	190
12.12 For statement	190
12.13 Return statement	192
12.14 Assert statement	192
12.15 Exceptions in X10	193
12.16 Throw statement	193
12.17 Try-catch statement	194
12.18 Assert	195
13 Places	196
13.1 The Structure of Places	196
13.2 <code>here</code>	196
13.3 <code>at</code> : Place Changing	197
13.3.1 Copying Values	198
13.3.2 How at Copies Values	199
13.3.3 <code>at</code> and Activities	199
13.3.4 Copying from <code>at</code>	200
13.3.5 Copying and Transient Fields	201
13.3.6 Copying and GlobalRef	202
13.3.7 Warnings about <code>at</code>	202
14 Activities	204
14.1 The X10 rooted exception model	205
14.2 <code>async</code> : Spawning an activity	205
14.3 <code>Finish</code>	206
14.4 Initial activity	207
14.5 <code>Ateach</code> statements	207
14.6 <code>vars</code> and Activities	208
14.7 Atomic blocks	208
14.7.1 Unconditional atomic blocks	210
14.7.2 Conditional atomic blocks	210
14.8 Use of Atomic Blocks	213
15 Clocks	215
15.1 Clock operations	217

15.1.1	Creating new clocks	217
15.1.2	Registering new activities on clocks	217
15.1.3	Resuming clocks	218
15.1.4	Advancing clocks	218
15.1.5	Dropping clocks	219
15.2	Deadlock Freedom	219
15.3	Program equivalences	220
15.4	Clocked Finish	220
16	Rails and Arrays	222
16.1	Overview	222
16.2	Rails	222
16.3	x10.array: Simple Arrays	224
16.3.1	Points	224
16.3.2	IterationSpace	225
16.3.3	Array	225
16.3.4	DistArray	226
16.4	x10.regionarray: Flexible Arrays	226
16.4.1	Regions	227
16.4.2	Arrays	229
16.4.3	Distributions	230
16.4.4	Distributed Arrays	232
16.4.5	Distributed Array Construction	232
16.4.6	Operations on Arrays and Distributed Arrays	233
17	Annotations	236
17.1	Annotation syntax	236
17.2	Annotation declarations	237
18	Interoperability with Other Languages	239
18.1	Embedded Native Code Fragments	239
18.1.1	Native static Methods	239
18.1.2	Native Blocks	241
18.2	Interoperability with External Java Code	242
18.2.1	How Java program is seen in X10	242
18.2.2	How X10 program is translated to Java	244
18.3	Interoperability with External C and C++ Code	247
18.3.1	Auxiliary C++ Files	249
18.3.2	C++ System Libraries	249
19	Definite Assignment	251
19.1	Asynchronous Definite Assignment	252
19.2	Characteristics of Definite Assignment	253
20	Grammar	258

References	277
Alphabetic index of definitions of concepts, keywords, and procedures	279
A Deprecations	289
B Change Log	290
B.1 Changes from X10 v2.5	290
B.2 Changes from X10 v2.4	290
B.3 Changes from X10 v2.3	291
B.3.1 Integral Literals	291
B.3.2 Arrays	291
B.3.3 Other Changes from X10 v2.3	292
B.4 Changes from X10 v2.2	292
B.5 Changes from X10 v2.1	293
B.6 Changes from X10 v2.0.6	294
B.6.1 Object Model	294
B.6.2 Constructors	295
B.6.3 Implicit clocks for each finish	295
B.6.4 Asynchronous initialization of val	296
B.6.5 Main Method	296
B.6.6 Assorted Changes	296
B.6.7 Safety of atomic and when blocks	296
B.6.8 Removed Topics	297
B.6.9 Deprecated	297
B.7 Changes from X10 v2.0	297
B.8 Changes from X10 v1.7	298
C Options	299
C.1 Compiler Options: Common	299
C.1.1 Optimization: -O or -optimize	299
C.1.2 Debugging: -DEBUG=boolean	299
C.1.3 Call Style: -STATIC_CHECKS, -VERBOSE_CHECKS	299
C.1.4 Help: -help and --help	300
C.1.5 Source Path: -sourcepath path	300
C.1.6 Output Directory: -d directory	300
C.1.7 Executable File: -o path	300
C.2 Compiler Option: C++	300
C.2.1 Runtime: -x10rt impl	300
C.3 Compiler Option: Java	300
C.3.1 Class Path: -classpath path	300
C.4 Execution Options: Java	301
C.4.1 Class Path: -classpath path	301
C.4.2 Library Path: -libpath path	301
C.4.3 Heap Size: -mssize and -mxsize	301
C.4.4 Stack Size: -sssize	301

CONTENTS	11
-----------------	-----------

C.4.5	Places: <code>-np count</code>	301
C.4.6	Hosts: <code>-host host1,host2,...</code> or <code>-hostfile file</code> . .	301
C.4.7	Runtime: <code>-x10rt impl</code>	301
C.4.8	Help: <code>-h</code>	301
C.5	Running X10	302
C.6	Managed X10	302
C.7	Native X10	302
D	Acknowledgments and Trademarks	303

1 Introduction

Background

The era of the mighty single-processor computer is over. Now, when more computing power is needed, one does not buy a faster uniprocessor—one buys another processor just like those one already has, or another hundred, or another million, and connects them with a high-speed communication network. Or, perhaps, one rents them instead, with a cloud computer. This gives one whatever number of computer cycles that one can desire and afford.

The problem, then, is how to use those computer cycles effectively. One must understand how to divide up the available work into chunks that can be executed simultaneously without introducing undesirable indeterminacy, cycles of “deadly embrace” which jam up processors or causing processors to spin uselessly waiting for conditions that may never materialize.

One response to this problem has been to move to a *fragmented memory model*. Multiple processors are programmed largely as if they were uniprocessors, but are made to interact via a relatively language-neutral message-passing format such as MPI [12]. This model has enjoyed some success: several high-performance applications have been written in this style. Unfortunately, this model leads to a *loss of programmer productivity*: the message-passing format is integrated into the host language by means of an application-programming interface (API), the programmer must explicitly represent and manage the interaction between multiple processes and choreograph their data exchange; large data-structures (such as distributed arrays, graphs, hash-tables) that are conceptually unitary must be thought of as fragmented across different nodes; all processors must generally execute the same code in an Single Program Multiple Data (SPMD) fashion etc.

One response to this problem has been the advent of the *partitioned global address space* (PGAS) model underlying languages such as UPC, Titanium and Co-Array Fortran [3, 16]. These languages permit the programmer to think of a single computation running across multiple processors, sharing a common address space. All data resides at some processor, which is said to have *affinity* to the data. Each processor may operate directly on the data it contains but must use some indirect mechanism to access or update data at other processors. Some kind of global *barriers* are used to ensure that processors remain roughly synchronized.

X10 is the first of the second generation of PGAS languages. It is a modern object-oriented programming language that introduces new constructs that significantly simplify scale out programming. The fundamental goal of X10 is to enable scalable, high-performance, high-productivity transformational programming for high-end computers—for traditional numerical computation workloads (such as weather simulation, molecular dynamics, particle transport problems etc) as well as commercial server workloads, such as big data analytics.

X10 is based on state-of-the-art object-oriented programming ideas primarily to take advantage of their proven flexibility and ease-of-use for a wide spectrum of programming problems. X10 takes advantage of several years of research (e.g., in the context of the Java Grande forum, [9, 1]) on how to adapt such languages to the context of high-performance numerical computing. Thus X10 provides support for user-defined *struct types* (such as `Int`, `Float`, `Complex` etc) and function literals, supports a very flexible form of multi-dimensional arrays (based on ideas in ZPL [4]) and supports IEEE-standard floating point arithmetic. Some capabilities for overloading operator are also provided.

X10 introduces a flexible treatment of concurrency, distribution and locality, within an integrated type system. X10 extends the PGAS model with *asynchrony* (yielding the *APGAS* programming model). X10 introduces *places* as an abstraction for a computational context with a locally synchronous view of shared memory. An X10 computation runs over a large collection of places. Each place hosts some data and runs one or more *activities*. Activities can be dynamically created. Activities are lightweight threads of execution. An activity may synchronously (and *atomically*) use one or more memory locations in the place in which it resides, leveraging current symmetric multiprocessor (SMP) technology. A distributed termination construct `finish` enables code to execute after all activities in the given statement have terminated, thus ensuring that all their side-effects have already taken place. An activity may shift to another place to execute a statement block. X10 provides weaker ordering guarantees for inter-place data access, enabling applications to scale. Multiple memory locations in multiple places cannot be accessed atomically. *Immutable* data needs no consistency management and may be freely copied by the implementation between places. One or more *clocks* may be used to order activities running in multiple places. *DistArrays*, distributed arrays, may be distributed across multiple places and support parallel collective operations. A novel exception flow model ensures that exceptions thrown by asynchronous activities can be caught at a suitable parent activity. Asynchronous initialization of variables is supported. Linking with native code is supported.

2 Overview of X10

X10 is a statically typed object-oriented language, extending a sequential core language with *places*, *activities*, *clocks*, (distributed, multi-dimensional) *arrays* and *struct* types. All these changes are motivated by the desire to use the new language for high-end, high-performance, high-productivity computing.

2.1 Object-oriented features

The sequential core of X10 is a *container-based* object-oriented language similar to Java and C++, and more recent languages such as Scala. Programmers write X10 code by defining containers for data and behavior called *classes* (§8) and *structs* (§9), often abstracted as *interfaces* (§7). X10 provides inheritance and subtyping in fairly traditional ways.

Example:

Normed describes entities with a `norm()` method. *Normed* is intended to be used for entities with a position in some coordinate system, and `norm()` gives the distance between the entity and the origin. A *Slider* is an object which can be moved around on a line; a *PlanePoint* is a fixed position in a plane. Both *Sliders* and *PlanePoints* have a sensible `norm()` method, and implement *Normed*.

```
interface Normed {
    def norm():Double;
}
class Slider implements Normed {
    var x : Double = 0;
    public def norm() = Math.abs(x);
    public def move(dx:Double) { x += dx; }
}
struct PlanePoint implements Normed {
    val x : Double; val y:Double;
    public def this(x:Double, y:Double) {
        this.x = x; this.y = y;
    }
}
```

```
public def norm() = Math.sqrt(x*x+y*y);
}
```

Interfaces An X10 interface specifies a collection of abstract methods; `Normed` specifies just `norm()`. Classes and structs can be specified to *implement* interfaces, as `Slider` and `PlanePoint` implement `Normed`, and, when they do so, must provide all the methods that the interface demands.

Interfaces are purely abstract. Every value of type `Normed` must be an instance of some class like `Slider` or some struct like `PlanePoint` which implements `Normed`; no value can be `Normed` and nothing else.

Classes and Structs There are two kinds of containers: *classes* (§8) and *structs* (§9). Containers hold data in *fields*, and give concrete implementations of methods, as `Slider` and `PlainPoint` above.

Classes are organized in a single-inheritance tree: a class may have only a single parent class, though it may implement many interfaces and have many subclasses. Classes may have mutable fields, as `Slider` does.

In contrast, structs are headerless values, lacking the internal organs which give objects their intricate behavior. This makes them less powerful than objects (*e.g.*, structs cannot inherit methods, though objects can), but also cheaper (*e.g.*, they can be inlined, and they require less space than objects). Structs are immutable, though their fields may be immutably set to objects which are themselves mutable. They behave like objects in all ways consistent with these limitations; *e.g.*, while they cannot *inherit* methods, they can have them – as `PlanePoint` does.

X10 has no primitive classes per se. However, the standard library `x10.lang` supplies structs and objects `Boolean`, `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Complex` and `String`. The user may defined additional arithmetic structs using the facilities of the language.

Functions. X10 provides functions (§10) to allow code to be used as values. Functions are first-class data: they can be stored in lists, passed between activities, and so on. `square`, below, is a function which squares an `Long`. `of4` takes an `Long`-to-`Long` function and applies it to the number 4. So, `fourSquared` computes `of4(square)`, which is `square(4)`, which is 16, in a fairly complicated way.

```
val square = (i:Long) => i*i;
val of4 = (f: (Long)=>Long) => f(4);
val fourSquared = of4(square);
```

Functions are used extensively in X10 programs. For example, a common way to construct and initialize an `Rail[Long]` – that is, a fixed-length one-dimensional array of numbers, like an `long[]` in Java – is to pass two arguments to a factory method: the first argument being the length of the rail, and the second being a function which

computes the initial value of the i^{th} element. The following code constructs a 1-dimensional rail initialized to the squares of 0,1,...,9: $r(0) == 0$, $r(5) == 25$, etc.

```
val r : Rail[Long] = new Rail[Long](10, square);
```

Constrained Types X10 containers may declare *properties*, which are fields bound immutably at the creation of the container. The static analysis system understands properties, and can work with them logically.

For example, an implementation of matrices `Mat` might have the numbers of rows and columns as properties. A little bit of care in definitions allows the definition of a `+` operation that works on matrices of the same shape, and `*` that works on matrices with appropriately matching shapes.

```
abstract class Mat(rows:Long, cols:Long) {
    static type Mat(r:Long, c:Long) = Mat{rows==r&&cols==c};
    abstract operator this + (y:Mat(this.rows, this.cols))
        :Mat(this.rows, this.cols);
    abstract operator this * (y:Mat) {this.cols == y.rows}
        :Mat(this.rows, y.cols);
```

The following code typechecks (assuming that `makeMat(m,n)` is a function which creates an $m \times n$ matrix). However, an attempt to compute `axb1 + bxc` or `bxc * axb1` would result in a compile-time type error:

```
static def example(a:Long, b:Long, c:Long) {
    val axb1 : Mat(a,b) = makeMat(a,b);
    val axb2 : Mat(a,b) = makeMat(a,b);
    val bxc : Mat(b,c) = makeMat(b,c);
    val axc : Mat(a,c) = (axb1 + axb2) * bxc;
    //ERROR: val wrong1 = axb1 + bxc;
    //ERROR: val wrong2 = bxc * axb1;
}
```

The “little bit of care” shows off many of the features of constrained types. The `(rows:Long, cols:Long)` in the class definition declares two properties, `rows` and `cols`.¹

A constrained type looks like `Mat{rows==r && cols==c}`: a type name, followed by a Boolean expression in braces. The type declaration on the second line makes `Mat(r,c)` be a synonym for `Mat{rows==r && cols==c}`, allowing for compact types in many places.

Functions can return constrained types. The `makeMat(r,c)` method returns a `Mat(r,c)` – a matrix whose shape is given by the arguments to the method. In particular, constructors can have constrained return types to provide specific information about the constructed values.

¹The class is officially declared abstract to allow for multiple implementations, like sparse and band matrices, but in fact is abstract to avoid having to write the actual definitions of `+` and `*`.

The arguments of methods can have type constraints as well. The operator `this +` line lets `A+B` add two matrices. The type of the second argument `y` is constrained to have the same number of rows and columns as the first argument `this`. Attempts to add mismatched matrices will be flagged as type errors at compilation.

At times it is more convenient to put the constraint on the method as a whole, as seen in the operator `this *` line. Unlike for `+`, there is no need to constrain both dimensions; we simply need to check that the columns of the left factor match the rows of the right. This constraint is written in `{...}` after the argument list. The shape of the result is computed from the shapes of the arguments.

And that is all that is necessary for a user-defined class of matrices to have shape-checking for matrix addition and multiplication. The `example` method compiles under those definitions.

Generic types Containers may have type parameters, permitting the definition of *generic types*. Type parameters may be instantiated by any X10 type. It is thus possible to make a list of integers `List[Long]`, a list of non-zero integers `List[Long{self != 0}]`, or a list of people `List[Person]`. In the definition of `List`, `T` is a type parameter; it can be instantiated with any type.

```
class List[T] {
    var head: T;
    var tail: List[T];
    def this(h: T, t: List[T]) { head = h; tail = t; }
    def add(x: T) {
        if (this.tail == null)
            this.tail = new List[T](x, null);
        else
            this.tail.add(x);
    }
}
```

The constructor (`def this`) initializes the fields of the new object. The `add` method appends an element to the list. `List` is a generic type. When instances of `List` are allocated, the type parameter `T` must be bound to a concrete type. `List[Long]` is the type of lists of element type `Long`, `List[List[String]]` is the type of lists whose elements are themselves lists of string, and so on.

2.2 The sequential core of X10

The sequential aspects of X10 are mostly familiar from C and its progeny. X10 enjoys the familiar control flow constructs: `if` statements, `while` loops, `for` loops, `switch` statements, `throw` to raise exceptions and `try...catch` to handle them, and so on.

X10 has both implicit coercions and explicit conversions, and both can be defined on user-defined types. Explicit conversions are written with the `as` operation: `n as Long`. The types can be constrained: `n as Long{self != 0}` converts `n` to a non-zero integer, and throws a runtime exception if its value as an integer is zero.

2.3 Places and activities

The full power of X10 starts to emerge with concurrency. An X10 program is intended to run on a wide range of computers, from uniprocessors to large clusters of parallel processors supporting millions of concurrent operations. To support this scale, X10 introduces the central concept of *place* (§13). A place can be thought of as a virtual shared-memory multi-processor: a computational unit with a finite (though perhaps changing) number of hardware threads and a bounded amount of shared memory, uniformly accessible by all threads.

An X10 computation acts on *values*(§8.1) through the execution of lightweight threads called *activities*(§14). An *object* has a small, statically fixed set of fields, each of which has a distinct name. A scalar object is located at a single place and stays at that place throughout its lifetime. An *aggregate* object has many fields (the number may be known only when the object is created), uniformly accessed through an index (*e.g.*, an integer) and may be distributed across many places. The distribution of an aggregate object remains unchanged throughout the computation, though different aggregates may be distributed differently. Objects are garbage-collected when no longer useable; there are no operations in the language to allow a programmer to explicitly release memory.

X10 has a *unified* or *global address space*. This means that an activity can reference objects at other places. However, an activity may synchronously access data items only in the current place, the place in which it is running. It may atomically update one or more data items, but only in the current place. If it becomes necessary to read or modify an object at some other place `q`, the *place-shifting* operation `at(q; F)` can be used, to move part of the activity to `q`. `F` is a specification of what information will be sent to `q` for use by that part of the computation. It is easy to compute across multiple places, but the expensive operations (*e.g.*, those which require communication) are readily visible in the code.

Atomic blocks. X10 has a control construct `atomic S` where `S` is a statement with certain restrictions. `S` will be executed atomically, without interruption by other activities. This is a common primitive used in concurrent algorithms, though rarely provided in this degree of generality by concurrent programming languages.

More powerfully – and more expensively – X10 allows conditional atomic blocks, `when(B)S`, which are executed atomically at some point when `B` is true. Conditional atomic blocks are one of the strongest primitives used in concurrent algorithms, and one of the least-often available.

Asynchronous activities. An asynchronous activity is created by a statement `async S`, which starts up a new activity running `S`. It does not wait for the new activity to finish; there is a separate statement (`finish`) to do that.

2.4 Distributed heap management

X10 is the language for parallel and distributed computing, which is based on the APGAS (Asynchronous Partitioned Global Address Space) programming model. In (A)PGAS, the address space is partitioned into multiple semi-spaces. The semi-space is called *place* in X10. In Managed X10 (X10 on Java VMs), a place is represented as a single Java VM and the semi-space is mapped to the heap of the Java VM.

X10 supports garbage collection. Objects in a local heap (local objects) are collected with (local) garbage collection and there is no way to explicitly free them. The reference to local objects is called *local reference*.

In addition, X10 has another type of reference called *remote reference*. Unlike local reference, remote reference can reference objects at both local and remote places.

With remote reference, an activity (something like thread, it runs on a place at a time but it can move itself to different places) can access objects at a remote place (remote objects) when the activity has moved to the remote place. The place where an object is created is the home place of the object and it does not change for the lifetime.

To guarantee an activity can access remote objects at their home place, the objects with remote reference are protected from (local) garbage collection at their home place even if they have no local reference. Objects can be garbage collected only when they have neither local nor remote reference. The garbage collection that takes care of remote reference is called distributed garbage collection and it is supported in Managed X10.

Distributed garbage collection in Managed X10 [8] tracks the lifetime of remote reference with reference counting. When the local garbage collection at a remote place detects the remote reference is no longer needed at the place, the count is decremented. When the count becomes zero, the local garbage collection at the home place is ready to collect the referenced object in the ordinary way.

This mechanism works in most cases, but when there is unbalance in heap allocation rate between places, there is a risk of out of memory error at a frequently allocating place. This is because remote reference from infrequently allocating (i.e. infrequently garbage collected) places could retain remotely referenced objects longer than needed.

To avoid the out of memory error even with unbalanced heap allocation rate, there is a way to explicitly release remote reference.

A single call of `PlaceLocalHandle.destroy()` (`PlaceLocalHandle` is an X10 type that bundles multiple remote references to the objects at different places) releases all remote references immediately, thus the local garbage collection at each place becomes ready to collect the referenced object in the ordinary way. It can be called at the point where the all objects referenced by the handle are no longer needed to be accessible with the handle. Local reference to the object at each place won't be affected.

2.5 Clocks

The MPI style of coordinating the activity of multiple processes with a single barrier is not suitable for the dynamic network of heterogeneous activities in an X10 computation. X10 allows multiple barriers in a form that supports determinate, deadlock-free parallel computation, via the `Clock` type.

A single `Clock` represents a computation that occurs in phases. At any given time, an activity is *registered* with zero or more clocks. The static method `Clock.advanceAll` tells all of an activity's registered clocks that the activity has finished the current phase, and causes it to wait for the next phase. Other operations allow waiting on a single clock, starting new clocks or new activities registered on an extant clock, and so on.

Clocks act as barriers for a dynamically varying collection of activities. They generalize the barriers found in MPI style program in that an activity may use multiple clocks simultaneously. Yet programs using clocks properly are guaranteed not to suffer from deadlock.

2.6 Arrays, regions and distributions

X10 provides `DistArrays`, *distributed arrays*, which spread data across many places. An underlying `Dist` object provides the *distribution*, telling which elements of the `DistArray` go in which place. `Dist` uses subsidiary `Region` objects to abstract over the shape and even the dimensionality of arrays. Specialized X10 control statements such as `at each` provide efficient parallel iteration over distributed arrays.

2.7 Annotations

X10 supports annotations on classes and interfaces, methods and constructors, variables, types, expressions and statements. These annotations may be processed by compiler plugins.

2.8 Translating MPI programs to X10

While X10 permits considerably greater flexibility in writing distributed programs and data structures than MPI, it is instructive to examine how to translate MPI programs to X10.

Each separate MPI process can be translated into an X10 place. Async activities may be used to read and write variables located at different processes. A single clock may be used for barrier synchronization between multiple MPI processes. X10 collective operations may be used to implement MPI collective operations. X10 is more general than MPI in (a) not requiring synchronization between two processes in order to

enable one to read and write the other’s values, (b) permitting the use of high-level atomic blocks within a process to obtain mutual exclusion between multiple activities running in the same node (c) permitting the use of multiple clocks to combine the expression of different physics (e.g., computations modeling blood coagulation together with computations involving the flow of blood), (d) not requiring an SPMD style of computation.

2.9 Summary and future work

2.9.1 Design for scalability

X10 is designed for scalability, by encouraging working with local data, and limiting the ability of events at one place to delay those at another. For example, an activity may atomically access only multiple locations in the current place. Unconditional atomic blocks are dynamically guaranteed to be non-blocking, and may be implemented using non-blocking techniques that avoid mutual exclusion bottlenecks. Data-flow synchronization permits point-to-point coordination between reader/writer activities, obviating the need for barrier-based or lock-based synchronization in many cases.

2.9.2 Design for productivity

X10 is designed for productivity.

Safety and correctness. Programs written in X10 are guaranteed to be statically *type safe*, *memory safe* and *pointer safe*, with certain exceptions given in §4.15.

Static type safety guarantees that every location contains only values whose dynamic type agrees with the location’s static type. The compiler allows a choice of how to handle method calls. In strict mode, method calls are statically checked to be permitted by the static types of operands. In lax mode, dynamic checks are inserted when calls may or may not be correct, providing weaker static correctness guarantees but more programming convenience.

Memory safety guarantees that an object may only access memory within its representation, and other objects it has a reference to. X10 does not permit pointer arithmetic, and bound-checks array accesses dynamically if necessary. X10 uses garbage collection to collect objects no longer referenced by any activity. X10 guarantees that no object can retain a reference to an object whose memory has been reclaimed. Further, X10 guarantees that every location is initialized at run time before it is read, and every value read from a word of memory has previously been written into that word.

X10 programs that use only the common, specified clock idioms and unconditional atomic blocks are guaranteed not to deadlock. Unconditional atomic blocks are non-blocking, hence cannot introduce deadlocks. Many concurrent programs can be shown to be determinate (hence race-free) statically.

Integration. A key issue for any new programming language is how well it can be integrated with existing (external) languages, system environments, libraries and tools.

We believe that X10, like Java, will be able to support a large number of libraries and tools. An area where we expect future versions of X10 to improve on Java like languages is *native integration* (§18). Specifically, X10 will permit multi-dimensional local arrays to be operated on natively by native code.

2.9.3 Conclusion

X10 is considerably higher-level than thread-based languages in that it supports dynamically spawning lightweight activities, the use of atomic operations for mutual exclusion, and the use of clocks for repeated quiescence detection.

Yet it is much more concrete than languages like HPF in that it forces the programmer to explicitly deal with distribution of data objects. In this the language reflects the designers' belief that issues of locality and distribution cannot be hidden from the programmer of high-performance code in high-end computing. A performance model that distinguishes between computation and communication must be made explicit and transparent.² At the same time we believe that the place-based type system and support for generic programming will allow the X10 programmer to be highly productive; many of the tedious details of distribution-specific code can be handled in a generic fashion.

²In this X10 is similar to more modern languages such as ZPL [4].

3 Lexical and Grammatical structure

Lexically a program consists of a stream of white space, comments, identifiers, keywords, literals, separators and operators, all of them composed of Unicode characters in the UTF-8 (or US-ASCII) encoding.

3.1 Whitespace

ASCII space, horizontal tab (HT), form feed (FF) and line terminators constitute white space.

3.2 Comments

All text included within the ASCII characters “`/*`” and “`*/`” is considered a comment and ignored; nested comments are not allowed. All text from the ASCII characters “`//`” to the end of line is considered a comment and is ignored.

3.3 Identifiers

Identifiers consist of a single letter followed by zero or more letters or digits. The letters are the ASCII characters `a` through `z`, `A` through `Z`, and `_`. Digits are defined as the ASCII characters `0` through `9`. Case is significant; `a` and `A` are distinct identifiers, `as` is a keyword, but `As` and `AS` are identifiers. (However, case is insignificant in the hexadecimal numbers, exponent markers, and type-tags of numeric literals – `0xbabe` = `0XBABE`.)

In addition, any string of characters may be enclosed in backquotes ‘ to form an identifier – though the backquote character itself, and the backslash character, must be quoted by a backslash if they are to be included. This allows, for example, keywords to be used as identifiers. The following are backquoted identifiers:

```
'while', '!', '(unbalanced(', '\'', '0'
```

Certain back ends and compilation options do not support all choices of identifier.

3.4 Keywords

X10 uses the following keywords:

abstract	as	assert	async	at
athome	ateach	atomic	break	case
catch	class	clocked	continue	def
default	do	else	extends	false
final	finally	finish	for	goto
haszero	here	if	implements	import
in	instanceof	interface	native	new
null	offer	offers	operator	package
private	property	protected	public	return
self	static	struct	super	switch
this	throw	transient	true	try
type	val	var	void	when
while				

Keywords may be used as identifiers by enclosing them in backquotes: ‘new’ is an identifier, `new` is a keyword but not an identifier.

Note that the primitive type names are not considered keywords.

3.5 Literals

Briefly, X10 v2.4 uses fairly standard syntax for its literals: integers, unsigned integers, floating point numbers, booleans, characters, strings, and `null`. The most exotic points are (1) unsigned numbers are marked by a `u` and cannot have a sign; (2) `true` and `false` are the literals for the booleans; and (3) floating point numbers are `Double` unless marked with an `f` for `Float`.

Less briefly, we use the following abbreviations:

d	= one or more decimal digits only starting with 0 if it is 0
d_8	= one or more octal digits
d_{16}	= one or more hexadecimal digits, using a-f or A-F for 10-15
i	= $d \mid 0d_8 \mid 0xd_{16} \mid 0Xd_{16}$
s	= optional + or -
b	= $d \mid d. \mid d.d \mid .d$
x	= $(e \mid E)sd$
f	= bx

- `true` and `false` are the Boolean literals.

- `null` is a literal for the null value. It has type `Any{self==null}`.
- Int literals have the form `sin` or `siN`. E.g., `123n`, `-321N` are decimal Ints, `0123N` and `-0321n` are octal Ints, and `0x123n`, `-0X321N`, `0xEDN`, and `0XEBCN` are hexadecimal Ints.
- Long literals have the form `si`, `sil` or `siL`. E.g., `1234567890` and `0xBABEL` are Long literals.
- UInt literals have the form `iun` or `inu`, or capital versions of those. E.g., `123un`, `0123un`, and `0xBEAUN` are UInt literals.
- ULong literals have the form `iu`, `iul` or `ilu`, or capital versions of those. For example, `123u`, `0124567012u`, `0xFU`, `0Xba1efu`, and `0xDecafC0ffeeFU` are ULong literals.
- Short literals have the form `sis` or `siS`. E.g., `414S`, `0xACES` and `7001s` are short literals.
- UShort literals form `ius` or `isu`, or capital versions of those. For example, `609US`, `107us`, and `0xBeaus` are unsigned short literals.
- Byte literals have the form `siy` or `siY`. (The letter B cannot be used for bytes, as it is a hexadecimal digit.) `50Y` and `0xBABY` are byte literals.
- UByte literals have the form `iuy` or `iyu`, or capitalized versions of those. For example, `9uy` and `0xBUY` are UByte literals.
- Float literals have the form `sff` or `sfF`. Note that the floating-point marker letter `f` is required: unmarked floating-point-looking literals are Double. E.g., `1f`, `6.023E+32f`, `6.626068E-34F` are Float literals.
- Double literals have the form `sf1`, `sfD`, and `sfD`. E.g., `0.0`, `0e100`, `1.3D`, `229792458d`, and `314159265e-8` are Double literals.
- Char literals have one of the following forms:
 - `'c'` where `c` is any printing ASCII character other than `\` or `'`, representing the character `c` itself; e.g., `'!'`;
 - `'\b'`, representing backspace;
 - `'\t'`, representing tab;
 - `'\n'`, representing newline;
 - `'\f'`, representing form feed;
 - `'\r'`, representing return;
 - `'\'`, representing single-quote;

¹Except that literals like `1` which match both `i` and `f` are counted as integers, not Double; Doubles require a decimal point, an exponent, or the `d` marker.

- '\\"', representing double-quote;
 - '\\\\', representing backslash;
 - '\\dd', where dd is one or more octal digits, representing the one-byte character numbered dd; it is an error if dd > 0377.
- String literals consist of a double-quote ", followed by zero or more of the contents of a Char literal, followed by another double quote. E.g., "hi!", "".

3.6 Separators

X10 has the following separators and delimiters:

() { } [] ; , .

3.7 Operators

X10 has the following operator, type constructor, and miscellaneous symbols. (?) and : comprise a single ternary operator, but are written separately.)

```
== != < > <= >=
&& || & | ^
<< >> >>>
+ - * / %
++ -- !
&= |= ^= 
<<= >>= >>>=
+= -= *= /= %=
= ? : => ->
<: :> @ ..
** ! ~ -< >-
```

The precedence of the operators is as follows. Earlier rows of the table have higher precedence than later rows, binding more tightly. For example, a+b*c<d parses as (a+(b*c))<d, and -1 as Byte parses as -(1 as Byte).

```

postfix ()  

as T, postfix ++, postfix --  

unary -, unary +, prefix ++, prefix --  

unary operators !, ~, ^, *, |, &, /, and %  

..  

*      /      %      **  

+      -  

<<    >>    >>>   ->   >-   -<   <-   !  

>    >=    <    <=  instanceof  

==    !=    !    !~  

&  

^  

|  

&&  

||  

?:  

=, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=

```

3.8 Grammatical Notation

In this manual, ordinary BNF notation is used to specify grammatical constructions, with a few minor extensions. Grammatical rules look like this:

```

Adj ::= Adv? happy
      | Adv? sad
Adv  ::= very
      | Adv Adv

```

Terms in *italics* are called **non-terminals**. They represent kinds of phrases; for example, *ForStmt* (20.74)² describes all *for* statements. Equation numbers refer to the full X10 grammar, in §20. The small example has two non-terminals, *Adv* and *Adj*.

Terms in **fixed-width font** are **terminals**. They represent the words and symbols of the language itself. In X10, the terminals are the words described in this chapter.

A single grammatical rule has the form $A ::= X_1 X_2 \dots X_n$, where the X_i 's are either terminals or nonterminals. This indicates that the non-terminal A could be an instance of X_1 , followed by an instance of X_2 , ..., followed by an instance of X_n . Multiple rules for the same A are allowed, giving several possible phrasings of A 's. For brevity, two rules with the same left-hand side are written with the left-hand side appearing once, and the right-hand sides separated by |.

In the *Adj* example, there are two rules for *Adv*, $Adv ::= \text{very}$ and $Adv ::= Adv\ Adv$. So, an adverb could be *very*, or (by three uses of the rule) *very very*, or, one or more *very*'s.

The notation $A^?$ indicates an optional A . This is an ordinary non-terminal, defined by the rules:

²Grammar rules are given in §20, and referred to by equation number in that section.

$$\begin{array}{l} A^? \quad ::= \\ | \quad A \end{array}$$

The first rule says that $A^?$ can amount to nothing; the second, that it can amount to an A . This concept shows up so often that it is worth having a separate notation for it. In the *Adj* example, an adjective phrase may be preceded by an optional adverb. Thus, it may be **happy**, or **very happy**, or **very very sad**, etc.

4 Types

X10 is a *strongly typed* object-oriented language: every variable and expression has a type that is known at compile-time. Types limit the values that variables can hold.

X10 supports four kinds of values, *objects*, *struct values*, *functions*, and `null`. Objects are in the grand tradition of object-oriented languages, and the heart of most X10 computations. They are instances of *classes* (§8); they hold zero or more data fields that may be mutable. They respond to methods, and can inherit behavior from their superclass.

Struct values are similar to objects, though more restricted in ways that make them more efficient in space and time. Their fields cannot be mutable, and, although they respond to methods, they do not inherit behavior. They are instances of struct types (§9).

Together, objects and struct values are called *containers*, because they hold data.

Functions, called closures, lambda-expressions, and blocks in other languages, are instances of *function types* (§10). A function has zero or more *formal parameters* (or *arguments*) and a *body*, which is an expression that can reference the formal parameters and also other variables in the surrounding block. For instance, `(x:Long)=>x*y` is a unary integer function which multiplies its argument by the variable `y` from the surrounding block. Functions may be freely copied from place to place and may be repeatedly applied.

Finally, `null` is a constant, often found as the default value of variables of object type. While it is not an object, it may be stored in variables of class type – except for types which have a constraint (§4.5) which specifically excludes `null`.

These runtime values are classified by *types*. Types are used in variable declarations (§12.2), coercions and explicit conversions (§11.9.1), object creation (§11.21), static state and method accessors (§11.4), generic classes, structs, interfaces, and methods (§4.3), type definitions (§4.4), closures (§10), class, struct, and interface declarations (§8.1.2), subtyping expressions (§11.25), and `instanceof` and `as` expressions (§11.24).

The basic relationship between values and types is the *is a value in* relation: `e` is a value in `T`. We also often say “`e` has type `T`” or “`e` is an element of type `T`”. For example, `1` has type `Long` (the type of all integers representable in 64 bits). It has the more general

type `Any` (since all entities have type `Any`). Furthermore, it has such types as “Nonzero integer” and “Integer equal to one”, and many others. These types are expressable in X10 using constrained types (§4.5). `Long{self!=0}` is the type of `Longs` `self`¹ which are not equal to zero, and `Long{self==1}` is the type of the `Longs` which are equal to one.

The basic relationship between types is *subtyping*: $T <: U$ holds if every value in T is also a value in U . Two important kinds of subtyping are *subclassing* and *strengthening*. Subclassing is a familiar notion from object-oriented programming. Here we use it to refer to the relationship between a class and another class it extends or an interface (§7) it implements. For instance, in a class hierarchy with classes `Animal` and `Cat` such that `Cat` extends `Mammal` and `Mammal` extends `Animal`, every instance of `Cat` is by definition an instance of `Animal` (and `Mammal`). We say that `Cat` is a subclass of `Animal`, or $\text{Cat} <: \text{Animal}$ by subclassing. If `Animal` implements `Thing`, then `Cat` also implements `Thing`, and we say $\text{Cat} <: \text{Thing}$ by subclassing.

Strengthening is an equally familiar notion from logic. The instances of `Long{self == 1}` are all elements of `Long{self != 0}` as well, because `self == 1` logically implies `self != 0`; so $\text{Long}\{\text{self} == 1\} <: \text{Long}\{\text{self} != 0\}$ by strengthening. X10 uses both notions of subtyping. See §4.9 for the full definition of subtyping in X10.

4.1 Type System

X10 has several sorts of types. In this section, S , T , and T_i range over types. X ranges over type variables, M and x_i over identifiers, c over constraint expressions (§4.5), and e_i over expressions. For compactness, slanted brackets are used to indicate optional elements.²

<i>Type</i>	$::=$	T	
T	$::=$	M	(1)
		X	(2)
		$M [T_1, \dots, T_n]$	(3)
		$T_1 . T_2$	(4)
		F	(5)
		$M / [T_1, \dots, T_n] / (e_1, \dots, e_k)$	(6)
		$T\{c\}$	(7)
F	$::=$	$([x_1 :] T_1, \dots, [x_n :] T_n) / \{c\} / \Rightarrow T$	
		$([x_1 :] T_1, \dots, [x_n :] T_n) / \{c\} / \Rightarrow void$	

A type given by (1) is an identifier `M`, like `Point`, `Long`, or `long`. It refers to a unit – a class, struct type, or interface, (§4.2). Or, it can refer to a name defined by a `type` statement (§4.4);

¹X10 automatically uses the identifier `self` for the element of the type being constrained.

²The actual grammar, as given in §20, is slightly more intricate for technical reasons. The set of types is the same, however, and this grammar is better for exposition.

Example: *String refers to the standard class of strings, Long to the standard struct type of integers, and Any to the interface that describes all X10 values. long is an alias for the type Long, for the comfort of programmers used to other languages in the C family.*

A type of the form (2), a type variable X , refers to a parameter type of a generic (parameterized) type, as described in §4.3.

Example: *The class Pair[X] below provides a simplistic way to keep two things of the same type together.³ Pair[Long] holds two integers; Pair[Pair[String]] holds two pairs of strings. Within the definition of Pair, the type variable X is the parameter type of Pair – that is, the type which this pair is a pair of.*

```
class Pair[X]{
    public val first : X;
    public val second: X;
    public def this(f:X, s:X) {first = f; second = s;}
}
```

A type of form (3), $M[T,U]$, is a use of a generic type, also described in §4.3, or a generic type-defined type without value parameters (§4.4). The types inside the brackets are the actual parameters corresponding to the formal parameters of the parameterized type M . `Pair[Long]`, above, is an example of a use of the generic type `Pair`.

A type of form (4), $T.U$, is a qualified type: a unit U appearing inside of the unit T , as described in §8.14.

Example:

```
class Outer {
    class Inner { /* ... */ }
}
```

then (new Outer()).new Inner() creates a value of type Outer.Inner.

A type of form (5), F , such as `(x:Long)=>Long`, is a function type. Its values are functions, e.g., the squaring function taking integers to integers. Function types are described in §4.6, and computing with functions is described in §10.

Example: `square` is the squaring function on integers. It is used in the assert line.

```
val square : (x:Long)=>Long
            = (x:Long)=>x*x;
assert square(5) == 25;
```

A term of form (6), such as $M[T](e)$, is an instance of a parameterized type definition. Such types may be parameterized by both types and values. This is described in §4.4.

Example: `Array[Long](1)` is the type of one-dimensional arrays of integers. It has one type parameter giving the type of element, here `Long`. It has one value parameter

³In practice, most people would use an `Rail` rather than making a new `Pair` class.

giving the number of dimensions, here 1. `Region(1)` is the type of one-dimensional regions of points (§16.4.1).

In the function types (6), the variable names are bound. As with all bound variables in X10, they can be renamed. So, for example, the types `(x:Long)=>Long{self!=x}` and `(y:Long)=>Long{self!=y}` are equivalent, as they differ by nothing but the names of bound variables. This is more visible with types than with, say, methods or functions, because we can test equality of types.

Furthermore, if a variable `x` does not appear anywhere in a function type `F` save as an argument name, it (and its “`:`”) can be omitted. E.g., the types `(x:Long)=>Long` and `(Long)=>Long` are equivalent.

Example:

```
val f : (x:Long)=>Long{self!=x} = (x:Long) => (x+1) as Long{self!=x};
val g : (y:Long)=>Long{self!=y} = f;
val t : (x:Long)=>Long           = (x:Long) => x;
val u : ( Long )=>Long          = t;
```

A term of form (7), `T{c}`, is a type whose values are the values of type `T` for which the constraint `c` is true. This is described in §4.5.

Example: A variable of class `Point`, unconstrained, can contain null:

```
var gotNPE: Boolean = false;
val p : Point = null;
try {
    val q = p * 2; // method invocation, NPE
}
catch(NullPointerException) {
    gotNPE = true;
}
assert gotNPE;
```

A suitable constraint on that type will prevent a null from ever being assigned to the variable. The variable `self`, in a constraint, refers to the value being constrained, so the constraint `self != null` means “which is not null”. So, adding a `{self!=null}` constraint to `Point` results in a compile-time error, rather than a runtime null pointer exception.

```
// ERROR: p : Point{self!=null} = null;
```

4.2 Unit Types: Classes, Struct Types, and Interfaces

Most X10 computation manipulates values via the *unit* types: classes, struct types, and interfaces. These types share a great deal of structure, though there are important differences.

4.2.1 Class types

A *class declaration* declares a *class type* (§8), giving its name, behavior, and data. It may inherit from zero or one *parent class*. It may also implement zero or more interfaces, each one of which becomes a supertype of it.

Example: *The Position class below could describe the position of a slider control. The example method uses Position as a type. Position is a subtype of the type Poser.*

```
interface Poser {
    def pos():Long;
}
class Position implements Poser {
    private var x : Long = 0;
    public def move(dx:Long) { x += dx; }
    public def pos() : Long = x;
    static def example() {
        var p : Position;
    }
}
```

The null value, represented by the literal `null`, is a value of every class type C. The type whose values are all instances of C except `null` can be defined as `C{self != null}`.

4.2.2 Struct Types

A *struct declaration* (§9) introduces a *struct type* containing all instances of the struct. Struct types can include nearly all the features that classes have. They can implement interfaces, which become their supertypes just as for classes; but they do not have superclasses, and cannot extend anything.

Example: *The Coords struct gives an immutable position in 3-space. It is used as a type in example():*

```
struct Position {
    public val x:Double; public val y:Double; public val z:Double;
    def this(x:Double, y:Double, z:Double) {
        this.x = x; this.y = y; this.z = z;
    }
    static def example(p: Position, q: Rail[Position]) {
        var r : Position = p;
    }
}
```

4.2.3 Interface types

An *interface declaration* (§7) defines an *interface type*, specifying a set of instance method signatures and property method signatures which must be provided by any container declared to implement the interface. They can also declare static `val` fields, which are provided to all units implementing or extending the interface. They do not have code, and cannot implement anything. An interface may extend multiple interfaces. Each interface it extends becomes one of its superclasses.

Example: *Named and Mobile are interfaces, each specifying a single method. Person and NamedPoint are subtypes of both of them. They are used as types in the example method.*

```
interface Named {
    def name():String;
}
interface Mobile {
    def where():Long;
    def move(howFar:Long):void;
}
interface NamedPoint extends Named, Mobile {}
class Person implements Named, Mobile {
    var name:String; var pos: Long;
    public def name() = this.name;
    public def move(howFar:Long) { pos += howFar; }
    public def where() = this.pos;
    public def example(putAt:Mobile) {
        this.pos = putAt.where();
    }
}
```

4.2.4 Properties

Classes, interfaces, and structs may have *properties*, specified in parentheses after the type name. Properties are much like public `val` instance fields. They have certain restrictions on their use, however, which allows the compiler to understand them much better than other public `val` fields. In particular, they can be used in types. *E.g.*, the number of elements in a rail is a property of the rail, and an X10 program can specify that two rails have the same number of elements.

Example: *The following code declares a class named Coords with properties x and y and a move method. The properties are bound using the property statement in the constructor.*

```
class Coords(x: Long, y: Long) {
    def this(x: Long, y: Long) :
        Coords{self.x==x, self.y==y} {
```

```

    property(x, y);
}

def move(dx: Long, dy: Long) = new Coords(x+dx, y+dy);
}

```

Properties of `self` can be used in constraints. This places certain restrictions on how properties can be used, but allows a great deal of compile-time constraint checking. For a simple example, `new Coords(0, 0)` is known to be an instance of `Coords{self.x==0}`. Details of this substantial topic are found in §4.5.

4.3 Type Parameters and Generic Types

A class, interface, method, or type definition may have type parameters. Type parameters can be used as types, and will be bound to types on instantiation. For example, a generic stack class may be defined as `Stack[T]{...}`. Stacks can hold values of any type; e.g., `Stack[Long]` is a stack of longs, and `Stack[Point {self!=null}]` is a stack of non-null Points. Generics *must* be instantiated when they are used: `Stack`, by itself, is not a valid type. Type parameters may be constrained by a guard on the declaration (§4.4, §8.4.6, §10.3).

A *generic class* (or struct, interface, or type definition) is a class (resp. struct, interface, or type definition) declared with $k \geq 1$ type parameters. A generic class (or struct, interface, or type definition) can be used to form a type by supplying k types as type arguments within `[...]`.

Example: `Bottle[T]` is a generic class. A `Bottle[T]` can hold a value of type `T`; the variable `yup` in `example()` is of type `Bottle[Boolean]` and thus can hold a Boolean. However, `Bottle` alone is not a type.⁴

```

class Bottle[T] {
    var contents : T;
    public def this(t:T) { contents = t; }
    public def putIn(t:T) { contents = t; }
    public def get() = contents;
    static def example() {
        val yup : Bottle[Boolean] = new Bottle[Boolean](true);
        //ERROR: var nope : Bottle = null;
    }
}

```

A class (whether generic or not) may have generic methods.

Example: `NonGeneric` has a generic method `first[T](x:List[T])`. An invocation of such a method may supply the type parameters explicitly (e.g., `first[Long](z)`).

⁴By contrast, in Java, the equivalent of `Bottle` alone would be a type, via type erasure of generics.

In certain cases (e.g., `first(z)`) type parameters may be omitted and are inferred by the compiler (§4.12).

```
class NonGeneric {
    static def first[T](x>List[T]):T = x(0);
    def m(z>List[Long]) {
        val f = first[Long](z);
        val g = first(z);
        return f == g;
    }
}
```

Limitation: X10 v2.4's C++ back end requires generic methods to be static or final; the Java back end can accomodate generic instance methods as well.

4.4 Type definitions

A type definition can be thought of as a type-valued function, mapping type parameters and value parameters to a concrete type.

TypeDefDecln ::= Mods[?] type Id TypeParams[?] Guard[?] = Type ; (20.170)

| Mods[?] type Id TypeParams[?] (FormalList) Guard[?] = Type ;

TypeParams ::= [TypeParamList] (20.176)

Formals ::= (FormalList[?]) (20.80)

Guard ::= DepParams (20.83)

During type-checking the compiler replaces the use of such a defined type with its body, substituting the actual type and value parameters in the call for the formals. This replacement is performed recursively until the type no longer contains a defined type or a predetermined compiler limit is reached (in which case the compiler declares an error). Thus, recursive type definitions are not permitted.

Type definitions are considered applicative and not generative – they do not define new types, only aliases for existing types.

Type definitions may have guards: an invocation of a type definition is illegal unless the guard is satisfied when formal types and values are replaced by the actual parameters.

Type definitions may be overloaded: two type definitions with the same name are permitted provided that they have a different number of type parameters or different number or type of value parameters. The rules for type definition resolution are identical to those for method resolution.

However, `T()` is not allowed. If there is an argument list, it must be nonempty. This avoids a possible confusion between type `T = ...` and type `T() =`

A type definition for a type `T` can appear:

- As a top-level definition in a file named `T.x10`; or

- As a static member in a container definition; or
- In a block statement.

Use of type definitions in constructor invocations If a type definition has no type parameters and no value parameters and is an alias for a container type, a `new` expression may be used to create an instance of the class using the type definition's name. Similarly, a parameterless alias for an interface can be used to construct an instance of an anonymous class. Given the following type definition:

```
type A = C[T1, ..., Tk]{c};
```

where `C[T1, ..., Tk]` is a class type, a constructor of `C` may be invoked with `new A(e1, ..., en)`, if the invocation `new C[T1, ..., Tk](e1, ..., en)` is legal and if the constructor return type is a subtype of `A`.

Example: *The names of the class `Cont[X]` and the interface `Inte[X]` can be used to create an object `a` of type `Cont[Long]`, and an object `b` which implements `Inte[Long]`. The two types may be given aliases `A` and `B`, which may then be used in more compact expressions to construct objects `aa` and `bb` of the same types.*

```
class ConstructorExample {
    static class Cont[X]{
        static interface Inte[X]{
            def meth():X;
        }
        public static def example() {
            val a = new Cont[Long]();
            val b = new Inte[Long](){public def meth()=3;};
            type A = Cont[Long];
            val aa = new A();
            type B = Inte[Long];
            val bb = new B(){public def meth()=4;};
        }
    }
}
```

Automatically imported type definitions The collection of type definitions in `x10.lang..` is automatically imported in every compilation unit.

4.4.1 Motivation and use

The primary purpose of type definitions is to provide a succinct, meaningful name for complex types and combinations of types. With value arguments, type arguments, and constraints, the syntax for X10 types can often be verbose. For example, a non-null list of non-null strings is

```
List[String{self!=null}]{self!=null}.
```

We could name that type:

```
static type LnSn = List[String{self!=null}]{self!=null};
```

Or, we could abstract it somewhat, defining a type constructor `Nonnull[T]` for the type of T's which are not null:

```
class Example {
    static type Nonnull[T]{T isref} = T{self!=null};
    var example : Nonnull[Example] = new Example();
}
```

Type definitions can also refer to values, in particular, inside constraints. The type of n-element `Array[Long](1)s` is `x10.regionarray.Array[Long]{self.rank==1 && self.size == n}` but it is often convenient to give a shorter name:

```
type Vec(n:Long) = x10.regionarray.Array[Long]{self.rank==1, self.size == n};
var example : Vec(78L);
```

The following examples are legal type definitions,

```
import x10.util.*;
import x10.regionarray.*;
class TypeExamples {
    static type StringSet = Set[String];
    static type MapToList[K,V] = Map[K,List[V]];
    static type Long(x: Long) = Long{self==x};
    static type Dist(r: Long) = Dist{self.rank==r};
    static type Dist(r: Region) = Dist{self.region==r};
    static type Redund(n:Long, r:Region){r.rank==n}
        = Dist{rank==n && region==r};
}
```

The following code illustrates that type definitions are applicative rather than generative. B and C are both aliases for `String`, rather than new types, and so are interchangeable with each other and with `String`. Similarly, A and Long are equivalent.

```
def someTypeDefs () {
    type A = Long;
    type B = String;
    type C = String;
    a: A = 3;
    b: B = new C("Hi");
    c: C = b + ", Mom!";
}
```

4.5 Constrained types

Basic types, like `Long` and `List[String]`, provide useful descriptions of data.

However, one frequently wants to say more. One might want to know that a `String` variable is not `null`, or that a matrix is square, or that one matrix has the same number of columns as another has rows (so they can be multiplied). In the multicore setting, one might wish to know that two values are located at the same processor, or that one is located at the same place as the current computation.

In most languages, there is simply no way to say and check these things statically. Programmers must make do with comments, `assert` statements, and dynamic tests. X10 programs can do better, with *constraints* on types, and guards on class, method and type definitions.

A constraint expression is a Boolean expression `e` of a quite limited form (§4.5.2). A constraint expression `c` may be attached to a basic type `T`, giving a *constrained type* `T{c}`. The values of type `T{c}` are the values of `T` for which `c` is true. Constraint expressions also serve as guards on methods (§8.4) and functions (§10.3), and invariants on unit types (§8.9).

When constraining a value of type `T`, `self` refers to the object of type `T` which is being constrained. For example, `Long{self == 4}` is the type of `Longs` which are equal to 4 – the best possible description of 4, and a very difficult type to express without using `self`.

Example:

- `Long{self != 0}` is the type of non-zero `Longs`.
- `Long{self == 0}` is the type of `Longs` which are zero.
- `Long{self != 0, self != 1}` is the type of `Longs` which are neither zero nor one.
- `Long{self == 0, self == 1}` is the type of `Longs` which are both zero and one. There are no such values, so it is an empty type.
- `String{self != null}` is the type of non-null strings.
- Suppose that `Matrix` is a matrix class with properties `rows` and `cols`. `Matrix{self.rows == self.cols}` is the type of square matrices.
- One way to say that `a` has the same number of columns that `b` has rows (so that `a*b` is a valid matrix product), one could say:

```
val a : Matrix = someMatrix() ;
var b : Matrix{b.rows == a.cols} ;
```

$T\{e\}$ is a *dependent type*, that is, a type dependent on values. The type T is called the *base type* and e is called the *constraint*. If the constraint is omitted, it is `true`—that is, the base type is unconstrained.

Constraints may refer to immutable values in the local environment:

```
val n = 1;
var p : Point{rank == n};
```

In a `val` variable declaration, the variable itself is in scope in its type, and can be used in constraints.

Example: For example, `val nz : Long{nz != 0} = 1;` declares a non-zero variable `nz`. In this case, `nz` could have been declared as `val nz : Long{self != 0} = 1.`

4.5.1 Examples of Constraints

Example of entailment and subtyping involving constraints.

- $\text{Long}\{\text{self} == 3\} <: \text{Long}\{\text{self} != 14\}$. The only value of $\text{Long}\{\text{self} == 3\}$ is 3. All integers but 14 are members of $\text{Long}\{\text{self} != 14\}$, and in particular 3 is.
- Suppose we have classes `Child <: Person`, and `Person` has a `ssn:Long` property. If `rhys : Child{ssn == 123456789}`, then `rhys` is also a `Person`. `rhys`'s `ssn` field is the same, 123456789, whether `rhys` is regarded as a `Child` or a `Person`. Thus, `rhys : Person{ssn==123456789}` as well. So,

```
Child{ssn == 123456789} <: Person{ssn == 123456789}.
```

- Furthermore, since $123456789 != 55555555$, it is clear that `rhys : Person{ssn != 55555555}`. So,

```
Child{ssn == 123456789} <: Person{ssn != 55555555}.
```

- $T\{e\} <: T$ for any type T . That is, if you have a value v of some base type T which satisfied e , then v is of that base type T (with the constraint ignored).
- If $A <: B$, then $A\{c\} <: B\{c\}$ for every constraint $\{c\}$ for which $A\{c\}$ and $B\{c\}$ are defined. That is, if every A is also a B , and $a : A\{c\}$, then a is an A and c is true of it. So a is also a B (and c is still true of it), so $a : B\{c\}$.

Constraints can be used to express simple relationships between objects, enforcing some class invariants statically. For example, in geometry, a line is determined by two *distinct* points; a `Line` struct can specify the distinctness in a type constraint.⁵

⁵We call them `Position` to avoid confusion with the built-in class `Point`. Also, `Position` is a struct rather than a class so that the non-equality test `start != end` compares the coordinates. If `Position` were a class, `start != end` would check for different `Position` objects, which might have the same coordinates.

```
struct Position(x: Long, y: Long) {}
struct Line(start: Position, end: Position){start != end}
{}
```

Extending this concept, a `Triangle` can be defined as a figure with three line segments which match up end-to-end. Note that the degenerate case in which two or three of the triangle's vertices coincide is excluded by the constraint on `Line`. However, not all degenerate cases can be excluded by the type system; in particular, it is impossible to check that the three vertices are not collinear.

```
struct Triangle
(a: Line,
 b: Line{a.end == b.start},
 c: Line{b.end == c.start && c.end == a.start})
{}
```

The `Triangle` class automatically gets a ternary constructor which takes suitably constrained `a`, `b`, and `c` and produces a new triangle.

A constrained type may be constrained further: the type `S{c}{d}` is the same as the type `S{c,d}`. Multiple constraints are equivalent to conjoined constraints: `S{c,d}` in turn is the same as `S{c && d}`.

4.5.2 Syntax of constraints

Only a few kinds of expressions can appear in constraints. For fundamental reasons of mathematical logic, the more kinds of expressions that can appear in constraints, the harder it is to compute the essential properties of constrained types – in particular, the harder it is to compute `A{c} <: B{d}` or even `E : T{c}`. It doesn't take much to make this basic fact undecidable. In order to make sure that it stays decidable, X10 places stringent restrictions on constraints.

Only the following forms of expression are allowed in constraints.

Value expressions in constraints may be:

1. Literal constants, like `3` and `true`;
2. Accessible, immutable (`val`) variables and parameters;
3. `this`, if the constraint is at a point in the program where `this` is defined, but not in `extends` or `implements` clauses or class invariants;
4. `here`, if the constraint is at a point in the program where `here` is defined;
5. `self`;
6. A field selection expression `t.f`, where `t` is a value expression allowed in constraints, and `f` is a field of `t`'s type. If `t` is `self`, then `f` must be a property, not an arbitrary field.

7. Invocations of property methods, $p(a, b, \dots, c)$ or $a.p(b, c, \dots d)$, where the receiver and arguments must be value expressions acceptable in constraints, as long as the expansion (*viz.*, the expression obtained by taking the body of the definition of p , and replacing the formal parameters by the actual parameters) of the invocation is allowed as a value expression in constraints.

For an expression `self.p` to be legal in a constraint, p must be a property. However terms $t.f$ may be used in constraints (where t is a term other than `self` and f is an immutable field.)

Constraints may be any of the following, where all value expressions are of the forms which may appear in constraints:

1. Equalities $e == f$;
2. Inequalities of the form $e != f$;⁶
3. Conjunctions of Boolean expressions that may appear in constraints (but only in top-level constraints, not in Boolean expressions in constraints);
4. Subtyping and supertyping expressions: $T <: U$ and $T :> U$;
5. Type equalities and inequalities: $T == U$ and $T != U$;
6. Invocations of a property method, $p(a, b, \dots, c)$ or $a.p(b, c, \dots d)$, where the receiver and arguments must be value expressions acceptable in constraints, as long as the expansion of the invocation is allowed as a constraint.
7. Testing a type for a default: $T \text{ haszero}$.

Note that constraints on methods may include private, protected, or package-protected fields. It is possible to have a method whose guard cannot be directly checked, or even whose result type cannot be expressed as a clause in the program, at some call sites. Nonetheless, X10 uses a broader *internal* type representation, not limited by access rules, and can work with fields in types even though those fields cannot be used in executable code.

Example: *This phenomenon can be used to implement a form of compile-type capability checking. We give a minimal example, providing only security by obscurity: users unaware that the `key` method returns the required key will be unable to use the `secret` method. This approach can be strengthened to provide better security.*

The class `Keyed` has a private field `k`. The method `secret(q)` can only be called when `q==k`. In a larger example, `secret` could be some privileged behavior or secret, available only to callers with proper authority.

At the call site in `Snooper`, `keyed.secret()` is called. It can't be called as `keyed.secret(keyed.k)`, because `k` is a private field. It can't be called as `keyed.secret(8)`, even though

⁶Currently inequalities of the form $e < f$ are not supported.

`keyed.k==8`, because there is no proof available that `keyed.k==8` — indeed, at this point in the code, the requirement that `keyed.k==8` cannot even be expressed in X10.

However, the value of `keyed.k` can be retrieved, using `keyed.key()`. The type of `kk` cannot be expressed in Snooper, because it refers to a private field of `keyed`. However, the compiler's internal representation is not bound by the rules of privacy, and can track the fact that `kk` is the same as `keyed.k`. So, the call `keyed.secret(kk)` succeeds.

```
class Keyed {
    private val k : Long;
    public def this(k : Long) {
        this.k = k;
    }
    public def secret(q:Long){q==this.k} = 11;
    public def key():Long{self==this.k} = this.k;
}
class Snooper {
    public static def main(argv: Rail[String]) {
        val keyed : Keyed = new Keyed(8);
        //ERROR: keyed.secret(keyed.k);
        //ERROR: keyed.secret(8);
        val kk = keyed.key();
        keyed.secret(kk);
    }
}
```

Note: Constraints may not contain casts. In particular, comparisons of values of incompatible types are not allowed. If `i:Long`, then `i==0` is allowed as a constraint, but `i==0L` is an error, and `i as Long==0L` is outside of the constraint language.

Semantics of constraints

The logic of constraints is designed to allow a common and important X10 idiom:

```
class Thing(p:Long){}
static def example(){
    var x : Thing{x.p==3} = null;
}
```

That is, `null` must be an instance of `Thing{x.p==3}`. Of course, it cannot be the case that `null.p==3` — nor can it equal anything else. When evaluated at runtime, `null.p` must throw a `NullPointerException` rather than returning any value at all.

So, X10's logic of constraints — *unlike* the logic of runtime — allows `x=null` to satisfy `x.p==3`. Building this logic requires a few definitions.

The property graph, at an instant in an X10 execution, is the graph whose nodes are all objects in existence at that instance, plus `null`, with an edge from `x` to `y` if `x` is

an object with a property whose value is y . The rules for constructors guarantee that property graphs are acyclic, which is crucial for decidability.

As is standard in mathematical logic, we introduce the concept of a *valuation* v , which is a mapping from variable names to their values – in our case, nodes of an X10 property graph. A valuation v can be extended to values to all constraint formulas. The crucial definitions are:

$$\begin{aligned} v(a.b\dots.l.m == n.o\dots.y.z) = \\ \text{a=null} \vee \text{a.b=null} \vee \dots \text{a.b\dots.l=null} \\ \vee \text{n=null} \vee \text{n.o=null} \vee \dots \text{n.o\dots.y=null} \\ \vee v(a).b\dots.l.m = v(n).o\dots.y. \end{aligned}$$

$$\begin{aligned} v(a.b\dots.l.m != n.o\dots.y.z) = \\ \text{a=null} \vee \text{a.b=null} \vee \dots \text{a.b\dots.l=null} \\ \vee \text{n=null} \vee \text{n.o=null} \vee \dots \text{n.o\dots.y=null} \\ \vee v(a).b\dots.l.m \neq v(n).o\dots.y. \end{aligned}$$

For example, $v(a.b==1)$ is true if either $v(a) = \text{null}$ or if $v(a)$ is a container whose b -field is equal to 1. While such a valuation is perfectly well-defined, it has properties that need to be understood in light of the fact that $==$ is *not* mathematical equality.⁷ Given any valuation in which $v(a) = \text{null}$, both $v(a.b==1 \ \&\ a.b==2)$ and $v(a.b==1 \ \&\ a.b!=1)$ are true. This does not contradict logic and mathematics, it does not imply that $v(\text{false})$ is true (it's not), and it does not assert that in X10 there is a number which is both 1 and 2. It simply reflects the fact that, while $==$ is similar to mathematical equality in many respects, it is ultimately a different operation, and in constraints it is given a **null-safe** interpretation.

From this definition of valuation, we define *entailment* in the standard way. Given constraints c and d , we define c *entails* d , sometimes written $c \models d$, if for all valuations v such that $v(c)$ is true, $v(d)$ is also true.

Limitation: Although nearly-contradictory conjunctions like $x.a==1 \ \&\ x.a==2$ entail $x==\text{null}$, X10's constraint solver does not currently use this rule. If you want $x==\text{null}$, write $x==\text{null}$.

Subtyping of constrained types is defined in terms of entailment. $S[S_1, \dots, S_m]\{c\}$ is a subtype of $T[T_1, \dots, T_n]\{d\}$ if $S[S_1, \dots, S_m]$ is a subtype of $T[T_1, \dots, T_n]$ and c entails d .

For examples of constraints and entailment, see (§4.5.1)

4.5.3 Constraint solver: incompleteness and approximation

The constraint solver is sound in that if it claims that c entails d then in fact it is the case that every valuation that satisfies c satisfies d .

⁷No experienced programmer should actually think that $==$ is mathematical equality in any case. It is quite common for two objects to appear identical but not be $==$. X10's discrepancy between the two concepts is orthogonal to the familiar one.

Limitation: X10's constraint solver is incomplete. There are situations in which c entails d but the solver cannot establish it. For instance it cannot establish that $a \neq b \&& a \neq c \&& b \neq c$ entails `false` if a , b , and c are of type `Boolean`. Similarly, although $a.b==1 \&& a.b==2$ entails $a==\text{null}$, the constraint solver does not deduce this fact.

4.5.4 Acyclicity of Properties

To ensure that typechecking is decidable, X10 requires that the graph whose nodes are types, with edges from types to the properties of those types, be *acyclic*. This is often stated as “properties are acyclic.” That is, given a container type T , T cannot have a property of type T , nor a property which has a property of type T , nor a property which has a property with a property of type T , etc.

Example: *The following is forbidden by the acyclicity requirement, as `ERRORList[T]` would have a property, `tail`, which is also an `ERRORList[T]`.*

```
class ERRORList[T](head:T, tail: ERRORList[T]) {}
```

Without this restriction, typechecking becomes undecidable.

4.5.5 Limitation: Generics and Constraints at Runtime

The X10 runtime does not maintain a representation of constraints as part of the runtime representation of a type. While there various approaches which could be used, they would require far higher prices in space or time than they are worth. A representation suitable for one use of types (such as keeping a closure for testing membership in the type) is unsuitable for others (such as determining if one type is a subtype of another). Furthermore, it would be necessary to compute entailment at runtime, which is currently impractical.

Rather than pay the runtime costs for keeping and manipulating constraints (which can be considerable), X10 omits them. However, this renders certain type checks uncertain: X10 needs some information at runtime, but does not have it. In particular, **casts to instances of generic types, and to type variables, are potentially troublesome**.

Example: *The following code illustrates the dangers of casting to generic types. It constructs a rail `a` of `Long{self==3}`'s – integers which are statically known to be 3. The only number that can be stored into `a` is 3. Then it tricks the ocompiler into thinking that it is a rail of `Long`, without restriction on the elements, giving it the name `b` at that type. The cast `aa` as `Rail[Long]` is a cast to an instance of a generic type, which is the problem.*

But, it can store any `Long` into the elements of `b`, thereby violating the invariant that all the elements of the rail are 3. This could lead to program failures, as illustrated by the failing assertion.

With the `-VERBOSE` compiler option, X10 prints a warning about the declaration of `b`.

```

val a = new Rail[Long{self==3}](10, 3);
// a(0) = 1; would be illegal
a(0) = 3; // LEGAL
val aa = a as Any;
val b = aa as Rail[Long]; // WARNED with -VERBOSE
b(0) = 1;
val x : Long{self==3} = a(0);
assert x == 3 : "This fails at runtime.";

```

Since constraints are not preserved at runtime, `instanceof` and `as` cannot pay attention to them. When types are used generically, they may not behave as one would expect were one to imagine that their constraints were kept. Specifically, constraints at runtime are, in effect, simply replaced by `true`.

Example: *The following code defines generic methods `inst` and `cast`, which look like generic versions of `instanceof` and `as`. The `example()` code shows that `inst` and `cast` behave quite differently from `instanceof` and `as`, due to the loss of constraint information.*

The first section of asserts shows the behavior of `instanceof` and `at`. We have a value `pea`, such that `pea.p==1`. It behaves as if its `p` field were 1: it answers `true` to `self.p==1`, and `false` to `self.p==2`. This is entirely as desired.

The following section of `assert` and `val` statements does the analogous thing, but using the generic methods `inst` and `cast` rather than the built-in operations `instanceof` and `cast`. `pea` answers `true` to `inst` checks concerning both `Pea{p==1}` and `Pea{p==2}`, and can be `cast()` into both these types. This behavior is not what one would expect from runtime types that keep constraint information. It is, however, precisely what one would expect from runtime types that have their constraints replaced by `true`.

The `cast2` line shows how to use this fact to violate the constraint system at runtime. This dynamic cast produces an object of type `Pea{p==2}` for which `p!=2`.

Note that the `-VERBOSE` compiler flag will produce a warning that `cast` is unsound.

```

class Generic {
    public static def inst[T](x:Any):Boolean = x instanceof T;
    // With -VERBOSE, the following line gets a warning
    public static def cast[T](x:Any):T      = x as T;
}
class Pea(p:Long) {}
class Example{
    static def example() {
        val pea : Pea = new Pea(1);
        // These are what you'd expect:
        assert (pea instanceof Pea{p==1});
        assert (pea as Pea{p==1}).p == 1;
        assert ! (pea instanceof Pea{p==2});
        // 'val x = pea as Pea{p==2};'
        // throws a FailedDynamicCheckException.
    }
}

```

```

// But the genericized versions don't do the same thing:
assert Generic.inst[Pea{p==1}](pea);
assert Generic.inst[Pea{p==2}](pea);
// No exception here!
val cast1: Pea{p==1} = Generic.cast[Pea{p==1}](pea);
val cast2: Pea{p==2} = Generic.cast[Pea{p==2}](pea);
assert cast2.p == 1;
assert !(cast2 instanceof Pea{p==2});
}
}

```

While in some cases it would be possible to keep constraints around at runtime and operate efficiently on them, in other cases it would not.

4.6 Function types

$$\text{FunctionType} ::= \text{TypeParams}^? (\text{FormalList}^?) \text{Guard}^? \Rightarrow \text{Type} \quad (20.82)$$

For every sequence of types T_1, \dots, T_n , T , and n distinct variables x_1, \dots, x_n and constraint c , the expression $(x_1:T_1, \dots, x_n:T_n)\{c\} \Rightarrow T$ is a *function type*. It stands for the set of all functions f which can be applied to a list of values (v_1, \dots, v_n) provided that the constraint $c[v_1, \dots, v_n, p/x_1, \dots, x_n]$ is true, and which returns a value of type $T[v_1, \dots, v_n/x_1, \dots, x_n]$. When c is true, the clause $\{c\}$ can be omitted. When x_1, \dots, x_n do not occur in c or T , they can be omitted. Thus the type $(T_1, \dots, T_n) \Rightarrow T$ is actually shorthand for $(x_1:T_1, \dots, x_n:T_n)\{\text{true}\} \Rightarrow T$, for some variables x_1, \dots, x_n .

Limitation: Constraints on closures are not supported. They parse, but are not checked.

X10 functions, like mathematical functions, take some arguments and produce a result. X10 functions, like other X10 code, can change mutable state and throw exceptions. Closures (§10) are of function type – and so are rails and arrays.

Example: Typical functions are the reciprocal function:

```
val recip = (x : Double) => 1/x;
```

and a function which increments element i of a rail r , or throws an exception if there is no such element, where, for the sake of example, we constrain the type of i to avoid one of the many longs which are not possible subscripts:

```
val inc = (r:Rail[Long], i: Long{i != r.size}) => {
    if (i < 0 || i >= r.size) throw new DoomExn();
    r(i)++;
};
```

In general, a function type needs to list the types T_i of all the formal parameters, and their distinct names x_i in case other types refer to them; a constraint c on the function as a whole; a return type T .

```
(x1: T1, ..., xn: Tn) {c} => T
```

The names of the formal parameters, x_i , are bound in the type. As usual with bound variables, they can be given new names without changing the meaning of the type. In particular, the names of formals in a function type do not need to be the same as the names in the function in a value of that type.

Example: *The type of id uses the bound variable x. The type of ie uses the bound variable z, but is otherwise identical to that of id. The two types are the same, as shown by the assignment of id to ie. Also, id's type uses x, and id's value uses y.*

```
val id : (x:Long) => Long{self==x}
    = (y:Long) => y;
val ie : (z:Long) => Long{self==z}
    = id;
```

Limitation: Function types differing only in the names of bound variables may wind up being considered different in X10 v2.2, especially if the variables appear in constraints.

The formal parameter names are in scope from the point of definition to the end of the function type—they may be used in the types of other formal parameters and in the return type. Value parameters names may be omitted if they are not used; the type of the reciprocal function can be written as (Double)=>Double.

A function type is covariant in its result type and contravariant in each of its argument types. That is, let $S_1, \dots, S_n, S, T_1, \dots, T_n, T$ be any types satisfying $S_i <: T_i$ and $S <: T$. Then $(x_1:T_1, \dots, x_n:T_n) \{c\} => S$ is a subtype of $(x_1:S_1, \dots, x_n:S_n) \{c\} => T$.

A class or struct definition may use a function type

```
F = (x1:T1, ..., xn:Tn) {c} => T
```

in its implements clause; this is equivalent to implementing an interface requiring the single operator

```
public operator this(x1:T1, ..., xn:Tn) {c}: T
```

Similarly, an interface definition may specify a function type F in its extends clause. Values of a class or struct implementing F can be used as functions of type F in all ways. In particular, applying one to suitable arguments calls the apply method.

Limitation: A class or struct may not implement two different instantiations of a generic interface. In particular, a class or struct can implement only one function type.

A function type F is not a class type in that it does not extend any type or implement any interfaces, or support equality tests. F may be implemented, but not extended, by a class or function type. Nor is it a struct type, for it has no predefined notion of equality.

4.7 Default Values

Some types have default values, and some do not. Default values are used in situations where variables can legitimately be used without having been initialized; types without default values cannot be used in such situations. For example, a field of an object `var x:T` can be left uninitialized if T has a default value; it cannot be if T does not. Similarly, a transient (§8.2.3) field `transient val x:T` is only allowed if T has a default value.

Default values, or lack of them, is defined thus:

- The fundamental numeric types (`Int`, `UInt`, `Long`, `ULong`, `Short`, `UShort`, `Byte`, `UByte`, `Float`, `Double`) all have default value `0`.
- `Boolean` has default value `false`.
- `Char` has default value `'\0'`.
- If every field of a struct type T has a default value, then T has a default value. If any field of T has no default value, then T does not. (§9.7)
- A function type has a default value of `null`.
- A class type has a default value of `null`.
- The constrained type `T{c}` has the same default value as T if that default value satisfies c. If the default value of T doesn't satisfy c, then `T{c}` has no default value.

Example: `var x: Long{x != 4}` has default value 0, which is allowed because `0 != 4` satisfies the constraint on x. `var y : Long{y==4}` has no default value, because 0 does not satisfy `y==4`. The fact that `Long{y==4}` has precisely one value, viz. 4, doesn't matter; the only candidate for its default value, as for any subtype of `Long`, is 0. y must be initialized before it is used.

The predicate `T haszero` tells if the type T has a default value. `haszero` may be used in constraints.

Example: The following code defines a sort of cell holding a single value of type T. The cell is initially empty – that is, has T's zero value – but may be filled later.

```
class Cell0[T]{T haszero} {
    public var contents : T;
    public def put(t:T) { contents = t; }
}
```

The built-in type `Zero` has the method `get[T]()` which returns the default value of type T.

Example: As a variation on a theme of `Cell0`, we define a class `Cell1[T]` which can be initialized with a value of an arbitrary type T, or, if T has a default value, can be created with the default value. Note that `T haszero` is a constraint on one of the constructors, not the whole type:

```
class Cell1[T] {
    public var contents: T;
    def this(t:T) { contents = t; }
    def this(){T haszero} { contents = Zero.get[T](); }
    public def put(t:T) {contents = t;}
}
```

4.8 Annotated types

Any X10 type may be annotated with zero or more user-defined *type annotations* (§17).

Annotations are defined as (constrained) interface types and are processed by compiler plugins, which may interpret the annotation symbolically.

A type T is annotated by interface types A_1, \dots, A_n using the syntax $@A_1 \dots @A_n \ T$.

4.9 Subtyping and type equivalence

Intuitively, type T_1 is a subtype of type T_2 , written $T_1 <: T_2$, if every instance of T_1 is also an instance of T_2 . For example, `Child` is a subtype of `Person` (assuming a suitably defined class hierarchy): every child is a person. Similarly, `Long{self != 0}` is a subtype of `Long` – every non-zero integer is an integer.

This section formalizes the concept of subtyping. Subtyping of types depends on a *type context*, *viz.*.. a set of constraints on type parameters and variables that occur in the type. For example:

```
class ConsTy[T,U] {
    def upcast(t:T){T <: U} :U = t;
}
```

Inside `upcast`, T is constrained to be a subtype of U , and so $T <: U$ is true, and t can be treated as a value of type U . Outside of `upcast`, there is no reason to expect any relationship between them, and $T <: U$ may be false. However, subtyping of types that have no free variables does not depend on the context. `Long{self != 0} <: Long` is always true.

Limitation: Subtyping of type variables does not work under all circumstances in the X10 2.2 implementation.

- **Reflexivity:** Every type T is a subtype of itself: $T <: T$.
- **Transitivity:** If $T <: U$ and $U <: V$, then $T <: V$.

- **Direct Subclassing:** Let \vec{X} be a (possibly empty) vector of type variables, and \vec{Y}, \vec{Y}_i be vectors of type terms over \vec{X} . Let \vec{T} be an instantiation of \vec{X} , and \vec{U}, \vec{U}_i the corresponding instantiation of \vec{Y}, \vec{Y}_i . Let c be a constraint, and c' be the corresponding instantiation. We elide properties, and interpret empty vectors as absence of the relevant clauses. Suppose that C is declared by one of the forms:

1. class $C[\vec{X}]\{c\}$ extends $D[\vec{Y}]\{d\}$
implements $I_1[\vec{Y}_1]\{i_1\}, \dots, I_n[\vec{Y}_n]\{i_n\}\{$
2. interface $C[\vec{X}]\{c\}$ extends $I_1[\vec{Y}_1]\{i_1\}, \dots, I_n[\vec{Y}_n]\{i_n\}\{$
3. struct $C[\vec{X}]\{c\}$ implements $I_1[\vec{Y}_1]\{i_1\}, \dots, I_n[\vec{Y}_n]\{i_n\}\{$

Then:

1. $C[\vec{T}] <: D[\vec{U}]\{d\}$ for a class
2. $C[\vec{T}] <: I_i[\vec{U}_i]\{i_i\}$ for all cases.
3. $C[\vec{T}] <: C[\vec{T}]\{c'\}$ for all cases.

- **Function types:**

$$(x_1: T_1, \dots, x_n: T_n)\{c\} \Rightarrow T$$

is a subtype of

$$(x'_1: T'_1, \dots, x'_n: T'_n)\{c'\} \Rightarrow T'$$

if:

1. Each $T_i <: T'_i$;
2. $c[x'_1, \dots, x'_n / x_1, \dots, x_n] \text{ entails } c'$;
3. $T' <: T$;

- **Constrained types:** $T\{c\}$ is a subtype of $T\{d\}$ if c entails d .
- **Any:** Every type T is a subtype of `x10.lang.Any`.
- **Type Variables:** Inside the scope of a constraint c which entails $A <: B$, we have $A <: B$. e.g., `upcast` above.

Two types are *equivalent*, $T == U$, if $T <: U$ and $U <: T$.

4.10 Common ancestors of types

There are several situations where X10 must find a type T that describes values of two or more different types. This arises when X10 is trying to find a good type for:

- Conditional expressions, like `test ? 0 : "non-zero"` or even
`test ? 0 : 1;`

- Rail construction, like `[0, "non-zero"]` and `[0,1]`;
- Functions with multiple returns, like

```
def f(a:Long) {
    if (a == 0) return 0;
    else return "non-zero";
}
```

In some cases, there is a unique best type describing the expression. For example, if `B` and `C` are direct subclasses of `A`, `pick` will have return type `A`:

```
static def pick(t:Boolean, b:B, c:C) = t ? b : c;
```

However, in many common cases, there is no unique best type describing the expression. For example, consider the expression `E`

```
b ? 0 : 1 // Call this expression E
```

The best type of `0` is `Long{self==0}`, and the best type of `1` is `Long{self==1}`. Certainly `E` could be given the type `Long`, or even `Any`, and that would describe all possible results. However, we actually know more. `Long{self != 2}` is a better description of the type of `E`—certainly the result of `E` can never be `2`. `Long{self != 2, self != 3}` is an even better description; `E` can't be `3` either. We can continue this process forever, adding integers which `E` will definitely not return and getting better and better approximations. (If the constraint sublanguage had `||`, we could give it the type `Long{self == 0 || self == 1}`, which would be nearly perfect. But `||` makes typechecking far more expensive, so it is excluded.) No X10 type is the best description of `E`; there is always a better one.

Similarly, consider two unrelated interfaces:

```
interface I1 {}
interface I2 {}
class A implements I1, I2 {}
class B implements I1, I2 {}
class C {
    static def example(t:Boolean, a:A, b:B) = t ? a : b;
}
```

`I1` and `I2` are both perfectly good descriptions of `t ? a : b`, but neither one is better than the other, and there is no single X10 type which is better than both. (Some languages have *conjunctive types*, and could say that the return type of `example` was `I1 && I2`. This, too, complicates typechecking.)

So, when confronted with expressions like this, X10 computes *some* satisfactory type for the expression, but not necessarily the *best* type. X10 provides certain guarantees about the common type `V{v}` computed for `T{t}` and `U{u}`:

- If $T\{t\} == U\{u\}$, then $V\{v\} == T\{t\} == U\{u\}$. So, if X10's algorithm produces an utterly untenable type for $a ? b : c$, and you want the result to have type $T\{t\}$, you can (in the worst case) rewrite it to

$$a ? b \text{ as } T\{t\} : c \text{ as } T\{t\}$$
- If $T == U$, then $V == T == U$. For example, X10 will compute the type of $b ? \emptyset : 1$ as $\text{Long}\{c\}$ for some constraint c —perhaps simply picking $\text{Long}\{\text{true}\}$, *viz.*, Long .
- X10 preserves place information about `GlobalRefs`, because it is so important. If both t and u entail `self.home==p`, then v will also entail `self.home==p`.
- X10 similarly preserves nullity information. If t and u both entail $x == \text{null}$ or $x != \text{null}$ for some variable x , then v will also entail it as well.
- The computed upper bound of function types with the *same* argument types is found by computing the upper bound of the result types. If $T = (T_1, \dots, T_n) \Rightarrow T'$ and $U = (T_1, \dots, T_n) \Rightarrow U'$, and V' is the computed upper bound of T' and U' , then the computed upper bound of T and U is $U = (T_1, \dots, T_n) \Rightarrow V'$. (But, if the argument types are different, the computed upper bound may be `Any`.)

4.11 Fundamental types

Certain types are used in fundamental ways by X10.

4.11.1 The interface Any

It is quite convenient to have a type which all values are instances of; that is, a supertype of all types.⁸ X10's universal supertype is the interface `Any`.

```
package x10.lang;
public interface Any {
    def toString():String;
    def typeName():String;
    def equals(Any):Boolean;
    def hashCode():Long;
}
```

`Any` provides a handful of essential methods that make sense and are useful for everything. `a.toString()` produces a string representation of `a`, and `a.typeName()` the string representation of its type; both are useful for debugging. `a.equals(b)` is the programmer-overridable equality test, and `a.hashCode()` an integer useful for hashing.

⁸Java, for one, suffers a number of inconveniences because some built-in types like `long` and `char` aren't subtypes of anything else.

4.12 Type inference

X10 v2.4 supports limited local type inference, permitting certain variable types and return types to be elided. It is a static error if an omitted type cannot be inferred or uniquely determined. Type inference does not consider coercions.

4.12.1 Variable declarations

The type of a `val` variable declaration can be omitted if the declaration has an initializer. The inferred type of the variable is the computed type of the initializer. For example, `val seven = 7;` is identical to

```
val seven: Long{self==7} = 7;
```

Note that type inference gives the most precise X10 type, which might be more specific than the type that a programmer would write.

Limitation: At the moment, `var` declarations may not have their types elided in this way.

4.12.2 Return types

The return type of a method can be omitted if the method has a body (*i.e.*, is not `abstract` or `native`). The inferred return type is the computed type of the body. In the following example, the return type inferred for `isTriangle` is `Boolean{self==false}`

```
class Shape {
    def isTriangle() = false;
}
```

Note that, as with other type inference, methods are given the most specific type. In many cases, this interferes with subtyping. For example, if one tried to write:

```
class Triangle extends Shape {
    def isTriangle() = true;
}
```

the compiler would reject this program for attempting to override `isTriangle()` by a method with the wrong type, *viz.*, `Boolean{self==true}`. In this case, supply the type that is actually intended for `isTriangle`:

```
def isTriangle(): Boolean = false;
```

The return type of a closure can be omitted. The inferred return type is the computed type of the body.

The return type of a constructor can be omitted if the constructor has a body. The inferred return type is the enclosing class type with properties bound to the arguments

in the constructor's `property` statement, if any, or to the unconstrained class type. For example, the `Spot` class has two constructors, the first of which has inferred return type `Spot{x==0}` and the second of which has inferred return type `Spot{x==xx}`.

```
class Spot(x:Long) {
    def this() {property(0);}
    def this(xx: Long) { property(xx); }
}
```

A method or closure that has expression-free `return` statements (`return;` rather than `return e;`) is said to be a `void` method or closure. `void` is not a type; there are no `void` values, nor can `void` be used as the argument of a generic type. However, `void` takes the syntactic place of a type in a few contexts. A `void` method can be specified by `def m():void`, and similarly for a closure:

```
def m():void {return;}
val f : () => void = () => {return;};
```

By a convenient abuse of language, `void` is sometimes lumped in with types; *e.g.*, we may say “return type of a method” rather than the formally correct but rather more awkward “return type of a method that is not a `void` method”. Despite this informal usage, `void` is not a type and cannot be used as the value of a type parameter. For example, given

```
static def eval[T] (f:()=>T):T = f();
```

The call `eval[void](f)` does *not* typecheck. There is no way in X10 to write a generic function which works with both functions which return a value and functions which do not. In such cases it may be convenient to define a type `Unit` thus:

```
struct Unit{}
```

Functions whose return type is `Unit` may simply return the expression `Unit()` which evaluates to the unique value of type `Unit`. (By definition of equality of structures `Unit()==Unit()`.)

X10 preserves known information when computing return types. A constraint on a method induces a corresponding constraint on its return type.

Example: *In the following code, the type inferred for `x` is `Numb{self.p==n, n!=0, self!=null}`. In particular, the conjunct `n != 0` is preserved from the cast of `n` to `Long{self != 0}`.*

```
class Numb(p:Long){
    static def dup(n:Long){n != 0} = new Numb(n);
    public static def example(n:Long) {
        val x = dup(n as Long{self != 0});
        val y : Numb{self.p==n, n!=0, self!=null} = x;
    }
}
```

4.12.3 Inferring Type Arguments

A call to a polymorphic method may omit the explicit type arguments. X10 will compute a type from the types of the actual arguments. Failure of the compiler to infer unique types for omitted type arguments is a compile-time error. For instance, given the method definition `def m[T] () { ... }`, an invocation `m()` is considered a compile-time error. The compiler has no idea what T the programmer intends.

Example: Consider the following method, which chooses one of its arguments. (A more sophisticated one might sometimes choose the second argument, but that does not matter for the sake of this example.)

```
static def choose[T](a: T, b: T): T = a;
```

The type argument T can always be supplied: `choose[Long](1, 2)` picks an integer, and `choose[Any](1, "yes")` picks a value that might be an integer or a string. However, the type argument can be elided. Suppose that Sub <: Super; then the following compiles:

```
static def choose[T](a: T, b: T): T = a;
static val j : Any = choose("string", 1);
static val k : Super = choose(new Sub(), new Super());
```

The type parameter doesn't need to be the type of a variable. It can be found inside of the type of a variable; X10 can extract it.

Example: The first method below returns the first element of a rail. The type parameter T represents the type of the rail's elements. There is no parameter of type T. There is one of type `Rail[String]{length==3}`. When doing type inference, the compiler is able to infer that T should be instantiated to String:

```
static def first[T](x: Rail[T]) = x(0);
static def example() {
    val ss <: Rail[String] = ["X10", "Java", "C++"]; // ok
    val s1 <: String = first(ss); // ok
    assert s1.equals("X10");
}
```

Sketch of X10 Type Inference for Method Calls

When the X10 compiler sees a method call

$$a.m(b_1, \dots, b_n)$$

and attempts to infer type parameters to see if it could be a use of a method

$$\text{def } m[X_1, \dots, X_t](y_1: S_1, \dots, y_n: S_n),$$

it reasons as follows.

Let

T_i be the type of b_i

Then, the compiler is seeking a set B of type bindings

$$B = \{ X_1 = U_1, \dots, X_t = U_t \}$$

such that $T_i <: S_i^*$ for $1 \leq i \leq n$, where S^* is S with each type variable X_j replaced by the corresponding U_j . If it can find such a B , it has a usable choice of type arguments and can do the type inference. If it cannot find B , then it cannot do type inference. (Note that X10's type inference algorithm is incomplete – there may be such a B that X10 cannot find. If this occurs in your program, you will have to write down the type arguments explicitly.)

Let B_0 be the set $\{T_i <: S_i | 1 \leq i \leq n\}$. Let B_{n+1} be B_n with one element $F <: G$ or $F = G$ removed, and $Strip(F <: G)$ or $Strip(F = G)$, where $Strip$ is defined below, added. Repeat this until B_n consists entirely of comparisons with type variables (viz., $Y_j = U$, $Y_j <: U$, and $Y_j :> U$), or until some n exceeds a predefined compiler limit.

The candidate inferred types may be read off of B_n . The guessed binding for X_j is:

- If there is an equality $X_j = W$ in B_n , then guess the binding $X_j = W$. Note that there may be several such equalities with different choices of W ; pick any one. If the chosen binding does not equal the others, the candidate binding will be rejected later and type inference will fail.
- Otherwise, if there is one or more upper bounds $X_j <: V_k$ in B_n , guess the binding $X_j = V_+$, where V_+ is the computed lower bound of all the V_k 's.
- Otherwise, if there is one or more lower bounds $R_k <: X_j$, guess that $X_j = R_+$, where R_+ is the computed upper bound of all the R_k 's.

If this does not yield a binding for some variable X_j , then type inference fails. Furthermore, if every variable X_j is given a binding U_j , but the bindings do not work — that is, if $a.m[U_1, \dots, U_t](b_1, \dots, b_n)$ is not a well-typed call of the original method $\text{def } m[X_1, \dots, X_t](y_1: S_1, \dots, y_n: S_n)$ — then type inference also fails.

Computation of the Replacement Elements Given a type relation r of the form $F <: G$ or $F = G$, we compute the set $Strip(r)$ of replacement constraints. There are a number of cases; we present only the interesting ones.

- If F has the form $F'\{c\}$, then $Strip(r)$ is defined to be $F' = G$ if r is an equality, or $F' <: G$ if r is a subtyping. That is, we erase type constraints. Validity is not an issue at this point in the algorithm, as we check at the end that the result is valid. Note that, if the equation had the form $Z\{c\} = A$, it could be solved by either $Z = A$ or by $Z = A\{c\}$. By dropping constraints in this rule, we choose the former solution, which tends to give more general types in results.
- Similarly, we drop constraints on G as well.

- If F has the form $K[F_1, \dots, F_k]$ and G has the form $K[G_1, \dots, G_k]$, then $\text{Strip}(r)$ has one type relation comparing each parameter of F with the corresponding one of G :

$$\text{Strip}(r) = \{F_l = G_l \mid 1 \leq l \leq k\}$$

For example, the constraint `List[X] = List[Y]` induces the constraint `X=Y`. `List[X] <: List[Y]` also induces the same constraint. The only way that `List[X]` could be a subtype of `List[Y]` in X10 is if `X=Y`. List of different types are incomparable.⁹

- Other cases are fairly routine. E.g., if F is a type-defined abbreviation, it is expanded.

Example: Consider the program:

```
import x10.util.*;
class C1[C1, C2, C3]{}
class Example {
    static def me[X1, X2](C1[Long, X1, X2]) =
        new C1[X1, X2, Point]();
    static def example() {
        val a = new C1[Long, Boolean, String]();
        val b : C1[Boolean, String, Point]
            = me[Boolean, String](a);
        val c : C1[Boolean, String, Point]
            = me(a);
    }
}
```

The method call for `b` has explicit type parameters. The call for `c` infers the parameters. The computation starts with one equation, saying that the formal parameter of `me` has to be able to accept the actual parameter `a`:

$$\text{C1[Long, Boolean, String]} <: \text{C1[Long, X1, X2]}$$

Note that both terms are `C1` of three things. This is broken into three equations:

$$\text{Long} = \text{Long}$$

which is easy to satisfy,

$$\text{X1} = \text{Boolean}$$

which suggests a possible value for `X1`, and

$$\text{X2} = \text{String}$$

⁹The situation would be more complex if X10 had covariant and contravariant types.

which suggests a value for X_2 . All of these equations are simple enough, so the algorithm terminates.

Then, X_{10} confirms that the binding $X_1=\text{Boolean}$, $X_2=\text{String}$ actually generates a correct call, which it does.

Example: When there is no way to infer types correctly, the type inference algorithm will fail. Consider the program:

```
public class Failsome {
    static def fail[X](a:Rail[X], b:Rail[X]):void {}
    public static def main(argv:Rail[String]) {
        val aint : Rail[Long]      = [1,2,3];
        val abool : Rail[Boolean] = [true, false];
        fail(aint, abool);      // THIS IS WRONG
    }
}
```

The type inference computation starts, as always, by insisting that the types of the formals to `fail` are capable of accepting the actuals:

$$B_0 = \{\text{Rail[Long]} <: \text{Rail[X]}, \text{Rail[Boolean]} <: \text{Rail[X]}\}$$

Arbitrarily picking the first relation to Strip first, we get:

$$B_1 = \{\text{Long} = X, \text{Rail[Boolean]} <: \text{Rail[X]}\}$$

and then

$$B_2 = \{\text{Long} = X, \text{Boolean} = X\}$$

(At this point it is clear to a human that B is inconsistent, but the algorithm's check comes a bit later.) B_2 consists entirely of comparisons with type variables, so the loop is over. Arbitrarily picking the first equality, it guesses the binding

$$B = \{X = \text{Long}\}.$$

In the validation step, it checks that

$$\text{fail[Long]}(\text{aint}, \text{abool})$$

is a well-typed call to `fail`. Of course it is not; `abool` would have to be a value of type `Rail[Long]`, which it is not. So type inference fails at this point. In this case it is correct: there is no way to give a proper type to this program.¹⁰

¹⁰ In particular, $X=\text{Any}$ doesn't work either. A `Rail[Long]` is not a `Rail[Any]` — and it must not be, for you can put a boolean value into a `Rail[Any]`, but you cannot put a boolean value into an `Rail[Long]`. However, if the types of the arguments had simply been X rather than `Rail[X]`, then type inference would correctly infer $X=\text{Any}$.

4.13 Type Dependencies

Type definitions may not be circular, in the sense that no type may be its own supertype, nor may it be a container for a supertype. This forbids interfaces like `interface Loop extends Loop`, and indirect self-references such as `interface A extends B.C` where `interface B extends A`. The formal definition of this is based on Java's.

An *entity type* is a class, interface, or struct type.

Entity type E *directly depends on* entity type F if F is mentioned in the `extends` or `implements` clause of E , either by itself or as a qualifier within a super-entity-type name.

Example: *In the following, A directly depends on B, C, D, E, and F. It does not directly depend on G.*

```
class A extends B.C implements D.E, F[G] {}
```

It is an ordinary programming idiom to use A as an argument to a generic interface that A implements. For example, ComparableTo[T] describes things which can be compared to a value of type T. Saying that A implements ComparableTo[A] means that one A can be compared to another, which is reasonable and useful:

```
interface ComparableTo[T] {
    def eq(T):Boolean;
}
class A implements ComparableTo[A] {
    public def eq(other:A) = this.equals(other);
}
```

Entity type E *depends on* entity type F if either E directly depends on F , or E directly depends on an entity type that depends on F . That is, the relation “depends on” is the transitive closure of the relation “directly depends on”.

It is a static error if any entity type E depends on itself.

4.14 Typing of Variables and Expressions

Variable declarations, field declarations, and some other expressions introduce constraints on their types. These extra constraints represent information that is known at the point of declaration. They are used in deductions and type inference later on – as indeed all constraints are, but the automatically-added constraints are added because they are particularly useful.

Any variable declaration of the form

```
val x : A ...
```

results in declaring `x` to have the type `A{self==x}`, rather than simply `A`. (var declarations get no such addition, because vars cannot appear in constraints.)

A field or property declaration of the form:

```
class A {
    ...
    val f : B ...
    ...
}
```

results in declaring `f` to be of type `B{self==this.f}`. And, if `y` has type `A{c}`, then the type for `y.f` has a constraint `self==y.f`, and, additionally, preserves the information from `c`.

Example:

The following code uses a method `typeIs[T](x)` to confirm, statically, that the type of `x` is `T` (or a subtype of `T`).

On line (A) we confirm that the type of `x` has a `self==x` constraint. The error line (!A) confirms that a different variable doesn't have the `self==x` constraint. (B) shows the extra information carried by a field's type.

(C) shows the extra information carried by a field's type when the object's type is constrained. Note that the constraint `ExtraConstraint{self.n==8}` on the type of `y` has to be rewritten for `y.f`, since the constraint `Long{self.n==8}` is not correct or even well-typed. In this case, the `ExtraConstraint` whose `n`-field is 8 has the name `y`, so we can write the desired type with a conjunct `y.n==8`.¹¹

Note that we use one of the extra constraints here – this reasoning requires the information that the type of `y` has the constraint `self==y`, so X10 can infer `y.n==8` from `self.n==8`. This sort of inference is the reason why X10 adds these constraints in the first place: without them, even the simplest data flows would be beyond the ability of the type system to detect.

```
class Extra(n:Long) {
    val f : Long;
    def this(n:Long, f:Long) { property(n); this.f = f; }
    static def typeIs[T](val x:T) {}
    public static def main(argv:Rail[String]) {
        val x : Extra = new Extra(1,2L);
        typeIs[ Extra{self==x} ] (x);      // (A)
        val nx: Extra = new Extra(1,2L);
        // ERROR: typeIs[ Extra{self==x} ] (nx); // (!A)
        typeIs[ Long{self == x.f} ]          (x.f); // (B)
        val y : Extra{self.n==8} = new Extra(8, 4L);
```

¹¹If `y` were an expression rather than a variable, there would be no good way to express its type in X10's type system. (The compiler has a more elaborate internal representation of types, not all of which are expressible in X10 version 2.2.)

```

        typeIs[ Long{self == y.f, y.n == 8}] (y.f); // (C)
    }
}

```

Once in a while, the additional information will interfere with other typechecking or type inference. In this case, use `as` (§11.23) to erase it, using expressions like `x as A`.

Example: *The following code creates a one-element rail (§11.26) containing x.*

If the `ERROR` line were to be used, X10 would infer that the type of this rail were `Rail[T]`, where T is the type of x — that is, `Rail[Extra{self==x}]`. `[x]` is a rail of x's, not a rail of Extras. Since `Rail[Extra{self==x}]` is not a subtype of `Rail[Extra]`, the rail `[x]` cannot be used in a place where an `Rail[Extra]` is called for.

The expression `[x as Extra]` uses a type cast to erase the automatically-added extra information about x. `x as Extra` simply has type `Extra`, and thus `[x as Extra]` is a `Rail[Extra]` as desired.

```

class Extra {
    static def useRail(Rail[Extra]) {}
    public static def main(argv: Rail[String]) {
        val x : Extra = new Extra();
        //ERROR: useRail([x]);
        useRail([x as Extra]);
    }
}

```

4.15 Limitations of Strict Typing

X10's type checking provides substantial guarantees. In most cases, a program that passes the X10 type checker will not have any runtime type errors. However, there are a modest number of compromises with practicality in the type system: places where a program can pass the typechecker and still have a type error.

1. As seen in §4.5.5, generic types do not have constraint information at runtime. This allows one to write code which violates constraints at runtime, as seen in the example in that section.
2. The library type `x10.util.IndexedMemoryChunk` provides a low-level interface to blocks of memory. A few methods on that class are not type-safe. See the API if you must.
3. Custom serialization (§13.3.2) allows user code to construct new objects in ways that can subvert the type system.
4. Code written to use the underlying Java or C++ (§18) can break X10's guarantees.

5 Variables

A *variable* is an X10 identifier associated with a value within some context. Variable bindings have these essential properties:

- **Type:** What sorts of values can be bound to the identifier;
- **Scope:** The region of code in which the identifier is associated with the entity;
- **Lifetime:** The interval of time in which the identifier is associated with the entity.
- **Visibility:** Which parts of the program can read or manipulate the value through the variable.

X10 has many varieties of variables, used for a number of purposes.

- Class variables, also known as the static fields of a class, which hold their values for the lifetime of the class.
- Instance variables, which hold their values for the lifetime of an object;
- Array elements, which are not individually named and hold their values for the lifetime of an array;
- Formal parameters to methods, functions, and constructors, which hold their values for the duration of method (etc.) invocation;
- Local variables, which hold their values for the duration of execution of a block.
- Exception-handler parameters, which hold their values for the execution of the exception being handled.

A few other kinds of things are called variables for historical reasons; *e.g.*, type parameters are often called type variables, despite not being variables in this sense because they do not refer to X10 values. Other named entities, such as classes and methods, are not called variables. However, all name bindings enjoy similar concepts of scope and visibility.

Example: *In the following example, n is an instance variable, and nxt is a local variable defined within the method bump.*¹

¹This code is unnecessarily turgid for the sake of the example. One would generally write `public def bump() = ++n;`.

```
class Counter {
    private var n : Long = 0;
    public def bump() : Long {
        val nxt = n+1;
        n = nxt;
        return nxt;
    }
}
```

Both variables have type `Long` (or perhaps something more specific). The scope of `n` is the body of `Counter`; the scope of `nxt` is the body of `bump`. The lifetime of `n` is the lifetime of the `Counter` object holding it; the lifetime of `nxt` is the duration of the call to `bump`. Neither variable can be seen from outside of its scope.

Variables whose value may not be changed after initialization are said to be *immutable*, or *constants* (§5.1), or simply `val` variables. Variables whose value may change are *mutable* or simply `var` variables. `var` variables are declared by the `var` keyword. `val` variables may be declared by the `val` keyword; when a variable declaration does not include either `var` or `val`, it is considered `val`.

A variable—even a `val`—can be declared in one statement, and then initialized later on. It must be initialized before it can be used (§19).

Example: The following example illustrates many of the variations on variable declaration:

```
val a : Long = 0;           // Full 'val' syntax
b : Long = 0;               // 'val' implied
val c = 0;                  // Type inferred
var d : Long = 0;           // Full 'var' syntax
var e : Long;                // Not initialized
var f : Long{self != 100} = 0; // Constrained type
val g : Long;                // Init. deferred
if (a > b) g = 1; else g = 2; // Init. done here.
```

5.1 Immutable variables

<i>LocVarDeclnStmt</i>	$::=$	<i>LocVarDecln</i> ;	(20.111)
<i>LocVarDecln</i>	$::=$	<i>Mods?</i> <code>VarKeyword</code> <i>VariableDecls</i>	(20.110)
	$ $	<i>Mods?</i> <i>VarDeclsWType</i>	
	$ $	<i>Mods?</i> <code>VarKeyword</code> <i>FormalDecls</i>	

An immutable (`val`) variable can be given a value (by initialization or assignment) at most once, and must be given a value before it is used. Usually this is achieved by declaring and initializing the variable in a single statement, such as `val x = 3`, with syntax (20.110) using the *VariableDeclarators* or *VarDeclsWType* alternatives.

Example: After these declarations, `a` and `b` cannot be assigned to further, or even redeclared:

```
val a : Long = 10;
val b = (a+1)*(a-1);
// ERROR: a = 11; // vals cannot be assigned to.
// ERROR: val a = 11; // no redeclaration.
```

In three special cases, the declaration and assignment are separate. One case is how constructors give values to `val` fields of objects. In this case, production (20.110) is taken, with the *FormalDeclarators* option, such as `var n:Long;`.

Example: The `Example` class has an immutable field `n`, which is given different values depending on which constructor was called. `n` can't be given its value by initialization when it is declared, since it is not knowable which constructor is called at that point.

```
class Example {
    val n : Long; // not initialized here
    def this() { n = 1; }
    def this(dummy:Boolean) { n = 2; }
}
```

The second case of separating declaration and assignment is in function and method call, described in §5.4. The formal parameters are bound to the corresponding actual parameters, but the binding does not happen until the function is called.

Example: In the code below, `x` is initialized to 3 in the first call and 4 in the second.

```
val sq = (x:Long) => x*x;
x10.io.Console.OUT.println("3 squared = " + sq(3));
x10.io.Console.OUT.println("4 squared = " + sq(4));
```

The third case is delayed initialization (§19), useful in cases where the code has to make decisions (possibly asynchronously) before assigning values to variables.

5.2 Initial values of variables

Every assignment, binding, or initialization to a variable of type `T{c}` must be an instance of type `T` satisfying the constraint `{c}`. Variables must be given a value before they are used. This may be done by initialization – giving a variable a value as part of its declaration.

Example: These variables are all initialized:

```
val immut : Long = 3;
var mutab : Long = immut;
val use = immut + mutab;
```

A variable may also be given a value by an assignment. `var` variables may be assigned to repeatedly. `val` variables may only be assigned once; the compiler will ensure that they are assigned before they are used (§19).

Example: *The variables in the following example are given their initial values by assignment. Note that they could not be used before those assignments, nor could `immu` be assigned repeatedly.*

```
var muta : Long;
// ERROR: println(muta);
muta = 4;
val use2A = muta * 10;
val immu : Long;
// ERROR: println(immu);
if (cointoss()) {immu = 1;}
else {immu = use2A;}
val use2B = immu * 10;
// ERROR: immu = 5;
```

Every class variable must be initialized before it is read, through the execution of an explicit initializer. Every instance variable must be initialized before it is read, through the execution of an explicit or implicit initializer or a constructor. Implicit initializers initialize vars to the default values of their types (§4.7). Variables of types which do not have default values are not implicitly initialized.

Each method and constructor parameter is initialized to the corresponding argument value provided by the invoker of the method. An exception-handling parameter is initialized to the object thrown by the exception. A local variable must be explicitly given a value by initialization or assignment, in a way that the compiler can verify using the rules for definite assignment (§19).

5.3 Destructuring syntax

X10 permits a *destructuring* syntax for local variable declarations with explicit initializers, and for formal parameters, of type `Point`, §16.3.1 and `Array`, §16. A point is a sequence of zero or more `Long`-valued coordinates; an array is an indexed collection of data. It is often useful to get at the coordinates or elements directly, in variables.

$\begin{aligned} \text{VariableDeclr} &::= \text{Id HasResultType}^? = \text{VariableInitializer} \\ &\quad \quad [\text{IdList}] \text{ HasResultType}^? = \text{VariableInitializer} \\ &\quad \quad \text{Id} [\text{IdList}] \text{ HasResultType}^? = \text{VariableInitializer} \end{aligned}$	(20.203)
--	----------

The syntax `val [a1, ..., an] = e;`, where `e` is a `Point`, declares n `Long` variables, bound to the precisely n components of the `Point` value of `e`; it is an error if `e` is not a `Point` with precisely n components. The syntax `val p[a1, ..., an] = e;` is similar, but also declares the variable `p` to be of type `Point(n)`.

The syntax `val [a1, ..., an] = e;`, where `e` is an `Array[T]` for some type `T`, declares n variables of type `T`, bound to the precisely n components of the `Array[T]` value of `e`; it is an error if `e` is not a `Array[T]` with `rank==1` and `size==n`. The syntax `val p[a1, ..., an] = e;` is similar, but also declares the variable `p` to be of type `Array[T]{rank==1,size==n}`.

Example: *The following code makes an anonymous point with one coordinate 11, and binds i to 11. Then it makes a point with coordinates 22 and 33, binds p to that point, and j and k to 22 and 33 respectively.*

```
val [i] : Point = Point.make(11);
assert i == 11L;
val p[j,k] = Point.make(22,33);
assert j == 22L && k == 33L;
val q[l,m] = [44,55] as Point;
assert l == 44L && m == 55L;
//ERROR: val [n] = p;
```

Destructuring is allowed wherever a Point or Array[T] variable is declared, e.g., as the formal parameters of a method. **Example:** *The methods below take a single argument each: a three-element point for example1, a three-element array for example2. The argument itself is bound to x in both cases, and its elements are bound to a, b, and c.*

```
static def example1(x[a,b,c]:Point){}
static def example2(x[a,b,c]:Array[String]{rank==1,size==3L}){}
```

5.4 Formal parameters

Formal parameters are the variables which hold values transmitted into a method or function. They are always declared with a type. (Type inference is not available, because there is no single expression to deduce a type from.) The variable name can be omitted if it is not to be used in the scope of the declaration, as in the type of the method `static def main(Rail[String]):void` executed at the start of a program that does not use its command-line arguments.

`var` and `val` behave just as they do for local variables, §5.5. In particular, the following `inc` method is allowed, but, unlike some languages, does *not* increment its actual parameter. `inc(j)` creates a new local variable `i` for the method call, initializes `i` with the value of `j`, increments `i`, and then returns. `j` is never changed.

```
static def inc(var i:Long) { i += 1; }
static def example() {
    var j : Long = 0;
    assert j == 0;
    inc(j);
    assert j == 0;
}
```

5.5 Local variables and Type Inference

Local variables are declared in a limited scope, and, dynamically, keep their values only for so long as the scope is being executed. They may be `var` or `val`. They may have initializer expressions: `var i:Long = 1;` introduces a variable `i` and initializes it to 1. If the variable is immutable (`val`) the type may be omitted and inferred from the initializer type (§4.12).

The variable declaration `val x:T=e;` confirms that `e`'s value is of type `T`, and then introduces the variable `x` with type `T`. For example, consider a class `Tub` with a property `p`.

```
class Tub(p:Long){
    def this(pp:Long):Tub{self.p==pp} {property(pp);}
    def example() {
        val t : Tub = new Tub(3);
    }
}
```

produces a variable `t` of type `Tub`, even though the expression `new Tub(3)` produces a value of type `Tub{self.p==3}` – that is, a `Tub` whose `p` field is 3. This can be inconvenient when the constraint information is required.

Including type information in variable declarations is generally good programming practice: it explains to both the compiler and human readers something of the intent of the variable. However, including types in `val t:T=e` can obliterate helpful information. So, X10 allows a *documentation type declaration*, written

```
val t <: T = e
```

This has the same effect as `val t = e`, giving `t` the full type inferred from `e`; but it also confirms statically that that type is at least `T`.

Example: *The following gives `t` the type `Tub{self.p==3}` as desired. However, a similar declaration with an inappropriate type will fail to compile.*

```
val t <: Tub = new Tub(3);
// ERROR: val u <: Long = new Tub(3);
```

5.6 Fields

<i>FieldDecls</i>	::=	<i>FieldDeclr</i>	(20.70)
		<i>FieldDecls</i> , <i>FieldDeclr</i>	
<i>FieldDecln</i>	::=	<i>Mods</i> ? <i>VarKeyword</i> <i>FieldDecls</i> ;	(20.68)
		<i>Mods</i> ? <i>FieldDecls</i> ;	
<i>FieldDeclr</i>	::=	<i>Id HasResultType</i>	(20.69)
		<i>Id HasResultType</i> ? = <i>VariableInitializer</i>	
<i>HasResultType</i>	::=	<i>ResultType</i>	(20.86)
		<: <i>Type</i>	
<i>Mod</i>	::=	<i>abstract</i>	(20.121)
		<i>Annotation</i>	
		<i>atomic</i>	
		<i>final</i>	
		<i>native</i>	
		<i>private</i>	
		<i>protected</i>	
		<i>public</i>	
		<i>static</i>	
		<i>transient</i>	
		<i>clocked</i>	

Like most other kinds of variables in X10, the fields of an object can be either `val` or `var`. `val` fields can be `static` (§8.2). Field declarations may have optional initializer expressions, as for local variables, §5.5. `var` fields without an initializer are initialized with the default value of their type. `val` fields without an initializer must be initialized by each constructor.

For `val` fields, as for `val` local variables, the type may be omitted and inferred from the initializer type (§4.12). `var` fields, like `var` local variables, must be declared with a type.

6 Names and packages

6.1 Names

An X10 program consists largely of giving names to entities, and then manipulating the entities by their names. The entities involved may be compile-time constructs, like packages, types and classes, or run-time constructs, like numbers and strings and objects.

X10 names can be *simple names*, which look like identifiers: `vj`, `x10`, `AndSoOn`. Or, they can be *qualified names*, which are sequences of two or more identifiers separated by dots: `x10.lang.String`, `somePack.someType`, `a.b.c.d.e.f`. Some entities have only simple names; some have both simple and qualified names.

Every declaration that introduces a name has a *scope*: the region of the program in which the named entity can be referred to by a simple name. In some cases, entities may be referred to by qualified names outside of their scope. *E.g.*, a `public` class `C` defined in package `p` can be referred to by the simple name `C` inside of `p`, or by the qualified name `p.C` from anywhere.

Many sorts of entities have *members*. Packages have classes, structs, and interfaces as members. Those, in turn, have fields, methods, types, and so forth as members. The member `x` of an entity named `E` (as a simple or qualified name) has the name `E.x`; it may also have other names.

6.1.1 Shadowing

One declaration `d` may *shadow* another declaration `d'` in part of the scope of `d'`, if `d` and `d'` declare variables with the same simple name `n`. When `d` shadows `d'`, a use of `n` might refer to `d`'s `n` (unless some `d''` in turn shadows `d`), but will never refer to `d'`'s `n`.

X10 has four namespaces:

- **Types:** for classes, interfaces, structs, and defined types.
- **Values:** for `val`- and `var`-bound variables; fields; and formal parameters of all sorts.

- **Methods:** for methods of classes, interfaces, and structs.
- **Packages:** for packages.

A declaration d in one namespace, binding a name n to an entity e , shadows all other declarations of that name n in scope at the point where d is declared. This shadowing is in effect for the entire scope of d . Declarations in different namespaces do not shadow each other. Thus, a local variable declaration may shadow a field declaration, but not a class declaration.

Declarations which only introduce qualified names — in X10, this is only package declarations — cannot shadow anything.

The rules for shadowing of imported names are given in §6.4.

6.1.2 Hiding

Shadowing is ubiquitous in X10. Another, and considerably rarer, way that one definition of a given simple name can render another definition of the same name unavailable is *hiding*. If a class `Super` defines a field named `x`, and a subclass `Sub` of `Super` also defines a field named `x`, and `b:Sub`, then `b.x` is `Sub`'s `x` field, not `Super`'s. In this case, `Super`'s `x` is said to be *hidden*.

Hiding is technically different from shadowing, because hiding applies in more circumstances: a use of class `Sub`, such as `sub.x`, may involve hiding of name `x`, though it could not involve shadowing of `x` because `x` need not be declared as a name at that point.

6.1.3 Obscuring

The third way in which a definition of a simple name may become unavailable is *obscuring*. This well-named concept says that, if `n` can be interpreted as two or more of: a variable, a type, and a package, then it will be interpreted as a variable if that is possible, or a type if it cannot be interpreted as a variable. In this case, the unavailable interpretations are *obscured*.

Example: *In the example method of the following code, both a struct and a local variable are named eg. Following the obscuring rules, the call eg.ow() in the first assert uses the variable rather than the struct. As the second assert demonstrates, the struct can be accessed through its fully-qualified name. Note that none of this would have happened if the coder had followed the convention that structs have capitalized names, Eg, and variables have lower-case ones, eg.*

```
package obscuring;
struct eg {
    static def ow()= 1;
    static struct Bite {
```

```

        def ow() = 2;
    }
    def example() {
        val eg = Bite();
        assert eg.ow() == 2;
        assert obscuring.eg.ow() == 1;
    }
}

```

Due to obscuring, it may be impossible to refer to a type or a package via a simple name in some circumstances. Obscuring does not block qualified names.

6.1.4 Ambiguity and Disambiguation

Neither simple nor qualified names are necessarily unique. There can be, in general, many entities that have the same name. This is perfectly ordinary, and, when done well, considered good programming practice. Various forms of *disambiguation* are used to tell which entity is meant by a given name; *e.g.*, methods with the same name may be disambiguated by the types of their arguments (§8.12).

Example: *In the following example, there are three static methods with qualified name DisambEx.Example.m; they can be disambiguated by their different arguments. Inside the body of the third, the simple name i refers to both the Long formal of m, and to the static method DisambEx.Example.i.*

```

package DisambEx;
class Example {
    static def m() = 1;
    static def m(Boolean) = 2;
    static def i() = 3;
    static def m(i:Long) {
        if (i > 10) {
            return i() + 1;
        }
        return i;
    }
    static def example() {
        assert m() == 1;
        assert m(true) == 2;
        assert m(3) == 3;
        assert m(20) == 4;
    }
}

```

6.2 Access Control

X10 allows programmers *access control*, that is, the ability to determine statically where identifiers of most sorts are visible. In particular, X10 allows *information hiding*, wherein certain data can be accessed from only limited parts of the program.

There are four access control modes: `public`, `protected`, `private` and uninflected package-specific scopes, much like those of Java. Most things can be public or private; a few things (*e.g.*, class members) can also be protected or package-scoped.

Accessibility of one X10 entity (package, container, member, etc.) from within a package or container is defined as follows:

- Packages are always accessible.
- If a container C is public, and, if it is inside of another container D, container D is accessible, then C is accessible.
- A member m of a container C is accessible from within another container E if C is accessible, and:
 - m is declared `public`; or
 - C is an interface; or
 - m is declared `protected`, and either the access is from within the same package that C is defined in, or from within the body of a subclass of C (but see §6.2.1 for some fine points); or
 - m is declared `private`, and the access is from within the top-level class which contains the definition of C — which may be C itself, or, if C is a nested container, an outer class around C; or
 - m has no explicit class declaration (hence using the implicit “package”-level access control), and the access occurs from the same package that C is declared in.

6.2.1 Details of protected

`protected` access has a few fine points. Within the body of a subclass D of the class C containing the definition of a protected member m,

- An access e.fld to a field, or e.m(...) to a method, is permitted precisely when the type of e is either D or a subtype of D. For example, the access to `that.f` in the following code is acceptable, but the access to `xhax.f` is not.

```
class C {
    protected var f : Long = 0;
}
class X extends C {}
```

```

class D extends C {
    def usef(that:D, xhax:X) {
        this.f += that.f;
        // ERROR: this.f += xhax.f;
    }
}

```

Limitation: The X10 compiler improperly allows access to `xhax` – as, indeed, some Java compilers do, despite Java having the analogous rule. The compiler allows you to do everything the spec says and a bit more.

- An access through a qualified name `Q.N` is permitted precisely when the type of `Q` is `D` or a subtype of `D`.

Qualified access to a protected constructor is subtle. Let `C` be a class with a protected constructor `c`, and let `S` be the innermost class containing a use `u` of `c`. There are three cases for `u`:

- Superclass construction invocations, `super(...)` or `E.super(...)`, are permitted.
- Anonymous class instance creations, of the forms `new C(...){...}` and `E.new C(...){...}`, are permitted.
- No other accesses are permitted.

6.3 Packages

A package is a named collection of top-level type declarations, *viz.*, class, interface, and struct declarations. Package names are sequences of identifiers, like `x10.lang` and `com.ibm.museum`. The multiple names are simply a convenience, though there is a tenuous relationship between packages `a`, `a.b`, and `a.c`. Packages can be accessed by name from anywhere: a package may contain private elements, but may not itself be private.

Packages and protection modifiers determine which top-level names can be used where. Only the `public` members of package `pack.age` can be accessed outside of `pack.age` itself.

```

package pack.age;
class Deal {
    public def make() {}
}
public class Stimulus {
    private def taxCut() = true;
    protected def benefits() = true;
}

```

```
public def jobCreation() = true;
/*package*/ def jumpstart() = true;
}
```

The class `Stimulus` can be referred to from anywhere outside of `pack.age` by its full name of `pack.age.Stimulus`, or can be imported and referred to simply as `Stimulus`. The public `jobCreation()` method of a `Stimulus` can be referred to from anywhere as well; the other methods have smaller visibility. The non-public class `Deal` cannot be used from outside of `pack.age`.

6.3.1 Name Collisions

It is a static error for a package to have two members with the same name. For example, package `pack.age` cannot define two classes both named `Crash`, nor a class and an interface with that name.

Furthermore, `pack.age` cannot define a member `Crash` if there is another package named `pack.age.Crash`, nor vice-versa. (This prohibition is the only actual relationship between the two packages.) This prevents the ambiguity of whether `pack.age.Crash` refers to the class or the package. Note that the naming convention that package names are lower-case and package members are capitalized prevents such collisions.

6.4 import Declarations

Any public member of a package can be referred to from anywhere through a fully-qualified name: `pack.age.Stimulus`.

Often, this is too awkward. X10 has two ways to allow code outside of a class to refer to the class by its short name (`Stimulus`): single-type imports and on-demand imports.

Imports of either kind appear at the start of the file, immediately after the package directive if there is one; their scope is the whole file.

6.4.1 Single-Type Import

The declaration `import TypeName ;` imports a single type into the current namespace. The type it imports must be a fully-qualified name of an extant type, and it must either be in the same package (in which case the `import` is redundant) or be declared `public`.

Furthermore, when importing `pack.age.T`, there must not be another type named `T` at that point: neither a `T` declared in `pack.age`, nor a `inst.ant.T` imported from some other package.

The declaration `import E.n;`, appearing in file *f* of a package named *P*, shadows the following types named *n* when they appear in *f*:

- Top-level types named `n` appearing in other files of `P`, and
- Types named `n` imported by automatic imports (§6.4.2) in `f`.

6.4.2 Automatic Import

The automatic import `import pack.age.*;`, loosely, imports all the public members of `pack.age`. In fact, it does so somewhat carefully, avoiding certain errors that could occur if it were done naively. Types defined in the current package, and those imported by single-type imports, shadow those imported by automatic imports. If two automatic imports provide the same short name `n`, it is an error to use `n` – but it is not an error if no conflicting name is ever used. Names automatically imported never shadow any other names.

6.4.3 Implicit Imports

The package `x10.lang` is automatically imported in all files without need for further specification. Furthermore, the public static members of the class named `_` in `x10.lang` are imported everywhere as well. This provides a number of aliases, such as `Console` and `int` for `x10.io.Console` and `Int`.

6.5 Conventions on Type Names

TypeName ::= *Id* (20.172)

| *TypeName* . *Id*

PackageName ::= *Id* (20.128)

| *PackageName* . *Id*

While not enforced by the compiler, classes and interfaces in the X10 library follow the following naming conventions. Names of types—including classes, type parameters, and types specified by type definitions—are in CamelCase and begin with an uppercase letter. (Type variables are often single capital letters, such as `T`.) For backward compatibility with languages such as C and Java, type definitions are provided to allow primitive types such as `int` and `boolean` to be written in lowercase. Names of methods, fields, value properties, and packages are in camelCase and begin with a lowercase letter. Names of `static val` fields are in all uppercase with words separated by `_`'s.

7 Interfaces

An interface specifies signatures for zero or more public methods, property methods, static vals, classes, structs, interfaces, types and an invariant.

The following puny example illustrates all these features:

```
interface Pushable{prio() != 0} {
    def push(): void;
    static val MAX_PRIO = 100;
    abstract class Pushedness{}
    struct Pushy{}
    interface Pushing{}
    static type Shove = Long;
    property text():String;
    property prio():Long;
}
class MessageButton(text:String)
    implements Pushable{self.prio()==Pushable.MAX_PRIO} {
    public def push() {
        x10.io.Console.OUT.println(text + " pushed");
    }
    public property text() = text;
    public property prio() = Pushable.MAX_PRIO;
}
```

`Pushable` defines two property methods, one normal method, and a static value. It also establishes an invariant, that `prio() != 0`. `MessageButton` implements a constrained version of `Pushable`, *viz.* one with maximum priority. It defines the `push()` method given in the interface, as a `public` method—interface methods are implicitly `public`.

Limitation: X10 may not always detect that type invariants of interfaces are satisfied, even when they obviously are.

A container—a class or struct—can *implement* an interface, typically by having all the methods and property methods that the interface requires, and by providing a suitable `implements` clause in its definition.

A variable may be declared to be of interface type. Such a variable has all the properties and normal methods declared (directly or indirectly) by the interface; nothing else is statically available. Values of any concrete type which implement the interface may be stored in the variable.

Example: *The following code puts two quite different objects into the variable star, both of which satisfy the interface Star.*

```
interface Star { def rise():void; }
class AlphaCentauri implements Star {
    public def rise() {}
}
class ElvisPresley implements Star {
    public def rise() {}
}
class Example {
    static def example() {
        var star : Star;
        star = new AlphaCentauri();
        star.rise();
        star = new ElvisPresley();
        star.rise();
    }
}
```

An interface may extend several interfaces, giving X10 a large fraction of the power of multiple inheritance at a tiny fraction of the cost.

Example:

```
interface Star{}
interface Dog{}
class Sirius implements Dog, Star{}
class Lassie implements Dog, Star{}
```

7.1 Interface Syntax

<i>InterfaceDecln</i>	$::= Mods^? \text{ interface } Id \text{ TypeParamsI}^? \text{ Properties}^? \text{ Guard}^? \quad (20.99)$
	$\text{ExtendsInterfaces}^? \text{ InterfaceBody}$
<i>TypeParamsI</i>	$::= [\text{TypeParamIList}] \quad (20.177)$
<i>Guard</i>	$::= \text{DepParams} \quad (20.83)$
<i>ExtendsInterfaces</i>	$::= \text{extends Type} \quad (20.66)$
	$ \text{ ExtendsInterfaces , Type}$
<i>InterfaceBody</i>	$::= \{ \text{InterfaceMemberDecls}^? \} \quad (20.98)$
<i>InterfaceMemberDecln</i>	$::= \text{MethodDecln} \quad (20.100)$
	$ \text{ PropMethodDecln}$
	$ \text{ FieldDecln}$
	$ \text{ TypeDecln}$

The invariant associated with an interface is the conjunction of the invariants associated with its superinterfaces and the invariant defined at the interface.

A class C implements an interface I if I, or a subtype of I, appears in the `implements` list of C. In this case, C implicitly gets all the methods and property methods of I, as `abstract public` methods. If C does not declare them explicitly, then they are `abstract`, and C must be `abstract` as well. If C does declare them all, C may be `concrete`.

If C implements I, then the class invariant (§8.9) for C, $inv(C)$, implies the class invariant for I, $inv(I)$. That is, if the interface I specifies some requirement, then every class C that implements it satisfies that requirement.

7.2 Access to Members

All interface members are `public`, whether or not they are declared `public`. There is little purpose to non-public methods of an interface; they would specify that implementing classes and structs have methods that cannot be seen.

7.3 Member Specification

An interface can specify that all containers implementing it must have certain instance methods. It cannot require constructors or static methods, though.

Example: *The Stat interface requires that its implementers provide an `ick` method. It can't insist that implementations provide a static method like `meth`, or a nullary constructor.*

```
interface Stat {
    def ick():void;
```

```
// ERROR: static def meth():Long;
// ERROR: static def this();
}
class Example implements Stat {
    public def ick() {}
    def example() {
        this.ick();
    }
}
```

7.4 Property Methods

An interface may declare `property` methods. All non-`abstract` containers implementing such an interface must provide all the `property` methods specified.

7.5 Field Definitions

An interface may declare a `val` field, with a value. This field is implicitly `public static val`. In particular, it is *not* an instance field.

```
interface KnowsPi {
    PI = 3.14159265358;
}
```

Classes and structs implementing such an interface get the interface's fields as `public static` fields. Unlike methods, there is no need for the implementing class to declare them.

```
class Circle implements KnowsPi {
    static def area(r:Double) = PI * r * r;
}
class UsesPi {
    def circumf(r:Double) = 2 * r * KnowsPi.PI;
}
```

7.5.1 Fine Points of Fields

If two parent interfaces give different static fields of the same name, those fields must be referred to by qualified names.

```
interface E1 {static val a = 1;}
interface E2 {static val a = 2;}
interface E3 extends E1, E2{}
```

```
class Example implements E3 {
    def example() = E1.a + E2.a;
}
```

If the *same* field `a` is inherited through many paths, there is no need to disambiguate it:

```
interface I1 { static val a = 1; }
interface I2 extends I1 {}
interface I3 extends I1 {}
interface I4 extends I2,I3 {}
class Example implements I4 {
    def example() = a;
}
```

The initializer of a field in an interface may be any expression. It is evaluated under the same rules as a `static` field of a class.

Example: *In this example, a class `TheOne` is defined, with an inner interface `WelshOrFrench`, whose field `UN` (named in either Welsh or French) has value 1. Note that `WelshOrFrench` does not define any methods, so it can be trivially added to the `implements` clause of any class, as it is for `Onesome`. This allows the body of `Onesome` to use `UN` through an unqualified name, as is done in `example()`.*

```
class TheOne {
    static val ONE = 1;
    interface WelshOrFrench {
        val UN = 1;
    }
    static class Onesome implements WelshOrFrench {
        static def example() {
            assert UN == ONE;
        }
    }
}
```

7.6 Generic Interfaces

Interfaces, like classes and structs, can have type parameters. The discussion of generics in §4.3 applies to interfaces, without modification.

Example:

```
interface ListOfFuncs[T,U] extends x10.util.List[(T)=>U] {}
```

7.7 Interface Inheritance

The *direct superinterfaces* of a non-generic interface I are the interfaces (if any) mentioned in the `extends` clause of I 's definition. If I is generic, the direct superinterfaces are of an instantiation of I are the corresponding instantiations of those interfaces. A *superinterface* of I is either I itself, or a direct superinterface of a superinterface of I , and similarly for generic interfaces.

I inherits the members of all of its superinterfaces. Any class or struct that has I in its `implements` clause also implements all of I 's superinterfaces.

Classes and structs may be declared to implement multiple interfaces. Semantically, the interface type is the set of all objects that are instances of classes or structs that implement the interface. A class or struct implements an interface if it is declared to and if it concretely or abstractly implements all the methods and properties defined in the interface. For example, `Kim` implements `Person`, and hence `Named` and `Mobile`. It would be a static error if `Kim` had no `name` method, unless `Kim` were also declared `abstract`.

```
class Kim implements Person {
    var pos : Long = 0;
    public def name() = "Kim (" + pos + ")";
    public def move(dPos:Long) { pos += dPos; }
}
```

7.8 Members of an Interface

The members of an interface I are the union of the following sets:

1. All of the members appearing in I 's declaration;
2. All the members of its direct super-interfaces, except those which are hidden (§6.1.2) by I
3. The members of `Any`.

Overriding for instances is defined as for classes, §8.4.8

8 Classes

8.1 Principles of X10 Objects

8.1.1 Basic Design

Objects are instances of classes: the most common and most powerful sort of value in X10. The other kinds of values, structs and functions, are more specialized.

Classes are structured in a forest of single-inheritance code hierarchies. Like C++, but unlike Java, there is no single root class (e.g. `java.lang.Object`) that all classes inherit from. Classes may have any or all of these features:

- Implementing any number of interfaces;
- Static and instance `val` fields;
- Instance `var` fields;
- Static and instance methods;
- Constructors;
- Properties;
- Static and instance nested containers.
- Static type definitions

X10 objects (unlike Java objects) do not have locks associated with them. Programmers may use atomic blocks (§14.7) for mutual exclusion and clocks (§15) for sequencing multiple parallel operations.

An object exists in a single location: the place that it was created. One place cannot use or even directly refer to an object in a different place. A special type, `GlobalRef[T]`, allows explicit cross-place references.

The basic operations on objects are:

- Construction (§8.11). Objects are created, their `var` and `val` fields initialized, and other invariants established.
- Field access (§11.4). The static, instance, and property fields of an object can be retrieved; `var` fields can be set.
- Method invocation (§11.6). Static, instance, and property methods of an object can be invoked.
- Casting (§11.22) and instance testing with `instanceof` (§11.24) Objects can be cast or type-tested.
- The equality operators `==` and `!=`. Objects can be compared for equality with the `==` operation. This checks object *identity*: two objects are `==` iff they are the same object.

8.1.2 Class Declaration Syntax

The *class declaration* has a list of type parameters, a list of properties, a constraint (the *class invariant*), zero or one superclass, zero or more interfaces that it implements, and a class body containing the the definition of fields, properties, methods, and member types. Each such declaration introduces a class type (§4.2).

<i>ClassDecln</i>	<code>::= Mod[?] class Id TypeParamsI[?] Properties[?] Guard[?] Super[?] Interfaces[?]</code>	(20.34)
	<code>ClassBody</code>	
<i>TypeParamsI</i>	<code>::= [TypeParamIList]</code>	(20.177)
<i>TypeParamIList</i>	<code>::= TypeParam</code>	(20.174)
	<code> TypeParamIList , TypeParam</code>	
	<code> TypeParamIList ,</code>	
<i>Properties</i>	<code>::= (PropList)</code>	(20.142)
<i>PropList</i>	<code>::= Prop</code>	(20.140)
	<code> PropList , Prop</code>	
<i>Prop</i>	<code>::= Annotations[?] Id ResultType</code>	(20.139)
<i>Guard</i>	<code>::= DepParams</code>	(20.83)
<i>Super</i>	<code>::= extends ClassType</code>	(20.156)
<i>Interfaces</i>	<code>::= implements InterfaceTypeList</code>	(20.103)
<i>InterfaceTypeList</i>	<code>::= Type</code>	(20.102)
	<code> InterfaceTypeList , Type</code>	
<i>ClassBody</i>	<code>::= { ClassMemberDecls[?] }</code>	(20.33)
<i>ClassMemberDecls</i>	<code>::= ClassMemberDecln</code>	(20.36)
	<code> ClassMemberDecls ClassMemberDecln</code>	
<i>ClassMemberDecln</i>	<code>::= InterfaceMemberDecln</code>	(20.35)
	<code> CtorDecln</code>	

8.2 Fields

Objects may have *instance fields*, or simply *fields* (called “instance variables” in C++ and Smalltalk, and “slots” in CLOS); places to store data that is pertinent to the object. Fields, like variables, may be mutable (var) or immutable (val).

A class may have *static fields*, which store data pertinent to the entire class of objects. See §8.6 for more information. Because of its emphasis on safe concurrency, X10 requires static fields to be immutable (val).

No two fields of the same class may have the same name. A field may have the same name as a method, although for fields of functional type there is a subtlety (§8.12.4).

8.2.1 Field Initialization

Fields may be given values via *field initialization expressions*: val f1 = E; and var f2 : Long = F;. Other fields of this may be referenced, but only those that precede the field being initialized.

Example: The following is correct, but would not be if the fields were reversed:

```
class Fld{
    val a = 1;
    val b = 2+a;
}
```

8.2.2 Field hiding

A subclass that defines a field f hides any field f declared in a superclass, regardless of their types. The superclass field f may be accessed within the body of the subclass via the reference super.f.

With inner classes, it is occasionally necessary to write Cls.super.f to get at a hidden field f of an outer class Cls.

Example: The f field in Sub hides the f field in Super. The superf method provides access to the f field in Super.

```
class Super{
    public val f = 1;
}
class Sub extends Super {
    val f = true;
    def superf() : Long = super.f; // 1
}
```

Example: Hidden fields of outer classes can be accessed by suitable forms:

```

class A {
    val f = 3;
}
class B extends A {
    val f = 4;
    class C extends B {
        // C is both a subclass and inner class of B
        val f = 5;
        def example() {
            assert f == 5 : "field of C";
            assert super.f == 4 : "field of superclass";
            assert B.this.f == 4 : "field of outer instance";
            assert B.super.f == 3 : "super.f of outer instance";
        }
    }
}

```

8.2.3 Field qualifiers

The behavior of a field may be changed by a field qualifier, such as `static` or `transient`.

`static` qualifier

A `val` field may be declared to be *static*, as described in §8.2.

`transient` Qualifier

A field may be declared to be *transient*. Transient fields are excluded from the deep copying that happens when information is sent from place to place in an `at` statement. The value of a transient field of a copied object is the default value of its type, regardless of the value of the field in the original. If the type of a field has no default value, it cannot be marked `transient`.

```

class Trans {
    val copied = "copied";
    transient var transy : String = "a very long string";
    def example() {
        at (here) { // causes copying of 'this'
            assert(this.copied.equals("copied"));
            assert(this.transy == null);
        }
    }
}

```

8.3 Properties

The properties of an object (or struct) are a restricted form of public val fields.¹ For example, every array has a rank telling how many subscripts it takes. User-defined classes can have whatever properties are desired.

Properties differ from public val fields in a few ways:

1. Property references are allowed on self in constraints: self.prop. Field references are not.
2. Properties are in scope for all instance initialization expressions. val fields are not.
3. The graph of values reachable from a given object by following only property links is acyclic. Conversely, it is possible (and routine) for two objects to point to each other with val fields.
4. Properties are declared in the class header; val fields are defined in the class body.
5. Properties are set in constructors by a property statement. val fields are set by assignment.

Properties are defined in parentheses, after the name of the class. They are given values by the property command in constructors.

Example: Proper has a single property, t. new Proper(4) creates a Proper object with t==4.

```
class Proper(t:Long) {
    def this(t:Long) {property(t);}
}
```

It is a static error for a class defining a property x: T to have a subclass class that defines a property or a field with the name x.

A property x:T induces a field with the same name and type, as if defined with:

```
public val x : T;
```

Properties are initialized in a constructor by the invocation of a special property statement. The requirement to use the property statement means that all properties must be given values at the same time: a container either has its properties or it does not.

```
property(e1, ..., en);
```

¹In many cases, a val field can be upgraded to a property, which entails no compile-time or runtime cost. Some cannot be, e.g., in cases where cyclic structures of val fields are required.

The number and types of arguments to the `property` statement must match the number and types of the properties in the class declaration, in order. Every constructor of a class with properties must invoke `property(...)` precisely once; it is a static error if X10 cannot prove that this holds.

By construction, the graph whose nodes are values and whose edges are properties is acyclic. *E.g.*, there cannot be values `a` and `b` with properties `c` and `d` such that `a.c == b` and `b.d == a`.

Example:

```
class Proper(a:Long, b:String) {
    def this(a:Long, b:String) {
        property(a, b);
    }
    def this(z:Long) {
        val theA = z+5;
        val theB = "X"+z;
        property(theA, theB);
    }
    static def example() {
        val p = new Proper(1, "one");
        assert p.a == 1 && p.b.equals("one");
        val q = new Proper(10);
        assert q.a == 15 && q.b.equals("X10");
    }
}
```

8.3.1 Properties and Field Initialization

Fields with explicit initializers are evaluated immediately after the `property` command, and all properties are in scope when initializers are evaluated.

Example: *Class `Init` initializes the field `a` to be a rail of `n` elements, where `n` is a property. When `new Init(4)` is executed, the constructor first sets `n` to 4 via the `property` statement, and then initializes `a` to a 4-element rail.*

However, `Outit` uses a field rather than a property for `n`. If the `ERROR` line were present, it would not compile. `n` has not been definitely assigned (§19) at this point, and `n` has not been given its value, so `a` cannot be computed. (If one insisted that `n` be a property, `a` would have to be initialized in the constructor, rather than by an initialization expression.)

```
class Init(n:Long) {
    val a = new Rail[String](n, "");
    def this(n:Long) { property(n); }
}
class Outit {
```

```

val n : Long;
//ERROR: val a = new Rail[String](n, "");
def this(m:Long) { this.n = m; }
}

```

8.3.2 Properties and Fields

A container with a property named `p`, or a nullary property method named `p()`, cannot have a field named `p` — either defined in that container, or inherited from a superclass.

8.3.3 Acyclicity of Properties

X10 has certain restrictions that, ultimately, require that properties are simpler than their containers. For example, `class A(a:A){}` is not allowed. Formally, this requirement is that there is a total order \preceq on all classes and structs such that, if A extends B , then $A \prec B$, and if A has a property of type B , then $A \prec B$, where $A \prec B$ means $A \preceq B$ and $A \neq B$. For example, the preceding class `A` is ruled out because we would need $A \prec A$, which violates the definition of \prec . The programmer need not (and cannot) specify \preceq , and rarely need worry about its existence.

Similarly, the type of a property may not simply be a type parameter. For example, `class A[X](x:X){}` is illegal.

8.4 Methods

As is common in object-oriented languages, objects can have *methods*, of two sorts. *Static methods* are functions, conceptually associated with a class and defined in its namespace. *Instance methods* are parameterized code bodies associated with an instance of the class, which execute with convenient access to that instance's fields.

Each method has a *signature*, telling what arguments it accepts, what type it returns, and what precondition it requires. Method definitions may be overridden by subclasses; the overriding definition may have a declared return type that is a subtype of the return type of the definition being overridden. Multiple methods with the same name but different signatures may be provided on a class (called “overloading” or “ad hoc polymorphism”). Methods may be declared `public`, `private`, `protected`, or given default package-level access rights.

<i>MethMods</i>	$::= \begin{array}{l} Mods^? \\ MethMods \text{ property} \\ MethMods \text{ Mod} \end{array}$	(20.115)
<i>MethodDecln</i>	$::= \begin{array}{l} MethMods \text{ def } Id \text{ TypeParams}^? \text{ Formals } Guard^? \text{ Throws}^? \\ HasResultType^? \text{ MethodBody} \\ BinOpDecln \\ PrefixOpDecln \\ ApplyOpDecln \\ SetOpDecln \\ ConversionOpDecln \\ KeywordOpDecln \end{array}$	(20.117)
<i>TypeParams</i>	$::= [\text{TypeParamList}]$	(20.176)
<i>Formals</i>	$::= (\text{FormalList}^?)$	(20.80)
<i>FormalList</i>	$::= \begin{array}{l} Formal \\ FormalList , Formal \end{array}$	(20.79)
<i>Throws</i>	$::= \text{throws ThrowList}$	(20.84)
<i>ThrowsList</i>	$::= \begin{array}{l} Type \\ ThrowsList , Type \end{array}$	(20.85)
<i>HasResultType</i>	$::= \begin{array}{l} ResultType \\ <: Type \end{array}$	(20.86)
<i>MethodBody</i>	$::= \begin{array}{l} = \text{LastExp} ; \\ Annotations^? \text{ Block} \\ ; \end{array}$	(20.116)
<i>BinOpDecln</i>	$::= \begin{array}{l} \text{MethMods operator TypeParams}^? (\text{Formal}) \text{ BinOp } (\text{Formal}) \text{ Guard}^? \text{ HasResultType}^? \text{ MethodBody} \\ \text{MethMods operator TypeParams}^? \text{ this BinOp } (\text{Formal}) \text{ Guard}^? \text{ HasResultType}^? \text{ MethodBody} \\ \text{MethMods operator TypeParams}^? (\text{Formal}) \text{ BinOp this Guard}^? \text{ HasResultType}^? \text{ MethodBody} \end{array}$	(20.24)
<i>PrefixOpDecln</i>	$::= \begin{array}{l} \text{MethMods operator TypeParams}^? \text{ PrefixOp } (\text{Formal}) \text{ Guard}^? \text{ HasResultType}^? \text{ MethodBody} \\ \text{MethMods operator TypeParams}^? \text{ PrefixOp this Guard}^? \text{ HasResultType}^? \text{ MethodBody} \end{array}$	(20.137)
<i>ApplyOpDecln</i>	$::= \text{MethMods operator this TypeParams}^? \text{ Formals } Guard^? \text{ HasResultType}^? \text{ MethodBody}$	(20.7)
<i>ConversionOpDecln</i>	$::= \begin{array}{l} ExplConvOpDecln \\ ImplConvOpDecln \end{array}$	(20.51)

A formal parameter may have a `val` or `var` modifier; `val` is the default. The body of the method is executed in an environment in which each formal parameter corresponds to a local variable (`var` iff the formal parameter is `var`) and is initialized with the value of the actual parameter.

8.4.1 Forms of Method Definition

There are several syntactic forms for defining methods. The forms that include a block, such as `def m() {S}`, allow an arbitrary block. These forms can define a `void` method, which does not return a value.

The forms that include an expression, such as `def m() = E`, require a syntactically and semantically valid expression. These forms cannot define a `void` method, because expressions cannot be `void`.

There are no other semantic differences between the two forms.

8.4.2 Method Return Types

A method with an explicit return type returns values of that type. A method without an explicit return type is given a return type by type inference. A *call* to a method has type given by substituting information about the actual `val` parameters for the formals.

Example:

In the example below, `met1` has an explicit return type `Ret{n==a}`. `met2` does not, so its return type is computed, also to be `Ret{n==a}`, because that's what the implicitly-defined constructor returns.

use3 requires that its argument have `n==3`. example shows that both `met1` and `met2` can be used to produce such an object. In both cases, the actual argument 3 is substituted for the formal argument `a` in the return type expression for the method `Ret{n==a}`, giving the type `Ret{n==3}` as required by `use3`.

```
class Ret(n:Long) {
    static def met1(a:Long) : Ret{n==a} = new Ret(a);
    static def met2(a:Long)           = new Ret(a);
    static def use3(Ret{n==3}) {}
    static def example() {
        use3(met1(3));
        use3(met2(3));
    }
}
```

8.4.3 Throws Clause

The `throws` clause indicates what checked exceptions may be raised during the execution of the method and are not handled by `catch` blocks within the method. If a checked exception may escape from the method, then it must be by a subtype of one of the types listed in the `throws` clause of the method. Checked exceptions are defined to be any subclass of `x10.lang.CheckedThrowable` that are not also subclasses of either `x10.lang.Exception` or `x10.lang.Error`.

If a method is implementing an interface or overriding a superclass method the set of types represented by its `throws` clause must be a (potentially improper) subset of the types of the `throws` clause of the method it is overriding.

8.4.4 Final Methods

An instance method may be given the `final` qualifier. `final` methods may not be overridden.

8.4.5 Generic Instance Methods

Limitation: In X10, an instance method may be generic:

```
class Example {
    def example[T](t:T) = "I like " + t;
}
```

However, the C++ back end does not currently support generic virtual instance methods like `example`. It does allow generic instance methods which are `final` or `private`, and it does allow generic static methods.

8.4.6 Method Guards

Often, a method will only make sense to invoke under certain statically-determinable conditions. These conditions may be expressed as a guard on the method.

Example: *For example, `example(x)` is only well-defined when `x != null`, as `null.toString()` throws a null pointer exception, and returns nothing:*

```
class Example {
    var f : String = "";
    def setF(x:Any){x != null} : void {
        this.f = x.toString();
    }
}
```

(We could have used a constrained type `Any{self!=null}` for `x` instead; in most cases it is a matter of personal preference or convenience of expression which one to use.)

The requirement of having a method guard is that callers must demonstrate to the X10 compiler and/or runtime that the guard is satisfied. With the `STATIC_CHECKS` compiler option in force (§C.1.3), this is checked at compile time, and there is no runtime cost. Indeed, this code can be more efficient than usual, as it is statically provable that `x != null`.

When STATIC_CHECKS is not in force, dynamic checks are generated as needed; method guards are checked at runtime. This is potentially more expensive, but may be more convenient.

Example: *The following code fragment contains a line which will not compile with STATIC_CHECKS on (assuming the guarded example method above). (X10's type system does not attempt to propagate information from ifs.) It will compile with STATIC_CHECKS off, but it may insert an extra null-test for x.*

```
def exam(e:Example, x:Any) {
    if (x != null)
        e.example(x as Any{x != null});
    // If STATIC_CHECKS is in force:
    // ERROR: if (x != null) e.example(x);
}
```

The guard $\{c\}$ in a guarded method `def m() {c} = E;` specifies a constraint c on the properties of the class C on which the method is being defined. The method, in effect, only exists for those instances of C which satisfy c . It is illegal for code to invoke the method on objects whose static type is not a subtype of $C\{c\}$.

Specifically: the compiler checks that every method invocation $o.m(e_1, \dots, e_n)$ is type correct. Each argument e_i must have a static type S_i that is a subtype of the declared type T_i for the i th argument of the method, and the conjunction of the constraints on the static types of the arguments must entail the guard in the parameter list of the method.

The compiler checks that in every method invocation $o.m(e_1, \dots, e_n)$ the static type of o , S , is a subtype of $C\{c\}$, where the method is defined in class C and the guard for m is equivalent to c .

Finally, if the declared return type of the method is $D\{d\}$, the return type computed for the call is $D\{a: S; x_1: S_1; \dots; x_n: S_n; d[a/this]\}$, where a is a new variable that does not occur in d , S , S_1 , \dots , S_n , and x_1 , \dots , x_n are the formal parameters of the method.

Limitation: Using a reference to an outer class, `Outer.this`, in a constraint, is not supported.

8.4.7 Property methods

$$\begin{aligned} PropMethodDecln ::= & \text{ Mods}^? \text{ property } Id \text{ TypeParams}^? \text{ Formals } Guard^? \text{ Throws}^? \quad (20.141) \\ & \text{ HasResultType}^? \text{ MethodBody} \\ | & \text{ Mods}^? \text{ property } Id \text{ Guard}^? \text{ HasResultType}^? \text{ MethodBody} \end{aligned}$$

Property methods are methods that can be evaluated in constraints, as properties can. They provide a means of abstraction over properties; *e.g.*, interfaces can specify property methods that implementing containers must provide, but, just as they cannot specify ordinary fields, they cannot specify property fields. Property methods are very

limited in computing power: they must obey the same restrictions as constraint expressions. In particular, they cannot have side effects, or even much code in their bodies.

Example: *The eq() method below tells if the x and y properties are equal; the is(z) method tells if they are both equal to z. The eq and is property methods are used in types in the example method.*

```
class Example(x:Long, y:Long) {
    def this(x:Long, y:Long) { property(x,y); }
    property eq() = (x==y);
    property is(z:Long) = x==z && y==z;
    def example( a : Example{eq()}, b : Example{is(3)} ) {}
}
```

A property method declared in a class must have a body and must not be void. The body of the method must consist of only a single return statement with an expression, or a single expression. It is a static error if the expression cannot be represented in the constraint system. Property methods may be abstract in abstract classes, and may be specified in interfaces, but are implicitly final in non-abstract classes.

The expression may contain invocations of other property methods. The compiler ensures that there are no circularities in property methods, so property method evaluations always terminate.

Property methods in classes are implicitly final; they cannot be overridden. It is a static error if a superclass has a property method with a given signature, and a subclass has a method or property method with the same signature. It is a static error if a superclass has a property with some name p, and a subclass has a nullary method of any kind (instance, static, or property) also named p.

A nullary property method definition may omit the def keyword. That is, the following are equivalent:

```
property def rail(): Boolean =
    rect && onePlace == here && zeroBased;
```

and

```
property rail(): Boolean =
    rect && onePlace == here && zeroBased;
```

Similarly, nullary property methods can be inspected in constraints without (). If ob's type has a property p, then ob.p is that property. Otherwise, if it has a nullary property method p(), ob.p is equivalent to ob.p(). As a consequence, if the type provides both a property p and a nullary method p(), then the property can be accessed as ob.p and the method as ob.p().²

²This only applies to nullary property methods, not nullary instance methods. Nullary property methods perform limited computations, have no side effects, and always return the same value, since they have to be expressed in the constraint sublanguage. In this sense, a nullary property method does not behave hugely different from a property. Indeed, a compilation scheme which cached the value of the property method would all but erase the distinction. Other methods may have more behavior, e.g., side effects, so we keep the () to make it clear that a method call is potentially complex.

`w.rail`, with either definition above, is equivalent to `w.rail()`

Limitation of Property Methods

Limitation: Currently, X10 forbids the use of property methods which have all the following features:

- they are abstract, and
- they have one or more arguments, and
- they appear as subterms in constraints.

Any two of these features may be combined, but the three together may not be.

Example: *The constraint in example1 is concrete, not abstract. The constraint in example2 is nullary, and has no arguments. The constraint in example3 appears at the top level, rather than as a subterm (cf. the equality expressions A==B in the other examples). However, example4 combines all three features, and is not allowed.*

```
class Cls {
    property concrete(a:Long) = 7;
}
interface Inf {
    property nullary(): Long;
    property topLevel(z:Long):Boolean;
    property allThree(z:Long):Long;
}
class Example{
    def example1(Cls{self.concrete(3)==7}) = 1;
    def example2(Inf{self.nullary()==7}) = 2;
    def example3(Inf{self.topLevel(3)}) = 3;
    //ERROR: def example4(Inf{self.allThree(3)==7}) = "fails";
}
```

8.4.8 Method overloading, overriding, hiding, shadowing and obscuring

The definitions of method overloading, overriding, hiding, shadowing and obscuring in X10 are familiar from languages such as Java, modulo the following considerations motivated by type parameters and dependent types.

Two or more methods of a class or interface may have the same name if they have a different number of type parameters, or they have formal parameters of different constraint-erased types (in some instantiation of the generic parameters).

Example: *The following overloading of m is unproblematic.*

```
class Mful{
    def m() = 1;
    def m[T]() = 2;
    def m(x:Long) = 3;
    def m[T](x:Long) = 4;
}
```

A class definition may include methods which are ambiguous in *some* generic instantiation. (It is a compile-time error if the methods are ambiguous in *every* generic instantiation, but excluding class definitions which are ambiguous in *some* instantiation would exclude useful cases.) It is a compile-time error to *use* an ambiguous method call.

Example: *The following class definition is acceptable. However, the marked method calls are ambiguous, and hence not acceptable.*

```
class Two[T,U]{
    def m(x:T)=1;
    def m(x:Long)=2;
    def m[X](x:X)=3;
    def m(x:U)=4;
    static def example() {
        val t12 = new Two[Long, Any]();
        // ERROR: t12.m(2);
        val t13 = new Two[String, Any]();
        t13.m("ferret");
        val t14 = new Two[Boolean,Boolean]();
        // ERROR: t14.m(true);
    }
}
```

The call t12.m(2) could refer to either the 1 or 2 definition of m, so it is not allowed. The call t14.m(true) could refer to either the 1 or 4 definition, so it, too, is not allowed.

The call t13.m("ferret") refers only to the 1 definition. If the 1 definition were absent, type argument inference would make it refer to the 3 definition. However, X10 will choose a fully-specified call if there is one, before trying type inference, so this call unambiguously refers to 1.

X10 v2.4 does not permit overloading based on constraints. That is, the following is *not legal*, although either method definition individually is legal:

```
def n(x:Long){x==1} = "one";
def n(x:Long){x!=1} = "not";
```

The definition of a method declaration m_1 “having the same signature as” a method declaration m_2 involves identity of types.

The *constraint erasure* of a type T, $ce(T)$, is obtained by removing all the constraints outside of functions in T, specifically:

$$ce(T) = T \text{ if } T \text{ is a container or interface} \quad (8.1)$$

$$ce(T\{c\}) = ce(T) \quad (8.2)$$

$$ce(T[S_1, \dots, S_n]) = ce(T)[ce(S_1), \dots, ce(S_n)] \quad (8.3)$$

$$ce((S_1, \dots, S_n) \Rightarrow T) = (ce(S_1), \dots, ce(S_n)) \Rightarrow ce(T) \quad (8.4)$$

Two methods are said to have *erasure equivalent signatures* if (a) they have the same number of type parameters, (b) they have the same number of formal (value) parameters, and (c) for each formal parameter the constraint erasure of its types are erasure equivalent. It is a compile-time error for there to be two methods with the same name and erasure equivalent signatures in a class (either defined in that class or in a superclass), unless the signatures are identical (without erasures) and one of the methods is defined in a superclass (in which case the superclass's method is overridden by the subclass's, and the subclass's method's return type must be a subtype of the superclass's method's return type).

In addition, the guard of an overridden method must entail the guard of the overriding method. This ensures that any virtual call to the method satisfies the guard of the callee.

Example: In the following example, the call to `s.recip(3)` in `example()` will invoke `Sub.recip(n)`. The call is legitimate because `Super.recip`'s guard, `n != 0`, is satisfied by 3. The guard on `Sub.recip(n)` is simply true, which is also satisfied. However, if we had used the `ERROR` line's definition, the guard on `Sub.recip(n)` would be `n != 0, n != 3`, which is not satisfied by 3, so – despite the call statically type-checking – at runtime the call would violate its guard and (in this case) throw an exception.

```
class Super {
    def recip(n:Long){n != 0} = 1.0/n;
}
class Sub extends Super{
    //ERROR: def recip(n:Long){n != 0, n != 3} = 1.0/(n * (n-3));
    def recip(m:Long){true} = 1.0/m;
}
class Example{
    static def example() {
        val s : Super = new Sub();
        s.recip(3);
    }
}
```

If a class C overrides a method of a class or interface B, the guard of the method in B must entail the guard of the method in C.

A class C inherits from its direct superclass and superinterfaces all their methods visible according to the access modifiers of the superclass/superinterfaces that are not hidden

or overridden. A method M_1 in a class C overrides a method M_2 in a superclass D if M_1 and M_2 have erasedly equivalent signatures. Methods are overridden on a signature-by-signature basis. It is a compile-time error if an instance method overrides a static method. (But it is permitted for an instance *field* to hide a static *field*; that's hiding (§8.2.2), not overriding, and hence totally different.)

8.5 Constructors

Instances of classes are created by the `new` expression:

$$\begin{aligned} \textit{ObCreationExp} ::= & \quad \text{new } \textit{TypeName} \text{ } \textit{TypeArgs}? \text{ (} \textit{ArgumentList}? \text{) } \textit{ClassBody} \\ & | \quad \textit{Primary} \text{ . new } \textit{Id} \text{ } \textit{TypeArgs}? \text{ (} \textit{ArgumentList}? \text{) } \textit{ClassBody}? \\ & | \quad \textit{FullyQualifiedNamespace} \text{ . new } \textit{Id} \text{ } \textit{TypeArgs}? \text{ (} \textit{ArgumentList}? \text{) } \textit{ClassBody}? \end{aligned} \tag{20.126}$$

This constructs a new object, and calls some code, called a *constructor*, to initialize the newly-created object properly.

Constructors are defined like methods, except that they must be named `this` and ordinary methods may not be. The content of a constructor body has certain capabilities (*e.g.*, `val` fields of the object may be initialized) and certain restrictions (*e.g.*, most methods cannot be called); see §8.11 for the details.

Example:

The following class provides two constructors. The unary constructor `def this()` allows initialization of the `a` field to an arbitrary value. The nullary constructor `def this(b : Long)` gives it a default value of 10. The `example` method illustrates both of these calls.

```
class C {
    public val a : Long;
    def this(b : Long) { a = b; }
    def this()           { a = 10; }
    static def example() {
        val two = new C(2);
        assert two.a == 2;
        val ten = new C();
        assert ten.a == 10;
    }
}
```

8.5.1 Automatic Generation of Constructors

Classes that have no constructors written in the class declaration are automatically given a constructor which sets the class properties and does nothing else. If this automatically-generated constructor is not valid (*e.g.*, if the class has `val` fields that

need to be initialized in a constructor), the class has no constructor, which is a static error.

Example: *The following class has no explicit constructor. Its implicit constructor is def this(x:Long){property(x);} This implicit constructor is valid, and so is the class.*

```
class C(x:Long) {
    static def example() {
        val c : C = new C(4);
        assert c.x == 4;
    }
}
```

The following class has the same default constructor. However, that constructor does not initialize d, and thus is invalid. This class does not compile; it needs an explicit constructor.

```
// THIS CODE DOES NOT COMPILE
class Cfail(x:Long) {
    val d: Long;
    static def example() {
        val wrong = new Cfail(40);
    }
}
```

8.5.2 Calling Other Constructors

The *first* statement of a constructor body may be a call of the form `this(a,b,c)` or `super(a,b,c)`. The former will execute the body of the matching constructor of the current class; the latter, of the superclass. This allows a measure of abstraction in constructor definitions; one may be defined in terms of another.

Example: *The following class has two constructors. `new Ctors(123)` constructs a new Ctors object with parameter 123. `new Ctors()` constructs one whose parameter has a default value of 100:*

```
class Ctors {
    public val a : Long;
    def this(a:Long) { this.a = a; }
    def this()      { this(100); }
}
```

In the case of a class which implements operator `()` — or any other constructor and application with the same signature — this can be ambiguous. If `this()` appears as the first statement of a constructor body, it could, in principle, mean either a constructor call or an operator evaluation. This ambiguity is resolved so that `this()` always

means the constructor invocation. If, for some reason, it is necessary to invoke an application operator as the first meaningful statement of a constructor, write the target of the application as `(this)`, e.g., `(this)(a,b);`.

8.5.3 Return Type of Constructor

A constructor for class C may have a return type `C{c}`. The return type specifies a constraint on the kind of object returned. It cannot change its *class* — a constructor for class C always returns an instance of class C. If no explicit return type is specified, the constructor's return type is inferred.

Example: *The constructor (A) below, having no explicit return type, has its return type inferred. n is set by the property statement to 1, so the return type is inferred as `Ret{self.n==1}`. The constructor (B) has `Ret{n==self.n}` as an explicit return type. The example() code shows both of these in action.*

```
class Ret(n:Long) {
    def this() { property(1); }      // (A)
    def this(n:Long) : Ret{n==self.n} { // (B)
        property(n);
    }
    static def typeIs[T](x:T){}
    static def example() {
        typeIs[Ret{self.n==1}](new Ret()); // uses (A)
        typeIs[Ret{self.n==3}](new Ret(3)); // uses (B)
    }
}
```

8.6 Static initialization

Static fields in X10 are immutable and are guaranteed to be initialized before they are accessed. Static fields are initialized on a per-Place basis; thus an activity that reads a static field in two different Places may read different values for the content of the field in each Place. Static fields are not eagerly initialized, thus if a particular static field is not accessed in a given Place then the initializer expression for that field may not be evaluated in that Place.

When an activity running in a Place P attempts to read a static field F that has not yet been initialized in P, then the activity will evaluate the initializer expression for F and store the resulting value in F. It is guaranteed that at most one activity in each Place will attempt to evaluate the initializer expression for a given static field. If a second activity attempts to read F while the first activity is still executing the initializer expression the second activity will be suspended until the first activity finishes evaluating the initializer and stores the resulting value in F.

The initializer expression may directly or indirectly read other static fields in the program. If there is a cycle in the field initialization dependency graph for a set of static fields, then any activities accessing those fields may deadlock, which in turn may result in the program deadlocking.³.

If an exception is raised during the evaluation of a static field's initializer expression, then the field is deemed uninitialized in that Place and any subsequent attempt to access the static field's value by another activity in the Place will also result in an exception being raised.⁴. Failure to initialize a field in one Place does not impact the initialization status of the field in other Places.

8.6.1 Compatability with Prior Versions of X10

Previous versions of X10 eagerly initialized all static fields in the program at Place 0 and serialized the resulting values to all other Places before beginning execution of the user main function. It is possible to simulate these serialization semantics for specific static fields under the lazy per-Place initialization semantics by using the idiom below:

```
// Pre X10 2.3 code
// expr evaluated once in Place 0 and resulting value
// serialized to all other places
public static val x:T = expr;

// X10 2.3 code when T haszero is false
private static val x_holder:Cell[T] =
    (here == Place.FIRST_PLACE) ? new Cell[T](expr): null;
public static val x:T = at (Place.FIRST_PLACE) x_holder();

// simpler X10 2.3 code when T haszero is true
private static val x_holder:T =
    (here == Place.FIRST_PLACE) ? expr : Zero.get[T]();
public static val x:T = at (Place.FIRST_PLACE) x_holder;
```

A slightly more complex variant of the above idiom in which the initializer expression for the public field conditionally does the `at` only when not executed at `Place.FIRST_PLACE` can be used to obtain exactly the same serialization behavior as the pre X10 v2.3 semantics. When necessary, eager initialization for specific static fields can be simulated by reading the static fields in `main` before executing the rest of the program.

³The current X10 runtime does not dynamically detect this situation. Future versions of X10 may be able to detect this and convert such a deadlock into the throwing of an `ExceptionInInitializer` exception.

⁴The implementation will make a best effort attempt to present stack trace information about the original cause of the exception in all subsequent raised exceptions

8.7 User-Defined Operators

```
MethodDecln ::= MethMods def Id TypeParams? Formals Guard? Throws? (20.117)
    HasResultType? MethodBody
    | BinOpDecln
    | PrefixOpDecln
    | ApplyOpDecln
    | SetOpDecln
    | ConversionOpDecln
    | KeywordOpDecln
```

It is often convenient to have methods named by symbols rather than words. For example, suppose that we wish to define a `Poly` class of polynomials – for the sake of illustration, single-variable polynomials with `Long` coefficients. It would be very nice to be able to manipulate these polynomials by the usual operations: `+` to add, `*` to multiply, `-` to subtract, and `p(x)` to compute the value of the polynomial at argument `x`. We would like to write code thus:

```
public static def main(Rail[String]):void {
    val X = new Poly([0L,1L]);
    val t <: Poly = 7 * X + 6 * X * X * X;
    val u <: Poly = 3 + 5*X - 7*X*X;
    val v <: Poly = t * u - 1;
    for( i in -3 .. 3) {
        x10.io.Console.OUT.println(
            "" + i + " X:" + X(i) + " t:" + t(i)
            + " u:" + u(i) + " v:" + v(i)
        );
    }
}
```

Writing the same code with method calls, while possible, is far less elegant:

```
public static def uglymain() {
    val X = new UglyPoly([0L,1L]);
    val t <: UglyPoly
        = X.mult(7).plus(
            X.mult(X).mult(X).mult(6));
    val u <: UglyPoly
        = const(3).plus(
            X.mult(5).minus(X.mult(X).mult(7)));
    val v <: UglyPoly = t.mult(u).minus(1);
    for( i in -3 .. 3) {
        x10.io.Console.OUT.println(
            "" + i + " X:" + X.apply(i) + " t:" + t.apply(i)
            + " u:" + u.apply(i) + " v:" + v.apply(i)
        );
    }
}
```

```

    );
}
}
```

The operator-using code can be written in X10, though a few variations are necessary to handle such exotic cases as $1+X$.

Most X10 operators can be given definitions.⁵ (However, `&&` and `||` are only short-circuiting for Boolean expressions; user-defined versions of these operators have no special execution behavior.)

The user-definable operations are (in order of precedence):

```

implicit type coercions
postfix ()
as T
these unary operators: - + ! ~ | & / ^ * %
..
*      /      %      **
+      -
<<     >>     >>>   ->     <-     >-     -<     !
>     >=     <     <=     ~     !~
&
^
|
&&
||
```

Several of these operators have no standard meaning on any library type, and are included purely for programmer convenience.

Many operators may be defined either in `static` or instance forms. Those defined in instance form are dynamically dispatched, just like an instance method. Those defined in static form are statically dispatched, just like a static method. Operators are scoped like methods; static operators are scoped like static methods.

Example:

```

static class Trace(n:Long){
    public static operator !(f:Trace)
        = new Trace(10 * f.n + 1);
    public operator -this = new Trace (10 * this.n + 2);
}
```

⁵Indeed, even for the standard types, these operators are defined in the library. Not even as basic an operation as integer addition is built into the language. Conversely, if you define a full-featured numeric type, it will have most of the privileges that the standard ones enjoy. The missing privileges are (1) literals; (2) * won't track ranks, as it does for Regions; (3) `&&` and `||` won't short-circuit, as they do for Booleans, (4) the built-in notion of equality `a==b` will only coincide with the programmable notion `a.equals(b)`, as they do for most library types, if coded that way; and (5) it is impossible to define an operation like `String.+` which converts both its left and right arguments from any type. For example, a `Polar` type might have many representations for the origin, as radius 0 and any angle; these will be `equals()`, but will not be `==`; whereas for the standard `Complex` type, the two equalities are the same.

```

static class Brace extends Trace{
    def this(n:Long) { super(n); }
    public operator -this = new Brace (10 * this.n + 3);
    static def example() {
        val t = new Trace(1);
        assert (!t).n == 11;
        assert (-t).n == 12 && (-t instanceof Trace);
        val b = new Brace(1);
        assert (!b).n == 11;
        assert (-b).n == 13 && (-b instanceof Brace);
    }
}

```

8.7.1 Binary Operators

Binary operators, illustrated by +, may be defined statically in a container A as:

```
static operator (b:B) + (c:C) = ...;
```

Or, it may be defined as as an instance operator by one of the forms:

```
operator this + (b:B) = ...;
operator (b:B) + this = ...;
```

Example:

Defining the sum P+Q of two polynomials looks much like a method definition. It uses the operator keyword instead of def, and this appears in the definition in the place that a Poly would appear in a use of the operator. So, operator this + (p:Poly) explains how to add this to a Poly value.

```

class Poly {
    public val coeff : Rail[Long];
    public def this(coeff: Rail[Long]) {
        this.coeff = coeff;}
    public def degree() = coeff.size-1;
    public def a(i:Long)
        = (i<0 || i>this.degree()) ? 0L : coeff(i);
    public operator this + (p:Poly) =  new Poly(
        new Rail[Long](
            Math.max(this.coeff.size, p.coeff.size),
            (i:Long) => this.a(i) + p.a(i)
        ));
    // ...
}

```

The sum of a polynomial and an integer, P+3, looks like an overloaded method definition.

```
public operator this + (n : Long)
= new Poly([n as Long]) + this;
```

However, we want to allow the sum of an integer and a polynomial as well: 3+P. It would be quite inconvenient to have to define this as a method on Long; changing Long is far outside of normal coding. So, we allow it as a method on Poly as well.

```
public operator (n : Long) + this
= new Poly([n as Long]) + this;
```

Furthermore, it is sometimes convenient to express a binary operation as a static method on a class. The definition for the sum of two Polys could have been written:

```
public static operator (p:Poly) + (q:Poly) = new Poly(
    new Rail[Long](
        Math.max(q.coeff.size, p.coeff.size),
        (i:Long) => q.a(i) + p.a(i)
    ));

```

When X10 attempts to typecheck a binary operator expression like P+Q, it first typechecks P and Q. Then, it looks for operator declarations for + in the types of P and Q. If there are none, it is a static error. If there is precisely one, that one will be used. If there are several, X10 looks for a *best-matching* operation, *viz.* one which does not require the operands to be converted to another type. For example, `operator this + (n:Long)` and `operator this + (n:Int)` both apply to `p+1n`, because `1n` can be converted from an Int to a Long. However, the Int version will be chosen because it does not require a conversion. If even the best-matching operation is not uniquely determined, the compiler will report a static error.

8.7.2 Unary Operators

Unary operators, illustrated by `!`, may be defined statically in container A as

```
static operator !(x:A) = ...;
```

or as instance operators by:

```
operator !this = ...;
```

The rules for typechecking a unary operation are the same as for methods; the complexities of binary operations are not needed.

Example: The operator to negate a polynomial is:

```
public operator - this = new Poly(
    new Rail[Long](coeff.size, (i:Long) => -coeff(i))
);
```

8.7.3 Type Conversions

Explicit type conversions, `e as A`, can be defined as operators on class `A`, or on the container type of `e`. These must be static operators.

To define an operator in `class A` (or `struct A`) converting values of type `B` into type `A`, use the syntax:

```
static operator (x:B) as ? {c} = ...
```

The `?` indicates the containing type `A`. The guard clause `{c}` may be omitted.

Example:

```
class Poly {
    public val coeff : Rail[Long];
    public def this(coeff: Rail[Long]) { this.coeff = coeff; }
    public static operator (a:Long) as ? = new Poly([a as Long]);
    public static def main(Rail[String]):void {
        val three : Poly = 3L as Poly;
    }
}
```

The `?` may be given a bound, such as `as ? <: Caster`, if desired.

There is little difference between an explicit conversion `e as T` and a method call `e.ast()`. The explicit conversion does say undeniably what the result type will be. However, as described in §11.22.3, sometimes the built-in meaning of `as` as a cast overrides the user-defined explicit conversion.

Explicit casts are most suitable for cases which resemble the use of explicit casts among the arithmetic types, where, for example, `1.0 as Int` is a way to turn a floating-point number into the corresponding integer. While there is nothing in X10 which requires it, `e as T` has the connotation that it gives a good approximation of `e` in type `T`, just as `1` is a good (indeed, perfect) approximation of `1.0` in type `Int`.

8.7.4 Implicit Type Coercions

An implicit type conversion from `U` to `T` may be specified in container `T`. The syntax for it is:

```
static operator (u:U) : T = e;
```

Implicit coercions are used automatically by the compiler on method calls (§8.12) and assignments (§11.7). Implicit coercions may be used when a value of type `T` appears in a context expecting a value of type `U`. If `T <: U`, no implicit coercion is needed; *e.g.*, a method `m` expecting an `Long` argument may be called as `m(3)`, with an argument of type `Long{self==3}`, which is a subtype of `m`'s argument type `Long`. However, if it is not the case that `T <: U`, but there is an implicit coercion from `T` to `U` defined in container `U`, then this implicit coercion will be applied.

Example: We can define an implicit coercion from Long to Poly, and avoid having to define the sum of an integer and a polynomial as many special cases. In the following example, we only define + on two polynomials. The calculation 1+x coerces 1 to a polynomial and uses polynomial addition to add it to x.

```
public static operator (c : Long) : Poly
= new Poly([c as Long]);

public static operator (p:Poly) + (q:Poly) = new Poly(
    new Rail[Long](
        Math.max(p.coeff.size, q.coeff.size),
        (i:Long) => p.a(i) + q.a(i)
    ));
}

public static def main(Rail[String]):void {
    val x = new Poly([0L,1L]);
    x10.io.Console.OUT.println("1+x=" + (1L+x));
}
```

8.7.5 Assignment and Application Operators

X10 allows types to implement the subscripting / function application operator, and indexed assignment. The Array-like classes take advantage of both of these in `a(i) = a(i) + 1`.

`a(b, c, d)` is an operator call, to an operator defined with `public operator this(b:B, c:C, d:D)`. It may be overloaded. For example, an ordered dictionary structure could allow subscripting by numbers with `public operator this(i:Long)`, and by strings with `public operator this(s:String)`.

`a(i, j)=b` is an operator as well, with zero or more indices `i, j`. It may also be overloaded.

The update operations `a(i) += b` (for all binary operators in place of `+`) are defined to be the same as the corresponding `a(i) = a(i) + b`. This applies for all arities of arguments, and all types, and all binary operations. Of course to use this, the `+`, application and assignment operators must be defined.

Example:

The Oddvec class of somewhat peculiar vectors illustrates this.

`a()` returns a string representation of the oddvec, which ordinarily would be done by `toString()` instead. `a(i)` sensibly picks out one of the three coordinates of `a`. `a()=b` sets all the coordinates of `a` to `b`. `a(i)=b` assigns to one of the coordinates. `a(i,j)=b` assigns different values to `a(i)` and `a(j)`.

```
class Oddvec {
    var v : Rail[Long] = new Rail[Long](3);
```

```

public operator this () =
    "(" + v(0) + "," + v(1) + "," + v(2) + ")";
public operator this () = (newval: Long) {
    for(p in v.range) v(p) = newval;
}
public operator this(i:Long) = v(i);
public operator this(i:Long, j:Long) = [v(i),v(j)];
public operator this(i:Long) = (newval:Long)
    {v(i) = newval;}
public operator this(i:Long, j:Long) = (newval:Long)
    {v(i) = newval; v(j) = newval+1;}
public def example() {
    this(1) = 6; assert this(1) == 6;
    this(1) += 7; assert this(1) == 13;
}

```

8.8 User-Defined Control Structures

KeywordOpDecln ::= *MethMods* operator *keywordOp* *TypeParams?* *Formals* *Guard?* (20.105)
Throws? *HasResultType?* *MethodBody*

KeywordOp ::= for (20.104)

- | if
- | try
- | throw
- | async
- | atomic
- | when
- | finish
- | at
- | continue
- | break
- | aeach
- | while
- | do

Similarly to user-defined operators (Section 8.7), it is possible to redefine the behavior of some control structures. For example, suppose that we want to define a *if* statement that randomly chooses which branch to execute. In a class *RandomIf*, we define a method named *if* (introduced with the keyword *operator*) that implements this behavior:

```

class RandomIf {
    val random = new Random();
    public operator if(then: ()=>void, else_: ()=>void) {

```

```

        if (random.nextBoolean()) {
            then();
        } else {
            else_();
        }
    }
}

```

Then, we can call this method using the syntax of the `if` statement by prefixing the `if` keyword by an object that implements this method:

```

val random = new RandomIf();
random.if () {
    Console.OUT.println("true");
} else {
    Console.OUT.println("false");
}

```

The blocks that represent the `then` and the `else` branches of the `if` are automatically turned into closures and are given as argument to the `RandomIf.if` method.

To distinguish the use of a user-defined control structure from the use of a built-in one, the first keyword of the control structure must be prefixed with the object that redefines its behavior. The scoping and dispatching rules of user-defined control structures are exactly the same as the rules for methods.

User-defined control structures can also be called as standard methods using the keyword `operator` as prefix (as for user-defined operators). For example, the previous code is equivalent to:

```

val random = new RandomIf();
random.operator if () => { Console.OUT.println("true"); },
                           () => { Console.OUT.println("false"); });

```

The correspondence between the two invocation syntaxes is formally specified in Figure 8.1 for all the control structures we support. It uses the following conventions: o is either a class path or an object; \bar{T} is a list of types; $\bar{x}:t$ is a list of variable declaration with their types; \bar{e} is a list of expressions; b is a closure body: a list of statements between curly braces that can optionally end with an expression (a return value); $()^?$ is an optional group.

Let's consider the rule for `if`:

$$\begin{aligned}
 o.\text{if}[\bar{T}]^?(\bar{e})\ b_1 (\text{else } b_2)^? &\equiv \\
 o.\text{operator if}[\bar{T}]^?(\bar{e}, ()=>b_1(), ()=>b_2)^?;
 \end{aligned}$$

Compared to the builtin `if` control structure, the user-defined one accepts type arguments and replaces one condition expression with a list of expressions, possibly empty. The branches of the user-defined `if` statement are lifted to no-arg closures and passed to the user-defined `if` method as arguments. The `else` branch is optional.

```

o.if[ $\bar{T}$ ]? $(\bar{e})$  b1 (else b2)? $\equiv$ 
    o.operator if[ $\bar{T}$ ]? $(\bar{e}, ()=>b_1(), ()=>b_2())$ ;
o.for[ $\bar{T}$ ]? $((x:t \text{ in})? \bar{e})$  b $\equiv$ 
    o.operator for[ $\bar{T}$ ]? $(\bar{e}, (\overline{x:t})=>b)$ ;
o.try[ $\bar{T}$ ]? $(\bar{e})$ ? b1 catch ( $\overline{x:t}$ ) b2 (finally b3)? $\equiv$ 
    o.operator try[ $\bar{T}$ ]? $((\bar{e},)? ()=>b_1(), (\overline{x:t})=>b_2(), ()=>b_3())$ ;
o.throw[ $\bar{T}$ ]? $e?$ ;  $\equiv$  o.operator throw[ $\bar{T}$ ]? $(e?)$ ;
o.async[ $\bar{T}$ ]? $(\bar{e}_1)$ ? (clocked ( $\bar{e}_2$ ))? $b$  $\equiv$ 
    o.operator async[ $\bar{T}$ ]? $((\bar{e}_1),)? (\bar{e}_2,) ()=>b$ ;
o.atomic[ $\bar{T}$ ]? $(\bar{e})$ ? b  $\equiv$  o.operator atomic[ $\bar{T}$ ]? $((\bar{e},)? ()=>b)$ ;
o.when[ $\bar{T}$ ]? $(\bar{e})$  b  $\equiv$  o.operator when[ $\bar{T}$ ]? $(\bar{e}, ()=>b)$ ;
o.finish[ $\bar{T}$ ]? $(\bar{e})$ ? b  $\equiv$  o.operator finish[ $\bar{T}$ ]? $((\bar{e},)? ()=>b)$ ;
o.at[ $\bar{T}$ ]? $(\bar{e})$  b  $\equiv$  o.operator at[ $\bar{T}$ ]? $(\bar{e}, ()=>b)$ ;
o.continue[ $\bar{T}$ ]? $e?$ ;  $\equiv$  o.operator continue[ $\bar{T}$ ]? $(e?)$ ;
o.break[ $\bar{T}$ ]? $e?$ ;  $\equiv$  o.operator break[ $\bar{T}$ ]? $(e?)$ ;
o.ateach[ $\bar{T}$ ]? $((x:t \text{ in})? \bar{e})$  b  $\equiv$ 
    o.operator ateach[ $\bar{T}$ ]? $(\bar{e}, (\overline{x:t})=>b)$ ;
o.while[ $\bar{T}$ ]? $(\bar{e})$  b  $\equiv$  o.operator while[ $\bar{T}$ ]? $(\bar{e}, ()=>b)$ ;
o.do[ $\bar{T}$ ]? $b$  while ( $\bar{e}$ );  $\equiv$  o.operator do[ $\bar{T}$ ]? $((()=>b, \bar{e}))$ ;

```

Figure 8.1: Correspondence between control structure and method call notations.

The correspondence is purely syntactic. In other words, the control structure syntax is simply rewritten into the regular method invocation syntax with no consideration of types or method lookup.

8.8.1 User-Defined for

```

o.for[ $\bar{T}$ ]? $((x:t \text{ in})? \bar{e})$  b  $\equiv$ 
    o.operator for[ $\bar{T}$ ]? $(\bar{e}, (\overline{x:t})=>b)$ ;

```

A **for** loop over a collection may be defined in a container **A** as:

```
operator for[T](c: Iterable[T], body: (T)=>void) = ...
```

The use of such a user-defined **for** loop would have the following form:

```
A.for (x: T in c) { ... }
```

and would correspond to the following method call:

```
A.operator for (c, (x: Long) => { ... }) ;
```

The body of the `for` is automatically translated into a closure that takes the iteration variable as parameter. Since there is no type inference for closure parameters, the type of the iteration variable must be given explicitly.

The second argument of a `for` method can be a closure without argument:

```
operator for[T](c: Iterable[T], body: ()=>void) = ...
```

In this case, the method is called using the syntax of a `for` loop without iteration variable:

```
A.for (c) { ... }
```

Example: *A naive implementation of a parallel loop can be:*

```
class Parallel {

    public static operator for[T](c: Iterable[T], body: (T)=>void) {
        finish {
            for(x in c) {
                async { body(x); }
            }
        }
    }

    public static def main(Rail[String]) {
        val cpt = new Cell[Long](0);
        Parallel.for(i:Long in 1..10) {
            atomic { cpt() = cpt() + i; }
        }
        Console.OUT.println(cpt());
    }
}
```

Example: *We can also use the user-defined for loops to define iterations over a two dimensional space. Let us define a loop that creates an activity for each element of the first dimension.*

```
class Parallel2 {
    public static operator for (space: DenseIterationSpace_2,
                                body: (i:Long, j:Long)=>void) {
        finish {
            for (i in space.min0 .. space.max0) {
                async for (j in space.min1 .. space.max1) {
                    body(i, j);
                }
            }
        }
    }
}
```

and it can be used as follows:

```
Parallel2.for (i:Long, j:Long in 1..10 * 1..10) { ... }
```

The list of variables before the `in` keyword becomes the parameters of the closure whose body is the body of the loop.

8.8.2 User-Defined if

$$\begin{aligned} o.\text{if}[\bar{T}]^?(\bar{e})\ b_1 (\text{else } b_2)^? &\equiv \\ o.\text{operator}\ \text{if}[\bar{T}]^?(\bar{e}, ()=>b_1(), ()=>b_2)^?; \end{aligned}$$

When we use a user-defined `if` statement, the condition is evaluated before calling the `if` method, but the then and else branches are implicitly lifted to closures without argument.

Note that the condition of a user-defined `if` statement can take an arbitrary number of arguments. This is why we were able to define the `Random.if` that does not take a condition.

8.8.3 User-Defined try

$$\begin{aligned} o.\text{try}[\bar{T}]^?(\bar{e})^? b_1 \text{ catch } (\bar{x}:\bar{t})\ b_2 (\text{finally } b_3)^? &\equiv \\ o.\text{operator}\ \text{try}[\bar{T}]^?((\bar{e},)^?)\ ()=>b_1, (\bar{x}:\bar{t})=>b_2(), ()=>b_3)^?; \end{aligned}$$

When we use a user-defined `try` statement, the body of the `try` is lifted to a closure without argument and handler is lifted to a closure that has the parameter of the `catch` as parameter. The `finally` block is also lifted to a closure without argument.

Example: The user-defined `try` construct can be used to provide a control structure that automatically removes the nesting of `MultipleExceptions`:

```
class Flatten {

    public static operator try(body:()=>void,
                               handler:(MultipleExceptions)=>void) {
        try { body(); }
        catch (me: MultipleExceptions) {
            val exns = new GrowableRail[CheckedThrowable]();
            flatten(me, exns);
            handler (new MultipleExceptions(exns));
        }
    }

    private static def flatten(me:MultipleExceptions,
                               acc:GrowableRail[CheckedThrowable]) {
        for (e in me.exceptions) {
```

```
        if (e instanceof MultipleExceptions) {  
            flatten(e as MultipleExceptions, acc);  
        } else {  
            acc.add(e);  
        }  
    }  
}
```

Used in the following example, the MultipleExceptions me contains the exceptions Exception("Exn 1"), Exception("Exn 2"), and Exception("Exn 3") instead of the exception Exception("Exn 1") and another MultipleExceptions.

```
public static def main(Rail[String]) {
    Flatten.try {
        finish {
            async { throw new Exception("Exn 1"); }
            async finish {
                async { throw new Exception("Exn 2"); }
                async { throw new Exception("Exn 3"); }
            }
        }
    } catch (me: MultipleExceptions) {
        Console.OUT.println(me.exceptions);
    }
}
```

8.8.4 User-Defined throw

o.throw[\overline{T}]?*e*?; \equiv *o.operator throw*[\overline{T}]?(*e*?);

The argument of a user-defined `throw` is evaluated before calling the `throw` method.

8.8.5 User-Defined `async`

o.operator **async**[\overline{T}]? $(\overline{e_1})$? (**clocked** ($\overline{e_2}$))? b \equiv
o.operator **async**[\overline{T}]? $((\overline{e_1},)$? $(\overline{e_2},)$? $(_) \Rightarrow b$);

The body of a user-defined `async` is lifted to a closure without argument. The clock arguments are evaluated before the call to the `async` method.

Example: An `async` that does not execute in the scope in which it is written. The task is created in the scope where the object that defines the `async` method is instantiated.

```

class Escape {
    private var task: ()=>void = null;
    private var stop: Boolean = false;

    public def this() {
        async {
            while (!stop) {
                val t: () => void;
                when (task != null || stop) {
                    t = task;
                    task = null;
                }
                if (t != null) {
                    async { t(); }
                }
            }
        }
    }

    public operator async (body: () => void) {
        when (task == null) {
            task = body;
        }
    }

    public def stop() {
        atomic { stop = true; }
    }
}

```

In the following example, the message "OK" is printed even if the created task never terminates because the task is executed outside of the scope of the finish.

```

public static def main(Rail[String]) {
    val toplevel = new Escape();
    finish {
        toplevel.async { when (false){} }
    }
    Console.OUT.println("OK");
}

```

8.8.6 User-Defined atomic

$$o.\text{atomic}[\bar{T}]^?(\bar{e})^? b \equiv o.\text{operator atomic}[\bar{T}]^?((\bar{e},)^? ()=> b);$$

The body of a user-defined atomic statement is lifted to a closure without argument.

8.8.7 User-Defined when

$o.\text{when}[\bar{T}]^?(\bar{e})\ b \equiv o.\text{operator when}[\bar{T}]^?(\bar{e}, \text{O}=>b);$

The arguments of a user-defined when statements are evaluated before the call of the when method and the body is lifted to a closure without argument. It means that if the argument of a user-defined when is of type Boolean, the condition is evaluated once and cannot be changed. To be able to update the condition, it can be an object with mutable field as in the following example or a closure.

Example: We can provide a when statement whose execution can be canceled while it is waiting:

```
class CancelableWhen {
    private var stop : Boolean = false;

    public operator when(condition:Cell[Boolean], body:O=>void) {
        when (condition() || stop) {
            if (!stop) { body(); }
        }
    }

    public def cancel() {
        atomic { stop = true; }
    }
}
```

The following example will not print the message "KO" but will terminate even if the condition b of the when remains false:

```
public static def main(Rail[String]) {
    val c = new CancelableWhen();
    val b = new Cell[Boolean](false);
    finish {
        async {
            c.when(b) { Console.OUT.println("KO"); }
        }
        c.cancel();
    }
}
```

8.8.8 User-Defined finish

$o.\text{finish}[\bar{T}]^?(\bar{e})^? b \equiv o.\text{operator finish}[\bar{T}]^?((\bar{e},)^? \text{O}=>b);$

The body of a user-defined finish is lifted to a closure.

Example: We define a finish that provide the ability to some parallel task to wait for its termination:

```

class SignalingFinish {
    private var terminated : Boolean = false;
    public operator finish(body: ()=>void) {
        finish {
            body();
        }
        atomic { terminated = true; }
    }
    public def join() {
        when (terminated) {}
    }
}

```

The following example will always print the message "before" before the message "after".

```

public static def main(Rail[String]) {
    val t = new SignalingFinish();
    async {
        t.join();
        Console.OUT.println("after");
    }
    t.finish {
        Console.OUT.println("before");
    }
}

```

8.8.9 User-Defined at

$$o.\text{at}[\bar{T}]^?(\bar{e})\ b \equiv o.\text{operator at}[\bar{T}]^?(\bar{e}, ()=>b);$$

The arguments of the user-defined at statement are evaluated before the call of the at method and the body of the statement is lifted to a closure without argument.

Example: We define a class Ring implementing an at statement without argument. Each call to this user-defined at statement moves the activity to the next place in the place group given when the object is instantiated.

```

class Ring {
    val places: PlaceGroup;

    public def this (places: PlaceGroup) {
        this.places = places;
    }

    public operator at(body: ()=>void) {

```

```

        at(places.next(here)) { body(); }
    }
}

public static def main(Rail[String]) {
    val r = new Ring(Place.places());
    r.at() {
        Console.OUT.println("Hello from "+here+"!");
        r.at() {
            Console.OUT.println("Hello from "+here+"!");
        }
    }
}

```

8.8.10 User-Defined `ateach`

$$\begin{aligned} o.\text{ateach}[\bar{T}]^?((\bar{x}:\bar{t} \text{ in})^? \bar{e}) \ b &\equiv \\ o.\text{operator ateach}[\bar{T}]^?(\bar{e}, (\bar{x}:\bar{t}^?) \Rightarrow b); \end{aligned}$$

The arguments of the user-defined `ateach` statement are evaluated before the call of the `ateach` method and the body of the statement is lifted to a closure without argument.

Example: *An `ateach` control structure that has the same behavior as the built-in `ateach`, except that the activities are executed in sequence instead of being executed in parallel.*

```

class Sequential {
    public static operator ateach (d: Dist, body:(Point)=>void) {
        for (place in d.places()) {
            at(place) {
                for (p in d|here) { body(p); }
            }
        }
    }
}

```

8.8.11 User-Defined `while` and `do`

$$\begin{aligned} o.\text{while}[\bar{T}]^?(\bar{e}) \ b &\equiv o.\text{operator while}[\bar{T}]^?(\bar{e}, () \Rightarrow b); \\ o.\text{do}[\bar{T}]^? b \text{ while } (\bar{e}); &\equiv o.\text{operator do}[\bar{T}]^?(() \Rightarrow b, \bar{e}); \end{aligned}$$

The arguments of the user-defined `while` (resp. `do`) are evaluated before the call of the `while` (resp. `do`) method and the body of the loop is lifted to a closure without argument. Note that compared to usual loop, the condition is evaluated once before the call of the method that implements the behavior of the loop.

Example: *A loop that iterates during at least a given number of milliseconds:*

```
class Timeout {
    public static operator while(ms: Long, body: ()=>void) {
        val deadline = System.currentTimeMillis() + ms;
        while (System.currentTimeMillis() < deadline) {
            body();
        }
    }
}
```

Here, we increment a counter during a period of at least 10 milliseconds:

```
public static def main(Rail[String]) {
    val cpt = new Cell[Long](0);
    Timeout.while(10) {
        atomic { cpt() = cpt() + 1; }
    }
    Console.OUT.println(cpt());
}
```

8.8.12 User-Defined continue

`o.continue[\overline{T}]? $e^?$; ≡ o.operator continue[\overline{T}]?($e^?$);`

The argument of a user-defined `continue` is evaluated before calling the corresponding method.

Example: The following code provides a parallel for loop with a `continue` statement that allows skipping an iteration.

```
class Par {
    private static class Continue extends Exception {}

    public static operator continue () {
        throw new Continue();
    }

    public static operator for[T](c: Iterable[T], body:(T)=>void) {
        finish {
            for(x in c) async {
                try {
                    body(x);
                } catch (Continue) {}
            }
        }
    }
}
```

The following example skips every iteration where the loop index is even.

```
public static def main(Rail[String]) {
    val cpt = new Cell[Long](0);
    Par.for(i:Long in 1..10) {
        if (i%2 == 0) { Par.continue; }
        atomic { cpt() = cpt() + 1; }
    }
    Console.OUT.println(cpt());
}
```

8.8.13 User-Defined break

$$o.\text{break}[\bar{T}]^? e?; \equiv o.\text{operator break}[\bar{T}]^?(e?);$$

The argument of a user-defined `break` is evaluated before calling the corresponding method.

Example: To break out of a user-defined loop, it is necessary to also define the `break` statement:

```
class Infinite {
    private static class Break extends Exception {}

    public static operator break () {
        throw new Break();
    }

    public static operator while (body:()=>void) {
        try {
            while(true) {
                body();
            }
        } catch (Break) {}
    }

    public static def main(Rail[String]) {
        Infinite.while() {
            Infinite.break;
        }
        Console.OUT.println("OK");
    }
}
```

8.9 Class Guards and Invariants

Classes (and structs and interfaces) may specify a *class guard*, a constraint which must hold on all values of the class. In the following example, a Line is defined by two distinct Pts⁶

```
class Pt(x:Long, y:Long){}
class Line(a:Pt, b:Pt){a != b} {}
```

In most cases the class guard could be phrased as a type constraint on a property of the class instead, if preferred. Arguably, a symmetric constraint like two points being different is better expressed as a class guard, rather than asymmetrically as a constraint on one type:

```
class Line(a:Pt, b:Pt{a != b}) {}
```

With every container or interface T we associate a *type invariant* $inv(T)$, which describes the guarantees on the properties of values of type T.

Every value of T satisfies $inv(T)$ at all times. This is somewhat stronger than the concept of type invariant in most languages (which only requires that the invariant holds when no method calls are active). X10 invariants only concern properties, which are immutable; thus, once established, they cannot be falsified.

The type invariant associated with `x10.lang.Any` is `true`.

The type invariant associated with any interface or struct I that extends interfaces I_1, \dots, I_k and defines properties $x_1: P_1, \dots, x_n: P_n$ and specifies a guard c is given by:

```
inv(I1) && ... && inv(Ik) &&
self.x1 instanceof P1 && ... && self.xn instanceof Pn
&& c
```

Similarly the type invariant associated with any class C that implements interfaces I_1, \dots, I_k , extends class D and defines properties $x_1: P_1, \dots, x_n: P_n$ and specifies a guard c is given by the same thing with the invariant of the superclass D conjoined:

```
inv(I1) && ... && inv(Ik)
&& self.x1 instanceof P1 && ... && self.xn instanceof Pn
&& c
&& inv(D)
```

Note that the type invariant associated with a class entails the type invariants of each interface that it implements (directly or indirectly), and the type invariant of each ancestor class. It is guaranteed that for any variable v of type $T\{c\}$ (where T is an interface name or a class name) the only objects o that may be stored in v are such that o satisfies $inv(T[o/this]) \wedge c[o/self]$.

⁶We use Pt to avoid any possible confusion with the built-in class Point.

8.9.1 Invariants for implements and extends clauses

Consider a class definition

```
ClassModifiers?
class C(x1: P1, ..., xn: Pn) {c} extends D{d}
    implements I1{c1}, ..., Ik{ck}
ClassBody
```

These two rules must be satisfied:

- The type invariant $inv(C)$ of C must entail $c_i[\text{this}/\text{self}]$ for each i in $\{1, \dots, k\}$
- The return type c of each constructor in a class C must entail the invariant $inv(C)$.

8.9.2 Timing of Invariant Checks

The invariants for a container are checked immediately after the `property` statement in the container's constructor. This is the earliest that the invariant could possibly be checked. Recall that an invariant can mention the properties of the container (which are set, forever, at that point in the code), but cannot mention the `val` or `var` fields (which might not be set at that point), or `this` (which might not have been fully initialized).

If X10 can prove that the invariant always holds given the `property` statement and other known information, it may omit the actual check.

8.9.3 Invariants and constructor definitions

A constructor for a class C is guaranteed to return an object of the class on successful termination. This object must satisfy $inv(C)$, the class invariant associated with C (§8.9). However, often the objects returned by a constructor may satisfy *stronger* properties than the class invariant. X10's dependent type system permits these extra properties to be asserted with the constructor in the form of a constrained type (the “return type” of the constructor):

```
CtorDecln ::= Mods? def this TypeParams? Formals Guard? HasResultType? Ctor-Body (20.54)
```

The parameter list for the constructor may specify a *guard* that is to be satisfied by the parameters to the list.

Example: Here is another example, constructed as a simplified version of `x10.regionarray.Region`. The `mockUnion` method has the type, though not the value, that a true union method would have.

```

class MyRegion(rank:Long) {
    static type MyRegion(n:Long)=MyRegion{rank==n};
    def this(r:Long):MyRegion(r) {
        property(r);
    }
    def this(diag:Rail[Long]):MyRegion(diag.size){
        property(diag.size);
    }
    def mockUnion(r:MyRegion(rank)):MyRegion(rank) = this;
    def example() {
        val R1 : MyRegion(3L) = new MyRegion([4,4,4 as Long]);
        val R2 : MyRegion(3L) = new MyRegion([5,4,1]);
        val R3 = R1.mockUnion(R2); // inferred type MyRegion(3)
    }
}

```

The first constructor returns the empty region of rank r. The second constructor takes a Rail[Long] of arbitrary length n and returns a MyRegion(n) (intended to represent the set of points in the rectangular parallelopiped between the origin and the diag.)

The code in example typechecks, and R3's type is inferred as MyRegion(3).

Let C be a class with properties $p_1: P_1, \dots, p_n: P_n$, and invariant c extending the constrained type D{d} (where D is the name of a class).

For every constructor in C the compiler checks that the call to super invokes a constructor for D whose return type is strong enough to entail d. Specifically, if the call to super is of the form super(e₁, ..., e_k) and the static type of each expression e_i is S_i, and the invocation is statically resolved to a constructor def this(x₁: T₁, ..., x_k: T_k) {c}: D{d} then it must be the case that

$$\begin{aligned} x_1: S_1, \dots, x_i: S_i &\text{ entails } x_i: T_i \quad (\text{for } i \in \{1, \dots, k\}) \\ x_1: S_1, \dots, x_k: S_k &\text{ entails } c \\ d_1[a/\text{self}], x_1: S_1, \dots, x_k: S_k &\text{ entails } d[a/\text{self}] \end{aligned}$$

where a is a constant that does not appear in $x_1: S_1 \wedge \dots \wedge x_k: S_k$.

The compiler checks that every constructor for C ensures that the properties p_1, \dots, p_n are initialized with values which satisfy $\text{inv}(T)$, and its own return type c' as follows. In each constructor, the compiler checks that the static types T_i of the expressions e_i assigned to p_i are such that the following is true:

$$p_1: T_1, \dots, p_n: T_n \text{ entails } \text{inv}(T) \wedge c'$$

(Note that for the assignment of e_i to p_i to be type-correct it must be the case that $p_i: T_i \wedge p_i: P_i$.)

The compiler must check that every invocation C(e₁, ..., e_n) to a constructor is type correct: each argument e_i must have a static type that is a subtype of the declared type T_i for the i th argument of the constructor, and the conjunction of static types of the arguments must entail the constraint in the parameter list of the constructor.

8.10 Generic Classes

Classes, like other units, can be generic. They can be parameterized by types. The parameter types are used just like ordinary types inside the body of the generic class – with a few exceptions.

Example: A `Colorized[T]` holds a thing of type `T`, and a string which is intended to represent its color. Any type can be used for `T`; the `example` method shows `Long` and `Boolean`. The `thing()` method retrieves the thing; note that its return type is the generic type variable `T`. `X10` is aware that `colLong.thing()` is an `Long` and `colTrue.thing()` is a `Boolean`, and uses those typings in `example`.

```
class Colorized[T] {
    private var thing:T;
    private var color:String;
    def this(thing:T, color:String) {
        this.thing = thing;
        this.color = color;
    }
    public def thing():T = thing;
    public def color():String = color;
    public static def example() {
        val colLong : Colorized[Long]
            = new Colorized[Long](3, "green");
        assert colLong.thing() == 3
            && colLong.color().equals("green");
        val colTrue : Colorized[Boolean]
            = new Colorized[Boolean](true, "blue");
        assert colTrue.thing()
            && colTrue.color().equals("blue");
    }
}
```

8.10.1 Use of Generics

An unconstrained type variable `X` can be instantiated by any type. All the operations of `Any` are available on a variable of type `X`. Additionally, variables of type `X` may be used with `==`, `!=`, in `instanceof`, and casts.

If a type variable is constrained, the operations implied by its constraint are available as well.

Example: The interface `Named` describes entities which know their own name. The class `NameMap[T]` is a specialized map which stores and retrieves `Named` entities by name. The call `t.name()` in `put()` is only valid because the constraint `{T <: Named}` implies that `T` is a subtype of `Named`, and hence provides all the operations of `Named`.

```

interface Named { def name():String; }
class NameMap[T]{T <: Named, T haszero} {
    val m = new HashMap[String, T]();
    def put(t:T) { m.put(t.name(), t); }
    def get(s:String):T = m.getOrThrow(s);
}

```

8.11 Object Initialization

X10 does object initialization safely. It avoids certain bad things which trouble some other languages:

1. Use of a field before the field has been initialized.
2. A program reading two different values from a `val` field of a container.
3. `this` escaping from a constructor, which can cause problems as noted below.

It should be unsurprising that fields must not be used before they are initialized. At best, it is uncertain what value will be in them, as in `x` below. Worse, the value might not even be an allowable value; `y`, declared to be nonzero in the following example, might be zero before it is initialized.

```

// Not correct X10
class ThisIsWrong {
    val x : Long;
    val y : Long{y != 0};
    def this() {
        x10.io.Console.OUT.println("x=" + x + "; y=" + y);
        x = 1; y = 2;
    }
}

```

One particularly insidious way to read uninitialized fields is to allow `this` to escape from a constructor. For example, the constructor could put `this` into a data structure before initializing it, and another activity could read it from the data structure and look at its fields:

```

class Wrong {
    val shouldBe8 : Long;
    static Cell[Wrong] wrongCell = new Cell[Wrong]();
    static def doItWrong() {
        finish {
            async { new Wrong(); } // (A)
            assert( wrongCell().shouldBe8 == 8); // (B)
        }
    }
}

```

```

    }
    def this() {
        wrongCell.set(this); // (C) - ILLEGAL
        this.shouldBe8 = 8; // (D)
    }
}

```

In this example, the underconstructed `Wrong` object is leaked into a storage cell at line (C), and then initialized. The `doItWrong` method constructs a new `Wrong` object, and looks at the `Wrong` object in the storage cell to check on its `shouldBe8` field. One possible order of events is the following:

1. `doItWrong()` is called.
2. (A) is started. Space for a new `Wrong` object is allocated. Its `shouldBe8` field, not yet initialized, contains some garbage value.
3. (C) is executed, as part of the process of constructing a new `Wrong` object. The new, uninitialized object is stored in `wrongCell`.
4. Now, the initialization activity is paused, and execution of the main activity proceeds from (B).
5. The value in `wrongCell` is retrieved, and is `shouldBe8` field is read. This field contains garbage, and the assertion fails.
6. Now let the initialization activity proceed with (D), initializing `shouldBe8` — too late.

The `at` statement (§13.3) introduces the potential for escape as well. The following class prints an uninitialized value:

```

// This code violates this chapter's constraints
// and thus will not compile in X10.
class Example {
    val a: Long;
    def this() {
        at(here.next()) {
            // Recall that 'this' is a copy of 'this' outside 'at'.
            Console.OUT.println("this.a = " + this.a);
        }
        this.a = 1;
    }
}

```

X10 must protect against such possibilities. The rules explaining how constructors can be written are somewhat intricate; they are designed to allow as much programming as possible without leading to potential problems. Ultimately, they simply are elaborations of the fundamental principles that uninitialized fields must never be read, and `this` must never be leaked.

8.11.1 Constructors and Non-Escaping Methods

In general, constructors must not be allowed to call methods with `this` as an argument or receiver. Such calls could leak references to `this`, either directly from a call to `cell.set(this)`, or indirectly because `toString` leaks `this`, and the concatenation “`Escaper = "+this'` calls `toString`.⁷

```
// This code violates this chapter's constraints
// and thus will not compile in X10.
class Escaper {
    static val Cell[Escaper] cell = new Cell[Escaper]();
    def toString() {
        cell.set(this);
        return "Evil!";
    }
    def this() {
        cell.set(this);
        x10.io.Console.OUT.println("Escaper = " + this);
    }
}
```

However, it is convenient to be able to call methods from constructors; *e.g.*, a class might have eleven constructors whose common behavior is best described by three methods. Under certain stringent conditions, it *is* safe to call a method: the method called must not leak references to `this`, and must not read `vals` or `vars` which might not have been assigned.

So, X10 performs a static dataflow analysis, sufficient to guarantee that method calls in constructors are safe. This analysis requires having access to or guarantees about all the code that could possibly be called. This can be accomplished in two ways:

1. Ensuring that only code from the class itself can be called, by forbidding overriding of methods called from the constructor: they can be marked `final` or `private`, or the whole class can be `final`.
2. Marking the methods called from the constructor by `@NonEscaping` or `@NoThisAccess`

Non-Escaping Methods

A method may be annotated with `@NonEscaping`. This imposes several restrictions on the method body, and on all methods overriding it. However, it is the only way that a method can be called from constructors. The `@NonEscaping` annotation makes explicit all the X10 compiler’s needs for constructor-safety.

A method can, however, be safe to call from constructors without being marked `@NonEscaping`. We call such methods *implicitly non-escaping*. Implicitly non-escaping methods need

⁷This is abominable behavior for `toString`, but it cannot be prevented – save by a scheme such as we present in this section.

to obey the same constraints on `this`, `super`, and variable usage as `@NonEscaping` methods. An implicitly non-escaping method *could* be marked as `@NonEscaping`; the compiler, in effect, infers the annotation. In addition, all non-escaping methods must be `private` or `final` or members of a `final` class; this corresponds to the hereditary nature of `@NonEscaping` (by forbidding inheritance of implicitly non-escaping methods).

We say that a method is *non-escaping* if it is either implicitly non-escaping, or annotated `@NonEscaping`.

The first requirement on non-escaping methods is that they do not allow `this` to escape. Inside of their bodies, `this` and `super` may only be used for field access and assignment, and as the receiver of non-escaping methods.

The following example uses the possible variations. `aplomb()` explicitly forbids reading any field but `a`. `boric()` is called after `a` and `b` are set, but `c` is not. The `@NonEscaping` annotation on `boric()` is optional, but the compiler will print a warning if it is left out. `cajoled()` is only called after all fields are set, so it can read anything; its annotation, too, is not required. `SeeAlso` is able to override `aplomb()`, because `aplomb()` is `@NonEscaping`; it cannot override the final method `boric()` or the private one `cajoled()`.

```
import x10.compiler.*;

final class C2 {
    protected val a:Long; protected val b:Long; protected val c:Long;
    protected var x:Long; protected var y:Long; protected var z:Long;
    def this() {
        a = 1;
        this.aplomb();
        b = 2;
        this.boric();
        c = 3;
        this.cajoled();
    }
    @NonEscaping def aplomb() {
        x = a;
        // this.boric(); // not allowed; boric reads b.
        // z = b; // not allowed -- only 'a' can be read here
    }
    @NonEscaping final def boric() {
        y = b;
        this.aplomb(); // allowed;
        // a is definitely set before boric is called
        // z = c; // not allowed; c is not definitely written
    }
    @NonEscaping private def cajoled() {
        z = c;
    }
}
```

```
    }
}
```

NoThisAccess Methods

A method may be annotated `@NoThisAccess`. `@NoThisAccess` methods may be called from constructors, and they may be overridden in subclasses. However, they may not refer to `this` in any way – in particular, they cannot refer to fields of `this`, nor to `super`.

Example:

The class `IDed` has an `Float`-valued `id` field. The method `count()` is used to initialize the `id`. For `IDed` objects, the `id` is the count of `IDeds` created with the same parity of its kind. Note that `count()` does not refer to `this`, though it does refer to a static field `counts`.

The subclass `SubIDed` has `ids` that depend on `kind%3` as well as the parity of `kind`. It overrides the `count()` method. The body of `count()` still cannot refer to `this`. Nor can it refer to `super` (which is `self` under another name). This precludes the use of a super call. This is why we have separated the body of `count` out as the static method `kind2count` – without that, we would have had to duplicate its body, as we could not call `super.count(kind)` in a `NoThisAccess` method, as is shown by the ERROR line (A).

Note that `NoThisAccess` is in `x10.compiler` and must be imported, and that the overriding method `SubIDed.count` must be declared `@NoThisAccess` as well as the overridden method. Line (B) is not allowed because `code` is a field of `this`, and field accesses are forbidden. Line (C) references `this` directly, which, of course, is forbidden by `@NoThisAccess`.

```
import x10.compiler.*;
class UseNoThisAccess {
    static class IDed {
        protected static val counts = [0 as Long, 0];
        protected var code : Long;
        val id: Float;
        public def this(kind:Long) {
            code = kind;
            this.id = this.count(kind);
        }
        protected static def kind2count(kind:Long) = ++counts(kind % 2);
        @NoThisAccess def count(kind:Long) : Float = kind2count(kind);
    }
    static class SubIDed extends IDed {
        protected static val subcounts = [0 as Long, 0, 0];
        public static val all = new x10.util.ArrayList[SubIDed]();
        public def this(kind:Long) {
```

```

        super(kind);
    }
    @NoThisAccess
    def count(kind:Long) : Float {
        val subcount <: Long = ++subcounts(kind % 3);
        val supercount <: Float = kind2count(kind);
        //ERROR: val badSuperCount = super.count(kind); // (A)
        //ERROR: code = kind;                         // (B)
        //ERROR: all.add(this);                      // (C)
        return supercount + 1.0f / subcount;
    }
}
}
}

```

8.11.2 Fine Structure of Constructors

The code of a constructor consists of four segments, three of them optional and one of them implicit.

1. The first segment is an optional call to `this(...)` or `super(...)`. If this is supplied, it must be the first statement of the constructor. If it is not supplied, the compiler treats it as a nullary super-call `super()`;
2. If the class or struct has properties, there must be a single `property(...)` command in the constructor, or a `this(...)` constructor call. Every execution path through the constructor must go through this `property(...)` command precisely once. The second segment of the constructor is the code following the first segment, up to and including the `property()` statement.
If the class or struct has no properties, the `property()` call must be omitted. If it is present, the second segment is defined as before. If it is absent, the second segment is empty.
3. The third segment is automatically generated. Fields with initializers are initialized immediately after the `property` statement. In the following example, `b` is initialized to `y*9000` in segment three. The initialization makes sense and does the right thing; `b` will be `y*9000` for every `Overdone` object. (This would not be possible if field initializers were processed earlier, before properties were set.)
4. The fourth segment is the remainder of the constructor body.

The segments in the following code are shown in the comments.

```

class Overlord(x:Long) {
    def this(x:Long) { property(x); }
} //Overlord
class Overdone(y:Long) extends Overlord {

```

```

val a : Long;
val b = y * 9000;
def this(r:Long) {
    super(r);                      // (1)
    x10.io.Console.OUT.println(r); // (2)
    val rp1 = r+1;
    property(rp1);                // (2)
    // field initializations here // (3)
    a = r + 2 + b;                // (4)
}
def this() {
    this(10);                     // (1), (2), (3)
    val x = a + b;                // (4)
}
//Overdone

```

The rules of what is allowed in the three segments are different, though unsurprising. For example, properties of the current class can only be read in segment 3 or 4—naturally, because they are set at the end of segment 2.

Initialization and Inner Classes

Constructors of inner classes are tantamount to method calls on `this`. For example, the constructor for `Inner` is acceptable. It does not leak `this`. It leaks `Outer.this`, which is an utterly different object. So, the call to `this.new Inner()` in the `Outer` constructor is illegal. It would leak `this`. There is no special rule in effect preventing this; a constructor call of an inner class is no different from a method as far as leaking is concerned.

```

class Outer {
    static val leak : Cell[Outer] = new Cell[Outer](null);
    class Inner {
        def this() {Outer.leak.set(Outer.this);}
    }
    def /*Outer*/this() {
        //ERROR: val inner = this.new Inner();
    }
}

```

Initialization and Closures

Closures in constructors may not refer to `this`. They may not even refer to fields of `this` that have been initialized. For example, the closure `bad1` is not allowed because it refers to `this`; `bad2` is not allowed because it mentions `a` — which is, of course, identical to `this.a`.

```
class C {
    val a:Long;
    def this() {
        this.a = 1;
        //ERROR: val bad1 = () => this;
        //ERROR: val bad2 = () => a*10;
    }
}
```

8.11.3 Definite Initialization in Constructors

An instance field `var x:T`, when `T` has a default value, need not be explicitly initialized. In this case, `x` will be initialized to the default value of type `T`. For example, a `Score` object will have its `currently` field initialized to zero, below:

```
class Score {
    public var currently : Long;
}
```

All other sorts of instance fields do need to be initialized before they can be used. `val` fields must be initialized in the constructor, even if their type has a default value. It would be silly to have a field `val z : Long` that was always given default value of `0` and, since it is `val`, can never be changed. `var` fields whose type has no default value must be initialized as well, such as `var y : Long{y != 0}`, since it cannot be assigned a sensible initial value.

The fundamental principles are:

1. `val` fields must be assigned precisely once in each constructor on every possible execution path.
2. `var` fields of defaultless type must be assigned at least once on every possible execution path, but may be assigned more than once.
3. No variable may be read before it is guaranteed to have been assigned.
4. Initialization may be by field initialization expressions (`val x : Long = y+z`), or by uninitialized fields `val x : Long;` plus an initializing assignment `x = y+z`. Recall that field initialization expressions are performed after the `property` statement, in segment 3 in the terminology of §8.11.2.

8.11.4 Summary of Restrictions on Classes and Constructors

The following table tells whether a given feature is (yes), is not (no) or is with some conditions (note) allowed in a given context. For example, a property method is allowed

with the type of another property, as long as it only mentions the preceding properties. The first column of the table gives examples, by line of the following code body.

	Example	Prop.	self==this(1)	Prop.Meth.	this	fields
Type of property	(A)	yes (2)	no	no	no	no
Class Invariant	(B)	yes	yes	yes	yes	no
Supertype (3)	(C), (D)	yes	yes	yes	no	no
Property Method Body	(E)	yes	yes	yes	yes	no
Static field (4)	(F) (G)	no	no	no	no	no
Instance field (5)	(H), (I)	yes	yes	yes	yes	yes
Constructor arg. type	(J)	no	no	no	no	no
Constructor guard	(K)	no	no	no	no	no
Constructor ret. type	(L)	yes	yes	yes	yes	yes
Constructor segment 1	(M)	no	yes	no	no	no
Constructor segment 2	(N)	no	yes	no	no	no
Constructor segment 4	(O)	yes	yes	yes	yes	yes
Methods	(P)	yes	yes	yes	yes	yes

Details:

- (1) Top-level `self` only.
- (2) The type of the i^{th} property may only mention properties 1 through i .
- (3) Super-interfaces follow the same rules as supertypes.
- (4) The same rules apply to types and initializers.

The example indices refer to the following code:

```
class Example (
    prop : Long,
    proq : Long{prop != proq}, // (A)
    pror : Long
)
{prop != 0} // (B)
extends Supertype[Long{self != prop}] // (C)
implements SuperInterface[Long{self != prop}] // (D)
{
    property def propmeth() = (prop == pror); // (E)
    static staticField
        : Cell[Long{self != 0}] // (F)
        = new Cell[Long{self != 0}](1); // (G)
    var instanceField
        : Long {self != prop} // (H)
        = (prop + 1) as Long{self != prop}; // (I)
    def this(
        a : Long{a != 0},
```

```

        b : Long{b != a}                                // (J)
    )
    {a != b}                                         // (K)
    : Example{self.prop == a && self.prop==b} // (L)
{
    super();                                         // (M)
    property(a,b,a);                               // (N)
    // fields initialized here
    instanceField = b as Long{self != prop}; // (O)
}

def someMethod() =
    prop + staticField() + instanceField; // (P)
}

```

8.12 Method Resolution

Method resolution is the problem of determining, statically, which method (or constructor or operator) should be invoked, when there are several choices that could be invoked. For example, the following class has two overloaded `zap` methods, one taking an `Any`, and the other a `Resolve`. Method resolution will figure out that the call `zap(1..4)` should call `zap(Any)`, and `zap(new Resolve())` should call `zap(Resolve)`.

Example:

```

class Res {
    public static interface Surface {}
    public static interface Deface {}

    public static class Ace implements Surface {
        public static operator (Boolean) : Ace = new Ace();
        public static operator (Place) : Ace = new Ace();
    }
    public static class Face implements Surface, Deface {}

    public static class A {}
    public static class B extends A {}
    public static class C extends B {}

    def m(x:A) = 0;
    def m(x:Long) = 1;
    def m(x:Boolean) = 2;
    def m(x:Surface) = 3;
    def m(x:Deface) = 4;
}

```

```

def example() {
    assert m(100) == 1 : "Long";
    assert m(new C()) == 0 : "C";
    // An Ace is a Surface, unambiguous best choice
    assert m(new Ace()) == 3 : "Ace";
    // ERROR: m(new Face());

    // The match must be exact.
    // ERROR: assert m(here) == 3 : "Place";

    // Boolean could be handled directly, or by
    // implicit coercion Boolean -> Ace.
    // Direct matches always win.
    assert m(true) == 2 : "Boolean";
}

```

In the "Long" line, there is a very close match. 100 is an Long. In fact, 100 is an Long{self==100}, so even in this case the type of the actual parameter is not precisely equal to the type of the method.

In the "C" line of the example, new C() is an instance of C, which is a subtype of A, so the A method applies. No other method does, and so the A method will be invoked.

Similarly, in the "Ace" line, the Ace class implements Surface, and so new Ace() matches the Surface method.

However, a Face is both a Surface and a Deface, so there is no unique best match for the invocation m(new Face()). This invocation would be forbidden, and a compile-time error issued.

The match must be exact. There is an implicit coercion from Place to Ace, and Ace implements Surface, so the code

```

val ace : Ace = here;
assert m(ace) == 3;

```

works, by using the Surface form of m. But doing it in one step requires a deeper search than X10 performs⁸, and is not allowed.

For m(true), both the Boolean and, with the implicit coercion, Ace methods could apply. Since the Boolean method applies directly, and the Ace method requires an implicit coercion, this call resolves to the Boolean method, without an error.

The basic concept of method resolution is:

1. List all the methods that could possibly be used, inferring generic types but not performing implicit coercions.
2. If one possible method is more specific than all the others, that one is the desired method.

⁸In general this search is unbounded, so X10 can't perform it.

3. If there are two or more methods neither of which is more specific than the others, then the method invocation is ambiguous. Method resolution fails and reports an error.
4. Otherwise, no possible methods were found without implicit coercions. Try the preceding steps again, but with coercions allowed: zero or one implicit coercion for each argument. If a single most specific method is found with coercions, it is the desired method. If there are several, the invocation is ambiguous and erroneous.
5. If no methods were found even with coercions, then the method invocation is undetermined. Method resolution fails and reports an error.

After method resolution is done, there is a validation phase that checks the legality of the call, based on the STATIC_CHECKS compiler flag. With STATIC_CHECKS, the method's constraints must be satisfied; that is, they must be entailed (§4.5.2) by the information in force at the point of the call. With DYNAMIC_CHECKS, if the constraint is not entailed at that point, a dynamic check is inserted to make sure that it is true at runtime.

In the presence of X10's highly-detailed type system, some subtleties arise. One point, at least, is *not* subtle. The same procedure is used, *mutatis mutandis* for method, constructor, and operator resolution.

8.12.1 Space of Methods

X10 allows some constructs, particularly operators, to be defined in a number of ways, and invoked in a number of ways. This section specifies which forms of definition could correspond to a given definiendum.

Method invocations `a.m(b)`, where `a` is an expression, can be either of the following forms. There may be any number of arguments.

- An instance method on `a`, of the form `def m(B)`.
- A static method on `a`'s class, of the form `static def m(B)`.

The meaning of an invocation of the form `m(b)`, with any number of arguments, depends slightly on its context. Inside of a constraint, it might mean `self.m(b)`. Outside of a constraint, there is no `self` defined, so it can't mean that. The first of these that applies will be chosen.

1. Invoke a method on `this`, *viz.* `this.m(b)`. Inside a constraint, it may also invoke a property method on `self`, *viz.* `self.m(b)`. It is an error if both `this.m(b)` and `self.m(b)` are possible.
2. Invoke a function named `m` in a local or field.

3. Construct a structure named `m`.

Static method invocations, `A.m(b)`, where `A` is a container name, can only be static. There may be any number of arguments.

- A static method on `A`, of the form `static def m(B)`.

Constructor invocations, `new A(b)`, must invoke constructors. There may be any number of arguments.

- A constructor on `A`, of the form `def this(B)`.

A unary operator `*` `a` may be defined as:

- An instance operator on `A`, defined as `operator * this()`.
- A static operator on `A`, defined as `operator *(a:A)`.

A binary operator `a *` `b` may be defined as:

- An instance operator on `A`, defined as `operator this *(b:B)`; or
- A right-hand operator on `B`, defined as `operator (a:A) * this`; or
- A static operator on `A`, defined as `operator (a:A) * (b:B)`; or
- A static operator on `B`, if `A` and `B` are different classes, defined as `operator (a:A) * (b:B)`

If none of those resolve to a method, then either operand may be implicitly coerced to the other. If one of the following two situations obtains, it will be done; if both, the expression causes a static error.

- An implicit coercion from `A` to `B`, and an operator `B * B` can be used, by coercing `a` to be of type `B`, and then using `B`'s `*`.
- An implicit coercion from `B` to `A`, and an operator `A * A` can be used, coercing `b` to be of type `A`, and then using `A`'s `*`.

An application `a(b)`, for any number of arguments, can come from a number of things.

- an application operator on `a`, defined as `operator this(b:B)`;
- If `a` is an identifier, `a(b)` can also be a method invocation equivalent to `this.a(b)`, which invokes `a` as either an instance or static method on `this`
- If `a` is a qualified identifier, `a(b)` can also be an invocation of a struct constructor.

An indexed assignment, `a(b)=c`, for any number of `b`'s, can only come from an indexed assignment definition:

- `operator this(b:B)=(c:C) { ... }`

An implicit coercion, in which a value `a:A` is used in a context which requires a value of some other non-subtype `B`, can only come from implicit coercion operation defined on `B`:

- an implicit coercion in `B`: `static operator (a:A):B;`

An explicit conversion `a as B` can come from an explicit conversion operator, or an implicit coercion operator. X10 tries two things, in order, only checking 2 if 1 fails:

1. An `as` operator in `B`: `static operator (a:A) as ?;`
2. or, failing that, an implicit coercion in `B`: `static operator (a:A):B.`

8.12.2 Possible Methods

This section describes what it means for a method to be a *possible* resolution of a method invocation.

Generics introduce several subtleties, especially with the inference of generic types. For the purposes of method resolution, all that matters about a method, constructor, or operator `M` — we use the word “method” to include all three choices for this section — is its signature, plus which method it is. So, a typical `M` might look like `def m[G1, ..., Gg](x1:T1, ..., xf:Tf){c} =....` The code body `...` is irrelevant for the purpose of whether a given method call means `M` or not, so we ignore it for this section.

All that matters about a method definition, for the purposes of method resolution, is:

1. The method name `m`;
2. The generic type parameters of the method `m`, G_1, \dots, G_g . If there are no generic type parameters, $g = 0$.
3. The types $x_1:T_1, \dots, x_f:T_f$ of the formal parameters. If there are no formal parameters, $f = 0$. In the case of an instance method, the receiver will be the first formal parameter.⁹
4. A *unique identifier id*, sufficient to tell the compiler which method body is intended. A file name and position in that file would suffice. The details of the identifier are not relevant.

⁹The variable names are relevant because one formal can be mentioned in a later type, or even a constraint:
`def f(a:Long, b:Point{rank==a})=....`

For the purposes of understanding method resolution, we assume that all the actual parameters of an invocation are simply variables: `x1.meth(x2,x3)`. This is done routinely by the compiler in any case; the code `tbl(i).meth(true, a+1)` would be treated roughly as

```
val x1 = tbl(i);
val x2 = true;
val x3 = a+1;
x1.meth(x2,x3);
```

All that matters about an invocation I is:

1. The method name m' ;
2. The generic type parameters G'_1, \dots, G'_g . If there are no generic type parameters, $g = 0$.
3. The names and types $x_1:T'_1, \dots, x_f:T'_f$ of the actual parameters. If there are no actual parameters, $f = 0$. In the case of an instance method, the receiver is the first actual parameter.

The signature of the method resolution procedure is: `resolve(invocation : Invocation, context : Set[Method]) : MethodID`. Given a particular invocation and the set `context` of all methods which could be called at that point of code, method resolution either returns the unique identifier of the method that should be called, or (conceptually) throws an exception if the call cannot be resolved.

The procedure for computing `resolve(invocation, context)` is:

1. Eliminate from `context` those methods which are not *acceptable*; viz., those whose name, type parameters, and formal parameters do not suitably match `invocation`. In more detail:
 - The method name m must simply equal the invocation name m' ;
 - X10 infers type parameters, by an algorithm given in §4.12.3.
 - The method's type parameters are bound to the invocation's for the remainder of the acceptability test.
 - The actual parameter types must be subtypes of the formal parameter types, or be coercible to such subtypes. Parameter i is a subtype if $T'_i <: T_i$. It is implicitly coercible to a subtype if either it is a subtype, or if there is an implicit coercion operator defined from T'_i to some type U , and $U <: T_i$. If coercions are used to resolve the method, they will be called on the arguments before the method is invoked.
2. Eliminate from `context` those methods which are not *available*; viz., those which cannot be called due to visibility constraints, such as methods from other classes marked `private`. The remaining methods are both acceptable and available; they might be the one that is intended.

3. If the method invocation is a `super` invocation appearing in class `C1`, methods of `C1` and its subclasses are considered unavailable as well.
4. From the remaining methods, find the unique `ms` which is more specific than all the others, *viz.*, for which `specific(ms, mo) = true` for all other methods `mo`. The specificity test `specific` is given next.
 - If there is a unique such `ms`, then `resolve(invocation, context)` returns the `id` of `ms`.
 - If there is not a unique such `ms`, then `resolve` reports an error.

The subsidiary procedure `specific(m1, m2)` determines whether method `m1` is equally or more specific than `m2`. `specific` is not a total order: it is possible for each one to be considered more specific than the other, or either to be more specific. `specific` is computed as:

1. Construct an invocation `invocation1` based on `m1`:
 - `invocation1`'s method name is `m1`'s method name;
 - `invocation1`'s generic parameters are those of `m1`—simply some type variables.
 - `invocation1`'s parameters are those of `m1`.
2. If `m2` is acceptable for the invocation `invocation1`, `specific(m1, m2)` returns true;
3. Construct an invocation `invocation2p`, which is `invocation1` with the generic parameters erased. Let `invocation2` be `invocation2p` with generic parameters as inferred by X10's type inference algorithm. If type inference fails, `specific(m1, m2)` returns false.
4. If `m2` is acceptable for the invocation `invocation2`, `specific(m1, m2)` returns true;
5. Otherwise, `specific(m1, m2)` returns false.

8.12.3 Field Resolution

An identifier `p` can refer to a number of things. The rules are somewhat different inside and outside of a constraint.

Outside of a constraint, the compiler chooses the first one from the following list which applies:

1. A local variable named `p`.
2. A field of `this`, *viz.* `this.p`.
3. A nullary property method, `this.p()`
4. A member type named `p`.

5. A package named p.

Inside of a constraint, the rules are slightly different, because `self` is available, and packages cannot be used per se.

1. A local variable named p.
2. A property of `this` or of `self`, viz. `this.p` or `self.p`. If both are available, report an error.
3. A nullary property method, `this.p()`
4. A member type named p.

8.12.4 Other Disambiguations

It is possible to have a field of the same name as a method. Indeed, it is a common pattern to have private field and a public method of the same name to access it: **Example:**

```
class Xhaver {
    private var x: Long = 0;
    public def x() = x;
    public def bumpX() { x ++; }
}
```

Example: *However, this can lead to syntactic ambiguity in the case where the field f of object a is a function, rail, array, list, or the like, and where a has a method also named f. The term a.f(b) could either mean “call method f of a upon b”, or “apply the function a.f to argument b”.*

```
class Ambig {
    public val f : (Long)=>Long = (x:Long) => x*x;
    public def f(y:int) = y+1;
    public def example() {
        val v = this.f(10);
        // is v 100, or 11?
    }
}
```

In the case where a syntactic form $E.m(F_1, \dots, F_n)$ could be resolved as either a method call, or the application of a field $E.m$ to some arguments, it will be treated as a method call. The application of $E.m$ to some arguments can be specified by adding parentheses: $(E.m)(F_1, \dots, F_n)$.

Example:

```
class Disambig {
    public val f : (Long)=>Long = (x:Long) => x*x;
    public def f(y:int) = y+1;
    public def example() {
        assert( this.f(10) == 11 );
        assert( (this.f)(10) == 100 );
    }
}
```

Similarly, it is possible to have a method with the same name as a struct, say `ambig`, giving an ambiguity as to whether `ambig()` is a struct constructor invocation or a method invocation. This ambiguity is resolved by treating it as a method invocation. If the constructor invocation is desired, it can be achieved by including the optional `new`. That is, `new ambig()` is struct constructor invocation; `ambig()` is a method invocation.

8.13 Static Nested Classes

One class (or struct or interface) may be nested within another. The simplest way to do this is as a `static` nested class, written by putting one class definition at top level inside another, with the inner one having a `static` modifier. For most purposes, a static nested class behaves like a top-level class. However, a static nested class has access to private static fields and methods of its containing class.

Nested interfaces and static structs are permitted as well.

```
class Outer {
    private static val priv = 1;
    private static def special(n:Long) = n*n;
    public static class StaticNested {
        static def reveal(n:Long) = special(n) + priv;
    }
}
```

8.14 Inner Classes

Non-static nested classes are called *inner classes*. An inner class instance can be thought of as a very elaborate member of an object — one with a full class structure of its own. The crucial characteristic of an inner class instance is that it has an implicit reference to an instance of its containing class.

Example: *This feature is particularly useful when an instance of the inner class makes no sense without reference to an instance of the outer, and is closely tied to it. For example, consider a range class, describing a span of integers m to n, and an*

iterator over the range. The iterator might as well have access to the range object, and there is little point to discussing iterators-over-ranges without discussing ranges as well. In the following example, the inner class RangeIter iterates over the enclosing Range.

It has its own private cursor field n, telling where it is in the iteration; different iterations over the same Range can exist, and will each have their own cursor. It is perhaps unwise to use the name n for a field of the inner class, since it is also a field of the outer class, but it is legal. (It can happen by accident as well – e.g., if a programmer were to add a field n to a superclass of the outer class, the inner class would still work.) It does not even interfere with the inner class’s ability to refer to the outer class’s n field: the cursor initialization refers to the Range’s lower bound through a fully qualified name Range.this.n. The initialization of its n field refers to the outer class’s m field, which is not shadowed and can be referred to directly, as m.

```
class Range(m:Long, n:Long) implements Iterable[Long]{
    public def iterator () = new RangeIter();
    private class RangeIter implements Iterator[Long] {
        private var n : Long = m;
        public def hasNext() = n <= Range.this.n;
        public def next() = n++;
    }
    public static def main(argv: Rail[String]) {
        val r = new Range(3,5);
        for(i in r) Console.OUT.println("i=" + i);
    }
}
```

An inner class has full access to the members of its enclosing class, both static and instance. In particular, it can access `private` information, just as methods of the enclosing class can.

An inner class can have its own members. Inside instance methods of an inner class, `this` refers to the instance of the *inner* class. The instance of the outer class can be accessed as `Outer.this` (where `Outer` is the name of the outer class). If, for some dire reason, it is necessary to have an inner class within an inner class, the innermost class can refer to the `this` of either outer class by using its name.

An inner class can inherit from any class in scope, with no special restrictions. `super` inside an inner class refers to the inner class’s superclass. If it is necessary to refer to the outer classes’s superclass, use a qualified name of the form `Outer.super`.

The members of inner classes must be instance members. They cannot be static members. Classes, interfaces, static methods, static fields, and typedefs are not allowed as members of inner classes. The same restriction applies to local classes (§8.15).

Consider an inner class `IC1` of some outer class `OC1`, being extended by another class `IC2`. However, since an `IC1` only exists as a dependent of an `OC1`, each `IC2` must be associated with an `OC1` — or a subtype thereof — as well. So, `IC2` must be an inner class of either `OC1` or some subclass `OC2 <: OC1`.

Example: For example, one often extends an inner class when one extends its outer class:

```
class OC1 {
    class IC1 {}
}
class OC2 extends OC1 {
    class IC2 extends IC1 {}
}
```

The hiding of method names has one fine point. If an inner class defines a method named `doit`, then *all* methods named `doit` from the outer class are hidden — even if they have different argument types than the one defined in the inner class. They are still accessible via `Outer.this.doit()`, but not simply via `doit()`. The following code is correct, but would not be correct if the `//ERROR` line were uncommented.

```
class Outer {
    def doit() {}
    def doit(String) {}
    class Inner {
        def doit(Boolean, Outer) {}
        def example() {
            doit(true, Outer.this);
            Outer.this.doit();
            //ERROR: doit("fails");
        }
    }
}
```

8.14.1 Constructors and Inner Classes

If `IC` is an inner class of `OC`, then instance code in the body of `OC` can create instances of `IC` simply by calling a constructor `new IC(...)`:

```
class OC {
    class IC {}
    def method(){
        val ic = new IC();
    }
}
```

Instances of `IC` can be constructed from elsewhere as well. Since every instance of `IC` is associated with an instance of `OC`, an `OC` must be supplied to the `IC` constructor. The syntax for doing so is: `oc.new IC()`. For example:

```
class OC {
    class IC {}
```

```

static val oc1 = new OC();
static val oc2 = new OC();
static val ic1 = oc1.new IC();
static val ic2 = oc2.new IC();
}
class Elsewhere{
    def method(oc : OC) {
        val ic = oc.new IC();
    }
}

```

8.15 Local Classes

Classes can be defined and instantiated in the middle of methods and other code blocks. A local class in a static method is a static class; a local class in an instance method is an inner class. Local classes are local to the block in which they are defined. They have access to almost everything defined at that point in the method; the one exception is that they cannot use `var` variables. Local classes cannot be `public`, `protected`, or `private`, because they are only visible from within the block of declaration. They cannot be `static`.

Example: *The following example illustrates the use of a local class `Local`, defined inside the body of method `m()`.*

```

class Outer {
    val a = 1;
    def m() {
        val a = -2;
        val b = 2;
        class Local {
            val a = 3;
            def m() = 100*Outer.this.a + 10*b + a;
        }
        val l : Local = new Local();
        assert l.m() == 123;
    } // end of m()
}

```

Note that the middle `a`, whose value is `-2`, is not accessible inside of `Local`; it is shadowed by `Local`'s `a` field. `Outer`'s `a` is also shadowed, but the notation `Outer.this` gives a reference to the enclosing `Outer` object. There is no corresponding notation to access shadowed local variables from the enclosing block; if you need to get them, rename the fields of `Local`.

The members of inner classes must be instance members. They cannot be static members. Classes, interfaces, static methods, static fields, and typedefs are not allowed as members of local classes. The same restriction applies to inner classes (§8.14).

8.16 Anonymous Classes

It is possible to define a new local class and instantiate it as part of an expression. The new class can extend an existing class or interface. Its body can include all of the usual members of a local class. It can refer to any identifiers available at that point in the expression — except for `var` variables. An anonymous class in a static context is a static inner class.

Anonymous classes are useful when you want to package several pieces of behavior together (a single piece of behavior can often be expressed as a function, which is syntactically lighter-weight), or if you want to extend and vary an extant class without going through the trouble of actually defining a whole new class.

The syntax for an anonymous class is a constructor call followed immediately by a braced class body: `new C{def foo()=2;}`.

Example: *In the following minimalist example, the abstract class `Choice` encapsulates a decision. A `Choice` has a `yes()` and a `no()` method. The `choose(b)` method will invoke one of the two. Choices also have names.*

The `main()` method creates a specific `Choice`. `c` is not a immediate instance of `Choice` — as an abstract class, `Choice` has no immediate instances. `c` is an instance of an anonymous class which inherits from `Choice`, but supplies `yes()` and `no()` methods. These methods modify the contents of the `Cell[Long]` `n`. (Note that, as `n` is a local variable, it would take a few lines more coding to extract `c`'s class, name it, and make it an inner class.) The call to `c.choose(true)` will call `c.yes()`, incrementing `n()`, in a rather roundabout manner.

```
abstract class Choice(name: String) {
    def this(name:String) {property(name);}
    def choose(b:Boolean) {
        if (b) this.yes(); else this.no(); }
    abstract def yes():void;
    abstract def no():void;
}

class Example {
    static def main(Rail[String]) {
        val n = new Cell[Long](0);
        val c = new Choice("Inc Or Dec") {
            def yes() { n() += 1; }
            def no() { n() -= 1; }
        };
        c.choose(true);
        Console.OUT.println("n=" + n());
    }
}
```

Anonymous classes have many of the features of classes in general. A few features are unavailable because they don't make sense.

- Anonymous classes don't have constructors. Since they don't have names, there's no way a constructor could get called in the ordinary way. Instead, the `new C(...)` expression must match a constructor of the parent class `C`, which will be called to initialize the newly-created object of the anonymous class.
- The `public`, `private`, and `protected` modifiers don't make sense for anonymous classes: Anonymous classes, being anonymous, cannot be referenced at all, so references to them can't be public, private, or protected.
- Anonymous classes cannot be `abstract`. Since they only exist in combination with a constructor call, they must be constructable. The parent class of the anonymous class may be abstract, or may be an interface; in this case, the anonymous class must provide all the methods that the parent demands.
- Anonymous classes cannot have explicit `extends` or `implements` clauses; there's no place in the syntax for them. They have a single parent and that is that.

9 Structs

X10 objects are a powerful general-purpose programming tool. However, the power must be paid for in space and time. In space, a typical object implementation requires some extra memory for run-time class information, as well as a pointer for each reference to the object. In time, a typical object requires an extra indirection to read or write data, and some run-time computation to figure out which method body to call.

For high-performance computing, this overhead may not be acceptable for all objects. X10 provides structs, which are stripped-down objects. They are less powerful than objects; in particular they lack inheritance and mutable fields. Without inheritance, method calls do not need to do any lookup; they can be implemented directly. Accordingly, structs can be implemented and used more cheaply than objects, potentially avoiding the space and time overhead. (Currently, the C++ back end avoids the overhead, but the Java back end implements structs as Java objects and does not avoid it.)

Structs and classes are interoperable. Both can implement interfaces; in particular, like all X10 values they implement `Any`. Subroutines whose arguments are defined by interfaces can take both structs and classes. (Some caution is necessary here: referring to a struct through an interface requires overhead similar to that required for an object.)

In many cases structs can be converted to classes or classes to structs, within the constraints of structs. If you start off defining a struct and decide you need a class instead, the code change required is simply changing the keyword `struct` to `class`. If you have a class that does not use inheritance or mutable fields, it can be converted to a struct by changing its keyword. Client code using the struct that was a class will need certain changes: *e.g.*, the `new` keyword must be added in constructor calls, and structs (unlike classes) cannot be `null`.

9.1 Struct declaration

<i>StructDecln</i>	$::=$	<i>Mods?</i> struct <i>Id</i> <i>TypeParamsI?</i> <i>Properties?</i> <i>Guard?</i> <i>Interfaces?</i> <i>Class-</i>	(20.154)
		<i>Body</i>	
<i>TypeParamsI</i>	$::=$	[<i>TypeParamIList</i>]	(20.177)
<i>Properties</i>	$::=$	(<i>PropList</i>)	(20.142)
<i>Guard</i>	$::=$	<i>DepParams</i>	(20.83)
<i>Interfaces</i>	$::=$	implements <i>InterfaceTypeList</i>	(20.103)
<i>ClassBody</i>	$::=$	{ <i>ClassMemberDecls?</i> }	(20.33)

All fields of a struct must be `val`.

A struct *S* cannot contain a field of type *S*, or a field of struct type *T* which, recursively, contains a field of type *S*. This restriction is necessary to permit *S* to be implemented as a contiguous block of memory of size equal to the sum of the sizes of its fields.

Values of a struct *C* type can be created by invoking a constructor defined in *C*. Unlike for classes, the `new` keyword is optional for struct constructors.

Example: *Leaving out new can improve readability in some cases:*

```
struct Polar(r:Double, theta:Double){
    def this(r:Double, theta:Double) {property(r,theta);}
    static val Origin = Polar(0,0);
    static val x0y1   = Polar(1, 3.14159/2);
    static val x1y0   = new Polar(1, 0);
}
```

When a struct and a method have the same name (often in violation of the X10 capitalization convention), new may be used to resolve to the struct's constructor.

```
struct Ambig(x:Long) {
    static def Ambig(x:Long) = "ambiguity please";
    static def example() {
        val useMethod      = Ambig(1);
        val useConstructor = new Ambig(2);
    }
}
```

Structs support the same notions of generics, properties, and constrained types that classes do.

Example:

```
struct Exam[T](nQuestions:Long){T <: Question} {
    public static interface Question {}
    // ...
}
```

9.2 Boxing of structs

If a struct S implements an interface I (e.g., Any), a value v of type S can be assigned to a variable of type I. The implementation creates an object o that is an instance of an anonymous class implementing I and containing v. The result of invoking a method of I on o is the same as invoking it on v. This operation is termed *auto-boxing*. It allows full interoperability of structs and objects—at the cost of losing the extra efficiency of the structs when they are boxed.

In a generic class or struct obtained by instantiating a type parameter T with a struct S, variables declared at type T in the body of the class are not boxed. They are implemented as if they were declared at type S.

Example: *The rail aa in the following example is a Rail[Any]. It initially holds two objects. Then, its elements are replaced by two structs, both of which are auto-boxed. Note that no fussing is required to put an integer into a Rail[Any]. However, a rail of structs, such as ah, holds unboxed structs and does not incur boxing overhead.*

```
struct Horse(x:Long){
    static def example(){
        val aa : Rail[Any] = ["a String" as Any, "another one"];
        aa(0) = Horse(8);
        aa(1) = 13;
        val ah : Rail[Horse] = [Horse(7), Horse(13)];
    }
}
```

9.3 Optional Implementation of Any methods

Two structs are equal (==) if and only if their corresponding fields are equal (==).

All structs implement x10.lang.Any. Structs are required to implement the following methods from Any. Programmers need not provide them; X10 will produce them automatically if the program does not include them.

```
public def equals[Any]:Boolean;
public def hashCode():Int;
public def typeName():String;
public def toString():String;
```

A programmer who provides an explicit implementation of equals[Any] for a struct S should also consider supplying a definition for equals(S):Boolean. This will often yield better performance since the cost of an upcast to Any and then a downcast to S can be avoided.

9.4 Primitive Types

Certain types that might be built in to other languages are in fact implemented as structs in package `x10.lang` in X10. Their methods and operations are often provided with `@Native` (§18) rather than X10 code, however. These types are:

```
Boolean, Char, Byte, Short, Int, Long
Float, Double, UByte, USHORT, UInt, ULong
```

9.4.1 Signed and Unsigned Integers

X10 has an unsigned integer type corresponding to each integer type: `UInt` is an unsigned `Int`, and so on. These types can be used for binary programming, or when an extra bit of precision for counters or other non-negative numbers is needed in integer arithmetic. However, X10 does not otherwise encourage the use of unsigned arithmetic.

9.5 Example structs

`x10.lang.Complex` provides a detailed example of a practical struct, suitable for use in a library. For a shorter example, we define the `Pair` struct. A `Pair` packages two values of possibly unrelated type together in a single value, *e.g.*, to return two values from a function.

`divmod` computes the quotient and remainder of $a \div b$ (naively). It returns both, packaged as a `Pair[UInt, UInt]`. Note that the constructor uses type inference, and that the quotient and remainder are accessed through the `first` and `second` fields.

```
struct Pair[T,U] {
    public val first:T;
    public val second:U;
    public def this(first:T, second:U):Pair[T,U] {
        this.first = first;
        this.second = second;
    }
    public def toString()
        = "(" + first + ", " + second + ")";
}
class Example {
    static def divmod(var a:UInt, b:UInt): Pair[UInt, UInt] {
        assert b > 0u;
        var q : UInt = 0un;
        while (a > b) {q += 1un; a -= b;}
        return Pair(q, a);
    }
}
```

```

static def example() {
    val qr = divmod(22un, 7un);
    assert qr.first == 3un && qr.second == 1un;
}
}

```

9.6 Nested Structs

Static nested structs may be defined, essentially as static nested classes except for making them structs (§8.13). Inner structs may be defined, essentially as inner classes except making them structs (§8.14). **Limitation:** Nested structs must be currently be declared static.

9.7 Default Values of Structs

If all fields of a struct have default values, then the struct has a default value, *viz.*, the struct whose fields are all set to their default values. If some field does not have a default value, neither does the struct.

Example:

In the following code, the Example struct has a default value whose i field is 0. If an Example is ever constructed by the constructor, its i field will be 1. This program does a slightly subtle dance to get ahold of a default Example, by having an instance var (which, unlike most kinds of variables, does not need to get initialized before use (though that exemption only applies if its type has a default value)). As the assert confirms, the default Example does indeed have an i field of 0.

```

class StructDefault {
    static struct Example {
        val i : Long;
        def this() { i = 1; }
    }
    var ex : Example;
    static def example() {
        val ex = (new StructDefault()).ex;
        assert ex.i == 0;
    }
}

```

9.8 Converting Between Classes And Structs

Code written using structs can be modified to use classes, or vice versa. Caution must be used in certain places.

Class and struct *definitions* are syntactically nearly identical: change the `class` keyword to `struct` or vice versa. Of course, certain important class features can't be used with structs, such as inheritance and `var` fields.

Converting code that *uses* the class or struct requires a certain amount of caution. Suppose, in particular, that we want to convert the class `Class2Struct` to a struct, and `Struct2Class` to a class.

```
class Class2Struct {
    val a : Long;
    def this(a:Long) { this.a = a; }
    def m() = a;
}
struct Struct2Class {
    val a : Long;
    def this(a:Long) { this.a = a; }
    def m() = a;
}
```

1. Class constructors require the `new` keyword; struct constructors allow it but do not require it. `Struct2Class(3)` to will need to be converted to `new Struct2Class(3)`.
2. Objects and structs have different notions of `==`. For objects, `==` means “same object”; for structs, it means “same contents”. Before conversion, both `asserts` in the following program succeed. After converting and fixing constructors, both of them fail.

```
val a = new Class2Struct(2);
val b = new Class2Struct(2);
assert a != b;
val c = Struct2Class(3);
val d = Struct2Class(3);
assert c==d;
```

3. Objects can be set to `null`. Structs cannot.
4. The rules for default values are quite different. The default value of an object type (if it exists) is `null`, which behaves quite differently from an ordinary object of that type; *e.g.*, you cannot call methods on `null`, whereas you can on an ordinary object. The default value for a struct type (if it exists) is a struct like any other of its type, and you can call methods on it as for any other.

10 Functions

10.1 Overview

Functions, the last of the three kinds of values in X10, encapsulate pieces of code which can be applied to a vector of arguments to produce a value. Functions, when applied, can do nearly anything that any other code could do: fail to terminate, throw an exception, modify variables, spawn activities, execute in several places, and so on. X10 functions are not mathematical functions: the `f(1)` may return `true` on one call and `false` on an immediately following call.

A *function literal* $(x_1:T_1, \dots, x_n:T_n)\{c\}:T \Rightarrow e$ creates a function of type $(x_1:T_1, \dots, x_n:T_n)\{c\} \Rightarrow T$ (§4.6). For example, $(x:\text{Long}):\text{Long} \Rightarrow x^*x$ is a function literal describing the squaring function on integers. Every function type also possesses the (default) value `null`.

Limitation: X10 functions cannot have type arguments or constraints.

Function application is written `f(a,b,c)`, following common mathematical usage.

The function body may be a block. To compute integer squares by repeated addition (inefficiently), one may write:

```
val sq: (Long) => Long
= (n:Long) => {
    var s : Long = 0;
    val abs_n = n < 0 ? -n : n;
    for (i in 1..abs_n) s += abs_n;
    s
};
```

A function literal evaluates to a function entity f . When f is applied to a suitable list of actual parameters a_1 through a_n , it evaluates e with the formal parameters bound to the actual parameters. So, the following are equivalent, where e is an expression involving x_1 and x_2

```
var result:T;
{
    val f = (x1:T1,x2:T2){true}:T => e;
```

```

    val a1 : T1 = arg1();
    val a2 : T2 = arg2();
    result = f(a1,a2);
}

```

and

```

var result:T;
{
    val a1 : T1 = arg1();
    val a2 : T2 = arg2();
    {
        val x1 : T1 = a1;
        val x2 : T2 = a2;
        result = e;
    }
}

```

This equivalence does not hold if the body is a statement rather than an expression. A few language features are forbidden (break or continue of a loop that surrounds the function literal) or mean something different (return inside a function returns from the function, not the surrounding block).

Function types may be used in `implements` clauses of class definitions. Suitable operator definitions must be supplied, with `public operator this(x1:T1, ..., xn:Tn)` declarations. Instances of such classes may be used as functions of the given type. Indeed, an object may behave like any (fixed) number of functions, since the class it is an instance of may implement any (fixed) number of function types. *e.g.* Instances of the `Funny` class behave like two functions: a constant function on Booleans, and a linear function on pairs of Longs.

```

class Funny implements (Boolean) => Long,
                    (Long, Long) => Long
{
    public operator this(Boolean) = 1;
    public operator this(x:Long, y:Long) = 10*x+y;
    static def example() {
        val f <: Funny = new Funny();
        assert f(true) == 1; // (Boolean)=>Long behavior
        assert f(1,2) == 12; // (Long,Long)=>Long behavior
    }
}

```

10.2 Function Application

The basic operation on functions is function application. (Since, *e.g.*, array lookup has the same type as function application, these rules are used for array lookup as well, and

so on.)

A function with type $(x_1:T_1, \dots, x_n:T_n)\{c\} \Rightarrow T$ can be applied to a sequence of expressions e_1, \dots, e_n if:

- e_1 is of type $T_1[e_1/x_1]$,
- ...,
- e_n is of type $T_n[e_1/x_1, \dots, e_n/x_n]$,
- X10 can prove that $c[e_1/x_1, \dots, e_n/x_n]$ holds.

In this case, if the application terminates normally, it returns a value of type $T[y_1/x_1, \dots, y_n/x_n]$ where y_1, \dots, y_n may be thought of as new variables defined as if by:

```
val y1=e1;
...
val yn=en;
```

Example: Consider

```
f : (a:Long{a!=0}, b:Long{b!=a}){b!=0} => Long{self != a}
```

Then the call $f(3, 4)$ is allowed, because:

- 3 is of type $\text{Long}\{a!=0\}$ with a replaced by 3, viz. $\text{Long}\{3 != 0\}$;
- 4 is of type $\text{Long}\{b!=a\}$ with a replaced by 3 and b replaced by 4, viz. $\text{Long}\{3 != 4\}$.
- The guard $b!=0$, with a replaced by 3 and b replaced by 4, is $4!=0$, which is true.

So, $f(3, 4)$ will return a value of type $\text{Long}\{\text{self } != a\}$ with a replaced by 3 and b replaced by 4, which is to say, $\text{Long}\{\text{self}!=3\}$.

10.3 Function Literals

X10 provides first-class, typed functions, often called *closures*.

ClosureExp ::= *Formals* *Guard?* *HasResultType?* \Rightarrow *ClosureBody* (20.42)

Formals ::= (FormalList?) (20.80)

Guard ::= *DepParams* (20.83)

HasResultType ::= *ResultType*
| *Type* (20.86)

ClosureBody ::= *Exp* (20.40)

| *ClosureBodyBlock* (20.41)

ClosureBodyBlock ::= *Annotations?* { *BlockStmts?* *LastExp* }
| *Annotations?* *Block*

Functions have zero or more formal parameters and an optional return type. The body has the same syntax as a method body; it may be either an expression, a block of statements, or a block terminated by an expression to return. Return statements may be used in the body of the function to return a value (§12.13).

The type of a function is a function type as described in §4.6. In some cases the return type T of the function can be omitted and defaults to the type of the body. If a formal x_i does not occur in any T_j , c , T or e , the declaration $x_i:T_i$ may be replaced by just T_i . E.g., $(\text{Long})=>7$ is the integer function returning 7 for all inputs.

As with methods, a function may declare a guard to constrain the actual parameters with which it may be invoked. The guard may refer to the type parameters, formal parameters, and any vals in scope at the function expression.

Example:

```
val n = 3;
val f : (x:Long){x != n} => Long
  = (x:Long){x != n} => (12/(n-x));
Console.OUT.println("f(5)=" + f(5));
```

The body of the function is evaluated when the function is invoked by a call expression (§11.6), not at the function's place in the program text.

As with methods, a function with return type `void` cannot have a terminating expression. If the return type is omitted, it is inferred, as described in §4.12. It is a static error if the return type cannot be inferred. E.g., $(\text{Long})=>\text{null}$ is not well-specified; X10 does not know which type of `null` is intended. But $(\text{Long}):T=>\text{null}$ is legal.

Example: *The following method takes a function parameter and uses it to test each element of the list, returning the first matching element. It returns no if no element matches.*

```
def find[T](f: (T) => Boolean, xs: List[T], no:T): T {
  for (x: T in xs)
    if (f(x)) return x;
  return no;
}
```

The method may be invoked thus, to find a positive element of xs, or return 0 if there is no positive element.

```
xs: List[Long] = new ArrayList[Long]();
x: Long = find((x: Long) => x>0, xs, 0);
```

10.3.1 Outer variable access

In a function $(x_1: T_1, \dots, x_n: T_n)\{c\} => \{ s \}$ the types T_i , the guard c and the body s may access the following variables from outer scopes:

- All fields of the enclosing object(s) and class(es);
- All type parameters;
- All `val` variables;

`var` variables cannot be accessed.

The function body may refer to instances of enclosing classes using the syntax `C.this`, where `C` is the name of the enclosing class. `this` refers to the instance of the immediately enclosing class, as usual.

e.g. The following is legal. Note that `a` is not a local `var` variable. It is a field of `this`. A reference to `a` is simply short for `this.a`, which is a use of a `val` variable (`this`).

```
class Lambda {
    var a : Long = 0;
    val b = 0;
    def m(var c : Long, val d : Long) {
        var e : Long = 0;
        val f : Long = 0;
        val closure = (var i: Long, val j: Long) => {
            // c and e are not usable here
            a + b + d + f + i
            + j + this.a + Lambda.this.a
        };
        return closure;
    }
}
```

10.4 Functions as objects of type Any

Two functions `f` and `g` are equal if both were obtained by the same evaluation of a function literal.¹ Further, it is guaranteed that if two functions are equal then they refer to the same locations in the environment and represent the same code, so their executions in an identical situation are indistinguishable. (Specifically, if `f == g`, then `f(1)` can be substituted for `g(1)` and the result will be identical. However, there is no guarantee that `f(1)==g(1)` will evaluate to true, since there is no guarantee that `f(1)==f(1)` will evaluate to true either, as `f` might be a function which returns `n` on its n^{th} invocation. However, `f(1)==f(1)` and `f(1)==g(1)` are interchangeable.)

Every function type implements all the methods of `Any`. `f.equals(g)` is equivalent to `f==g`. The behavior of `hashCode`, `toString`, and `typeName` is up to the implementation, but respect `equals` and the basic contracts of `Any`.

¹A literal may occur in program text within a loop, and hence may be evaluated multiple times.

11 Expressions

X10 has a rich expression language. Evaluating an expression produces a value, or, in a few cases, no value. Expression evaluation may have side effects, such as change of the value of a `var` variable or a data structure, allocation of new values, or throwing an exception.

11.1 Literals

Literals denote fixed values of built-in types. The syntax for literals is given in §3.5.

The type that X10 gives a literal often includes its value. *E.g.*, `1` is of type `Long{self==1}`, and `true` is of type `Boolean{self==true}`.

11.2 this

Primary ::= `this` (20.138)
| `ClassName . this`

The expression `this` is a local `val` containing a reference to an instance of the lexically enclosing class. It may be used only within the body of an instance method, a constructor, or in the initializer of a instance field – that is, the places where there is an instance of the class under consideration.

Within an inner class, `this` may be qualified with the name of a lexically enclosing class. In this case, it represents an instance of that enclosing class.

Example: *Outer is a class containing Inner. Each instance of Inner has a reference Outer.this to the Outer involved in its creation. Inner has access to the fields of Outer.this. Note that Inner has its own three field, which is different from and not even the same type as Outer.this.three.*

```
class Outer {  
    val three = 3;  
    class Inner {
```

```

val three = "THREE";
def example() {
    assert Outer.this.three == 3;
    assert three.equals("THREE");
    assert this.three.equals("THREE");
}
}
}

```

The type of a `this` expression is the innermost enclosing class, or the qualifying class, constrained by the class invariant and the method guard, if any.

The `this` expression may also be used within constraints in a class or interface header (the class invariant and `extends` and `implements` clauses). Here, the type of `this` is restricted so that only properties declared in the class header itself, and specifically not any members declared in the class body or in supertypes, are accessible through `this`.

11.3 Local variables

Id ::= IDENTIFIER

(20.90)

A local variable expression consists simply of the name of the local variable, a field of `this`, a formal parameter in scope, etc. It evaluates to the value of the local variable.

Example: *n in the second line below is a local variable expression. The n in the first line is not; it is part of a local variable declaration.*

```

val n = 22;
val m = n + 56;

```

11.4 Field access

*FieldAccess ::= Primary . Id
| super . Id
| ClassName . super . Id* (20.67)

A field of an object instance may be accessed with a field access expression.

The type of the access is the declared type of the field with the actual target substituted for `this` in the type.

Example: *The declaration of b below has a constraint involving this. The use of an instance of it, f.b, has the same constraint involving f instead of this, as required.*

```

class Fielded {
    public val a : Long = 1;
    public val b : Long{this.a == b} = this.a;
    static def example() {
        val f : Fielded = new Fielded();
        assert f.a == 1 && f.b == 1;
        val fb : Long{fb == f.a} = f.b;
        assert fb == 1;
    }
}

```

The field accessed is selected from the fields and value properties of the static type of the target and its superclasses.

If the field target is given by the keyword `super`, the target's type is the superclass of the enclosing class. This form is used to access fields of the parent class hidden by same-named fields of the current class.

If the field target is `Cls.super`, then the target's type is `Cls`, which must be an enclosing class. This (admittedly obscure) form is used to access fields of an ancestor class which are shadowed by same-named fields of some more recent ancestor.

Example: *This illustrates all four cases of field access.*

```

class Uncle {
    public static val f = 1;
}
class Parent {
    public val f = 2;
}
class Ego extends Parent {
    public val f = 3;
    class Child extends Ego {
        public val f = 4;
        def example() {
            assert Uncle.f == 1;
            assert Ego.super.f == 2;
            assert super.f == 3;
            assert this.f == 4;
            assert f == 4;
        }
    }
}

```

If the field target is `null`, a `NullPointerException` is thrown. If the field target is a class name, a static field is selected. It is illegal to access a field that is not visible from the current context. It is illegal to access a non-static field through a static field access expression. However, it is legal to access a static field through a non-static reference.

11.5 Function Literals

Function literals are described in §10.

11.6 Calls

MethodInvo ::= *MethodName* *TypeArgs?* (*ArgumentList?*) (20.118)

| *Primary* . *Id* *TypeArgs?* (*ArgumentList?*)
| *super* . *Id* *TypeArgs?* (*ArgumentList?*)
| *ClassName* . *super* . *Id* *TypeArgs?* (*ArgumentList?*)
| *Primary* *TypeArgs?* (*ArgumentList?*)

ArgumentList ::= *Exp* (20.8)

| *ArgumentList* , *Exp*

MethodName ::= *Id* (20.120)

| *FullyQualifiedname* . *Id*

A *MethodInvocation* may be to either a `static` method, an instance method, or a closure.

The syntax for method invocations is ambiguous. `ob.m()` could either be the invocation of a method named `m` on object `ob`, or the application of a function held in a field `ob.m`. If both are defined on the same class, X10 resolves `ob.m()` to the invocation of the method. If the application of a function in a field is desired, use an alternate syntax which makes the intent clear to X10, such as `(ob.m)()`.

Example:

```
class Callsome {
    static val closure : () => Long = () => 1;
    static def method() = 2;
    static def example() {
        assert Callsome.closure() == 1;
        assert Callsome.method() == 2;
    }
}
```

However, adding a static method [mis]named `closure` makes `Callsome.closure()` refer to the method, rather than the closure

```
static def closure () = 3;
static def example() {
    assert Callsome.closure() == 3;
    assert (Callsome.closure)() == 1;
}
```

The application form `e(f,g)`, when `e` evaluates to an object or struct, invokes the application operator, defined in the form

```
public operator this(f:F, g:G) = "value";
```

Method selection rules are given in §8.12.

Guard satisfaction depends on the STATIC_CHECKS compiler flag. With the flag on, it is a static error if a method's *Guard* is not statically satisfied by the caller. With STATIC_CHECKS off, the guard will be checked at runtime if necessary.

Example: *In this example, a DivideBy object provides the service of dividing numbers by denom — so long as denom is not zero. X10's strictness of checking this is under control of the STATIC_CHECKS compiler option (§C.1.3).*

With STATIC_CHECKS turned on, the example method will not compile. The call this.div(100) is not allowed; there is no guarantee that denom != 0. Casting this to a type whose constraint implies denom != 0 permits the method call.

With STATIC_CHECKS turned off, the call will compile. X10 will insert a dynamic check that the denominator is non-zero, and will fail at runtime if it is zero.

```
class DivideBy(denom:Long) {
    def div(numer:Long){denom != 0} = numer / denom;
    def example() {
        val thisCast = (this as DivideBy{self.denom != 0});
        thisCast.div(100);
        //ERROR (with STATIC_CHECKS): this.div(100);
    }
}
```

11.6.1 super calls

The expression `super.f(e1...en)` may appear in an instance method definition. This causes the method invocation to be a `super` invocation, as described in §8.12.

Informally, suppose the invocation appears in class C1, which extends class Sup. An invocation `this.f()` will call a nullary method named `f` that appears in class C1 itself, if there is one. An invocation `super.f()` will call the nullary `f` method in Sup or an ancestor thereof, but not one in C1. Note that `super.f()` may be used to invoke an `f` method in Sup which has been overridden by one appearing in C1.

Note that there's only one choice for which `f` is invoked by `super.f()` – *viz.* the lowest one in the class hierarchy above C1. So, `super.f()` performs static dispatch, like a static method call. This is generally more efficient than a dynamic dispatch, like an instance method call.

11.7 Assignment

<i>Assignment</i>	$::= LeftHandSide AsstOp AsstExp$	(20.12)
	$ ExpName (ArgumentList?) AsstOp AsstExp$	
	$ Primary (ArgumentList?) AsstOp AsstExp$	
<i>LeftHandSide</i>	$::= ExpName$	(20.108)
	$ FieldAccess$	
<i>AsstOp</i>	$::= =$	(20.14)
	$ *=$	
	$ /=$	
	$ %=$	
	$ +=$	
	$ -=$	
	$ <<=$	
	$ >>=$	
	$ >>>=$	
	$ &=$	
	$ ^=$	
	$ =$	

The assignment expression `x = e` assigns a value given by expression `e` to a variable `x`. Most often, `x` is mutable, a `var` variable. The same syntax is used for delayed initialization of a `val`, but `vals` can only be initialized once.

```
var x : Long;
val y : Long;
x = 1;
y = 2; // Correct; initializes y
x = 3;
// ERROR: y = 4;
```

There are three syntactic forms of assignment:

1. `x = e;`, assigning to a local variable, formal parameter, field of `this`, etc.
2. `x.f = e;`, assigning to a field of an object.
3. `a(i1,...,in) = v;`, where $n \geq 0$, assigning to an element of an array or some other such structure. This is an operator call (§8.7). For well-behaved classes it works like array assignment, *mutatis mutandis*, but there is no actual guarantee, and the compiler makes no assumptions about how this works for arbitrary `a`. Naturally, it is a static error if no suitable assignment operator for `a` exists..

For a binary operator \diamond , the \diamond -assignment expression `x \diamond = e` combines the current value of `x` with the value of `e` by \diamond , and stores the result back into `x`. `i += 2`, for example, adds 2 to `i`. For variables and fields,

```
x ::= e
```

behaves just like

```
x = x :: e.
```

The subscripting forms of `a(i) ::= b` are slightly subtle. Subexpressions of `a` and `i` are only evaluated once. However, `a(i)` and `a(i)=c` are each executed once—in particular, there is one call to the application operator, and one to the assignment operator. If subscripting is implemented strangely for the class of `a`, the behavior is *not* necessarily updating a single storage location. Specifically, `A() (I()) += B()` is tantamount to the following code, except for the unspecified order of evaluation of the expressions:

```
{
    // The order of these evaluations is not specified
    val aa = A(); // Evaluate A() once
    val ii = I(); // Evaluate I() once
    val bb = B(); // Evaluate B() once
    // But they happen before this:
    val tmp = aa(ii) + bb; // read aa(ii)
    aa(ii) = tmp; // write sum back to aa(ii)
}
```

11.8 Increment and decrement

The operators `++` and `--` increment and decrement a variable, respectively. `x++` and `++x` both increment `x`, just as the statement `x += (1 as T)` would (where `x:T`), and similarly for `--`.

The difference between the two is the return value. `++x` and `--x` return the *new* value of `x`, after incrementing or decrementing. `x++` and `x--` return the *old* value of `x`, before incrementing or decrementing.

These operators work for any `x` for which `1 as T` is defined, where `T` is the type of `x`.

11.9 Numeric Operations

Numeric types (`Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Complex`, and unsigned variants of fixed-point types) are normal X10 structs, though most of their methods are implemented via native code. They obey the same general rules as other X10 structs. For example, numeric operations, coercions, and conversions are defined by `operator` definitions, the same way you could for any struct.

Promoting a numeric value to a longer numeric type results in either sign extension or zero extension depending on whether the target type is signed or unsigned. For example, (255 as UByte) as UInt is 255 while (255 as Byte) as Int is -1.

Most of these operations can be defined on user-defined types as well. While it is good practice to keep such operations consistent with the numeric operations whenever possible, the compiler neither enforces nor assumes any particular semantics of user-defined operations.

11.9.1 Conversions and coercions

Specifically, each numeric type can be converted or coerced into each other numeric type, perhaps with loss of accuracy.

Example:

```
val n : Byte = 123 as Byte; // explicit
val f : (Long)=>Boolean = (Long) => true;
val ok = f(n); // implicit
```

11.9.2 Unary plus and unary minus

The unary + operation on numbers is an identity function. The unary - operation on signed numbers is a negation function. On unsigned numbers, these are two's-complement arithmetic; the unsigned number types are closed under unary -. For example, -(0x0F as UByte) is (0xF1 as UByte).

11.10 Bitwise complement

The unary ~ operator, only defined on integral types, complements each bit in its operand.

11.11 Binary arithmetic operations

The binary arithmetic operators perform the familiar binary arithmetic operations: + adds, - subtracts, * multiplies, / divides, and % computes remainder.

On integers, the operands are coerced to the longer of their two types, and then operated upon. Floating point operations are determined by the IEEE 754 standard. The integer / and % throw an exception if the right operand is zero.

11.12 Binary shift operations

When operands of the binary shift operations are of integral type, the expression performs bitwise shifts. The type of the result is the type of the left operand. The right operand, describing a number of bits, must be a Long: `x << y`.

If the promoted type of the left operand is `Int`, the right operand is masked with `0x1f` using the bitwise AND (`&`) operator, giving a number at most the number of bits in an `Int`. If the promoted type of the left operand is `Long`, the right operand is masked with `0x3f` using the bitwise AND (`&`) operator, giving a number at most the number of bits in a `Long`.

The `<<` operator left-shifts the left operand by the number of bits given by the right operand. The `>>` operator right-shifts the left operand by the number of bits given by the right operand. The result is sign extended; that is, if the right operand is k , the most significant k bits of the result are set to the most significant bit of the operand.

The `>>>` operator right-shifts the left operand by the number of bits given by the right operand. The result is not sign extended; that is, if the right operand is k , the most significant k bits of the result are set to `0`. This operation is deprecated, and may be removed in a later version of the language.

11.13 Binary bitwise operations

The binary bitwise operations operate on integral types, which are promoted to the longer of the two types. The `&` operator performs the bitwise AND of the promoted operands. The `|` operator performs the bitwise inclusive OR of the promoted operands. The `^` operator performs the bitwise exclusive OR of the promoted operands.

11.14 String concatenation

The `+` operator is used for string concatenation as well as addition. If either operand is of static type `x10.lang.String`, the other operand is converted to a `String`, if needed, and the two strings are concatenated. String conversion of a non-null value is performed by invoking the `toString()` method of the value. If the value is `null`, the value is converted to "null".

The type of the result is `String`.

For example, `"one " + 2 + true` evaluates to `one 2true`.

11.15 Logical negation

The unary `!` operator applied to type `x10.lang.Boolean` performs logical negation. The type of the result is `Boolean`. If the value of the operand is `true`, the result is `false`; if the value of the operand is `false`, the result is `true`.

11.16 Boolean logical operations

The binary operations `&` and `|` at type `Boolean` perform Boolean logical operations.

The `&` operator evaluates to `true` if both of its operands evaluate to `true`; otherwise, the operator evaluates to `false`.

The `|` operator evaluates to `false` if both of its operands evaluate to `false`; otherwise, the operator evaluates to `true`.

11.17 Boolean conditional operations

The binary `&&` and `||` operations, on `Boolean` values, give conditional or short-circuiting Boolean operations.

The `&&` operator evaluates to `true` if both of its operands evaluate to `true`; otherwise, the operator evaluates to `false`. Unlike the logical operator `&`, if the first operand is `false`, the second operand is not evaluated.

The `||` operator evaluates to `false` if both of its operands evaluate to `false`; otherwise, the operator evaluates to `true`. Unlike the logical operator `||`, if the first operand is `true`, the second operand is not evaluated.

11.18 Relational operations

The relational operations on numeric types compare numbers, producing `Boolean` results.

The `<` operator evaluates to `true` if the left operand is less than the right. The `<=` operator evaluates to `true` if the left operand is less than or equal to the right. The `>` operator evaluates to `true` if the left operand is greater than the right. The `>=` operator evaluates to `true` if the left operand is greater than or equal to the right.

Floating point comparison is determined by the IEEE 754 standard. Thus, if either operand is `NaN`, the result is `false`. Negative zero and positive zero are considered to be equal. All finite values are less than positive infinity and greater than negative infinity.

11.19 Conditional expressions

$$\text{ConditionalExp} ::= \text{ConditionalOrExp} ? \text{Exp} : \text{ConditionalExp} \quad (20.45)$$

A conditional expression evaluates its first subexpression (the condition); if `true` the second subexpression (the consequent) is evaluated; otherwise, the third subexpression (the alternative) is evaluated.

The type of the condition must be Boolean. The type of the conditional expression is some common ancestor (as constrained by §4.10) of the types of the consequent and the alternative.

Example: *a == b ? 1 : 2 evaluates to 1 if a and b are the same, and 2 if they are different. As the type of 1 is Long{self==1} and of 2 is Long{self==2}, the type of the conditional expression has the form Long{c}, where self==1 and self==2 both imply c. For example, it might be Long{true} – or perhaps it might be a more accurate type, like Long{self != 8}. Note that this term has no most accurate type in the X10 type system.*

The subexpression not selected is not evaluated.

Example: *The following use of the conditional expression prevents division by zero. If den==0, the division is not performed at all.*

```
(den == 0) ? 0 : num/den
```

Similarly, the following code performs a method call if op is non-null, and avoids the null pointer error if it is null. Defensive coding like this is quite common when working with possibly-null objects.

```
(ob == null) ? null : ob.toString();
```

11.20 Stable equality

$$\begin{aligned} EqualityExp &::= RelationalExp && (20.59) \\ &\quad | EqualityExp == RelationalExp \\ &\quad | EqualityExp != RelationalExp \\ &\quad | Type == Type \end{aligned}$$

The == and != operators provide a fundamental, though non-abstract, notion of equality. a==b is true if the values of a and b are extremely identical.

- If a and b are values of object type, then a==b holds if a and b are the same object.
- If one operand is null, then a==b holds iff the other is also null.
- The structs in `x10.lang` have unsurprising concepts of ==:
 - In Boolean, true == true and false == false.
 - In Char, c == d iff `c.ord() == d.ord()`.
 - Equality in Double and Float is IEEE floating-point equality.
 - Two GlobalRefs are == if they refer to the same object.

- The integral types, `Byte`, `Short`, `Int`, `Long`, and their unsigned versions, use binary equality.
- If the operands both have struct type and are not in `x10.lang`, then they must be structurally equal; that is, they must be instances of the same struct and all their fields or components must be `==`.
- The definition of equality for function types is specified in §10.4.
- No implicit coercions are performed by `==`.
- It is a static error to have an expression `a == b` if the types of `a` and `b` are disjoint.

`a != b` is true iff `a==b` is false.

The predicates `==` and `!=` may not be overridden by the programmer.

`==` provides a *stable* notion of equality. If two values are `==` at any time, they remain `==` forevermore, regardless of what happens to the mutable state of the program.

Example: *Regardless of the values and types of `a` and `b`, or the behavior of `any_code_at_all` (which may, indeed, be any code at all—not just a method call), the value of `a==b` does not change:*

```
val a = something();
val b = something_else();
val eq1 = (a == b);
any_code_at_all();
val eq2 = (a == b);
assert eq1 == eq2;
```

11.20.1 No Implicit Coercions for `==`

`==` is a primitive operation in X10 – one of very few. Most operations, like `+` and `<=`, are defined as operators. `==` and `!=` are not. As non-operators, they need not and do not follow the general method resolution procedure of §8.12. In particular, while operators perform implicit conversions on their arguments, `==` and `!=` do not.

The advantage of this restriction is that `==`’s behavior is as simple and efficient as possible. It never runs user-defined code, and the compiler can analyze and understand it in detail – and guarantee that it is efficient.

The disadvantage is that certain straightforward-looking idioms do not work. One may not test that an `Int` variable is `==` to a long like `0`:

```
//ERROR: for(var i : Int = 0n; i != 100; i++) {}
```

A `Int` like `i` can never `==` a `Long` like `100`. Because `==` does not permit implicit coercions, `i` stays a `Int`. The loop must be written with a comparison of two `Int`s:

```
for(var i : Int = 0n; i != 100n; i++) {}
```

Because the operation `<=` is a regular operator, and thus uses coercions in its arguments, it is legal (although not recommended) to write the loop as

```
for(var i : Int = 0n; i <= 100; i++) {}
```

In this formulation, `i` will be coerced to a `Long` on each loop iteration so it can be compared using `<=` against `100`.

Example: *If numbers are cast to `Any`, they are compared as values of type `Any`, not as numbers. For example, `1 as Any == 1ul as Any` is not a static error (because it is comparing two values of type `Any`), and returns `false` (because the two `Any` values refer to different values — indeed, to values of different types, `Int` and `ULong`).*

11.20.2 Non-Disjointness Requirement

It is, in many cases, a static error to have an expression `a==b` where `a` and `b` could not possibly be equal, based on their types. (In one case it is a static error even though they *could* be equal.) This is a practical codicil to §11.20.1. Consider the illegal code

```
// NOT ALLOWED
for(var i : Long = 0; i != 100; i++)
```

`100` and `100L` are different values; they are not `==`. A coercion could make them equal, but `==` does not allow coercions. So, if `100 == 100L` were going to return anything, it would have to return `false`. This would have the unfortunate effect of making the `for` loop run forever.

Since this and related idioms are so common, and since so many programmers are used to languages which are less precise about their numeric types, X10 avoids the mistake by declaring it a static error in most cases. Specifically, `a==b` is not allowed if, by inspection of the types, `a` and `b` could not possibly be equal.

Example: *Nonetheless, it is possible to wind up comparing values of different numeric types. Even though, say, `0n` and `0L` represent the same number, they are different values and of different types, and hence, `0n != 0L`. The expression `0n == 0L` does not compile. However, if you hide type information from X10, you can get a similar expression to compile:*

```
val a : Any = 0n;
val b : Any = 0L;
assert a != b;
```

- Numbers of different base types cannot be equal, and thus cannot be compared for equality. `100==100L` is a static error. To compare numbers, explicitly cast them to the same type: `100 as Long == 100L`.
- Indeed, structs of different types cannot be equal, and so they cannot be compared for equality.

- For objects, the story is different. Unconstrained object types can always be compared for equality. Given objects of unrelated classes `a:Person` and `b:Theory`, `a==b` could be true if `a==null` and `b==null`. Despite this, `a==b` is a static error, because it is generally a programming mistake. `a as Any == b as Any` can be used to express the equality, if it is necessary.
- Constraints are ignored in determining whether an equality is statically allowed. For example, the following is allowed:

```
def m(a:Long{self==1}, b:Long{self==2}) = (a==b);
```

- Explicit casts erase type information. If you wanted to have a comparison `a==b` for `a:Person{self!=null}` and `b:Theory`, you could write it as `a as Any == b as Any`. It would, of course, return `false`, but it would not be a compiler error.¹ A struct and an object may both be cast to `Any` and compared for equality, though they, too, will always be different.

11.21 Allocation

ObCreationExp ::= `new TypeName TypeArgs? (ArgumentList?) ClassBody?` (20.126)
 | `Primary . new Id TypeArgs? (ArgumentList?) ClassBody?`
 | `FullyQualifiedNamespace . new Id TypeArgs? (ArgumentList?) ClassBody?`

An allocation expression creates a new instance of a class and invokes a constructor of the class. The expression designates the class name and passes type and value arguments to the constructor.

The allocation expression may have an optional class body. In this case, an anonymous subclass of the given class is allocated. An anonymous class allocation may also specify a single super-interface rather than a superclass; such an anonymous class does not have a superclass.

If the class is anonymous—that is, if a class body is provided—then the constructor is selected from the superclass. The constructor to invoke is selected using the same rules as for method invocation (§11.6).

The type of an allocation expression is the return type of the constructor invoked, with appropriate substitutions of actual arguments for formal parameters, as specified in §11.6.

§8.14.1 describes allocation expressions for inner classes.

It is illegal to allocate an instance of an **abstract** class. The usual visibility rules apply to allocations: it is illegal to allocate an instance of a class or to invoke a constructor that is not visible at the allocation expression.

Note that instantiating a struct type can use function application syntax; `new` is optional. As structs do not have subclassing, there is no need or possibility of a *ClassBody*.

¹Code generators often find this trick to be useful.

11.22 Casts and Conversions

```
CastExp ::= Primary (20.30)
| ExpName
| CastExp as Type
```

The cast and conversion operation `e as T` may be used to force an expression into a given type `T`, if is permissible at run time, and either a compile-time error or a runtime exception (`x10.lang.ClassCastException`) if it is not.

The `e as T` operation comes in two forms. Which form applies depends on both the source type (the type of `e`) and the target type `T`.

- **Cast:** A cast makes a value have a different type, without changing the value's identity. For example, "`a String`" `as Any` simply reconsiders the `String` object as an `Any`. This cast does not need to do any run-time computation, since every `String` is an `Any`; a cast in the reverse direction, from `Any` to `String`, would need a run-time check that the `Any` was in fact a `String`. Casts are all system-defined, following from the X10 type system.
- **Conversions:** A conversion takes a value of one type and produces one of a different type which, conceptually, means the same thing. For example, `1 as Float` is a conversion. It performs some computation on `1` to come up with a `Float` value. Conversions are all library- or user-defined.

11.22.1 Casts

A cast `v as T2` re-imagines a value `v` of one type `T1` as being a value of another type `T2`. The value itself does not change, nor is a new value computed. The only run-time computation that happens is to check that `v` is indeed a value of type `T2` (which, in many cases, is unnecessary), and auto-boxing (§9.2).

Casts to generic types can be unsound. The instantiations of the generic types have constraints, but the runtime does not preserve the representation of these types. See §4.5.5 for more details.

There are two forms of casts. *Upcasts* happen when `T1 <: T2`, that is, when a value is being cast to a more general type. Upcasts often don't require any runtime computation at all, since, if `T1 <: T2` and `T2 isref`, every value of type `T1` is automatically one of type `T2`. For example, "`A String`" `as Any` is a trivial upcast: every `String` can simply be used as a value of type `Any` because it is already represented as a heap-allocated object. Upcasts from structs to interface types however do require auto-boxing, such as `1 as Any`.

Downcasts are casts which are not upcasts. Often they are recasting something from a more general to a more specific type, though casts that cross the type hierarchy laterally are also called downcasts.

```
val ob : Any = "a String" as Any; // upcast
val st : String = ob as String;   // downcast
assert st == ob;
```

Example:

In the following example, Snack and Crunchy are unrelated interfaces: neither inherits from the other. Some objects are both; some are one but not the other. Casting from a Crunchy to a Snack requires confirming that the value being cast is indeed a Snack.

```
interface Snack {}
interface Crunchy {}
class Pretzel implements Snack, Crunchy{}
class Apricot implements Snack{}
class Gravel implements Crunchy{}
class Example{
    def example(crunchy : Crunchy) {
        if (crunchy instanceof Snack) {
            val snack = crunchy as Snack;
        } } }
```

An upcast `v as T2` requires no computation. A downcast `v as T2` requires testing that `v` really is a value of type `T2`. In either case, the cast returns the value `v`; casts do not change value identity.

When evaluating `E as T{c}`, first the value of `E` is converted to type `T` (which may fail), and then the constraint `{c}` is checked (which may also fail).

- If `T` is a class, then the first half of the cast succeeds if the run-time value of `E` is an instance of class `T`, or of a subclass.
- If `T` is an interface, then the first half of the cast succeeds if the run-time value of `E` is an instance of a class or struct implementing `T`.
- If `T` is a struct type, then the first half of the cast succeeds if the run-time value of `E` is an instance of `T`.
- If `T` is a function type, then the first half of the cast succeeds if the run-time value of `X` is a function of that type, or an object or struct which implements it.

If the first half of the cast succeeds, the second half – the constraint `{c}` – must be checked. In general this will be done at runtime, though in special cases it can be checked at compile time. For example, `n as Long{self != w}` succeeds if `n != w` — even if `w` is a value read from input, and thus not determined at compile time.

The compiler may forbid casts that it knows cannot possibly work. If there is no way for the value of `E` to be of type `T{c}`, then `E as T{c}` can result in a static error, rather than a runtime error. For example, `1 as Long{self==2}` may fail to compile,

because the compiler knows that 1, which has type `Long{self==1}`, cannot possibly be of type `Long{self==2}`.

If, for some reason, you need to write one of these forbidden casts, cast to `Any` first. `(1 as Any) as Long{self==2}` always returns false, but compiles.

11.22.2 Explicit Conversions

Explicit conversions are written with the same syntax as casts: `v as T2`. Explicit conversions transform a value of one type `T1` to an unrelated type `T2`. Unlike casts, conversions *do* execute code, and *may* (and generally do) return new values.

Explicit conversions do not arise spontaneously, as casts do. They may be programmed directly, using the `operator` syntax of §8.7.3. Implicit coercions can also be called explicitly as conversions. (The reverse is not true – explicit conversions cannot be used as implicit conversions.)

The numeric types in `x10.lang` have explicit conversions, as described in §11.23.1. These conversions enable `1 as Float` and the like.

Example: *The following class has an explicit conversion from Long to Knot, and an implicit one from String to Knot. a uses the explicit conversion, b uses the implicit coercion, and c uses the implicit coercion explicitly.*

```
class Knot(s:String){
    public def is(t:String):Boolean = s.equals(t);
    // explicit conversion
    public static operator (n:Long) as Knot = new Knot("knot-" + n);
    // implicit coercion
    public static operator (s:String):Knot = new Knot(s);
    // using them
    public static def example() {
        val a : Knot = 1 as Knot;
        val b : Knot = "frayed";
        val c : Knot = "three" as Knot;
        assert a.is("knot-1") && b.is("frayed") && c.is("three");
    }
}
```

11.22.3 Resolving Ambiguity

If `v as T` could either be a cast or an explicit coercion, X10 treats its as a cast. With the `VERBOSE` compiler flag, this is flagged as a warning.

Example: *The Person class provides an explicit conversion from its subclass Fop to itself. However, since Fop is a subclass of Person, using the `as` operator invokes the upcast, rather than the explicit conversion. This is visible in the example because the*

user-defined operator f as Person returns new Person() (just like the asPerson method), while the upcast returns f itself.

```
class Person {
    static operator (f:Fop) as Person = new Person();
    static def asPerson(f:Fop) = new Person();
    public static def example() {
        val f = new Fop();
        val cast = f as Person; // WARNING on this line
        assert cast == f;
        val meth = asPerson(f);
        assert meth != f;
    }
}
class Fop extends Person {}
```

The definition of an explicit conversion in this case is of little value, since any use of it in the f as Person syntax will invoke the upcast.

11.23 Coercions and conversions

A *coercion* does not change object identity; a coerced object may be explicitly coerced back to its original type through a cast. A *conversion* may change object identity if the type being converted to is not the same as the type converted from. X10 permits both user-defined coercions and conversions (§11.23.2).

11.23.1 Coercions

$CastExp ::= Primary$ $ExpName$ $CastExp \text{ as } Type$	(20.30)
--	---------

Subsumption coercion. A value of a subtype may be implicitly coerced to any supertype.

Example: If $Child <: Person$ and val rhys:Child , then $rhys$ may be used in any context that expects a $Person$. For example,

```
class Example {
    def greet(Person) = "Hi!";
    def example(rhys: Child) {
        greet(rhys);
    }
}
```

Similarly, 2 (whose innate type is Long{self==2}) is usable in a context requiring a non-zero integer (Long{self != 0}).

Explicit Coercion (Casting with as) All classes and interfaces allow the use of the `as` operator for explicit type coercion. Any class or interface may be cast to any interface. Any interface may be cast to any class. Also, any interface can be cast to a struct that implements (directly or indirectly) that interface.

Example: *In the following code, a Person is cast to Childlike. There is nothing in the class definition of Person that suggests that a Person can be Childlike. However, the Person in question, p, is actually a HappyChild — a subclass of Person — and is, in fact, Childlike.*

Similarly, the Childlike value cl is cast to Happy. Though these two interfaces are unrelated, the value of cl is, in fact, Happy. And the Happy value hc is cast to the class Child, though there is no relationship between the two, but the actual value is a HappyChild, and thus the cast is correct at runtime.

Cyborg is a struct rather than a class. So, it cannot have substructs, and all the interfaces of all Cyborgs are known: a Cyborg is Personable, but not Childlike or Happy. So, it is correct and meaningful to cast r to Personable. There is no way that a cast to Childlike could succeed, so r as Childlike is a static error.

```
interface Personable {}
class Person implements Personable {}
interface Childlike extends Personable {}
class Child extends Person implements Childlike {}
struct Cyborg implements Personable {}
interface Happy {}
class HappyChild extends Child implements Happy {}
class Example {
    static def example() {
        var p : Person = new HappyChild();
        // class -> interface
        val cl : Childlike = p as Childlike;
        // interface -> interface
        val hc : Happy = cl as Happy;
        // interface -> class
        val ch : Child = hc as Child;
        var r : Cyborg = Cyborg();
        val rl : Personable = r as Personable;
        // ERROR: val no = r as Childlike;
    }
}
```

If the value coerced is not an instance of the target type, and no coercion operators that can convert it to that type are defined, a `ClassCastException` is thrown. Casting to a constrained type may require a run-time check that the constraint is satisfied.

It is a static error, rather than a `ClassCastException`, when the cast is statically determinable to be impossible.

Effects of explicit numeric coercion Coercing a number of one type to another type gives the best approximation of the number in the result type, or a suitable disaster value if no approximation is good enough.

- Casting a number to a *wider* numeric type is safe and effective, and can be done by an implicit conversion as well as an explicit coercion. For example, `4` as `Long` produces the `Long` value of `4`.
- Casting a floating-point value to an integer value truncates the digits after the decimal point, thereby rounding the number towards zero. `54.321` as `Int` is `54n`, and `-54.321` as `Int` is `-54n`. If the floating-point value is too large to represent as that kind of integer, the coercion returns the largest or smallest value of that type instead: `1e110` as `Int` is `Int.MAX_VALUE`, *viz.* `2147483647`.
- Casting a `Double` to a `Float` normally truncates binary digits: `0.12345678901234567890` as `Float` is approximately `0.12345679f`. This can turn a nonzero `Double` into `0.0f`, the zero of type `Float`: `1e-100` as `Float` is `0.0f`. Since `Doubles` can be as large as about `1.79E308` and `Floats` can only be as large as about `3.4E38f`, a large `Double` will be converted to the special `Float` value of `Infinity`: `1e100` as `Float` is `Infinity`.
- Integers are coerced to smaller integer types by truncating the high-order bits. If the value of the large integer fits into the smaller integer's range, this gives the same number in the smaller type: `12` as `Byte` is the `Byte`-sized `12`, `-12` as `Byte` is `-12`. However, if the larger integer *doesn't* fit in the smaller type, the numeric value and even the sign can change: `254` as `Byte` is the `Byte`-sized `-2y`.
- Casting an unsigned integer type to a signed integer type of the same size (*e.g.*, `UIInt` to `Int`) preserves 2's-complement bit pattern (*e.g.*, `UIInt.MAX_VALUE` as `Int` == `-1n`). Casting an unsigned integer type to a signed integer type of a different size is equivalent to first casting to an unsigned integer type of the target size, and then casting to a signed integer type.
- Casting a signed integer type to an unsigned one is similar.

User-defined Coercions

Users may define coercions from arbitrary types into the container type `B`, and coercions from `B` to arbitrary types, by providing `static` operator definitions for the `as` operator in the definition of `B`.

Example:

```

class Bee {
    public static operator (x:Bee) as Long = 1;
    public static operator (x:Long) as Bee = new Bee();
    def example() {
        val b:Bee = 2 as Bee;
        assert (b as Long) == 1;
    }
}

```

11.23.2 Conversions

Widening numeric conversion. A numeric type may be implicitly converted to a wider numeric type. In particular, an implicit conversion may be performed between a numeric type and a type to its right, below:

```

Byte < Short < Int < Long < Float < Double
UByte < UShort < UInt < ULong

```

Furthermore, an unsigned integer value may be implicitly coerced to a signed type large enough to hold any value of the type: UByte to Short, UShort to Int, UInt to Long. There are no implicit conversions from signed to unsigned numbers, since they cannot treat negatives properly.

There are no implicit conversions in cases when overflow is possible. For example, there is no implicit conversion between Int and UInt. If it is necessary to convert between these types, use `n as Int` or `n as UInt`, generally with a test to ensure that the value will fit and code to handle the case in which it does not.

String conversion. Any value that is an operand of the binary + operator may be converted to String if the other operand is a String. A conversion to String is performed by invoking the `toString()` method.

User defined conversions. The user may define implicit conversion operators from type A to a container type B by specifying an operator in B's definition of the form:

```
public static operator (r: A): T = ...
```

The return type T should be a subtype of B. The return type need not be specified explicitly; it will be computed in the usual fashion if it is not. However, it is good practice for the programmer to specify the return type for such operators explicitly. The return type can be more specific than simply B, for cases when there is more information available.

Example: The code for `x10.lang.Point` contains a conversion from a Rails of longs to Points of the same length:

```
public operator (r: Rail[Long]): Point(r.size)
    = make(r);
```

This conversion is used whenever a Rail of integers appears in a context that requires a Point, such as subscripting. Note that a requires a Point of rank 2 as a subscript, and that a two-element Rail (like [2, 4]) is converted to a Point(2).

```
val a = new Array[String](Region.make(2..3, 4..5), "hi!");
a([2,4]) = "converted!";
```

11.24 instanceof

X10 permits types to be used in an instanceof expression to determine whether an object is an instance of the given type:

$\begin{array}{l} \textit{RelationalExp} ::= \textit{ShiftExp} \\ \textit{HasZeroConstraint} \\ \textit{SubtypeConstraint} \\ \textit{RelationalExp} < \textit{ShiftExp} \\ \textit{RelationalExp} > \textit{ShiftExp} \\ \textit{RelationalExp} \leq \textit{ShiftExp} \\ \textit{RelationalExp} \geq \textit{ShiftExp} \\ \textit{RelationalExp} \text{ instanceof } \textit{Type} \end{array}$	(20.144)
---	---

In the above expression, *Type* is any type. At run time, the result of `e instanceof T` is `true` if the value of `e` is an instance of type `T`. Otherwise the result is `false`. This determination may involve checking that the constraint, if any, associated with the type is true for the given expression.

For example, `3 instanceof Long{self==x}` is an overly-complicated way of saying `3==x`.

However, it is a static error if `e` cannot possibly be an instance of `C{c}`; the compiler will reject `1 instanceof Long{self == 2}` because `1` can never satisfy `Long{self == 2}`. Similarly, `1 instanceof String` is a static error, rather than an expression always returning `false`.

If `x instanceof T` returns `true` for some value `x` and type `T`, then `x as T` will evaluate normally.

Limitation: X10 does not currently handle instanceof of generics in the way you might expect. For example, `r instanceof Array[Long{self != 0}]` does not test that every element of `r` is non-zero; instead, the compiler gives an unsound cast warning.

11.24.1 Nulls in Constraints in as and instanceof

Both `as` and `instanceof` expressions can throw `NullPointerExceptions`, if the constraints involve selecting fields or properties of variables which are bound to `null`.

These operations give some guarantees for any type `T`, constraint `c`, and class `SomeObj` with an `a` field:

1. `null instanceof T` always returns `false`. It never throws an exception. It never returns `true`, not even in cases where `null` could be assigned to a variable of type `T`.
2. `null` can be assigned to a variable of type `SomeObj{self.a==b}`, or, more broadly, to a variable of a constrained object type whose constraint does not explicitly exclude `null`. This is the case even though `null.a==b` would throw a `NullPointerException` rather than evaluate to either `true` or `false`.
3. If `x instanceof T` returns `true`, then `x as T` is a cast rather than an explicit conversion, and will succeed and have static type `T`.
4. If the static type of `x` is `T`, then `x instanceof T` and `x as T` will do one of these:
 - Succeed, with `x instanceof T` returning `true`, and `x as T` being a cast and returning value of type `T`; **or**
 - Throw a `NullPointerException`.
 - If `x==null`, then `x instanceof T` will always return `false`, and `x as T` will either return a `null` of type `T`, or, if `T` has a constraint which tries to extract a field of `x`, will throw a `NullPointerException`.
5. If `x instanceof SomeObj{self.a==b}` is `true`, then `x.a==b` evaluates to `true` (rather than a null pointer exception). Indeed, in general, if `x instanceof T{c}` succeeds, then `cc` evaluates to `true`, where `cc` is `c` with suitable occurrences of `self` replaced by `x`.

11.25 Subtyping expressions

$$\begin{array}{lcl} \text{SubtypeConstraint} & ::= & \text{Type} <: \text{Type} \\ & | & \text{Type} :> \text{Type} \end{array} \quad (20.155)$$

The subtyping expression `T1 <: T2` evaluates to `true` if `T1` is a subtype of `T2`.

The expression `T1 :> T2` evaluates to `true` if `T2` is a subtype of `T1`.

The expression `T1 == T2` evaluates to `true` if `T1` is a subtype of `T2` and if `T2` is a subtype of `T1`.

Example: Subtyping expressions are particularly useful in giving constraints on generic types. `x10.util.Ordered[T]` is an interface whose values can be compared with values of type T. In particular, `T <: x10.util.Ordered[T]` is true if values of type T can be compared to other values of type T. So, if we wish to define a generic class `OrderedList[T]`, of lists whose elements are kept in the right order, we need the elements to be ordered. This is phrased as a constraint on T:

```
class OrderedList[T]{T <: x10.util.Ordered[T]} {  
    // ...  
}
```

11.26 Rail Constructors

Primary ::= [*ArgumentList*?]

X10 includes short syntactic forms for constructing Rails. Enclose some expressions in brackets to put them in a Rail:

```
val ints <: Rail[Long] = [1,3,7,21];
```

The expression `[e1, ..., en]` produces an n-element `Rail[T]`, where T is the computed common supertype (§4.10) of the types of the expressions e_i .

Example: The type of `[0,1,2]` is `Rail[Long]`. The type of `[0]` is `Rail[Long{self==0}]`.

To make a `Rail[Long]` containing just a 0, use `[0 as Long]`. The as Long masks more detailed type information, such as the fact that 0 is zero.

Example: Occasionally one does actually need `Rail[Long{self==0}]`, or, say, `Rail[Eel{self != null}]`, a rail of non-null Eels. For these cases, cast one or more of the elements of the rail to the desired type, and the rail constructor will do the right thing.

```
val zero <: Rail[Long{self == 0}]  
    = [0];  
val non1 <: Rail[Long{self != 1}]  
    = [0 as Long{self != 1}];  
val eels <: Rail[Eel{self != null}]  
    = [new Eel() as Eel{self != null},  
        new Eel(), new Eel()];
```

11.27 Parenthesized Expressions

If E is any expression, (E) is an expression which, when evaluated, produces the same result as E.

Example: *The main use of parentheses is to write complex expressions for which the standard precedence order of operations is not appropriate: $1+2*3$ is 7, but $(1+2)*3$ is 9.*

Similarly, but perhaps less familiarly, parentheses can disambiguate other expressions. In the following code, `funny.f` is a field-selection expression, and so `(funny.f)()` means “select the `f` field from `funny`, and evaluate it”. However, `funny.f()` means “evaluate the `f` method on object `funny`.“

```
class Funny {
    def f () = 1;
    val f = () => 2;
    static def example() {
        val funny = new Funny();
        assert funny.f() == 1;
        assert (funny.f)() == 2;
    }
}
```

Note that this does *not* mean that `E` and `(E)` are identical in all respects; for example, if `i` is an `Long` variable, `i++` increments `i`, but `(i)++` is not allowed. `++` is an assignment; it operates on variables, not merely values, and `(i)` is simply an expression whose *value* is the same as that of `i`.

12 Statements

This chapter describes the statements in the sequential core of X10. Statements involving concurrency and distribution are described in §14.

12.1 Empty statement

The empty statement ; does nothing.

Example: *Sometimes, the syntax of X10 requires a statement in some position, but you do not actually want to do any computation there. The following code searches the rail a for the value v, assumed to appear somewhere in a, and returns the index at which it was found. There is no computation to do in the loop body, so we use an empty statement there.*

```
static def search[T](a: Rail[T], v: T):Long {
    var i : Long;
    for(i = 0L; a(i) != v; i++)
        ;
    return i;
}
```

12.2 Local variable declaration

<i>LocVarDecln</i>	::=	<i>Mods?</i> <i>VarKeyword</i> <i>VariableDecls</i>	(20.110)
		<i>Mods?</i> <i>VarDeclsWType</i>	
		<i>Mods?</i> <i>VarKeyword</i> <i>FormalDecls</i>	
<i>LocVarDeclnStmt</i>	::=	<i>LocVarDecln</i> ;	(20.111)
<i>VarDeclsWType</i>	::=	<i>VarDeclWType</i>	(20.201)
		<i>VarDeclsWType</i> , <i>VarDeclWType</i>	
<i>VariableDecls</i>	::=	<i>VariableDeclr</i>	(20.204)
		<i>VariableDecls</i> , <i>VariableDeclr</i>	
<i>VariableInitializer</i>	::=	<i>Exp</i>	(20.205)
<i>FormalDecls</i>	::=	<i>FormalDeclr</i>	(20.78)
		<i>FormalDecls</i> , <i>FormalDeclr</i>	

Short-lived variables are introduced by local variables declarations, as described in §12.2. Local variables may be declared only within a block statement (§12.3). The scope of a local variable declaration is the subsequent statements in the block.

```
if (a > 1) {
    val b = a/2;
    var c : Long = 0;
    // b and c are defined here
}
// b and c are not defined here.
```

Variables declared in such statements shadow variables of the same name declared elsewhere. A local variable of a given name, say *x*, cannot shadow another local variable or parameter named *x* unless there is an intervening method, constructor, initializer, or closure declaration.

Example: *The following code illustrates both legal and illegal uses of shadowing. Note that a shadowed field name *x* can still be accessed as *this.x*.*

```
class Shadow{
    var x : Long;
    def this(x:Long) {
        // Parameter can shadow field
        this.x = x;
    }
    def example(y:Long) {
        val x = "shadows a field";
        // ERROR: val y = "shadows a param";
        val z = "local";
        for (a in [1,2,3]) {
            // ERROR: val x = "can't shadow local var";
        }
        async {
```

```
// ERROR: val x = "can't shadow through async";
}
val f = () => {
    val x = "can shadow through closure";
    x
};
class Local {
    val f = at(here) { val x = "can here"; x };
    def this() { val x = "can here, too"; }
}
}
```

Example: Note that recursive definitions of local variables is not allowed. There are few useful recursive declarations of objects and structs; `x`, in the following example, has no meaningful definition. Recursive declarations of local functions is forbidden, even though (like `f` below) there are meaningful uses of it.

```
val x : Long = x + 1; // ERROR: recursive local declaration
val f : (Long)=>Long
= (n:Long) => (n <= 2) ? 1 : f(n-1) + f(n-2);
// ERROR: recursive local declaration
```

12.3 Block statement

$$\text{Block} ::= \{ \text{BlockStmts}^? \} \quad (20.25)$$

$$\text{BlockStmts} ::= \text{BlockInteriorStmt} \quad (20.27)$$

$$\begin{aligned} \text{BlockInteriorStmt} ::= & \text{LocVarDeclnStmt} \\ & | \text{ClassDecln} \\ & | \text{StructDecln} \\ & | \text{TypeDefDecln} \\ & | \text{Stmt} \end{aligned} \quad (20.26)$$

A block statement consists of a sequence of statements delimited by “`{`” and “`}`”. When a block is evaluated, the statements inside of it are evaluated in order. Blocks are useful for putting several statements in a place where X10 asks for a single one, such as the consequent of an `if`, and for limiting the scope of local variables.

```
if (b) {
    // This is a block
    val v = 1;
    S1(v);
    S2(v);
}
```

12.4 Expression statement

Any expression may be used as a statement.

<i>ExpStmt</i>	$::=$	<i>StmtExp</i> ;	(20.63)
<i>StmtExp</i>	$::=$	<i>Assignment</i>	(20.152)
		<i>PreIncrementExp</i>	
		<i>PreDecrementExp</i>	
		<i>PostIncrementExp</i>	
		<i>PostDecrementExp</i>	
		<i>MethodInvo</i>	
		<i>ObCreationExp</i>	

The expression statement evaluates an expression. The value of the expression is not used. Side effects of the expression occur, and may produce results used by following statements. Indeed, statement expressions which terminate without side effects cannot have any visible effect on the results of the computation.

Example:

```
class StmtEx {
    def this() {
        x10.io.Console.OUT.println("New StmtEx made"); }
    static def call() {
        x10.io.Console.OUT.println("call!");}
    def example() {
        var a : Long = 0;
        a = 1; // assignment
        new StmtEx(); // allocation
        call(); // call
    }
}
```

12.5 Labeled statement

LabeledStatement $::=$ *Id* : *Statement*

Statements may be labeled. The label may be used to describe the target of a `break` statement appearing within a substatement (which, when executed, ends the labeled statement), or, in the case of a loop, a `continue` as well (which, when executed, proceeds to the next iteration of the loop). The scope of a label is the statement labeled.

Example: *The label on the outer for statement allows continue and break statements to continue or break it. Without the label, continue or break would only continue or break the inner for loop.*

```

lbl : for (i in 1..10) {
    for (j in i..10) {
        if (a(i,j) == 0) break lbl;
        if (a(i,j) == 1) continue lbl;
        if (a(i,j) == a(j,i)) break lbl;
    }
}

```

In particular, a block statement may be labeled: L:{S}. This allows the use of `break` L within S to leave S, which can, if carefully used, avoid deeply-nested `ifs`.

Example:

```

multiphase: {
    if (!exists(filename)) break multiphase;
    phase1(filename);
    if (!suitable_for_phase_2(filename)) break multiphase;
    phase2(filename);
    if (!suitable_for_phase_3(filename)) break multiphase;
    phase3(filename);
}
// Now the file has been phased as much as possible

```

Limitation: Blocks cannot currently be labeled.

12.6 Break statement

BreakStmt ::= break Id[?] ; (20.29)

An unlabeled `break` statement exits the currently enclosing loop or switch statement. A labeled `break` statement exits the enclosing statement with the given label. It is illegal to break out of a statement not defined in the current method, constructor, initializer, or closure. `break` is only allowed in sequential code.

Example: *The following code searches for an element of a C-style two-dimensional array and breaks out of the loop when it is found:*

```

var found: Boolean = false;
outer: for (i in a.range)
    for (j in a(i).range)
        if (a(i)(j) == v) {
            found = true;
            break outer;
        }

```

12.7 Continue statement

ContinueStmt ::= `continue` *Id*? ; (20.50)

An unlabeled `continue` skips the rest of the current iteration of the innermost enclosing loop, and proceeds on to the next. A labeled `continue` does the same to the enclosing loop with that label. It is illegal to continue a loop not defined in the current method, constructor, initializer, or closure. `continue` is only allowed in sequential code.

12.8 If statement

IfThenStmt ::= `if` (*Exp*) *Stmt* (20.93)

IfThenElseStmt ::= `if` (*Exp*) *Stmt* `else` *Stmt* (20.92)

An if statement comes in two forms: with and without an else clause.

The if-then statement evaluates a condition expression, which must be of type Boolean. If the condition is `true`, it evaluates the then-clause. If the condition is `false`, the if-then statement completes normally.

The if-then-else statement evaluates a Boolean expression and evaluates the then-clause if the condition is `true`; otherwise, the `else`-clause is evaluated.

As is traditional in languages derived from Algol, the if-statement is syntactically ambiguous. That is,

```
if (B1) if (B2) S1 else S2
```

could be intended to mean either

```
if (B1) { if (B2) S1 else S2 }
```

or

```
if (B1) {if (B2) S1} else S2
```

X10, as is traditional, attaches an `else` clause to the most recent `if` that doesn't have one. This example is interpreted as `if (B1) { if (B2) S1 else S2 }`.

12.9 Switch statement

<i>SwitchStmt</i>	$::= \text{switch} (\text{Exp}) \text{SwitchBlock}$	(20.162)
<i>SwitchBlock</i>	$::= \{ \text{SwitchBlockGroups}^? \text{SwitchLabels}^? \}$	(20.157)
<i>SwitchBlockGroups</i>	$::= \text{SwitchBlockGroup}$ $\text{SwitchBlockGroups} \text{SwitchBlockGroup}$	(20.159)
<i>SwitchBlockGroup</i>	$::= \text{SwitchLabels} \text{BlockStmts}$	(20.158)
<i>SwitchLabels</i>	$::= \text{SwitchLabel}$ $\text{SwitchLabels} \text{SwitchLabel}$	(20.161)
<i>SwitchLabel</i>	$::= \text{case} \text{ConstantExp} :$ $\text{default} :$	(20.160)

A switch statement evaluates an index expression and then branches to a case whose value is equal to the value of the index expression. If no such case exists, the switch branches to the `default` case, if any.

Statements in each case branch are evaluated in sequence. At the end of the branch, normal control-flow falls through to the next case, if any. To prevent fall-through, a case branch may be exited using a `break` statement.

The index expression must be of type `Int`. Case labels must be of type `Int`, `Byte`, or `Short`, and must be compile-time constants. Case labels cannot be duplicated within the `switch` statement.

Example: *In this switch, case 1 falls through to case 2. The other cases are separated by breaks.*

```
switch (i) {
    case 1n: println("one, and ");
    case 2n: println("two");
        break;
    case 3n: println("three");
        break;
    default: println("Something else");
        break;
}
```

12.10 While statement

<i>WhileStmt</i>	$::= \text{while} (\text{Exp}) \text{Stmt}$	(20.208)
------------------	---	----------

A while statement evaluates a Boolean-valued condition and executes a loop body if `true`. If the loop body completes normally (either by reaching the end or via a `continue` statement with the loop header as target), the condition is reevaluated and the loop repeats if `true`. If the condition is `false`, the loop exits.

Example: A loop to execute the process in the Collatz conjecture (a.k.a. $3n+1$ problem, Ulam conjecture, Kakutani's problem, Thwaites conjecture, Hasse's algorithm, and Syracuse problem) can be written as follows:

```
while (n > 1) {
    n = (n % 2 == 1) ? 3*n+1 : n/2;
}
```

12.11 Do-while statement

DoStmt ::= do *Stmt* while (*Exp*) ; (20.56)

A **do-while** statement executes the loop body, and then evaluates a Boolean-valued condition expression. If **true**, the loop repeats. Otherwise, the loop exits.

12.12 For statement

<i>ForStmt</i>	::= <i>BasicForStmt</i>	(20.74)
	<i>EnhancedForStmt</i>	
<i>BasicForStmt</i>	::= for (<i>ForInit</i> ? ; <i>Exp</i> ? ; <i>ForUpdate</i> ?) <i>Stmt</i>	(20.22)
<i>ForInit</i>	::= <i>StmtExpList</i>	(20.73)
	<i>LocVarDecln</i>	
<i>ForUpdate</i>	::= <i>StmtExpList</i>	(20.75)
<i>StmtExpList</i>	::= <i>StmtExp</i>	(20.153)
	<i>StmtExpList</i> , <i>StmtExp</i>	
<i>EnhancedForStmt</i>	::= for (<i>LoopIndex</i> in <i>Exp</i>) <i>Stmt</i>	(20.58)
	for (<i>Exp</i>) <i>Stmt</i>	

for statements provide bounded iteration, such as looping over a list. It has two forms: a basic form allowing near-arbitrary iteration, *a la C*, and an enhanced form designed to iterate over a collection.

A basic **for** statement provides for arbitrary iteration in a somewhat more organized fashion than a **while**. The loop **for(init; test; step)body** is similar to:

```
{
    init;
    while(test) {
        body;
        step;
    }
}
```

except that `continue` statements which continue the `for` loop will perform the `step`, which, in the `while` loop, they will not do.

`init` is performed before the loop, and is traditionally used to declare and/or initialize the loop variables. It may be a single variable binding statement, such as `var i:Long = 0` or `var i:Long=0, j:Long=100`. (Note that a single variable binding statement may bind multiple variables.) Variables introduced by `init` may appear anywhere in the `for` statement, but not outside of it. Or, it may be a sequence of expression statements, such as `i=0, j=100`, operating on already-defined variables. If omitted, `init` does nothing.

`test` is a Boolean-valued expression; an iteration of the loop will only proceed if `test` is true at the beginning of the loop, after `init` on the first iteration or after `step` on later ones. If omitted, `test` defaults to `true`, giving a loop that will run until stopped by some other means such as `break`, `return`, or `throw`.

`step` is performed after the loop body, between one iteration and the next. It traditionally updates the loop variables from one iteration to the next: *e.g.*, `i++` and `i++,j--`. If omitted, `step` does nothing.

`body` is a statement, often a code block, which is performed whenever `test` is true. If omitted, `body` does nothing.

An enhanced for statement is used to iterate over a collection, or other structure designed to support iteration by implementing the interface `Iterable[T]`. The loop variable must be of type `T`, or destructurable from a value of type `T` (§5). Each iteration of the loop binds the iteration variable to another element of the collection. The loop `for(x in c)S` behaves like:

```
val iterator: Iterator[T] = c.iterator();
while (iterator.hasNext()) {
    val x : T = iterator.next();
    S();
}
```

A number of library classes implement `Iterable`, and thus can be iterated over. For example, iterating over a `Rail` iterates the elements stored in the rail.

The type of the loop variable may be supplied as `x <: T`. In this case the iterable `c` must have type `Iterable[U]` for some `U <: T`, and `x` will be given the type `U`.

Example: *This loop adds up the elements of a `List[Long]`. Note that iterating over a list yields the elements of the list, as specified in the List API.*

```
static def sum(a:x10.util.List[Long]):Long {
    var s : Long = 0;
    for(x in a) s += x;
    return s;
}
```

The following code sums the elements of an integer rail.

```
static def sum(a: Rail[Long]): Long {
    var s : Long = 0;
    for(v in a) s += v;
    return s;
}
```

Iteration over a LongRange is quite common. This allows looping while varying a long index:

```
var sum : Long = 0;
for(i in 1..10) sum += i;
assert sum == 55;
```

Iteration variables have the `for` statement as scope. They shadow other variables of the same names.

12.13 Return statement

ReturnStmt ::= return Exp[?] ; (20.146)

Methods and closures may return values using a `return` statement. `void` methods must return without a value; other methods must return a value of the return type.

Example: *The following code illustrates returning values from a closure and a method. The `return` inside of closure returns from closure, not from method.*

```
def method(x:Long) {
    val closure = (y:Long) => {return x+y;};
    val res = closure(0);
    assert res == x;
    return res == x;
}
```

12.14 Assert statement

AssertStmt ::= assert Exp ; (20.10)
| assert Exp : Exp ;

The statement `assert E` checks that the Boolean expression `E` evaluates to true, and, if not, throws an `x10.lang.Error` exception. The annotated assertion statement `assert E : F`; checks `E`, and, if it is false, throws an `x10.lang.Error` exception with `F`'s value attached to it.

Example: *The following code compiles properly.*

```
class Example {
    public static def main(argv: Rail[String]) {
        val a = 1;
        assert a != 1 : "Changed my mind about a.";
    }
}
```

However, when run, it prints a stack trace starting with

```
x10.lang.Error: Changed my mind about a.
```

12.15 Exceptions in X10

X10 programs can throw *exceptions* to indicate unusual or problematic situations; this is *abrupt termination*. Exceptions, as data values, are instances of `x10.lang.CheckedThrowable` or its subclasses. Note that for ease of implementation X10 does not permit subclasses of `x10.lang.CheckedThrowable` to be generic, that is, take type parameters.

Exceptions may be thrown intentionally with the `throw` statement. Many primitives and library functions throw exceptions if they encounter problems; e.g., dividing by zero throws an instance of `x10.lang.ArithException`.

When an exception is thrown, dynamically enclosing `try-catch` blocks in the same activity can attempt to handle it. If the throwing statement is inside some `try` clause, and some matching `catch` clause catches that type of exception, the corresponding `catch` body will be executed, and the process of throwing is finished. If no statically-enclosing `try-catch` block can handle the exception, the current method call returns (abnormally), throwing the same exception from the point at which the method was called.

This process continues until the exception is handled or there are no more calling methods in the activity. In the latter case, the activity will terminate abnormally, and the exception will propagate to the activity's root; see §14.1 for details.

X10 supports both *checked* and *unchecked* exceptions. Methods are obligated to declare via a `throws` clause any checked exceptions that they might throw. However, in X10, the class library design favors unchecked exceptions: virtually all exceptions in the standard library are unchecked. Checked exceptions are defined to be any subclass of `x10.lang.CheckedThrowable` that are not also subclasses of either `x10.lang.Exception` or `x10.lang.Error`. All of the concrete exception classes in the X10 standard library are subclasses of either `Exception` or `Error`.

12.16 Throw statement

`ThrowStmt ::= throw Exp ;`

(20.163)

`throw E` throws an exception whose value is `E`, which must be an instance of a subtype of `x10.lang.CheckedThrowable`.

Example: *The following code checks if an index is in range and throws an exception if not.*

```
if (i < 0 || i >= x.size)
    throw new MyIndexOutOfBoundsException();
```

12.17 Try–catch statement

<i>TryStmt</i>	<code>::= try Block Catches</code>	(20.164)
	<code> try Block Catches? Finally</code>	
<i>Catches</i>	<code>::= CatchClause</code>	(20.32)
	<code> Catches CatchClause</code>	
<i>CatchClause</i>	<code>::= catch (Formal) Block</code>	(20.31)
<i>Finally</i>	<code>::= finally Block</code>	(20.71)

Exceptions are handled with a `try` statement. A `try` statement consists of a `try` block, zero or more `catch` blocks, and an optional `finally` block.

First, the `try` block is evaluated. If the block throws an exception, control transfers to the first matching `catch` block, if any. A `catch` matches if the value of the exception thrown is a subclass of the `catch` block's formal parameter type.

The `finally` block, if present, is evaluated on all normal and exceptional control-flow paths from the `try` block. If the `try` block completes normally or via a `return`, a `break`, or a `continue` statement, the `finally` block is evaluated, and then control resumes at the statement following the `try` statement, at the branch target, or at the caller as appropriate. If the `try` block completes exceptionally, the `finally` block is evaluated after the matching `catch` block, if any, and when and if the `finally` block finishes normally, the exception is rethrown.

The parameter of a `catch` block has the block as scope. It shadows other variables of the same name.

Example: *The example() method below executes without any assertion errors*

```
class Example {
    class ThisExn extends Exception {}
    class ThatExn extends Exception {}
    var didFinally : Boolean = false;
    def example(b:Boolean) {
        try {
            throw b ? new ThatExn() : new ThisExn();
        }
        catch(ThatExn) {return true;}
```

```

        catch(ThisExn) {return false;}
        finally {
            this.didFinally = true;
        }
    }
    static def doExample() {
        val e = new Example();
        assert e.example(true);
        assert e.didFinally == true;
    }
}

```

Limitation: Constraints on exception types in `catch` blocks are not currently supported.

12.18 Assert

The `assert` statement `assert B;` checks that the Boolean expression `B` evaluates to true. If so, computation proceeds. If not, it throws `x10.lang.AssertionError`.

The extended form `assert B:A;` is similar, but provides more debugging information. The value of the expression `A` is available as part of the `AssertionError`, e.g., to be printed on the console.

Example: *assert is useful for confirming properties that you believe to be true and wish to rely on. In particular, well-chosen asserts make a program robust in the face of code changes and unexpected uses of methods. For example, the following method compute percent differences, but asserts that it is not dividing by zero. If the mean is zero, it throws an exception, including the values of the numbers as potentially useful debugging information.*

```

static def percentDiff(x:Double, y:Double) {
    val diff = x-y;
    val mean = (x+y)/2;
    assert mean != 0.0 : [x,y];
    return Math.abs(100 * (diff / mean));
}

```

At times it may be considered important not to check `assert` statements; e.g., if the test is expensive and the code is sufficiently well-tested. The `-noassert` command line option causes the compiler to ignore all `assert` statements.

13 Places

An X10 place is a repository for data and activities, corresponding loosely to a process or a processor. Places induce a concept of “local”. The activities running in a place may access data items located at that place with the efficiency of on-chip access. Accesses to remote places may take orders of magnitude longer. X10’s system of places is designed to make this obvious. Programmers are aware of the places of their data, and know when they are incurring communication costs, but the actual operation to do so is easy. It’s not hard to use non-local data; it’s simply hard to do so accidentally.

The set of places available to a computation is determined at the time that the program is started, and remains fixed through the run of the program. See the README documentation on how to set command line and configuration options to set the number of places.

Places are first-class values in X10, as instances `x10.lang.Place`. Place provides a number of useful ways to query places, such as `Place.places()`, which returns a `PlaceGroup` of the places available to the current run of the program.

Objects and structs are created in a single place – the place that the constructor call was running in. They cannot change places. They can be *copied* to other places, and the special library struct `GlobalRef` allows values at one place to point to values at another.

13.1 The Structure of Places

Places are numbered starting at 0; the number is stored in the field `p1.id`. The method `Place.numPlaces()` returns the number of Places in the current execution of the program. The program starts by executing a `main` method at `Place.FIRST_PLACE`, which is `Place.places()()`; see §14.4.

13.2 here

The variable `here` is always bound to the place at which the current computation is running, in the same way that `this` is always bound to the instance of the current

class (for non-static code), or `self` is bound to the instance of the type currently being constrained. `here` may denote different places in the same method body or even the same expression, due to place-shifting operations.

This is not unusual for automatic variables: `self` denotes two different values (one `List`, one `Long`) when one describes a non-null list of non-zero numbers as `List[Long{self!=0}]{self!=null}`. In the following code, `here` has one value at `h0`, and a different one at `h1` (unless there is only one place).

```
val h0 = here;
val world = Place.places();
at (world.next(here)) {
    val h1 = here;
    assert (h0 != h1);
}
```

(Similar examples show that `self` and `this` have the same behavior: `self` can be shadowed by constrained types appearing inside of type constraints, and `this` by inner classes.)

The following example looks through a list of references to `Things`. It finds those references to things that are `here`, and deals with them.

```
public static def deal(things: List[GlobalRef[Thing]]) {
    for(gr in things) {
        if (gr.home == here) {
            val grHere =
                gr as GlobalRef[Thing]{gr.home == here};
            val thing <: Thing = grHere();
            dealWith(thing);
        }
    }
}
```

13.3 at: Place Changing

An activity may change place synchronously using the `at` statement or `at` expression. Like any distributed operation, it is potentially expensive, as it requires, at a minimum, two messages and the copying of all data used in the operation, and must be used with care – but it provides the basis for distributed programming in X10.

AtStmt ::= `at (Exp) Stmt` (20.20)

AtExp ::= `at (Exp) ClosureBody` (20.19)

The *PlaceExp* must be an expression of type `Place` or some subtype. For programming convenience, if *PlaceExp* is of type `GlobalRef[T]` then the `home` property of `GlobalRef` is used as the value of *PlaceExp*.

An activity may also spawn an asynchronous remote child activity. For optimal performance, it is desirable for the spawning activity to continue executing locally without waiting for the message creating the remote child activity to arrive at the destination place. X10 supports this “fire-and-forget” style of remote activity creation by special handling of the combination of `at (P) async S`. In particular, any exceptions raised during deserialization (§13.3.2) at the remote place will be reported asynchronously (as if they occurred after the remote activity `async S` was spawned).

Example: *The following example creates a rail `a` located here, and copies it to another place. `a` in the second place refers to the copy. The copy is modified and examined. After the `at` finishes, the original is also examined, and (since only the copy, not the original, was modified) is observed to be unchanged.*

```
val a = [1,2,3];
val world = Place.places();
at(world.next(here)) {
    a(1) = 4;
    assert a(0)==1 && a(1)==4 && a(2)==3;
}
assert a(0)==1 && a(1)==2 && a(2)==3;
```

13.3.1 Copying Values

An activity executing `at(q)S` at a place `p` evaluates `q` at place `p`, which should be a `Place`. It then moves to place `q` to execute `S`. The values variables that `S` refers to are copied (§13.3.2) to `q`, and bound to the variables of the same name. If the `at` is inside of an instance method and `S` uses `this`, `this` is copied as well. Note that a field reference `this.fld` or a method call `this.meth()` will cause `this` to be copied — as will their abbreviated forms `fld` and `meth()`, despite the lack of a visible `this`.

Note that the value obtained by evaluating `q` is not necessarily distinct from `p` (*e.g.*, `q` may be `here`). This does not alter the behavior of `at`. `at(here)S` will copy all the values mentioned in `S`, even though there is no actual change of place, and even though the original values already exist there.

On normal termination of `S` control returns to `p` and execution is continued with the statement following `at (q) S`. If `S` terminates abruptly with exception `E`, `E` is serialized into a buffer, the buffer is communicated to `p` where it is deserialized into an exception `E1` and `at (p) S` throws `E1`.

Since `at(p) S` is a synchronous construct, usual control-flow constructs such as `break`, `continue`, `return` and `throw` are permitted in `S`. All concurrency related constructs – `async`, `finish`, `atomic`, `when` are also permitted.

The `at`-expression `at(p)E` is similar, except that, in the case of normal termination of `E`, the value that `E` produces is serialized into a buffer, transported to the starting place, and deserialized, and the value of the `at`-expression is the result of deserialization.

Limitation: X10 does not currently allow `break`, `continue`, or `return` to exit from an `at`.

13.3.2 How at Copies Values

The values mentioned in S are copied to place p by at(p)S as follows.

First, the original-expressions are evaluated to give a vector of X10 values. Consider the graph of all values reachable from these values (except for transient fields (§13.3.5, GlobalRefs (§13.3.6); also custom serialization (§13.3.2 may alter this behavior)).

Second this graph is *serialized* into a buffer and transmitted to place q. Third, the vector of X10 values is re-created at q by deserializing the buffer at q. Fourth, S is executed at q, in an environment in which each variable v declared in F refers to the corresponding deserialized value.

Note that since values accessed across an at boundary are copied, the programmer may wish to adopt the discipline that either variables accessed across an at boundary contain only structs or stateless objects, or the methods invoked on them do not access any mutable state on the objects. Otherwise the programmer has to ensure that side effects are made to the correct copy of the object. For this the struct `x10.lang.GlobalRef[T]` is often useful.

Serialization and deserialization.

The X10 runtime provides a default mechanism for serializing/deserializing an object graph with a given set of roots. This mechanism may be overridden by the programmer on a per class or struct basis as described in the API documentation for `x10.io.CustomSerialization`. The default mechanism performs a deep copy of the object graph (that is, it copies the object or struct and, recursively, the values contained in its fields), but does not traverse or copy transient fields. transient fields are omitted from the serialized data. On deserialization, transient fields are initialized with their default values (§4.7). The types of transient fields must therefore have default values.

The default serialization/deserialization mechanism will not (modulo error conditions like `OutOfMemoryError`) throw any exceptions. However, user code running during serialization/deserialization via `CustomSerialization` may raise exceptions. These exceptions are handled like any other exception raised during the execution of an X10 activity. However, due to the special treatment of at (p) async S (§13.3) any exception raised during deserialization will be handled as if it was raised by `async S`, not by the at statement itself.

A struct s of type `x10.lang.GlobalRef[T]` 13.3.6 is serialized as a unique global reference to its contained object o (of type T). Please see the documentation of `x10.lang.GlobalRef[T]` for more details.

13.3.3 at and Activities

`at(p)S` does *not* start a new activity. It should be thought of as transporting the current activity to p, running S there, and then transporting it back. `async` is the only construct

in the language that starts a new activity. In different contexts, each one of the following combination of `async` and `at(p)` makes sense: (1) `at(p) async S` and, (2) `async at(p) S`. In the first case, the expression `p` is evaluated synchronously by the current activity and then a single remote `async` is spawned. In the second case, `p` is semantically required to be evaluated asynchronously with the parent `async` as it is contained in the body of an `async`. Then the evaluation of `S` is transported to the new place. In most cases, the first form (`at(p) async S`) is preferred to second one (`async at(p) S`), since it enables a more efficient runtime implementation (it avoids the spawning a local `async` solely to evaluate `p`).

Since `at(p) S` does not start a new activity, `S` may contain constructs which only make sense within a single activity. For example,

```
for(x in globalRefsToThings)
    if (at(x.home) x().isNice())
        return x();
```

returns the first nice thing in a collection. If we had used `async at(x.home)`, this would not be allowed; you can't `return` from an `async`.

Limitation: X10 does not currently allow `break`, `continue`, or `return` to exit from an `at`.

13.3.4 Copying from at

`at(p)S` copies data required in `S`, and sends it to place `p`, before executing `S` there. The only things that are not copied are values only reachable through `GlobalRefs` and `transient` fields, and data omitted by custom serialization.

Example:

```
val c = new Cell[Long](9); // (1)
at (here) {                // (2)
    assert(c() == 9);       // (3)
    c.set(8);               // (4)
    assert(c() == 8);       // (5)
}
assert(c() == 9);           // (6)
```

The `at` statement copies the `Cell` and its contents. After (1), `c` is a `Cell` containing 9; call that cell c_1 . At (2), that cell is copied, resulting in another cell c_2 whose contents are also 9, as tested at (3). (Note that the copying behavior of `at` happens even when the destination place is the same as the starting place—even with `at(here)`.) At (4), the contents of c_2 are changed to 8, as confirmed at (5); the contents of c_1 are of course untouched. Finally, at (6), outside the scope of the `at` started at line (2), `c` refers to its original value c_1 rather than the copy c_2 .

The `at` statement induces a *deep copy*. Not only does it copy the values of variables, it copies values that they refer to through zero or more levels of reference. Structures are

preserved as well: if two fields `x.f` and `x.g` refer to the same object o_1 in the original, then `x.f` and `x.g` will both refer to the same object o_2 in the copy.

Example: In the following variation of the preceding example, `a`'s original value a_1 is a rail with two references to the same `Cell[Long]` c_1 . The fact that $a_1(0)$ and $a_1(1)$ are both identical to c_1 is demonstrated in (A)-(C), as $a_1(0)$ is modified and $a_1(1)$ is observed to change. In (D)-(F), the copy a_2 is tested in the same way, showing that $a_2(0)$ and $a_2(1)$ both refer to the same `Cell[Long]` c_2 . However, the test at (G) shows that c_2 is a different cell from c_1 , because changes to c_2 did not propagate to c_1 .

```
val c = new Cell[Long](5);
val a : Rail[Cell[Long]] = [c,c as Cell[Long]];
assert(a(0)() == 5 && a(1)() == 5);           // (A)
c.set(6);                                      // (B)
assert(a(0)() == 6 && a(1)() == 6);           // (C)
at(here) {
    assert(a(0)() == 6 && a(1)() == 6);       // (D)
    c.set(7);                                      // (E)
    assert(a(0)() == 7 && a(1)() == 7);       // (F)
}
assert(a(0)() == 6 && a(1)() == 6);           // (G)
```

13.3.5 Copying and Transient Fields

Recall that fields of classes and structs marked `transient` are not copied by `at`. Instead, they are set to the default values for their types. Types that do not have default values cannot be used in `transient` fields.

Example: Every `Trans` object has an `a`-field equal to 1. However, despite the initializer on the `b` field, it is not the case that every `Trans` has `b==2`. Since `b` is `transient`, when the `Trans` value `this` is copied at `at(here){...}` in `example()`, its `b` field is not copied, and the default value for an `Long`, 0, is used instead. Note that we could not make a `transient` field `c : Long{c != 0}`, since the type has no default value, and copying would in fact set it to zero.

```
class Trans {
    val a : Long = 1;
    transient val b : Long = 2;
    //ERROR: transient val c : Long{c != 0} = 3;
    def example() {
        assert(a == 1 && b == 2);
        at(here) {
            assert(a == 1 && b == 0);
        }
    }
}
```

13.3.6 Copying and GlobalRef

A `GlobalRef[T]` (say `g`) contains a reference to a value `v` of type `T`, in a form which can be transmitted, and a `Place` `g.home` indicating where the value lives. When a `GlobalRef` is serialized an opaque, globally unique handle to `v` is created.

Example: *The following example does not copy the value `huge`. However, `huge` would have been copied if it had been put into a `Cell`, or simply used directly.*

```
val huge = "A potentially big thing";
val href = GlobalRef(huge);
at (here) {
    use(href);
}
}
```

Values protected in `GlobalRefs` can be retrieved by the application operation `g()`. `g()` is guarded; it can only be called when `g.home == here`. If you want to do anything other than pass a global reference around or compare two of them for equality, you need to placeshift back to the home place of the reference, often with `at(g.home)`.

Example: *The following program, for reasons best known to the programmer, modifies the command-line argument array.*

```
public static def main(argv: Rail[String]) {
    val argref = GlobalRef[Rail[String]](argv);
    val world = Place.places();
    at(world.next(here))
        use(argref);
}
static def use(argref : GlobalRef[Rail[String]]) {
    at(argref) {
        val argv = argref();
        argv(0) = "Hi!";
    }
}
```

There is an implicit coercion from `GlobalRef[T]` to `Place`, so `at(argref)` goes to `argref.home`.

13.3.7 Warnings about at

There are two dangers involved with `at`:

- Careless use of `at` can result in copying and transmission of very large data structures. In particular, it is very easy to capture `this` – a field reference will do it – and accidentally copy everything that `this` refers to, which can be very large. A disciplined use of copy specifiers to make explicit just what gets copied can ameliorate this issue.

- As seen in the examples above, a local variable reference `x` may refer to different objects in different nested `at` scopes. The programmer must either ensure that a variable accessed across an `at` boundary has no mutable state or be prepared to reason about which copy gets modified. A disciplined use of copy specifiers to give different names to variables can ameliorate this concern.

14 Activities

An *activity* is a statement being executed, independently, with its own local variables; it may be thought of as a very light-weight thread. An X10 computation may have many concurrent activities executing at any give time. All X10 code runs as part of an activity; when an X10 program is started, the `main` method is invoked in an activity, called the *root activity*.

Activities progress by executing control structures. For example, `when(x==0);` blocks the current activity until some other activity sets `x` to zero. However, activities determine the loca at which they may be blocked and resumed, using `when` and similar constructs. There are no means by which one activity can arbitrarily interrupt, block, kill or resume another.

An activity may be *running*, *blocked* on some condition or *terminated*. An activity terminates when it has no more statements to execute; it terminates *normally (abruptly)* if the last statement it executes terminates normally (abruptly) (§14.1).

Activities can be long-running, and may access a lot of data. In particular they can call recursive methods (and therefore have runtime stacks). However, activities can also perform very few actions, such as incrementing some variables.

An activity may asynchronously and in parallel launch activities. Every activity except the initial `main` activity is spawned by another. Thus, at any instant, the activities in a program form a tree.

X10 uses this tree in crucial ways. First is the distinction between *local* termination and *global* termination of a statement. The execution of a statement by an activity is said to terminate *locally* when the activity has finished all its computation. (For instance the creation of an asynchronous activity terminates locally when the activity has been created.) It is said to terminate *globally* when it has terminated locally and all activities that it may have spawned have, recursively, terminated globally. For example, consider:

```
async {s1();}  
async {s2();}
```

The primary activity spawns two child activities and then terminates locally, very quickly. The child activities may take arbitrary amounts of time to terminate (and may spawn grandchildren). When `s1()`, `s2()`, and all their descendants terminate locally, then the primary activity terminates globally.

The program as a whole terminates when the root activity terminates globally. In particular, X10 does not permit the creation of daemon threads—threads that outlive the lifetime of the root activity. We say that an X10 computation is *rooted* (§14.4).

Future Extensions. *We may permit the initial activity to be a daemon activity to permit reactive computations, such as webservers, that may not terminate.*

14.1 The X10 rooted exception model

The rooted nature of X10 computations permits the definition of a *rooted exception model*. In multi-threaded programming languages there is a natural parent-child relationship between a thread and a thread that it spawns. Typically the parent thread continues execution in parallel with the child thread. Therefore the parent thread cannot serve to catch any exceptions thrown by the child thread.

The presence of a root activity and the concept of global termination permits X10 to adopt a more powerful exception model. In any state of the computation, say that an activity A is a *root of* an activity B if A is an ancestor of B and A is blocked at a statement (such as the `finish` statement §14.3) awaiting the termination of B (and possibly other activities). For every X10 computation, the *root-of* relation is guaranteed to be a tree. The root of the tree is the root activity of the entire computation. If A is the nearest root of B , the path from A to B is called the *activation path* for the activity.¹

We may now state the exception model for X10. An uncaught exception propagates up the activation path to its nearest root activity, where it may be handled locally or propagated up the *root-of* tree when the activity terminates (based on the semantics of the statement being executed by the activity).² There is always a good place to put a `try-catch` block to catch exceptions thrown by an asynchronous activity.

14.2 `async`: Spawning an activity

Asynchronous activities serve as a single abstraction for supporting a wide range of concurrency constructs such as message passing, threads, DMA, streaming, and data prefetching. (In general, asynchronous operations are better suited for supporting scalability than synchronous operations.)

An activity is created by executing the `async` statement:

AsyncStmt ::= `async ClockedClause?` *Stmt* (20.15)

| `clocked async Stmt`

ClockedClause ::= `clocked Arguments` (20.39)

¹Note that depending on the state of the computation the activation path may traverse activities that are running, blocked or terminated.

²In X10 v2.4 the `finish` statement is the only statement that marks its activity as a root activity. Future versions of the language may introduce more such statements.

The basic form of `async` is `async S`, which starts a new activity located here executing `S`. (For the clocked form, see §15.4.)

Multiple activities launched by a single activity in a place are not ordered in any way. They are added to the set of activities running in the place and will be executed based on the local scheduler's decisions. If some particular sequencing of events is needed, `when`, `atomic`, `finish`, clocks, and other X10 constructs can be used. X10 implementations are not required to have fair schedulers, though every implementation should make a best faith effort to ensure that every activity eventually gets a chance to make forward progress.

The statement in the body of an `async` is subject to the restriction that it must be acceptable as the body of a `void` method for an anonymous inner class declared at that point in the code. For example, it may reference `val` variables in lexically enclosing scopes, but not `var` variables. Similarly, it cannot `break` or `continue` surrounding loops.

14.3 Finish

The statement `finish S` converts global termination to local termination.

<i>FinishStmt</i>	<code>::= finish Stmt</code>	(20.72)
	<code> clocked finish Stmt</code>	

An activity `A` executes `finish S` by executing `S` and then waiting for all activities spawned by `S` (directly or indirectly, here or at other places) to terminate. An activity may terminate normally, or abruptly, i.e. by throwing an exception. All exceptions thrown by spawned activities are caught and accumulated.

`finish S` terminates locally when all activities spawned by `S` terminate globally (either abruptly or normally). If `S` terminates normally, then `finish S` terminates normally and `A` continues execution with the next statement after `finish S`. If `S` or one of the activities spawned by it terminate abruptly, then `finish S` terminates abruptly and throws a single exception, of type `x10.lang.MultipleExceptions`, formed from the collection of exceptions accumulated at `finish S`.

Thus `finish S` statement serves as a collection point for uncaught exceptions generated during the execution of `S`.

Note that repeatedly finishing a statement has little effect after the first `finish`: `finish finish S` is indistinguishable from `finish S` if `S` terminates normally. If `S` throws exceptions, `finish S` collects the exceptions and wraps them in a `MultipleExceptions`, whereas `finish finish S` does the same, and then puts that `MultipleExceptions` inside of a second `MultipleExceptions`.

14.4 Initial activity

An X10 computation is initiated from the command line on the presentation of a class or struct name C. The container must have a `main` method:

```
public static def main(a: Rail[String]):void
```

method, otherwise an exception is thrown and the computation terminates. The single statement

```
finish async at (Place.FIRST_PLACE) {
    C.main(s);
}
```

is executed where `s` is a one-dimensional `Rail` of strings created from the command line arguments. This single activity is the root activity for the entire computation. (See §13 for a discussion of places.)

14.5 Ateach statements

Deprecated: The `ateach` construct is deprecated.

AtEachStmt ::= `ateach (LoopIndex in Exp) ClockedClause?` Stmt (20.18)

| `ateach (Exp) Stmt`

LoopIndexDeclr ::= *Id HasResultType?* (20.113)

| `[IdList] HasResultType?`

| `Id [IdList] HasResultType?`

LoopIndex ::= *Mods? LoopIndexDeclr* (20.112)

| `Mods? VarKeyword LoopIndexDeclr`

In `ateach(p in D) S`, `D` must be either of type `Dist` (see §16.4.3) or of type `DistArray[T]` (see §16), and `p` will be of type `Point` (see §16.3.1). If `D` is an `DistArray[T]`, then `ateach (p in D)S` is identical to `ateach(p in D.dist)S`; the iteration is over the array's underlying distribution.

Instead of writing `ateach (p in D) S` the programmer could write

```
for (place in D.places()) at (place) async {
    for (p in D|here) async {
        S(p);
    }
}
```

For each point `p` in `D`, statement `S` is executed concurrently at place `D(p)`.

`break` and `continue` statements may not be applied to `ateach`.

14.6 vars and Activities

X10 restricts the use of local var variables in activities, to make programs more deterministic. Specifically, a local var variable `x` defined outside of `async S` cannot appear inside `async S` unless there is a `finish` surrounding `async S` with the definition of `x` outside of it.

Example: *The following code is fine; the definition of `result` appears outside of the `finish` block:*

```
var result : Long = 0;
finish {
    async result = 1;
}
assert result == 1;
```

This code is deterministic: the `async` will finish before the `assert` starts, and the `assert`'s test will be true.

However, without the `finish`, the code would not compile in X10. If it were allowed to compile, the activity might finish or might not finish before the `println`, and the program would not be deterministic.

14.7 Atomic blocks

X10's atomic blocks (`atomic S` and `when (c) S`) provide a high-level construct for coordinating the mutation of shared data. An atomic block is executing as if in a single step, with respect to atomic blocks executed by all other activities in the same place. That is, all `atomic` blocks execute in a *serializable* order. Hence no `atomic` block can see the intermediate state within the execution of some other `atomic` block.

Code executed inside of `atomic S` and `when(E)S` is subject to certain restrictions. A violation of these restrictions causes an `IllegalOperationException` to be thrown at the point of the violation.

- `S` may not spawn another activity.
- `S` may not use any blocking statements; `when`, `Clock.advanceAll()`, `finish`. (The use of a nested `atomic` is permitted.)
- `S` may not `force()` a `Future`.
- `S` may not use `at` expressions.

That is `S` must be sequential, single-place and non-blocking.

A programmer may use atomic blocks to guarantee that invariants of shared data-structures are maintained even as they are being accessed simultaneously by multiple

activities running in the same place. An X10 program in which all accesses (both reads and writes) of shared variables appear in `atomic` or `when` blocks is guaranteed to use all shared variables atomically. Equivalently, if two accesses to some shared variable `v` could collide at runtime, and one is in an atomic block, then the other must be in an atomic block as well to guarantee atomicity of the accesses to `v`. If some accesses to shared variables are not protected by `atomic` or `when`, then race conditions or deadlocks may occur. In particular, atomic blocks at the same place are atomic with respect to each other. They may not be atomic with respect to non-atomic code, or with respect to atomic sections at different places.

AtomicStmt ::= atomic Stmt (20.21)
WhenStmt ::= when (Exp) Stmt (20.207)

Example: Consider a class `Redund[T]`, which encapsulates a list `list` and, (redundantly) keeps the size of the list in a second field `size`. Then `r:Redund[T]` has the invariant `r.list.size() == r.size`, which must be true at any point at which no method calls on `r` are active.

If the `add` method on `Redund` (which adds an element to the list) were defined as:

```
def add(x:T) { // Incorrect
    this.list.add(x);
    this.size = this.size + 1;
}
```

Then two activities simultaneously adding elements to the same `r` could break the invariant. Suppose that `r` starts out empty. Let the first activity perform the `list.add`, and compute `this.size+1`, which is 1, but not store it back into `this.size` yet. (At this point, `r.list.size()==1` and `r.size==0`; the invariant expression is false, but, as the first call to `r.add()` is active, the invariant does not need to be true – it only needs to be true when the call finishes.) Now, let the second activity do its call to add to completion, which finishes with `r.size==1`. (As before, the invariant expression is false, but a call to `r.add()` is still active, so the invariant need not be true.) Finally, let the first activity finish, which assigns the 1 computed before back into `this.size`. At the end, there are two elements in `r.list`, but `r.size==1`. Since there are no calls to `r.add()` active, the invariant is required to be true, but it is not.

In this case, the invariant can be maintained by making the increment atomic. Doing so forbids that sequence of events; the `atomic` block cannot be stopped partway.

```
def add(x:T) {
    atomic {
        this.list.add(x);
        this.size = this.size + 1;
    }
}
```

14.7.1 Unconditional atomic blocks

The simplest form of an atomic block is the *unconditional atomic block*: `atomic S.` It executes S as if in a single step with respect to all other concurrently executing atomic blocks. S may throw an exception; when control leaves `atomic S` the side-effects executed so far are made visible to other atomic blocks. The programmer may surround S with a `try/finally` block and try to undo assignments when an exception is thrown.

Atomic blocks are closely related to non-blocking synchronization constructs [7], and can be used to implement non-blocking concurrent algorithms.

Note an important property of an (unconditional) atomic block:

$$\text{atomic } \{s1; \text{atomic } s2\} = \text{atomic } \{s1; s2\} \quad (14.1)$$

Unconditional atomic blocks do not introduce deadlocks. They may exhibit all the bad behavior of sequential programs, including throwing exceptions and running forever, but they are guaranteed not to deadlock.

Example: *The following class method implements a (generic) compare and swap (CAS) operation:*

```
var target:Any = null;
public atomic def CAS(old1: Any, y: Any):Boolean {
    if (target.equals(old1)) {
        target = y;
        return true;
    }
    return false;
}
```

14.7.2 Conditional atomic blocks

Conditional atomic blocks are of the form `when(b)S;` b is called the *guard*, and S the *body*.

An activity executing such a statement suspends until such time as the guard is true in the current state. In that state, the body is executed. The checking of the guards and the execution of the corresponding guarded statement is done atomically.

X10 does not guarantee that a conditional atomic block will execute if its condition holds only intermittently. For, based on the vagaries of the scheduler, the precise instant at which a condition holds may be missed. Therefore the programmer is advised to ensure that conditions being tested by conditional atomic blocks are eventually stable, *i.e.*, they will continue to hold until the block executes (the action in the body of the block may cause the condition to not hold any more).

The statement `when (true) S` is behaviorally identical to `atomic S`: it never suspends.

The body `S` of `when(b)S` is subject to the same restrictions that the body of `atomic S` is. The guard is subject to the same restrictions as well. Furthermore, guards should not have side effects. Note that this implies that guarded statements are required to be *flat*, that is, they may not contain conditional atomic blocks. (The implementation of nested conditional atomic blocks may require sophisticated operational techniques such as rollbacks.)

Conditional atomic blocks allow the activity to wait for some condition to be satisfied before executing an atomic block. For example, consider a `Redund` class holding a list `r.list` and, redundantly, its length `r.size`. A `pop` operation will delay until the `Redund` is nonempty, and then remove an element and update the length.

```
def pop():T {
    var ret : T;
    when(size>0) {
        ret = list.removeAt(0);
        size--;
    }
    return ret;
}
```

The execution of the test is atomic with the execution of the block. This is important; it means that no other activity can sneak in and falsify the condition after the test was seen to be true, but before the block is executed. In this example, two pops executing on a list with one element would work properly. Without the conditional atomic block – even doing the decrement atomically – one call to `pop` could pass the `size>0` guard; then the other call could run to completion (removing the only element of the list); then, when the first call proceeds, its `removeAt` will fail.

Note that `if` would not work here.

```
if(size>0) atomic{size--; return list.removeAt(0);}
```

allows another activity to act between the test and the atomic block. And

```
atomic{ if(size>0) {size--; ret = list.removeAt(0);}}
```

does not wait for `size>0` to become true.

Example: *The following class shows how to implement a bounded buffer of size 1 in X10 for repeated communication between a sender and a receiver. The call `buf.send(ob)` waits until the buffer has space, and then puts `ob` into it. Dually, `buf.receive()` waits until the buffer has something in it, and then returns that thing.*

```
class OneBuffer[T] {
    var datum: T;
    def this(t:T) { this.datum = t; this.filled = true; }
    var filled: Boolean;
```

```

public def send(v: T) {
    when (!filled) {
        this.datum = v;
        this.filled = true;
    }
}
public def receive(): T {
    when (filled) {
        v: T = datum;
        filled = false;
        return v;
    }
}
}

```

When when is Tested

Suppose that activity *A* is blocked waiting on `when(e) S`, because *e* is `false`. If some other activity *B* changes the state in an `atomic` section in a way that makes *e* become `true`, then either:

- *A* will eventually execute *S*, or
- Some activity *C* $\neq A$ will cause *e* to become `false` again, or
- Some activity at that place will execute an infinite number of steps.

In particular, if no other activity ever falsifies *e*, then *A* will, eventually, discover that *e* evaluates to `true` and run *S* (provided that no other activity at that place is running forever).

Two caveats are worth noting:

- X10 has no guarantees of fairness or liveness.
- X10 only makes guarantees about state changes *in atomic blocks* alerting `whens`. State changes outside of atomic blocks might not cause *A* to re-evaluate *e*.

Example: *The method good below will always terminate (assuming no other activities are executing an infinite number of steps). In particular, if the when statement is allowed to run first and block on c(), the atomic will alert it that c() has changed.*

The method bad has no such guarantee: it might terminate if the compiler and scheduler are in a generous mood, or the when might wait forever to be told that c() is now true. Without an atomic, the when statement might not be notified about the change in c().

```

static def good() {
    val c = new Cell[Boolean](false);
    async {
        atomic {c() = true;}
    }
    when( c() );
}
static def bad() {
    val c = new Cell[Boolean](false);
    async {
        c() = true;
    }
    when( c() );
}

```

14.8 Use of Atomic Blocks

The semantics of atomicity is chosen as a compromise between programming simplicity and efficient implementation. Unlike some possible definitions of “atomic”, atomic blocks do not provide absolute atomicity.

Atomic blocks are atomic with respect to *each other*.

```

var n : Long = 0;
finish {
    async atomic n = n + 1; // (a)
    async atomic n = n + 2; // (b)
}

```

This program has only two possible interleavings: either (a) entirely precedes (b) or (b) entirely precedes (a). Both end up with $n=3$.

However, atomic blocks are not atomic with respect to non-atomic code. If we remove the `atomics` on (a), we get far messier semantics.

```

var n : Long = 0;
finish {
    // LEGAL BUT UNWISE
    async n = n + 1;           // (a)
    async atomic n = n + 2;    // (b)
}

```

If X10 had absolute (“strong”) atomic semantics, this program would be guaranteed to treat the atomic increment as a single statement. This would permit three interleavings: the two possible from the fully atomic program, or a third one with the events: (a)’s read of 0 from n, the entirety of (b), and then (a)’s write of 0+1 back to n. This

interleaving results in $n==1$. So, with absolute atomic semantics, $n==1$ or $n==3$ are the possible results.

However, atomic blocks in X10 are “weak”. Atomic blocks are atomic with respect to each other — but there is no guarantee about how they interact with non-atomic statements at all. They might even break up the atomicity of an atomic block. In particular, the following fourth interleaving is possible: (a)’s read of \emptyset from n , (b)’s read of \emptyset from n , (a)’s write of 1 to n , and (b)’s write of 2 to n . Thus, $n==2$ is permissible as a result in X10.

However, X10’s semantics do impose a certain burden on the programmer. A sufficient rule of thumb is that, if *any* access to a variable is done in an atomic section, then *all* accesses to it must be in atomic sections.

Atomic sections are a powerful and convenient general solution. Classes in the package `x10.util.concurrent` may be more efficient and more convenient in particular cases. For example, an `AtomicInteger` provides an atomic integer cell, with atomic get, set, compare-and-set, and add operations. Each `AtomicInteger` takes care of its own locking. Accesses to one `AtomicInteger` a only block activities which try to access a — not others, not even if they are using different `AtomicIntegers` or even atomic blocks.

15 Clocks

Many concurrent algorithms proceed in phases: in phase k , several activities work independently, but synchronize together before proceeding on to phase $k + 1$. X10 supports this communication structure (and many variations on it) with a generalization of barriers called *clocks*. Clocks are designed to be dynamic (new activities can be registered with clocks, and terminated activities automatically deregister from clocks), and to support a simple syntactic discipline for deadlock-free and race-free conditions. Just like `finish`, X10's clocks can both be used within a single place and to synchronize activities across multiple places.

The following minimalist example of clocked code has two worker activities A and B, and three phases. In the first phase, each worker activity says its name followed by 1; in the second phase, by a 2, and in the third, by a 3. So, if `say` prints its argument, `A-1 B-1 A-2 B-2 B-3 A-3` would be a legitimate run of the program, but `A-1 A-2 B-1 B-2 A-3 B-3` (with A-2 before B-1) would not.

The program creates a clock `c1` to manage the phases. Each participating activity does the work of its first phase, and then executes `Clock.advanceAll()`; to signal that it is finished with that work. `Clock.advanceAll()`; is blocking, and causes the participant to wait until all participants have finished with the phase – as measured by the clock `c1` to which they are both registered. Then they do the second phase, and another `Clock.advanceAll()`; to make sure that neither proceeds to the third phase until both are ready. This example uses `finish` to wait for both participants to finish.

```
class ClockEx {
    static def say(s:String) {
        Console.OUT.println(s);
    }
    public static def main(argv:Rail[String]) {
        finish async{
            val c1 = Clock.make();
            async clocked(c1) { // Activity A
                say("A-1");
                Clock.advanceAll();
                say("A-2");
                Clock.advanceAll();
                say("A-3");
            }
        }
    }
}
```

```

} // Activity A

async clocked(c1) { // Activity B
    say("B-1");
    Clock.advanceAll();
    say("B-2");
    Clock.advanceAll();
    say("B-3");
} // Activity B
}
}
}

```

This chapter describes the syntax and semantics of clocks and statements in the language that have parameters of type `Clock`.

The key invariants associated with clocks are as follows. At any stage of the computation, a clock has zero or more *registered* activities. An activity may perform operations only on those clocks it is registered with (these clocks constitute its *clock set*). An attempt by an activity to operate on a clock it is not registered with will cause a `ClockUseException` to be thrown. An activity is registered with zero or more (existing) clocks when it is created. During its lifetime, the only additional clocks it can possibly be registered with are exactly those that it creates. In particular it is not possible for an activity to register itself with a clock it discovers by reading a data structure.

The primary operations that an activity `a` may perform on a clock `c` that it is registered upon are:

- It may spawn and simultaneously *register* a new activity on `c`, with the statement `async clocked(c)S`.
- It may *unregister* itself from `c`, with `c.drop()`. After doing so, it can no longer use most operations on `c`.
- It may *resume* the clock, with `c.resume()`, indicating that it has finished with the current phase associated with `c` and is ready to move on to the next one.
- It may *wait* on the clock, with `c.advance()`. This first does `c.resume()`, and then blocks the current activity until the start of the next phase, *viz.*, until all other activities registered on that clock have called `c.resume()`.
- It may *block* on all the clocks it is registered with simultaneously, by the command `Clock.advanceAll();`. This, in effect, calls `c.advance()` simultaneously on all clocks `c` that the current activity is registered with.
- Other miscellaneous operations are available as well; see the `Clock API`.

15.1 Clock operations

There are two language constructs for working with clocks. `async clocked(c1) S` starts a new activity registered on one or more clocks. `Clock.advanceAll()`; blocks the current activity until all the activities sharing clocks with it are ready to proceed to the next clock phase. Clocks are objects, and have a number of useful methods on them as well.

15.1.1 Creating new clocks

Clocks are created using a factory method on `x10.lang.Clock`:

```
val c: Clock = Clock.make();
```

The current activity is automatically registered with the newly created clock. It may deregister using the `drop` method on clocks (see the documentation of `x10.lang.Clock`). On (normal or abrupt) termination, an activity is automatically deregistered from all clocks it is registered with.

15.1.2 Registering new activities on clocks

`AsyncStmt ::= async ClockedClause? Stmt` (20.15)

`| clocked async Stmt`

`ClockedClause ::= clocked Arguments` (20.39)

The `async` statement with a `clocked` clause of either form, say

```
async clocked (c1, c2, c3) S
```

starts a new activity, initially registered with clocks `c1`, `c2`, and `c3`, and running `S`. The activity running this code must be registered on those clocks. Violations of these conditions are punished by the throwing of a `ClockUseException`.

If an activity `a` that has executed `c.resume()` then starts a new activity `b` also registered on `c` (*e.g.*, via `async clocked(c) S`), the new activity `b` starts out having also resumed `c`, as if it too had executed `c.resume()`. That is, `a` and `b` are in the same phase of the clock.

```
// ACTIVITY a
val c = Clock.make();
c.resume();
async clocked(c) {
    // ACTIVITY b
    c.advance();
    b_phase_two();
    // END OF ACTIVITY b
```

```

}
c.advance();
a_phase_two();
// END OF ACTIVITY a

```

In the proper execution, *a* and *b* both perform `c.advance()` and then their phase-2 actions. However, if *b* were not initially in the resume state for *c*, there would be a race condition; *b* could perform `c.advance()` and proceed to `b_phase_two` before *a* performed `c.advance()`.

An activity may check whether or not it is registered on a clock *c* by the method call `c.registered()`

NOTE: X10 does not contain a “register” operation that would allow an activity to discover a clock in a datastructure and register itself (or another process) on it. Therefore, while a clock *c* may be stored in a data structure by one activity *a* and read from it by another activity *b*, *b* cannot do much with *c* unless it is already registered with it. In particular, it cannot register itself on *c*, and, lacking that registration, cannot register a sub-activity on it with `async clocked(c) S`.

15.1.3 Resuming clocks

X10 permits *split phase* clocks. An activity may wish to indicate that it has completed whatever work it wishes to perform in the current phase of a clock *c* it is registered with, without suspending itself altogether. It may do so by executing `c.resume()`.

An activity may invoke `resume()` only on a clock it is registered with, and has not yet dropped (§15.1.5). A `ClockUseException` is thrown if this condition is violated. Nothing happens if the activity has already invoked a `resume` on this clock in the current phase.

An activity may invoke `Clock.resumeAll()` to resume all the clocks that it is registered with and has not yet dropped. This `resume()`s all these clocks in parallel, or, equivalently, sequentially in some arbitrary order.

15.1.4 Advancing clocks

An activity may execute the following method call to signal that it is done with the current phase.

```
Clock.advanceAll();
```

Execution of this call blocks until all the clocks that the activity is registered with (if any) have advanced. (The activity implicitly issues a `resume` on all clocks it is registered with before suspending.)

`Clock.advanceAll();` may be thought of as calling `c.advance()` in parallel for all clocks that the current activity is registered with. (The parallelism is conceptually

important: if activities *a* and *b* are both registered on clocks *c* and *d*, and *a* executes *c.advance()*; *d.advance()* while *b* executes *d.advance()*; *c.advance()*, then the two will deadlock. However, if the two clocks are waited on in parallel, as *Clock.advanceAll()*; does, *a* and *b* will not deadlock.)

Equivalently, *Clock.advanceAll()*; sequentially calls *c.resume()* for each registered clock *c*, in arbitrary order, and then *c.advance()* for each clock, again in arbitrary order.

An activity blocked on *advance()* resumes execution once it is marked for progress by all the clocks it is registered with.

15.1.5 Dropping clocks

An activity may drop a clock by executing *c.drop()*;

The activity is no longer considered registered with this clock. A *ClockUseException* is thrown if the activity has already dropped *c*.

15.2 Deadlock Freedom

In general, programs using clocks can deadlock, just as programs using loops can fail to terminate. However, programs written with a particular syntactic discipline *are* guaranteed to be deadlock-free, just as programs which use only bounded loops are guaranteed to terminate. The syntactic discipline is:

- The *advance()* instance method shall not be called on any clock. (The *Clock.advanceAll()*; method is allowed for this discipline.)
- Inside of *finish{S}*, all clocked *asyncs* shall be in the scope some unclocked *async* in *S*.

X10 does not enforce this discipline. Doing so would exclude useful programs, many of which are deadlock-free for reasons more subtle than the straightforward syntactic discipline. Still, this discipline is useful for simple cases.

The first clause of the discipline prevents a deadlock in which an activity is registered on two clocks, advances one of them, and ignores the other. The second clause prevents the following deadlock.

```
val c:Clock = Clock.make();
async clocked(c) { // (A)
    finish async clocked(c) { // (B) Violates clause 2
        Clock.advanceAll(); // (Bnext)
    }
    Clock.advanceAll(); // (Anext)
}
```

(A), first of all, waits for the `finish` containing (B) to finish. (B) will execute its `advance` at (`Bnext`), and then wait for all other activities registered on `c` to execute their `advance()`s. However, (A) is registered on `c`. So, (B) cannot finish until (A) has proceeded to (`Anext`), and (A) cannot proceed until (B) finishes. Thus, the activities are deadlocked.

15.3 Program equivalences

From the discussion above it should be clear that the following equivalences hold:

$$c.resume(); \text{Clock.advanceAll}(); = \text{Clock.advanceAll}(); \quad (15.1)$$

$$c.resume(); d.resume(); = d.resume(); c.resume(); \quad (15.2)$$

$$c.resume(); c.resume(); = c.resume(); \quad (15.3)$$

Note that `Clock.advanceAll(); Clock.advanceAll();` is not the same as `Clock.advanceAll();`. The first will wait for clocks to advance twice, and the second once.

15.4 Clocked Finish

In the most common case of a single clock coordinating a few behaviors, X10 allows coding with an implicit clock. `finish` and `async` statements may be qualified with `clocked`.

A `clocked finish` introduces a new clock. It executes its body in the usual way that a `finish` does— except that, when its body completes, the activity executing the `clocked finish` drops the clock, while it waits for asynchronous spawned `asyncs` to terminate.

A `clocked async` registers its `async` with the implicit clock of the surrounding `clocked finish`.

The bodies of the `clocked finish` and `clocked async` statements may use the `Clock.advanceAll()` method call to advance the implicit clock. Since the implicit clock is not available in a variable, it cannot be manipulated directly. (If you want to manipulate the clock directly, use an explicit clock.)

Example: *The following code starts two activities, each of which perform their first phase, wait for the other to finish phase 1, and then perform their second phase.*

```
clocked finish {
    clocked async {
        phase("A", 1);
        Clock.advanceAll();
```

```

        phase("A", 2);
    }
    clocked async {
        phase("B", 1);
        Clock.advanceAll();
        phase("B", 2);
    }
}

```

The `clocked async` and `clocked finish` constructs can be combined with `at` in the same ways as their unclocked counterparts.

Example: *The following code creates one clocked activity in every Place that synchronize to execute `iter` steps of a two phase computation. The clock ensures that every activity has completed the call to `before(N)` before any activity calls `after(N)`. Note that executions of `after(N)` and `before(N+1)` in different places may overlap; if this is not desired an additional call to `Clock.advanceAll()` could be added after the call to `after(count)`.*

```

clocked finish {
    for (p in Place.places()) {
        at (p) clocked async {
            for (count in 1..iters) {
                before(count);
                Clock.advanceAll();
                after(count);
            }
        }
    }
}

```

Clocked finishes may be nested. The inner `clocked finish` operates in a single phase of the outer one.

16 Rails and Arrays

16.1 Overview

Indexable memory is a fundamental abstraction provided by a programming language. To enable the programmer to best balance performance and flexibility, X10 provides a layered implementation of indexable memory. The foundation of all indexable storage in X10 is `x10.lang.Rail`, an intrinsic one-dimensional array analogous to the built-in arrays provided by languages such as C or Java. On top of `Rail`, more sophisticated array abstractions can be constructed completely as user-defined X10 classes. Two such families of user-defined array abstractions are included in the X10 core class libraries in the `x10.array` and `x10.regionarray` packages. Both families of arrays provide both local (single place) and distributed (multi-place) arrays.

The next section specifies `Rail`. Subsequent sections outline the `x10.array` and `x10.regionarray` packages. As discussed, in more detail below, `Rail` and the classes of `x10.array` provide significantly higher performance operations than the corresponding classes of `x10.regionarray`. Therefore we recommend that programmers only use the more general arrays of `x10.regionarray` where the increased flexibility justifies the reduced performance. We also encourage programmers to use the classes of `x10.array` as an example of how to define high-performance array abstractions in X10 and use them as templates for defining additional high-performance array abstractions (for example column-major arrays as in Fortran or 1-based arrays as in MATLAB).

16.2 Rails

The `Rail` class provides a basic abstraction of fixed-sized indexed storage. If `r` is a `Rail[T]`, then `r` contains `r.size` elements of type `T` that may be accessed using the Long values `0` to `r.size-1` as indices. All accesses to the elements of a `Rail` are checked: attempting to use an index that is less than `0` or greater than `r.size-1` will result in an `ArrayIndexOutOfBoundsException` being thrown.

As shown in the example below, instances of `Rail[T]` may be created using one of several constructors that initialize the data elements to the zero-value of `T`, a single initial value, or to a different initial value for each element of the `Rail`.

```
// A zero-initialized Rail of 10 doubles
val r1 = new Rail[Double](10);

// A Rail of 10 doubles, all initialized to pi
val r2 = new Rail[Double](10, Math.PI);

// A Rail of 10 doubles, r3(i) == i*pi
val r3 = new Rail[Double](10, (i:long)=>i*Math.PI);
```

As described in more detail in section 11.26, X10 also includes a shorthand form for Rail construction: simply put brackets around a list of expressions.

```
// A Rail[Long] containing the first 5 primes
val r1 = [2,3,5,7,11];

// A Rail[Double] such that r2(i) == i*pi
val r2 = [Math.PI, 2*Math.PI, 3*Math.PI, 4*Math.PI];
```

Accessing and updating single elements of a Rail is done using application syntax. For example, `a(i)=a(i+1)`; sets the `i`th element of `a` to the value of the `i+1` element of `a`. If `T` supports the `+` or `-` operation, then the usual pre/post increment/decrement operations are also available on individual array elements. For example, `a(i)++` is equivalent to `a(i) = a(i)+1`

Iteration over the elements of a Rail can be accomplished using several equivalent idioms. Furthermore, via some modest compiler support, each of these for loops will actually result in identical generated code. In the examples below, `r` is a Rail[Long] and `sum` is a local variable of type Long.

```
// A classic C-style for loop
for (var i:long=0; i<r.size; i++) {
    sum += r(i);
}

// Iterate over the LongRange 0..(r.size-1)
for (i in 0..(r.size-1)) {
    sum += r(i);
}

// Get the LongRange to iterate over from r
for (i in r.range()) {
    sum += r(i);
}

// Directly iterate over the values of r
for (v in r) {
    sum += v;
}
```

Basic bulk operations such as clearing (setting to zero), filling with a single value, and copying are provided by methods of the `Rail` class. Efficient copying of Rails across places is supported via the combination of the `x10.lang.GlobalRail` struct¹ and the `asyncCopy` method of `Rail`.

Additional complex bulk operations on `Rail` such as sorting, searching, map, and reduce are provided by the `x10.util.RailUtils` class.

When implementing higher-level data structures that use `Rail` as their backing storage, there may be significant performance advantage in performing unsafe operations on `Rail`. For example, when building a multi-dimensional `Array` class, the `Array` level bounds checking and initialization logic make the `Rail` level operations redundant. To support such scenarios the class `x10.lang.Unsafe` provides methods to allocate uninitialized `Rail` objects and to access the elements of a `Rail` without bounds checking. These unsafe extensions should be used judiciously, as improper use can result in memory safety violations that would not be possible in pure X10 code.

16.3 x10.array: Simple Arrays

Classes in the `x10.array` package provide high-performance implementations of both local and distributed multi-dimension arrays. The array implementations in this package are restricted to the case of rectangular, dense, zero-based index spaces. By making this restriction, the indexing calculations for both single-place and multi-place arrays can be optimized, resulting in an array implementation that should obtain equivalent performance to the corresponding array abstractions in C or Fortran. More general index spaces are supported by the classes in the `x10.regionarray` package (see Section 16.4).

The three main concepts of this package: iteration spaces, arrays, and distributed arrays are outlined below. All three concepts are implemented as simple class hierarchies with an abstract superclass and a collection of concrete final subclasses that contain the performance-critical operations. Client code using the abstractions can be written (with lower performance) using the abstract APIs provided by the superclass, but performance sensitive code should be written using the more specific type of the concrete subclasses. This enables compile time optimization and inlining of key operations, resulting in optimal code sequences.

16.3.1 Points

Both kinds of arrays are indexed by `Points`, which are n -dimensional tuples of integers. The `rank` property of a point gives its dimensionality. Points can be constructed from integers, or `Rail[Long]` by the `\xcdPoint.make`[‘] factory methods:

¹a specialized version of `x10.lang.GlobalRef` that includes the `size` of the referenced `Rail` to enable bounds checking

```
val origin_1 : Point{rank==1} = Point.make(0);
val origin_2 : Point{rank==2} = Point.make(0,0);
val origin_5 : Point = Point.make(new Rail[Long](5));
```

There is an implicit conversion from `Rail[Long]` to `Point`, giving a convenient syntax for constructing points:

```
val p : Point = [1,2,3];
val q : Point{rank==5} = [1,2,3,4,5];
val r : Point(3) = [11,22,33];
```

The coordinates of a point are available by function application, or, if you prefer, by subscripting; `p(i)` is the *i*th coordinate of the point `p`. `Point(n)` is a type-defined shorthand for `Point{rank==n}`.

16.3.2 IterationSpace

An `IterationSpace` is a generalization of `LongRange` to multiple dimensions. The `rank` property of the `IterationSpace` corresponds to its dimensionality. An `IterationSpace` represents an ordered collection of `Points` of the same `rank` as the `IterationSpace`. The primary purpose of an `IterationSpace` is to represent the indices of an `Array` or `DistArray`.

16.3.3 Array

The abstract `Array` class provides rank-generic operations for single place multi-dimensional arrays. Its concrete subclasses `Array_1`, `Array_2`, etc. provide rank-specific operations such as efficient element access. The APIs of the classes are designed to be a natural extension of the `Rail` API to multiple dimensions. In most usage scenarios, programmers should operate using the types of the concrete subclasses, not of `Array` itself.

The example below illustrates the construction and indexing operations of rank 2 arrays using a simple matrix multiply kernel where `N` is a `Long`.

```
val a = new Array_2[double](N, N, (i:long,j:long)=>(i*j) as double);
val b = new Array_2[double](N, N, (i:long,j:long)=>(i-j) as double);
val c = new Array_2[double](N, N, (i:long,j:long)=>(i+j) as double);

for (i in 0..(N-1))
    for (j in 0..(N-1))
        for (k in 0..(N-1))
            a(i,j) += b(i,k)*c(k,j);
```

Similarly to `Rail`, iteration over the elements of a `Array` can be accomplished using several equivalent idioms. Furthermore, via some modest compiler support, each of these for loops will actually result in identical generated code. In the examples below, `a` is a `Array_2[Long]` and `sum` is a local variable of type `Long`.

```
// A classic C-style for loop
for (var i:long=0; i<a.numElems_1; i++) {
    for (var j:long=0; j<a.numElems_2; j++) {
        sum += a(i,j);
    }
}

// Iterate over the indices of a using Point destructuring
for ([i,j] in a.indices()) {
    sum += a(i,j);
}

// Directly iterate over the values of a
for (v in a) {
    sum += v;
}
```

An additional idiom which iterates over the Points in the `IterationSpace` of a without destructuring the Points is also supported, but should be avoided in the current implementation of X10 as the compiler does not optimize away all of the overheads associated with explicit use of Point objects.

```
// Iterate over the indices of a using Points
for (p in a.indices()) {
    sum += a(p);
}
```

16.3.4 DistArray

The abstract class `DistArray` and its concrete subclasses represent an extension of the `Array` API to multiple places. In X10 version 2.4.0 `DistArrays` are still a work in progress. Only three concrete implementations are available: `DistArray_Unique` which has one data element per `Place`, `DistArray_Block_1` which block distributes a rank-1 array across the `Places` in its `PlaceGroup`, and `DistArray_BlockBlock_2` which distributes blocks of a rank-2 array across the `Places` in its `PlaceGroup`.

The API of `DistArray` is preliminary in this release of X10 we anticipate extending it with more operations and supporting a wider range of ranks and distributions in future release of X10.

16.4 x10.regionarray: Flexible Arrays

Classes in the `x10.regionarray` package provide the most general and flexible array abstraction that support mapping arbitrary multi-dimensional index spaces to data elements. Although they are significantly more flexible than `Rails` or the classes of the

`x10.array` package, this flexibility does carry with it an expectation of lower runtime performance.

Arrays provide indexed access to data at a single `Place`, *via* `Points`—indices of any dimensionality. `DistArrays` is similar, but spreads the data across multiple `Places`, *via* `Dists`.

16.4.1 Regions

A *region* is a set of points of the same rank. X10 provides a built-in class, `x10.regionarray.Region`, to allow the creation of new regions and to perform operations on regions. Each region `R` has a property `R.rank`, giving the dimensionality of all the points in it.

Example:

```
val MAX_HEIGHT=20;
val Null = Region.makeUnit(); //Empty 0-dimensional region
val R1 = Region.make(1, 100); // Region 1..100
val R2 = Region.make(1..100); // Region 1..100
val R3 = Region.make(0..99, -1..MAX_HEIGHT);
val R4 = Region.makeUpperTriangular(10);
val R5 = R4 && R3; // intersection of two regions
```

The `LongRange` value `1..100` can be used to construct the one-dimensional `Region` consisting of the points $\{[m], \dots, [n]\}$ `Region` by using the `Region.make` factory method. `LongRanges` are useful in building up regions, especially rectangular regions.

By a special dispensation, the compiler knows that, if `r : Region(m)` and `s : Region(n)`, then `r*s : Region(m+n)`. (The X10 type system ordinarily could not specify the sum; the best it could do would be `r*s : Region`, with the rank of the region unknown.) This feature allows more convenient use of arrays; in particular, one does not need to keep track of ranks nearly so much.

Various built-in regions are provided through factory methods on `Region`.

- `Region.makeEmpty(n)` returns an empty region of rank `n`.
- `Region.makeFull(n)` returns the region containing all points of rank `n`.
- `Region.makeUnit()` returns the region of rank 0 containing the unique point of rank 0. It is useful as the identity for Cartesian product of regions.
- `Region.makeHalfspace(normal, k)`, where `normal` is a `Point` and `k` an `Long`, returns the unbounded half-space of rank `normal.rank`, consisting of all points `p` satisfying the vector inequality $p \cdot \text{normal} \leq k$.
- `Region.makeRectangular(min, max)`, where `min` and `max` are rank-1 length-`n` integer arrays, returns a `Region(n)` equal to: $[min(0) \dots max(0), \dots, min(n-1) \dots max(n-1)]$.

- `Region.makeBanded(size, a, b)` constructs the banded `Region(2)` of size `size`, with `a` bands above and `b` bands below the diagonal.
- `Region.makeBanded(size)` constructs the banded `Region(2)` with just the main diagonal.
- `Region.makeUpperTriangular(N)` returns a region corresponding to the non-zero indices in an upper-triangular $N \times N$ matrix.
- `Region.makeLowerTriangular(N)` returns a region corresponding to the non-zero indices in a lower-triangular $N \times N$ matrix.
- If `R` is a region, and `p` a Point of the same rank, then `R+p` is `R` translated forwards by `p` – the region whose points are $r+p$ for each r in `R`.
- If `R` is a region, and `p` a Point of the same rank, then `R-p` is `R` translated backwards by `p` – the region whose points are $r-p$ for each r in `R`.

All the points in a region are ordered canonically by the lexicographic total order. Thus the points of the region $(1..2)^*(1..2)$ are ordered as

$(1,1), (1,2), (2,1), (2,2)$

Sequential iteration statements such as `for` (§12.12) iterate over the points in a region in the canonical order.

A region is said to be *rectangular* if it is of the form $(T_1 * \dots * T_k)$ for some set of intervals $T_i = l_i \dots h_i$. In particular an `LongRange` turned into a `Region` is rectangular: `Region.make(1..10)`. Such a region satisfies the property that if two points p_1 and p_3 are in the region, then so is every point p_2 between them (that is, it is *convex*). (Banded and triangular regions are not rectangular.) The operation `R.boundingBox()` gives the smallest rectangular region containing `R`.

Operations on regions

Let `R` be a region. A *sub-region* is a subset of `R`.

Let `R1` and `R2` be two regions whose types establish that they are of the same rank. Let `S` be another region; its rank is irrelevant.

`R1 && R2` is the intersection of `R1` and `R2`, *viz.*, the region containing all points which are in both `R1` and `R2`. For example, `Region.make(1..10) && Region.make(2..20)` is `2..10`.

`R1 * S` is the Cartesian product of `R1` and `S`, formed by pairing each point in `R1` with every point in `S`. Thus, `Region.make(1..2)*Region.make(3..4)*Region.make(5..6)` is the region of rank 3 containing the eight points with coordinates $[1, 3, 5], [1, 3, 6], [1, 4, 5], [1, 4, 6], [2, 3, 5], [2, 3, 6], [2, 4, 5], [2, 4, 6]$.

For a region `R` and point `p` of the same rank, `R+p` and `R-p` represent the translation of the region forward and backward by `p`. That is, `R+p` is the set of points $p+q$ for all `q` in `R`, and `R-p` is the set of `q-p`.

More `Region` methods are described in the API documentation.

16.4.2 Arrays

Arrays are organized data, arranged so that the data can be accessed with subscripts. An `Array[T]` `A` has a `Region A.region`, specifying which `Points` are in `A`. For each point `p` in `A.region`, `A(p)` is the datum of type `T` associated with `p`. X10 implementations should attempt to store `Arrays` efficiently, and to make array element accesses quick—*e.g.*, avoiding constructing `Points` when unnecessary.

This generalizes the concepts of arrays appearing in many other programming languages. A `Point` may have any number of coordinates, so an `Array` can have, in effect, any number of integer subscripts.

Example: *Indeed, it is possible to write code that works on `Arrays` regardless of dimension. For example, to add one `Array[Long]` `src` into another `dest`,*

```
static def addInto(src: Array[Long], dest: Array[Long])
  {src.region == dest.region}
  {
    for (p in src.region)
      dest(p) += src(p);
  }
```

Since `p` is a `Point`, it can hold as many coordinates as are necessary for the arrays `src` and `dest`.

The basic operation on arrays is subscripting: if `A` is an `Array[T]` and `p` a point with the same rank as `A.region`, then `A(p)` is the value of type `T` associated with point `p`. This is the same operation as function application (§10.2); arrays implement function types, and can be used as functions.

Array elements can be changed by assignment. If `t:T`,

`A(p) = t;`

modifies the value associated with `p` to be `t`, and leaves all other values in `A` unchanged.

An `Array[T]` named `a` has:

- `a.region`: the `Region` upon which `a` is defined.
- `a.size`: the number of elements in `a`.
- `a.rank`, the rank of the points usable to subscript `a`. `a.rank` is a cached copy of `a.region.rank`.

Array Constructors

To construct an array whose elements all have the same value `init`, call `new Array[T](R, init)`. For example, an array of a thousand "oh!"'s can be made by: `new Array[String](1000, "oh!")`.

To construct and initialize an array, call the two-argument constructor. `new Array[T](R, f)` constructs an array of elements of type T on region R, with `a(p)` initialized to `f(p)` for each point p in R. f must be a function taking a point of rank R.rank to a value of type T.

Example: One way to construct the array [11, 22, 33] is with an array constructor `new Array[Long](3, (i:long)=>11*i)`. To construct a multiplication table, call `new Array[Long](Region.make(0..9, 0..9), (p:Point(2)) => p(0)*p(1))`.

Other constructors are available; see the API documentation and §11.26.

Array Operations

The basic operation on `Arrays` is subscripting. If `a:Array[T]` and `p:Point{rank == a.rank}`, then `a(p)` is the value of type T appearing at position p in a. The syntax is identical to function application, and, indeed, arrays may be used as functions. `a(p)` may be assigned to, as well, by the usual assignment syntax `a(p)=t`. (This uses the application and setting syntactic sugar, as given in §8.7.5.)

Sometimes it is more convenient to subscript by longs. Arrays of rank 1-4 can, in fact, be accessed by longs:

```
val A1 = new Array[Long](10, 0);
A1(4) = A1(4) + 1;
val A4 = new Array[Long](Region.make(1..2, 1..3, 1..4, 1..5), 0);
A4(2,3,4,5) = A4(1,1,1,1)+1;
```

Iteration over an `Array` is defined, and produces the `Points` of the array's region. If you want to use the values in the array, you have to subscript it. For example, you could take the logarithm of every element of an `Array[Double]` by:

```
for (p in a) a(p) = Math.log(a(p));
```

The method `a.values()` can be used to enumerate all the values of an `Array[T]` array a.

```
static def allNonNegatives(a:Array[Double]):Boolean {
  for (v in a.values()) if (v < 0.0D) return false;
  return true;
}
```

16.4.3 Distributions

Distributed arrays are spread across multiple `Places`. A *distribution*, a mapping from a region to a set of places, describes where each element of a distributed array is kept. Distributions are embodied by the class `x10.regionarray.Dist` and its subclasses.

The *rank* of a distribution is the rank of the underlying region, and thus the rank of every point that the distribution applies to.

Example:

```
val R <: Region = Region.make(1..100);
val D1 <: Dist = Dist.makeBlock(R);
val D2 <: Dist = Dist.makeConstant(R, here);
```

D1 distributes the region R in blocks, with a set of consecutive points at each place, as evenly as possible. D2 maps all the points in R to here.

Let D be a distribution. D.region denotes the underlying region. Given a point p, the expression D(p) represents the application of D to p, that is, the place that p is mapped to by D. The evaluation of the expression D(p) throws an `ArrayIndexOutOfBoundsException` if p does not lie in the underlying region.

PlaceGroups

A PlaceGroup represents an ordered set of Places. PlaceGroups exist for performance and scalability: they are more efficient, in certain critical places, than general collections of Place. PlaceGroup implements `Sequence[Place]`, and thus provides familiar operations – pg.size() for the number of places, pg.iterator() to iterate over them, etc.

PlaceGroup is an abstract class. The concrete class `SparsePlaceGroup` is intended for a small group of places. `new SparsePlaceGroup(somePlace)` is a good PlaceGroup containing one place. `new SparsePlaceGroup(seqPlaces)` constructs a sparse place group from a Rail of places.

Operations returning distributions

Let R be a region, Q a PlaceGroup, and P a place.

Unique distribution The distribution `Dist.makeUnique(Q)` is the unique distribution from the region `Region.make(1..k)` to Q mapping each point i to pi.

Constant distributions. The distribution `Dist.makeConstant(R,P)` maps every point in region R to place P. The special case `Dist.makeConstant(R)` maps every point in R to here.

Block distributions. The distribution `Dist.makeBlock(R)` distributes the elements of R, in approximately-even blocks, over all the places available to the program. There are other `Dist.makeBlock` methods capable of controlling the distribution and the set of places used; see the API documentation.

Domain Restriction. If D is a distribution and R is a sub-region of $D.\text{region}$, then $D \mid R$ represents the restriction of D to R —that is, the distribution that takes each point p in R to $D(p)$, but doesn’t apply to any points but those in R .

Range Restriction. If D is a distribution and P a place expression, the term $D \mid P$ denotes the sub-distribution of D defined over all the points in the region of D mapped to P .

Note that $D \mid \text{here}$ does not necessarily contain adjacent points in $D.\text{region}$. For instance, if D is a cyclic distribution, $D \mid \text{here}$ will typically contain points that differ by the number of places. An implementation may find a way to still represent them in contiguous memory, *e.g.*, using an arithmetic function to map from the region index to an index into the array.

16.4.4 Distributed Arrays

Distributed arrays, instances of `DistArray[T]`, are very much like Arrays, except that they distribute information among multiple Places according to a `Dist` value passed in as a constructor argument.

Example: *The following code creates a distributed array holding a thousand cells, each initialized to 0.0, distributed via a block distribution over all places.*

```
val R <: Region = Region.make(1..1000);
val D <: Dist = Dist.makeBlock(R);
val da <: DistArray[Float]
  = DistArray.make[Float](D, (Point(1))=>0.0f);
```

16.4.5 Distributed Array Construction

`DistArrays` are instantiated by invoking one of the `make` factory methods of the `DistArray` class. A `DistArray` creation must take either an `Long` as an argument or a `Dist`. In the first case, a distributed array is created over the distribution `Dist.makeConstant(Region.make(0, N-1), here)`; in the second over the given distribution.

Example: *A distributed array creation operation may also specify an initializer function. The function is applied in parallel at all points in the domain of the distribution. The construction operation terminates locally only when the `DistArray` has been fully created and initialized (at all places in the range of the distribution).*

For instance:

```
val ident = ([i]:Point(1)) => i;
val data : DistArray[Long]
  = DistArray.make[Long](Dist.makeConstant(Region.make(1, 9)), ident);
val blk = Dist.makeBlock(Region.make(1..9, 1..9));
val data2 : DistArray[Long]
  = DistArray.make[Long](blk, ([i,j]:Point(2)) => i*j);
```

The first declaration stores in `data` a reference to a mutable distributed array with 9 elements each of which is located in the same place as the array. The element at `[i]` is initialized to its index `i`.

The second declaration stores in `data2` a reference to a mutable two-dimensional distributed array, whose coordinates both range from 1 to 9, distributed in blocks over all Places, initialized with $i \cdot j$ at point `[i, j]`.

16.4.6 Operations on Arrays and Distributed Arrays

Arrays and distributed arrays share many operations. In the following, let `a` be an array with base type `T`, and `da` be an array with distribution `D` and base type `T`.

Element operations

The value of `a` at a point `p` in its region of definition is obtained by using the indexing operation `a(p)`. The value of `da` at `p` is similarly `da(p)`. This operation may be used on the left hand side of an assignment operation to update the value: `a(p)=t`; and `da(p)=t`; The operator assignments, `a(i) += e` and so on, are also available.

It is a runtime error to access arrays, with `da(p)` or `da(p)=v`, at a place other than `da.dist(p)`, viz. at the place that the element exists.

Arrays of Single Values

For a region `R` and a value `v` of type `T`, the expression `new Array[T](R, v)` produces an array on region `R` initialized with value `v`. Similarly, for a distribution `D` and a value `v` of type `T` the expression

```
DistArray.make[T](D, (Point(D.rank))=>v)
```

constructs a distributed array with distribution `D` and base type `T` initialized with `v` at every point.

Note that `Arrays` are constructed by constructor calls, but `DistArrays` are constructed by calls to the factory methods `DistArray.make`. This is because `Arrays` are fairly simple objects, but `DistArrays` may be implemented by different classes for different distributions. The use of the factory method gives the library writer the freedom to select appropriate implementations.

Restriction of an array

Let `R` be a sub-region of `da.region`. Then `da | R` represents the sub-`DistArray` of `da` on the region `R`. That is, `da | R` has the same values as `da` when subscripted by a point in region `R && da.region`, and is undefined elsewhere.

Recall that a rich set of operators are available on distributions (§16.4.3) to obtain sub-distributions (e.g. restricting to a sub-region, to a specific place etc).

Operations on Whole Arrays

Pointwise operations The unary `map` operation applies a function to each element of a distributed or non-distributed array, returning a new distributed array with the same distribution, or a non-distributed array with the same region.

The following produces an array of cubes:

```
val A = new Array[Long](11, (i:long)=>i);
assert A(3) == 3 && A(4) == 4 && A(10) == 10;
val cube = (i:Long) => i*i*i;
val B = A.map(cube);
assert B(3) == 27 && B(4) == 64 && B(10) == 1000;
```

A variant operation lets you specify the array `B` into which the result will be stored,

```
val A = new Array[Long](11, (i:long)=>i);
assert A(3) == 3 && A(4) == 4 && A(10) == 10;
val cube = (i:Long) => i*i*i;
val B = new Array[Long](A.region); // B = 0,0,0,0,0,0,0,0,0,0,0
A.map(B, cube);
assert B(3) == 27 && B(4) == 64 && B(10) == 1000;
```

This is convenient if you have an already-allocated array lying around unused. In particular, it can be used if you don't need `A` afterwards and want to reuse its space:

```
val A = new Array[Long](11, (i:long)=>i);
assert A(3) == 3 && A(4) == 4 && A(10) == 10;
val cube = (i:Long) => i*i*i;
A.map(A, cube);
assert A(3) == 27 && A(4) == 64 && A(10) == 1000;
```

The binary `map` operation takes a binary function and another array over the same region or distributed array over the same distribution, and applies the function pointwise to corresponding elements of the two arrays, returning a new array or distributed array of the same shape. The following code adds two distributed arrays:

```
static def add(da:DistArray[Long], db: DistArray[Long])
{da.dist==db.dist}
= da.map(db, (a:Long,b:Long)=>a+b);
```

Reductions Let `f` be a function of type $(T, T) \Rightarrow T$. Let `a` be an array over base type `T`. Let `unit` be a value of type `T`. Then the operation `a.reduce(f, unit)` returns a value of type `T` obtained by combining all the elements of `a` by use of `f` in some unspecified order (perhaps in parallel). The following code gives one method which meets the definition of `reduce`, having a running total `r`, and accumulating each value `a(p)` into it using `f` in turn. (This code is simply given as an example; `Array` has this operation defined already.)

```
def oneWayToReduce[T](a:Array[T], f:(T,T)=>T, unit:T):T {
    var r : T = unit;
    for(p in a.region) r = f(r, a(p));
    return r;
}
```

For example, the following sums an array of longs. `f` is addition, and `unit` is zero.

```
val a = new Array[Long](4, (i:long)=>i+1);
val sum = a.reduce((a:Long,b:Long)=>a+b, 0);
assert(sum == 10); // 10 == 1+2+3+4
```

Other orders of evaluation, degrees of parallelism, and applications of `f(x,unit)` and `f(unit,x)` are also correct. In order to guarantee that the result is precisely determined, the function `f` should be associative and commutative, and the value `unit` should satisfy `f(unit,x) == x == f(x,unit)` for all `x:T`.

`DistArrays` have the same operation. This operation involves communication between the places over which the `DistArray` is distributed. The X10 implementation guarantees that only one value of type `T` is communicated from a place as part of this reduction process.

Scans Let `f:(T,T)=>T`, `unit:T`, and `a` be an `Array[T]` or `DistArray[T]`. Then `a.scan(f,unit)` is the array or distributed array of type `T` whose *i*th element in canonical order is the reduction by `f` with unit `unit` of the first *i* elements of `a`.

This operation involves communication between the places over which the distributed array is distributed. The X10 implementation will endeavour to minimize the communication between places to implement this operation.

Other operations on arrays, distributed arrays, and the related classes may be found in the `x10.regionarray` package.

17 Annotations

X10 provides an annotation system for the compiler to be extended with new static analyses and new transformations.

Annotations are constraint-free interface types that decorate the abstract syntax tree of an X10 program. The X10 type-checker ensures that an annotation is a legal interface type. In X10, interfaces may declare both methods and properties. Therefore, like any interface type, an annotation may instantiate one or more of its interface's properties.

17.1 Annotation syntax

The annotation syntax consists of an “@” followed by an interface type.

Annotations ::= *Annotation* (20.6)

| *Annotations Annotation*

Annotation ::= @ *NamedTypeNoConstraints* (20.4)

Annotations can be applied to most syntactic constructs in the language including class declarations, constructors, methods, field declarations, local variable declarations and formal parameters, statements, expressions, and types. Multiple occurrences of the same annotation (i.e., multiple annotations with the same interface type) on the same entity are permitted.

Recall that interface types may have dependent parameters.

The following examples illustrate the syntax:

- Declaration annotations:

```
// class annotation
@Value
class Cons { ... }

// method annotation
@PreCondition(0 <= i && i < this.size)
public def get(i: Long): T { ... }
```

```
// constructor annotation
@Where(x != null)
def this(x: T) { ... }

// constructor return type annotation
def this(x: T): C@Initialized { ... }

// variable annotation
@Unique x: A;
```

- Type annotations:

```
List@Nonempty

Int@Range(1n,4n)

Rail[Rail[Double]]@Size(n * n)
```

- Expression annotations:

```
m() @RemoteCall
```

- Statement annotations:

```
@Atomic { ... }

@MinIterations(0)
@MaxIterations(n)
for (var i: Long = 0; i < n; i++) { ... }

// An annotated empty statement ;
@Assert(x < y);
```

17.2 Annotation declarations

Annotations are declared as interfaces. They must be subtypes of the interface `x10.lang.annotation.Annotation`. Annotations on particular static entities must extend the corresponding `Annotation` subclasses, as follows:

- Expressions—`ExpressionAnnotation`
- Statements—`StatementAnnotation`
- Classes—`ClassAnnotation`

- Fields—`FieldAnnotation`
- Methods—`MethodAnnotation`
- Imports—`ImportAnnotation`
- Packages—`PackageAnnotation`

18 Interoperability with Other Languages

The ability to interoperate with other programming languages is an essential feature of the X10 implementation. Cross-language interoperability enables both the incremental adoption of X10 in existing applications and the usage of existing libraries and frameworks by newly developed X10 programs.

There are two primary interoperability scenarios that are supported by X10 v2.4: inline substitution of fragments of foreign code for X10 program fragments (expressions, statements) and external linkage to foreign code.

18.1 Embedded Native Code Fragments

The `@Native(lang,code)` Construct annotation from `x10.compiler.Native` in X10 can be used to tell the X10 compiler to substitute code for whatever it would have generated when compiling Construct with the `lang` back end.

The compiler cannot analyze native code the same way it analyzes X10 code. In particular, `@Native` fields and methods must be explicitly typed; the compiler will not infer types.

18.1.1 Native static Methods

`static` methods can be given native implementations. Note that these implementations are syntactically *expressions*, not statements, in C++ or Java. Also, it is possible (and common) to provide native implementations into both Java and C++ for the same method.

```
import x10.compiler.Native;
class Son {
    @Native("c++", "printf(\"Hi!\\\")")
    @Native("java", "System.out.println(\"Hi!\\\")")
    static def printNatively():void {}
}
```

If only some back-end languages are given, the X10 code will be used for the remaining back ends:

```
import x10.compiler.Native;
class Land {
    @Native("c++", "printf(\"Hi from C++!\")")
    static def example():void {
        x10.io.Console.OUT.println("Hi from X10!");
    }
}
```

The `native` modifier on methods indicates that the method must not have an X10 code body, and `@Native` implementations must be given for all back ends:

```
import x10.compiler.Native;
class Plants {
    @Native("c++", "printf(\"Hi!\")")
    @Native("java", "System.out.println(\"Hi!\")")
    static native def printNatively():void;
}
```

Values may be returned from external code to X10. Scalar types in Java and C++ correspond directly to the analogous types in X10.

```
import x10.compiler.Native;
class Return {
    @Native("c++", "1")
    @Native("java", "1")
    static native def one():Int;
}
```

Types are *not* inferred for methods marked as `@Native`.

Parameters may be passed to external code. (#1) is the first parameter, (#2) the second, and so forth. (#0) is the name of the enclosing class, or the `this` variable.) Be aware that this is macro substitution rather than normal parameter passing; *e.g.*, if the first actual parameter is `i++`, and (#1) appears twice in the external code, `i` will be incremented twice. For example, a (ridiculous) way to print the sum of two numbers is:

```
import x10.compiler.Native;
class Species {
    @Native("c++","printf(\"Sum=%d\", ((#1)+(#2)) )")
    @Native("java","System.out.println(\"\" + ((#1)+(#2)))")
    static native def printNatively(x:Int, y:Int):void;
}
```

Static variables in the class are available in the external code. For Java, the static variables are used with their X10 names. For C++, the names must be mangled, by use of the FMGL macro.

```
import x10.compiler.Native;
class Ability {
    static val A : Int = 1n;
    @Native("java", "A+2")
    @Native("c++", "Ability::FMGL(A)+2")
    static native def fromStatic():Int;
}
```

18.1.2 Native Blocks

Any block may be annotated with `@Native(lang, stmt)`, indicating that, in the given back end, it should be implemented as `stmt`. All variables from the surrounding context are available inside `stmt`. For example, the method call `born.example(10n)`, if compiled to Java, changes the field `y` of a `Born` object to 10. If compiled to C++ (for which there is no `@Native`), it sets it to 3.

```
import x10.compiler.Native;
class Born {
    var y : Int = 1n;
    public def example(x:Int):Int{
        @Native("java", "y=x;")
        {y = 3n;}
        return y;
    }
}
```

Note that the code being replaced is a statement – the block `{y = 3n;}` in this case – so the replacement should also be a statement.

Other X10 constructs may or may not be available in Java and/or C++ code. For example, type variables do not correspond exactly to type variables in either language, and may not be available there. The exact compilation scheme is *not* fully specified. You may inspect the generated Java or C++ code and see how to do specific things, but there is no guarantee that fancy external coding will continue to work in later versions of X10.

The full facilities of C++ or Java are available in native code blocks. However, there is no guarantee that advanced features behave sensibly. You must follow the exact conventions that the code generator does, or you will get unpredictable results. Furthermore, the code generator's conventions may change without notice or documentation from version to version. In most cases the code should either be a very simple expression, or a method or function call to external code.

18.2 Interoperability with External Java Code

With Managed X10, we can seamlessly call existing Java code from X10, and call X10 code from Java. We call this the *Java interoperability* [13] feature.

By combining Java interoperability with X10's distributed execution features, we can even make existing Java applications, which are originally designed to run on a single Java VM, scale-out with minor modifications.

18.2.1 How Java program is seen in X10

Managed X10 does not pre-process the existing Java code to make it accessible from X10. X10 programs compiled with Managed X10 will call existing Java code as is.

Types In X10, both at compile time and run time, there is no way to distinguish Java types from X10 types. Java types can be referred to with regular `import` statement, or their qualified names. The package `java.lang` is not auto-imported into X10. In Managed X10, the resolver is enhanced to resolve types with X10 source files in the source path first, then resolve them with Java class files in the class path. Note that the resolver does not resolve types with Java source files, therefore Java source files must be compiled in advance. To refer to Java types listed in Tables 18.1, and 18.2, which include all Java primitive types, use the corresponding X10 type (e.g. use `x10.lang.Int` (or in short, `Int`) instead of `int`).

Fields Fields of Java types are seen as fields of X10 types.

Managed X10 does not change the static initialization semantics of Java types, which is per-class, at load time, and per-place (Java VM), therefore, it is subtly different than the per-field lazy initialization semantics of X10 static fields.

Methods Methods of Java types are seen as methods of X10 types.

Generic types Generic Java types are seen as their raw types (§4.8 in [6]). Raw type is a mechanism to handle generic Java types as non-generic types, where the type parameters are assumed as `java.lang.Object` or their upperbound if they have it. Java introduced generics and raw type at the same time to facilitate generic Java code interfacing with non-generic legacy Java code. Managed X10 uses this mechanism for a slightly different purpose. Java erases type parameters at compile time, whereas X10 preserves their values at run time. To manifest this semantic gap in generics, Managed X10 represents Java generic types as raw types and eliminates type parameters at source code level. For more detailed discussions, please refer to [14, 15].

```

import x10.interop.Java;
public class XClass {
    public static def main(args: Rail[String]): void {
        try {
            val a = Java.newArray[Int](2n);
            a(0n) = 0n;
            a(1n) = 1n;
            a(2n) = 2n;
        } catch (e:x10.lang.ArrayIndexOutOfBoundsException) {
            Console.OUT.println(e);
        }
    }
}

> x10c -d bin src/XClass.x10
> x10 -cp bin XClass
x10.lang.ArrayIndexOutOfBoundsException: Array index out of range: 2

```

Figure 18.1: Java exceptions in X10

Arrays X10 rail and array types are generic types whose representation is different from Java array types.

Managed X10 provides a special X10 type `x10.interop.Java.array[T]` which represents Java array type `T[]`. Note that for X10 types in Table 18.1, this type means the Java array type of their primary type. For example, `array[Int]` and `array[String]` mean `int[]` and `java.lang.String[]`, respectively. Managed X10 also provides conversion methods between X10 Rails and Java arrays (`Java.convert[T](a:Rail[T]):array[T]` and `Java.convert[T](a:array[T]):Rail[T]`), and creation methods for Java arrays (`Java.newArray[T](d0:Int):array[T]` etc.).

Exceptions The X10 v2.4 exception hierarchy has been designed so that there is a natural correspondence with the Java exception hierarchy. As shown in Table 18.2, many commonly used Java exception types are directly mapped to X10 exception types. Exception types that are thus aliased may be caught (and thrown) using either their Java or X10 types. In X10 code, it is stylistically preferable to use the X10 type to refer to the exception as shown in Figure 18.1.

Compiling and executing X10 programs We can compile and run X10 programs that call existing Java code with the same `x10c` and `x10` command by specifying the location of Java class files or jar files that your X10 programs refer to, with `-classpath` (or in short, `-cp`) option.

18.2.2 How X10 program is translated to Java

Managed X10 translates X10 programs to Java class files.

X10 does not provide a Java reflection-like mechanism to resolve X10 types, methods, and fields with their names at runtime, nor a code generation tool, such as `javadoc`, to generate stub code to access them from other languages. Java programmers, therefore, need to access X10 types, methods, and fields in the generated Java code directly as they access Java types, methods, and fields. To make it possible, Java programmers need to understand how X10 programs are translated to Java.

Some aspects of the X10 to Java translation scheme may change in future version of X10; therefore in this document only a stable subset of translation scheme will be explained. Although it is a subset, it has been extensively used by X10 core team and proved to be useful to develop Java Hadoop interop layer for a Main-memory Map Reduce (M3R) engine [11] in X10.

In the following discussions, we deliberately ignore generic X10 types because the translation of generics is an area of active development and will undergo some changes in future versions of X10. For those who are interested in the implementation of generics in Managed X10, please consult [15]. We also don't cover function types, function values, and all non-static methods. Although slightly outdated, another paper [14], which describes translation scheme in X10 2.1.2, is still useful to understand the overview of Java code generation in Managed X10.

Types X10 classes and structs are translated to Java classes with the same names. X10 interfaces are translated to Java interfaces with the same names.

Table 18.1 shows the list of special types that are mapped to Java primitives. Primitives are their primary representations that are useful for good performance. Wrapper classes are used when the reference types are needed. Each wrapper class has two static methods `$box()` and `$unbox()` to convert its value from primary representation to wrapper class, and vice versa, and Java backend inserts their calls as needed. As you notice, every unsigned type uses the same Java primitive as its corresponding signed type for its representation.

Table 18.2 shows a non-exhaustive list of another kind of special types that are mapped (not translated) to Java types. As you notice, since an interface `Any` is mapped to a class —`java.lang.Object`— and `Object` is hidden from the language, there is no direct way to create an instance of `Object`. As a workaround, `Java.newObject()` is provided.

As you also notice, `x10.lang.Comparable[T]` is mapped to `java.lang.Comparable`. This is needed to map `x10.lang.String`, which implements `x10.lang.Comparable[String]`, to `java.lang.String` for performance, but as a trade off, this mapping results in the loss of runtime type information for `Comparable[T]` in Managed X10. The runtime of Managed X10 has built-in knowledge for `String`, but for other Java classes that implement `java.lang.Comparable`, `instanceof Comparable[Int]` etc. may return incorrect results. In principle, it is impossible to map X10 generic type to the existing Java generic type without losing runtime type information.

X10		Java (primary)	Java (wrapper class)
x10.lang.Byte	1y	byte (byte)1	x10.core.Byte
x10.lang.UByte	1uy	byte (byte)1	x10.core.UByte
x10.lang.Short	1s	short (short)1	x10.core.Short
x10.lang.UShort	1us	short (short)1	x10.core.UShort
x10.lang.Int	1n	int 1	x10.core.Int
x10.lang.UInt	1un	int 1	x10.core.UInt
x10.lang.Long	1	long 1l	x10.core.Long
x10.lang.ULong	1u	long 1l	x10.core.ULong
x10.lang.Float	1.0f	float 1.0f	x10.core.Float
x10.lang.Double	1.0	double 1.0	x10.core.Double
x10.lang.Char	'c'	char 'c'	x10.core.Char
x10.lang.Boolean	true	boolean true	x10.core.Boolean

Table 18.1: X10 types that are mapped to Java primitives

X10	Java
x10.lang.Any	java.lang.Object
x10.lang.Comparable[T]	java.lang.Comparable
x10.lang.String	java.lang.String
x10.lang.CheckedThrowable	java.lang.Throwable
x10.lang.CheckedException	java.lang.Exception
x10.lang.Exception	java.lang.RuntimeException
x10.lang.ArithmaticException	java.lang.ArithmaticException
x10.lang.ClassCastException	java.lang.ClassCastException
x10.lang.IllegalArgumentException	java.lang.IllegalArgumentException
x10.util.NoSuchElementException	java.util.NoSuchElementException
x10.lang.NullPointerException	java.lang.NullPointerException
x10.lang.NumberFormatException	java.lang.NumberFormatException
x10.lang.UnsupportedOperationException	java.lang.UnsupportedOperationException
x10.lang.IndexOutOfBoundsException	java.lang.IndexOutOfBoundsException
x10.lang.ArrayIndexOutOfBoundsException	java.lang.ArrayIndexOutOfBoundsException
x10.lang.StringIndexOutOfBoundsException	java.lang.StringIndexOutOfBoundsException
x10.lang.Error	java.lang.Error
x10.lang.AssertionError	java.lang.AssertionError
x10.lang.OutOfMemoryError	java.lang.OutOfMemoryError
x10.lang.StackOverflowError	java.lang.StackOverflowError
void	void

Table 18.2: X10 types that are mapped (not translated) to Java types

```

class C {
    static val a:Int = ...;
    var b:Int;
}
interface I {
    val x:Int = ...;
}

class C {
    static int get$a() { return ...; }
    int b;
}
interface I {
    abstract static class $Shadow {
        static int get$x() { return ...; }
    }
}

```

Figure 18.2: X10 fields in Java

Fields As shown in Figure 18.2, instance fields of X10 classes and structs are translated to the instance fields of the same names of the generated Java classes. Static fields of X10 classes and structs are translated to the static methods of the generated Java classes, whose name has `get$` prefix. Static fields of X10 interfaces are translated to the static methods of the special nested class named `$Shadow` of the generated Java interfaces.

Note on name resolution In X10, fields (both static and instance), local variables, and types are in the same name space, and fields and local variables have higher precedence than types in resolving names. This is same as in Java and it is programmers' responsibility to avoid name conflict. Figure 18.3 explains how name conflict occurs. Uncommenting either `a` or `b`, or replacing `c` with `c'` causes *Field C not found in type "x10.lang.Long"* at `x`. Even if there is no name conflict in X10 code, it may occur in the generated Java code since Java backend generates static field access to get runtime type in some situation. To avoid potential name conflict, the best practice is not to use the same name for variables and package prefix.

Methods As shown in Figure 18.4, methods of X10 classes or structs are translated to the methods of the same names of the generated Java classes. Methods of X10 interfaces are translated to the methods of the same names of the generated Java interfaces.

Every method whose return type has two representations, such as the types in Table 18.1, will have `$0` suffix with its name. For example, `def f():Int` in X10 will

Main.x10:

```
public class Main {
    //static val p:Long = 1; // a
    //val p:Long = 1;        // b
    //def func(p:Long) {    // c'
    def func() {           // c
        val f = p.C.f;     // x
    }
}
```

p/C.x10:

```
package p;
public class C {
    public static val f = true;
}
```

Figure 18.3: Name conflict in X10

be compiled as `int f$0()` in Java. For those who are interested in the reason, please look at our paper [15].

Compiling Java programs To compile Java program that calls X10 code, you should use `x10cj` command instead of `javac` (or whatever your Java compiler). It invokes the post Java-compiler of `x10c` with the appropriate options. You need to specify the location of X10-generated class files or jar files that your Java program refers to.

```
x10cj -cp MyX10Lib.jar MyJavaProg.java
```

Executing Java programs Before executing any X10-generated Java code, the runtime of Managed X10 needs to be set up properly at each place. To set up the runtime, a special launcher named `runjava` is used to run Java programs. All Java programs that call X10 code should be launched with it, and no other mechanisms, including direct execution with `java` command, are supported.

Usage: `runjava <Java-main-class> [arg0 arg1 ...]`

18.3 Interoperability with External C and C++ Code

C and C++ code can be linked to X10 code, either by writing auxiliary C++ files and adding them with suitable annotations, or by linking libraries.

```
interface I {
    def f():Int;
    def g():Any;
}
class C implements I {
    static def s():Int = 0n;
    static def t():Any = null;
    public def f():Int = 1n;
    public def g():Any = null;
}

interface I {
    int f$0();
    java.lang.Object g();
}
class C implements I {
    static int s$0() { return 0; }
    static java.lang.Object t() { return null; }
    public int f$0() { return 1; }
    public java.lang.Object g() { return null; }
}
```

Figure 18.4: X10 methods in Java

18.3.1 Auxiliary C++ Files

Auxiliary C++ code can be written in .h and .cc files, which should be put in the same directory as the X10 file using them. Connecting with the library uses the `@NativeCPPInclude(dot_h_file_name)` annotation to include the header file, and the `@NativeCPPCompilationUnit(dot_cc_file_name)` annotation to include the C++ code proper. For example:

MyCppCode.h:

```
void foo();
```

MyCppCode.cc:

```
#include <cstdlib>
#include <cstdio>
void foo() {
    printf("Hello World!\n");
}
```

Test.x10:

```
import x10.compiler.Native;
import x10.compiler.NativeCPPInclude;
import x10.compiler.NativeCPPCompilationUnit;

@NativeCPPInclude("MyCPPCode.h")
@NativeCPPCompilationUnit("MyCPPCode.cc")
public class Test {
    public static def main (args: Rail[String]) {
        { @Native("c++", "foo()") {} }
    }
}
```

18.3.2 C++ System Libraries

If we want to additionally link to more libraries in /usr/lib for example, it is necessary to adjust the post-compilation directly. The post-compilation is the compilation of the C++ which the X10-to-C++ compiler `x10c++` produces.

The primary mechanism used for this is the `-cxx-prearg` and `-cxx-postarg` command line arguments to `x10c++`. The values of any `-cxx-prearg` commands are placed in the post compiler command before the list of .cc files to compile. The values of any `-cxx-postarg` commands are placed in the post compiler command after the list of .cc files to compile. Typically pre-args are arguments intended for the C++ compiler itself, while post-args are arguments intended for the linker.

The following example shows how to compile blas into the executable via these commands. The command must be issued on one line.

```
x10c++ Test.x10 -cxx-prearg -I/usr/local/blas  
-cxx-postarg -L/usr/local/blas -cxx-postarg -lblas'
```

19 Definite Assignment

X10 requires that every variable be set before it is read. Sometimes this is easy, as when a variable is declared and assigned together:

```
var x : Long = 0;
assert x == 0;
```

However, it is convenient to allow programs to make decisions before initializing variables.

```
def example(a:Long, b:Long) {
    val max:Long;
    //ERROR: assert max==max; // can't read 'max'
    if (a > b) max = a;
    else max = b;
    assert max >= a && max >= b;
}
```

This is particularly useful for `val` variables. `vars` could be initialized to a default value and then reassigned with the right value. `vals` must be initialized once and cannot be changed, so they must be initialized with the correct value.

However, one must be careful – and the X10 compiler enforces this care. Without the `else` clause, the preceding code might not give `max` a value by the time `assert` is invoked.

This leads to the concept of *definite assignment* [5]. A variable is *definitely assigned* at a point in code if, no matter how that point in code is reached, the variable has been assigned to. In X10, variables must be definitely assigned before they can be read.

As X10 requires that `val` variables *not* be initialized twice, we need the dual concept as well. A variable is *definitely unassigned* at a point in code if it cannot have been assigned no matter how that point in code is reached. For example, immediately after `val x:Long`, `x` is definitely unassigned.

Finally, we need the concept of *singly* and *multiply assigned*. A variable is singly assigned in a block if it is assigned precisely once; it is multiply assigned if it could possibly be assigned more than once. `vars` can multiply assigned as desired. `vals`

must be singly assigned. For example, the code `x = 1; x = 2;` is perfectly fine if `x` is a `var`, but incorrect (even in a constructor) if `x` is a `val`.

At some points in code, a variable might be neither definitely assigned nor definitely unassigned. Such states are not always useful.

```
def example(flag : Boolean) {
    var x : Long;
    if (flag) x = 1;
    // x is neither def. assigned nor unassigned.
    x = 2;
    // x is def. assigned.
```

This shows that we cannot simply define “definitely unassigned” as “not definitely assigned”. If `x` had been a `val` rather than a `var`, the previous example would not be allowed.

Unfortunately, a completely accurate definition of “definitely assigned” or “definitely unassigned” is undecidable – impossible for the compiler to determine. So, X10 takes a *conservative approximation* of these concepts. If X10’s definition says that `x` is definitely assigned (or definitely unassigned), then it will be assigned (or not assigned) in every execution of the program.

However, there are programs which X10’s algorithm says are incorrect, but which actually would behave properly if they were executed. In the following example, `flag` is either `true` or `false`, and in either case `x` will be initialized. However, X10’s analysis does not understand this — thought it *would* understand if the example were coded with an `if-else` rather than a pair of `ifs`. So, after the two `if` statements, `x` is not definitely assigned, and thus the `assert` statement, which reads it, is forbidden.

```
def example(flag:Boolean) {
    var x : Long;
    if (flag) x = 1;
    if (!flag) x = 2;
    // ERROR: assert x < 3;
}
```

19.1 Asynchronous Definite Assignment

Local variables and instance fields allow *asynchronous assignment*. A local variable can be assigned in an `async` statement, and, when the `async` is `finished`, the variable is definitely assigned.

Example:

```
val a : Long;
finish {
    async {
```

```

    a = 1;
}
// a is not definitely assigned here
}
// a is definitely assigned after 'finish'
assert a==1;

```

This concept supports a core X10 programming idiom. A `val` variable may be initialized asynchronously, thereby providing a means for returning a value from an `async` to be used after the enclosing `finish`.

19.2 Characteristics of Definite Assignment

The properties “definitely assigned”, “singly assigned”, and “definitely unassigned” are computed by a conservative approximation of X10’s evaluation rules.

The precise details are up to the implementation. Many basic cases must be handled accurately; *e.g.*, `x=1`; definitely and singly assigns `x`.

However, in more complicated cases, a conforming X10 may mark as invalid some code which, when executed, would actually be correct. For example, the following program fragment will always result in `x` being definitely and singly assigned:

```

val x : Long;
var b : Boolean = mysterious();
if (b) x = cryptic();
if (!b) x = unknown();

```

However, most conservative approximations of program execution won’t mark `x` as properly initialized, though it is. For `x` to be properly initialized, precisely one of the two assignments to `x` must be executed. If `b` were true initially, it would still be true after the call to `cryptic()` — since methods cannot modify their caller’s local variables – and so the first but not the second assignment would happen. If `b` were false initially, it would still be false when `!b` is tested, and so the second but not the first assignment would happen. Either way, `x` is definitely and singly assigned.

However, for a slightly different program, this analysis would be wrong. *E.g.*, if `b` were a field of `this` rather than a local variable, `cryptic()` could change `b`; if `b` were true initially, both assignments might happen, which is incorrect for a `val`.

This sort of reasoning is beyond most conservative approximation algorithms. (Indeed, many do not bother checking that `!b` late in the program is the opposite of `b` earlier.) Algorithms that pay attention to such details and subtleties tend to be fairly expensive, which would lead to very slow compilation for X10 – for the sake of obscure cases.

X10’s analysis provides at least the following guarantees. We describe them in terms of a statement `S` performing some collection of possible numbers of assignments to

variables — on a scale of “0”, “1”, and “many”. For example, `if (b) x=1; else {x=1;x=2;y=2;}` might assign to `x` one or many times, and might assign to `y` zero or one time. Hence, after it, `x` is definitely assigned and may be multiply assigned, and `y` is neither definitely assigned nor definitely unassigned.

These descriptions are combined in natural ways. For example, if `R` says that `x` will be assigned 0 or 1 times, and `S` says it will be assigned precisely once, then `R;S` will assign it one or many times. If only one or `R` or `S` will occur, as from `if (b) R; else S;`, then `x` may be assigned 0 or 1 times.

This information is sufficient for the tests X10 makes. If `x` can be assigned one or many times in `S`, it is definitely assigned. It is an error if `x` is ever read at a point where it has been assigned zero times. It is an error if a `val` may be assigned many times.

We do not guarantee that any particular X10 compiler uses this algorithm; indeed, as of the time of writing, the X10 compiler uses a somewhat more precise one. However, any conformant X10 compiler must provide results which are at least as accurate as this analysis.

Assignment: `x = e`

`x = e` assigns to `x`, in addition to whatever assignments `e` makes. For example, if `this.setX(y)` sets a field `x` to `y` and returns `y`, then `x = this.setX(y)` definitely and multiply assigns `x`.

async and finish

By itself, `async S` provides few guarantees. After an activity executes `async{x=1;}` we know that there is a separate activity which (on being scheduled) will set `x` to 1. We do not know that this has happened yet.

However, if there is a `finish` around the `async`, the situation is clearer. After `finish` `async x=1;, x` has definitely been assigned.

In general, if an `async S` appears in the body of a `finish` in a way that guarantees that it will be executed, then, after the `finish`, the assignments made by `S` will have occurred. For example, if `S` definitely assigns to `x`, and the body of the `finish` guarantees that `async S` will be executed, then `finish{...async S...}` definitely assigns `x`.

if and switch

When `if(E) S else T` finishes, it will have performed the assignments of `E`, together with those of either `S` or `T` but not both. For example, `if (b) x=1; else x=2;` definitely assigns `x`, but `if (b) x=1;` does not.

`switch` is more complex, but follows the same principles as `if`. For example, `switch(E){case 1: A; break; case 2: B; default: C;}` performs the assignments of `E`, and those

of precisely one of A, or B;C, or C. Note that case 2 falls through to the default case, so it performs the same assignments as B;C.

Sequencing

When R;S finishes, it will have performed the assignments of R and those of S, if R and S terminate normally. If R terminates abruptly, then only the assignments of R executed till the point of termination will have been executed. If R terminates normally, but S terminates abruptly then the assignments of R will have been executed and those of S executed till the point of termination.

For example, `x=1;y=2;` definitely assigns x and y, and `x=1;x=2;` multiply assigns x.

Loops

`while(E)S` performs the assignments of E one or more times, and those of S zero or more times. For example, if `while(b()){x=1;}` might assign to x zero, one, or many times. `do S while(E)` performs the assignments of E one or more times, and those of S one or more times.

`for(A;B;C)D` performs the assignments of A once, those of B one or more times, and those of C and D one or more times. `for(x in E)S` performs the assignments of E once and those of S zero or more times.

Loops are of very little value for providing definite assignments, since X10 does not in general know how many times they will be executed.

`continue` and `break` inside of a loop are hard to describe in simple terms. They may be conservatively assumed to cause the loop to give no information about the variables assigned inside of it. For example, the analysis may conservatively conclude that `do{x = 1; if (true) break; } while(true)` may assign to x zero, one, or many times, overlooking the more precise fact that it is assigned once.

Method Calls

A method call `E.m(A,B)` performs the assignments of E, A, and B once each, and also those of m. This implies that X10 must be aware of the possible assignments performed by each method.

If X10 has complete information about m (as when m is a `private` or `final` method), this is straightforward. When such information is fundamentally impossible to acquire, as when m is a non-final method invocation, X10 has no choice but to assume that m might do anything that a method can do. (For this reason, the only methods that can be called from within a constructor on a raw – incompletely-constructed – object) are the `private` and `final` ones.)

- m cannot assign to local variables of the caller; methods have no such power.

- Let m be an instance method. m can assign to `var` fields of `this` freely,
- Let m be an instance method. m cannot initialize `val` fields of `this`. (But see §8.5.2; when one constructor calls another as the first statement of its body, the other constructor can initialize `vval` fields. This is a constructor call, not a method call.)

Recall that every container must be fully initialized upon exit from its constructor. X10 places certain restrictions on which methods can be called from a constructor; see §8.11.1. One of these restrictions is that methods called before object initialization is complete must be `final` or `private` — and hence, available for static analysis. So, when checking field initialization, X10 will ensure:

1. Each `val` field is initialized before it is read. A method that does not read a `val` field f *may* be called before f is initialized; a method that reads f must not be called until f is initialized. For example, a constructor may have the form:

```
class C {
    val f : Long;
    val g : String;
    def this() {
        f = fless();
        g = useF();
    }
    private def fless() = "f not used here".length();
    private def useF() = "f=" + this.f;
}
```

2. `var` fields require a deeper analysis. Consider a `var` field `var x:T` without initializer. If T has a default value, x may be read inside of a constructor before it is otherwise written, and it will have its default value.

If T has no default value, an analysis like that used for `vals` must be performed to determine that x is initialized before it is used. The situation is more complex than for `vals`, however, because a method can assign to x as well read from it. The X10 compiler computes a conservative approximation of which methods read and write which `var` fields. (Doing this carefully requires finding a solution of a set of equations over sets of variables, with each callable method having equations describing what it reads and writes.)

`at`

`at(p)S` performs precisely the assignments of p and those of S . Note that S is executed at the place named by p in an environment in which all variables used in S but defined outside S are bound to copies (made at p) of the values they had at the `at(p)S` statement (§13.3).

atomic

atomic S performs the assignments of S, and **when(E)S** performs those of E and S. Note that E or S may terminate abruptly.

try

try S catch(x:T1) E1 ... catch(x:Tn) En finally F performs some or all of the assignments of S, plus all the assignments of zero or one of the E's, plus those of F. For example,

```
try {
    x = boomy();
    x = Ø;
}
catch(e:Boom) { y = 1; }
finally { z = 1; }
```

assigns x zero, one, or many times¹, assigns y zero or one time, and assigns z exactly once.

Expression Statements

Expression statements **E;**, and other statements that execute an expression and do something innocuous with it (local variable declaration and **assert**) have the same effects as E.

return, throw

Statements that do not finish normally, such as **return** and **throw**, do not initialize anything (though the computation of the return or thrown value may). They also terminate a line of computation. For example, **if(b) {x=1; return;} x=2;** definitely and singly assigns x.

¹A more precise analysis could discover that x cannot be initialized only once.

20 Grammar

In this grammar, $X^?$ denotes an optional X element.

- (0) $AdditiveExp ::= MultiplicativeExp$
| $AdditiveExp + MultiplicativeExp$
| $AdditiveExp - MultiplicativeExp$
- (1) $AndExp ::= EqualityExp$
| $AndExp \& EqualityExp$
- (2) $AnnotatedType ::= Type Annotations$
- (3) $Annotation ::= @ NamedTypeNoConstraints$
- (4) $AnnotationStmt ::= Annotations? NonExpStmt$
- (5) $Annotations ::= Annotation$
| $Annotations Annotation$
- (6) $ApplyOpDecln ::= MethMods operator this TypeParams? Formals Guard?$
 $HasResultType? MethodBody$
- (7) $ArgumentList ::= Exp$
| $ArgumentList , Exp$
- (8) $Arguments ::= (ArgumentList)$
- (9) $AssertStmt ::= assert Exp ;$
| $assert Exp : Exp ;$
- (10) $AssignPropCall ::= property TypeArgs? (ArgumentList?) ;$

- (11) *Assignment* ::= *LeftHandSide AsstOp AsstExp*
 | *ExpName (ArgumentList?) AsstOp AsstExp*
 | *Primary (ArgumentList?) AsstOp AsstExp*
- (12) *AsstExp* ::= *Assignment*
 | *ConditionalExp*
- (13) *AsstOp* ::= =
 | *=
 | /=
 | %=
 | +=
 | -=
 | <<=
 | >>=
 | >>>=
 | &=
 | ^=
 | |=
- (14) *AsyncStmt* ::= *async ClockedClause? Stmt*
 | *clocked async Stmt*
- (15) *AtCaptureDeclr* ::= *Mods? VarKeyword? VariableDeclr*
 | *Id*
 | *this*
- (16) *AtCaptureDecls* ::= *AtCaptureDeclr*
 | *AtCaptureDecls , AtCaptureDeclr*
- (17) *AtEachStmt* ::= *ateach (LoopIndex in Exp) ClockedClause? Stmt*
 | *ateach (Exp) Stmt*
- (18) *AtExp* ::= *at (Exp) ClosureBody*
- (19) *AtStmt* ::= *at (Exp) Stmt*
- (20) *AtomicStmt* ::= *atomic Stmt*
- (21) *BasicForStmt* ::= *for (ForInit? ; Exp? ; ForUpdate?) Stmt*

(22) *BinOp* ::= +
 | -
 | *
 | /
 | %
 | &
 | |
 | ^
 | &&
 | ||
 | <<
 | >>
 | >>>
 | >=
 | <=
 | >
 | <
 | ==
 | !=
 | ..
 | ->
 | <-
 | -<
 | >-
 | **
 | ~
 | ! ~
 | !

(23) *BinOpDecln* ::= *MethMods operator TypeParams?* (*Formal*) *BinOp* (*Formal*)
 Guard? *HasResultType?* *MethodBody*
 | *MethMods operator TypeParams?* *this BinOp* (*Formal*) *Guard?*
 HasResultType? *MethodBody*
 | *MethMods operator TypeParams?* (*Formal*) *BinOp this Guard?*
 HasResultType? *MethodBody*

(24) *Block* ::= { *BlockStmts?* }

(25) *BlockInteriorStmt* ::= *LocVarDeclnStmt*
 | *ClassDecln*
 | *StructDecln*
 | *TypeDefDecln*
 | *Stmt*

- (26) *BlockStmts* ::= *BlockInteriorStmt*
 | *BlockStmts BlockInteriorStmt*
- (27) *BooleanLiteral* ::= **true**
 | **false**
- (28) *BreakStmt* ::= **break** *Id*? ;
- (29) *CastExp* ::= *Primary*
 | *ExpName*
 | *CastExp as Type*
- (30) *CatchClause* ::= **catch** (*Formal*) *Block*
- (31) *Catches* ::= *CatchClause*
 | *Catches CatchClause*
- (32) *ClassBody* ::= { *ClassMemberDeclns*? }
- (33) *ClassDecln* ::= *Mods*? **class** *Id* *TypeParamsI*? *Properties*? *Guard*? *Super*? *Interfaces*?
 ClassBody
- (34) *ClassMemberDecln* ::= *InterfaceMemberDecln*
 | *CtorDecln*
- (35) *ClassMemberDeclns* ::= *ClassMemberDecln*
 | *ClassMemberDeclns ClassMemberDecln*
- (36) *ClassName* ::= *TypeName*
- (37) *ClassType* ::= *NamedType*
- (38) *ClockedClause* ::= **clocked** *Arguments*
- (39) *ClosureBody* ::= *Exp*
 | *ClosureBodyBlock*
- (40) *ClosureBodyBlock* ::= *Annotations*? { *BlockStmts*? *LastExp* }
 | *Annotations*? *Block*

- (41) *ClosureExp* ::= *Formals Guard?* *HasResultType?* => *ClosureBody*
- (42) *CompilationUnit* ::= *PackageDecln?* *TypeDeclns?*
 | *PackageDecln?* *ImportDeclns TypeDeclns?*
 | *ImportDeclns PackageDecln ImportDeclns?* *TypeDeclns?*
 | *PackageDecln ImportDeclns PackageDecln ImportDeclns?*
 | *TypeDeclns?*
- (43) *ConditionalAndExp* ::= *InclusiveOrExp*
 | *ConditionalAndExp && InclusiveOrExp*
- (44) *ConditionalExp* ::= *ConditionalOrExp*
 | *ClosureExp*
 | *AtExp*
 | *ConditionalOrExp ? Exp : ConditionalExp*
- (45) *ConditionalOrExp* ::= *ConditionalAndExp*
 | *ConditionalOrExp || ConditionalAndExp*
- (46) *ConstantExp* ::= *Exp*
- (47) *ConstrainedType* ::= *NamedType*
 | *AnnotatedType*
- (48) *ConstraintConjunction* ::= *Exp*
 | *ConstraintConjunction , Exp*
- (49) *ContinueStmt* ::= **continue** *Id?* ;
- (50) *ConversionOpDecln* ::= *ExplConvOpDecln*
 | *ImplConvOpDecln*
- (51) *CtorBlock* ::= { *ExplicitCtorInvo?* *BlockStmts?* }
- (52) *CtorBody* ::= *CtorBlock*
 | = *ExplicitCtorInvo*
 | = *AssignPropCall*
 | ;
- (53) *CtorDecln* ::= *Mods?* **def this** *TypeParams?* *Formals Guard?* *HasResultType?* *CtorBody*

(54) *DepNamedType* ::= *SimpleNamedType DepParams*
 | *ParamizedNamedType DepParams*

(55) *DoStmt* ::= **do** *Stmt while (Exp) ;*

(56) *EmptyStmt* ::= **;**

(57) *EnhancedForStmt* ::= **for (LoopIndex in Exp) Stmt**
 | **for (Exp) Stmt**

(58) *EqualityExp* ::= *RelationalExp*
 | *EqualityExp == RelationalExp*
 | *EqualityExp != RelationalExp*
 | *Type == Type*
 | *EqualityExp ~ RelationalExp*
 | *EqualityExp !~ RelationalExp*

(59) *ExclusiveOrExp* ::= *AndExp*
 | *ExclusiveOrExp ^ AndExp*

(60) *Exp* ::= *AsstExp*

(61) *ExpName* ::= *Id*
 | *FullyQualifiedNames . Id*

(62) *ExpStmt* ::= *StmtExp ;*

(63) *ExplConvOpDecln* ::= *MethMods operator TypeParams? (Formal) as Type Guard?*
 | *MethMods operator TypeParams? (Formal) as ? Guard?*
 HasResultType? MethodBody

(64) *ExplicitCtorInvo* ::= *this TypeArgs? (ArgumentList?) ;*
 | *super TypeArgs? (ArgumentList?) ;*
 | *Primary . this TypeArgs? (ArgumentList?) ;*
 | *Primary . super TypeArgs? (ArgumentList?) ;*

(65) *ExtendsInterfaces* ::= **extends** *Type*
 | *ExtendsInterfaces , Type*

- (66) *FieldAccess* ::= *Primary* . *Id*
 | **super** . *Id*
 | *ClassName* . **super** . *Id*
- (67) *FieldDecln* ::= *Mods*? **VarKeyword** *FieldDecls* ;
 | *Mods*? *FieldDecls* ;
- (68) *FieldDeclr* ::= *Id HasResultType*
 | *Id HasResultType*? = *VariableInitializer*
- (69) *FieldDecls* ::= *FieldDeclr*
 | *FieldDecls* , *FieldDeclr*
- (70) *Finally* ::= **finally** *Block*
- (71) *FinishStmt* ::= **finish Stmt**
 | **clocked** **finish Stmt**
- (72) *ForInit* ::= *StmtExpList*
 | *LocVarDecln*
- (73) *ForStmt* ::= *BasicForStmt*
 | *EnhancedForStmt*
- (74) *ForUpdate* ::= *StmtExpList*
- (75) *Formal* ::= *Mods*? *FormalDeclr*
 | *Mods*? **VarKeyword** *FormalDeclr*
 | *Type*
- (76) *FormalDeclr* ::= *Id ResultType*
 | [*IdList*] *ResultType*
 | *Id* [*IdList*] *ResultType*
- (77) *FormalDecls* ::= *FormalDeclr*
 | *FormalDecls* , *FormalDeclr*
- (78) *FormalList* ::= *Formal*
 | *FormalList* , *Formal*

- (79) *Formals* ::= (*FormalList*[?])
- (80) *FullyQualifiedNames* ::= *Id*
| *FullyQualifiedNames* . *Id*
- (81) *FunctionType* ::= *TypeParams*[?] (*FormalList*[?]) *Guard*[?] => *Type*
- (82) *Guard* ::= *DepParams*
- (83) *Throws* ::= throws *ThrowsList*
- (84) *ThrowsList* ::= *Type*
| *ThrowsList* , *Type*
- (85) *HasResultType* ::= *ResultType*
| <: *Type*
- (86) *HasZeroConstraint* ::= *Type* haszero
- (87) *HomeVariable* ::= *Id*
| this
- (88) *HomeVariableList* ::= *HomeVariable*
| *HomeVariableList* , *HomeVariable*
- (89) *Id* ::= IDENTIFIER
- (90) *IdList* ::= *Id*
| *IdList* , *Id*
- (91) *IfThenElseStmt* ::= if (*Exp*) *Stmt* else *Stmt*
- (92) *IfThenStmt* ::= if (*Exp*) *Stmt*
- (93) *ImplConvOpDecln* ::= *MethMods* operator *TypeParams*[?] (*Formal*) *Guard*[?]
HasResultType[?] *MethodBody*
- (94) *ImportDecln* ::= *SingleTypeImportDecln*
| *TypeImportOnDemandDecln*

- (95) *ImportDeclns* ::= *ImportDecln*
 | *ImportDeclns ImportDecln*
- (96) *InclusiveOrExp* ::= *ExclusiveOrExp*
 | *InclusiveOrExp* | *ExclusiveOrExp*
- (97) *InterfaceBody* ::= {*InterfaceMemberDeclns?*}
- (98) *InterfaceDecln* ::= *Mods?* interface *Id* *TypeParamsI?* *Properties?* *Guard?*
 ExtendsInterfaces? *InterfaceBody*
- (99) *InterfaceMemberDecln* ::= *MethodDecln*
 | *PropMethodDecln*
 | *FieldDecln*
 | *TypeDecln*
- (100) *InterfaceMemberDeclns* ::= *InterfaceMemberDecln*
 | *InterfaceMemberDeclns InterfaceMemberDecln*
- (101) *InterfaceTypeList* ::= *Type*
 | *InterfaceTypeList , Type*
- (102) *Interfaces* ::= implements *InterfaceTypeList*
- (103) *KeywordOp* ::= for
 | if
 | try
 | throw
 | async
 | atomic
 | when
 | finish
 | at
 | continue
 | break
 | aeach
 | while
 | do
- (104) *KeywordOpDecln* ::= *MethMods operator keywordOp TypeParams?* *Formals Guard?*
 Throws? HasResultType? MethodBody

(105) *LabeledStmt* ::= *Id* : *LoopStmt*

(106) *LastExp* ::= *Exp*

(107) *LeftHandSide* ::= *ExpName*
| *FieldAccess*

(108) *Literal* ::= *IntegerLiteral*
| *LongLiteral*
| *ByteLiteral*
| *UnsignedByteLiteral*
| *ShortLiteral*
| *UnsignedShortLiteral*
| *UnsignedIntegerLiteral*
| *UnsignedLongLiteral*
| *FloatingPointLiteral*
| *DoubleLiteral*
| *BooleanLiteral*
| *CharacterLiteral*
| *StringLiteral*
| *null*

(109) *LocVarDecln* ::= *Mods*? *VarKeyword* *VariableDecls*
| *Mods*? *VarDeclsWType*
| *Mods*? *VarKeyword* *FormalDecls*

(110) *LocVarDeclnStmt* ::= *LocVarDecln* ;

(111) *LoopIndex* ::= *Mods*? *LoopIndexDeclr*
| *Mods*? *VarKeyword* *LoopIndexDeclr*

(112) *LoopIndexDeclr* ::= *Id HasResultType*?
| [*IdList*] *HasResultType*?
| *Id* [*IdList*] *HasResultType*?

(113) *LoopStmt* ::= *ForStmt*
| *WhileStmt*
| *DoStmt*
| *AtEachStmt*

- (114) *MethMods* ::= *Mods*[?]
 | *MethMods* *property*
 | *MethMods* *Mod*
- (115) *MethodBody* ::= = *LastExp* ;
 | *Annotations*? *Block*
 | ;
- (116) *MethodDecln* ::= *MethMods* def *Id* *TypeParams*? *Formals* *Guard*? *Throws*?
 HasResultType? *MethodBody*
 | *BinOpDecln*
 | *PrefixOpDecln*
 | *ApplyOpDecln*
 | *SetOpDecln*
 | *ConversionOpDecln*
 | *KeywordOpDecln*
- (117) *MethodInvo* ::= *MethodName* *TypeArgs*? (*ArgumentList*?)
 | *Primary* . *Id* *TypeArgs*? (*ArgumentList*?)
 | super . *Id* *TypeArgs*? (*ArgumentList*?)
 | *ClassName* . super . *Id* *TypeArgs*? (*ArgumentList*?)
 | *Primary* *TypeArgs*? (*ArgumentList*?)
- (118) *MethodInvoStmt* ::= *MethodName* *TypeArgs*? *Arguments*? *ClosureBodyBlock*
 | *Primary* . *Id* *TypeArgs*? *Arguments*? *ClosureBodyBlock*
 | super . *Id* *TypeArgs*? *Arguments*? *ClosureBodyBlock*
 | *ClassName* . super . *Id* *TypeArgs*? *Arguments*? *ClosureBodyBlock*
 | *Primary* *TypeArgs*? *Arguments*? *ClosureBodyBlock*
- (119) *MethodName* ::= *Id*
 | *FullyQualifiedNames* . *Id*
- (120) *Mod* ::= abstract
 | *Annotation*
 | atomic
 | final
 | native
 | private
 | protected
 | public
 | static
 | transient
 | clocked

- (121) *MultiplicativeExp* ::= *RangeExp*
 | *MultiplicativeExp* * *RangeExp*
 | *MultiplicativeExp* / *RangeExp*
 | *MultiplicativeExp* % *RangeExp*
 | *MultiplicativeExp* ** *RangeExp*
- (122) *NamedType* ::= *NamedTypeNoConstraints*
 | *DepNamedType*
- (123) *NamedTypeNoConstraints* ::= *SimpleNamedType*
 | *ParamizedNamedType*
- (124) *NonExpStmt* ::= *Block*
 | *EmptyStmt*
 | *AssertStmt*
 | *SwitchStmt*
 | *DoStmt*
 | *BreakStmt*
 | *ContinueStmt*
 | *ReturnStmt*
 | *ThrowStmt*
 | *TryStmt*
 | *LabeledStmt*
 | *IfThenStmt*
 | *IfThenElseStmt*
 | *WhileStmt*
 | *ForStmt*
 | *AsyncStmt*
 | *AtStmt*
 | *AtomicStmt*
 | *WhenStmt*
 | *AtEachStmt*
 | *FinishStmt*
 | *AssignPropCall*
 | *userStmtPrefix userStmt*
 | *MethodInvoStmt*
- (125) *ObCreationExp* ::= *new TypeName TypeArgs?* (*ArgumentList?*) *ClassBody?*
 | *Primary . new Id TypeArgs?* (*ArgumentList?*) *ClassBody?*
 | *FullyQualifiedname . new Id TypeArgs?* (*ArgumentList?*)
 | *ClassBody?*
- (126) *PackageDecln* ::= *Annotations?* **package** *PackageName* ;

(127) *PackageName* ::= *Id*
 | *PackageName . Id*

(128) *PackageOrTypeName* ::= *Id*
 | *PackageOrTypeName . Id*

(129) *ParamizedNamedType* ::= *SimpleNamedType Arguments*
 | *SimpleNamedType TypeArgs*
 | *SimpleNamedType TypeArgs Arguments*

(130) *PostDecrementExp* ::= *PostfixExp* --

(131) *PostIncrementExp* ::= *PostfixExp* ++

(132) *PostfixExp* ::= *CastExp*
 | *PostIncrementExp*
 | *PostDecrementExp*

(133) *PreDecrementExp* ::= -- *UnaryExpNotPlusMinus*

(134) *PreIncrementExp* ::= ++ *UnaryExpNotPlusMinus*

(135)	<i>PrefixOp</i>	::=
	-	+
	!	~
	^	^
	&	*
	*	/
	%	%

(136) *PrefixOpDecln* ::= *MethMods operator TypeParams?* *PrefixOp* (*Formal*) *Guard?*
HasResultType? *MethodBody*
 | *MethMods operator TypeParams?* *PrefixOp* *this* *Guard?*
HasResultType? *MethodBody*

(137) Primary	::=	here
		[<i>ArgumentList</i> ?]
		Literal
		self
		this
		<i>ClassName</i> . this
		(<i>Exp</i>)
		<i>ObCreationExp</i>
		<i>FieldAccess</i>
		<i>MethodInvo</i>

(138) *Prop* ::= *Annotations?* *Id ResultType*

(139) *PropList* ::= *Prop*
 | *PropList , Prop*

(140) *PropMethodDecln* ::= *Mods*? *property* *Id* *TypeParams*? *Formals* *Guard*?
HasResultType? *MethodBody*
| *Mods*? *property* *Id* *Guard*? *HasResultType*? *MethodBody*

(141) *Properties* ::= (*PropList*)

(142) *RangeExp* ::= *UnaryExp*
 | *RangeExp* .. *UnaryExp*

$(143) \quad RelationalExp \quad ::=$	$ShiftExp$ $HasZeroConstraint$ $SubtypeConstraint$ $RelationalExp < ShiftExp$ $RelationalExp > ShiftExp$ $RelationalExp \leq ShiftExp$ $RelationalExp \geq ShiftExp$ $RelationalExp \text{ instanceof } Type$
---------------------------------------	--

(144) *ResultType* ::= : *Type*

(145) *ReturnStmt* ::= **return** *Exp*? ;

(146) *SetOpDecln* ::= *MethMods operator this TypeParams? Formals = (Formal) Guard?*
HasResultType? MethodBody

- (147) *ShiftExp* ::= *AdditiveExp*
 | *ShiftExp* << *AdditiveExp*
 | *ShiftExp* >> *AdditiveExp*
 | *ShiftExp* >>> *AdditiveExp*
 | *ShiftExp* -> *AdditiveExp*
 | *ShiftExp* <- *AdditiveExp*
 | *ShiftExp* -< *AdditiveExp*
 | *ShiftExp* >- *AdditiveExp*
 | *ShiftExp* ! *AdditiveExp*
- (148) *SimpleNamedType* ::= *TypeName*
 | *Primary* . *Id*
 | *ParamizedNamedType* . *Id*
 | *DepNamedType* . *Id*
- (149) *SingleTypeImportDecln* ::= **import** *TypeName* ;
- (150) *Stmt* ::= *AnnotationStmt*
 | *ExpStmt*
- (151) *StmtExp* ::= *Assignment*
 | *PreIncrementExp*
 | *PreDecrementExp*
 | *PostIncrementExp*
 | *PostDecrementExp*
 | *MethodInvo*
 | *ObCreationExp*
- (152) *StmtExpList* ::= *StmtExp*
 | *StmtExpList* , *StmtExp*
- (153) *StructDecln* ::= *Mods*? **struct** *Id* *TypeParamsI*? *Properties*? *Guard*? *Interfaces*?
ClassBody
- (154) *SubtypeConstraint* ::= *Type* <: *Type*
 | *Type* :> *Type*
- (155) *Super* ::= **extends** *ClassType*
- (156) *SwitchBlock* ::= { *SwitchBlockGroups*? *SwitchLabels*? }

(157) *SwitchBlockGroup* ::= *SwitchLabels BlockStmts*

(158) *SwitchBlockGroups* ::= *SwitchBlockGroup*
 | *SwitchBlockGroups SwitchBlockGroup*

(159) *SwitchLabel* ::= **case** *ConstantExp* :
 | **default** :

(160) *SwitchLabels* ::= *SwitchLabel*
 | *SwitchLabels SwitchLabel*

(161) *SwitchStmt* ::= **switch** (*Exp*) *SwitchBlock*

(162) *ThrowStmt* ::= **throw** *Exp* ;

(163) *TryStmt* ::= **try** *Block Catches*
 | **try** *Block Catches?* *Finally*

(164) *Type* ::= *FunctionType*
 | *ConstrainedType*
 | *Void*

(165) *TypeArgs* ::= [*TypeArgumentList*]

(166) *TypeArgumentList* ::= *Type*
 | *TypeArgumentList , Type*

(167) *TypeDecln* ::= *ClassDecln*
 | *StructDecln*
 | *InterfaceDecln*
 | *TypeDefDecln*
 | ;

(168) *TypeDecls* ::= *TypeDecln*
 | *TypeDecls TypeDecln*

(169) *TypeDefDecln* ::= *Mods?* **type** *Id TypeParams?* *Guard?* = *Type* ;
 | *Mods?* **type** *Id TypeParams?* (*FormalList*) *Guard?* = *Type* ;

(170) *TypeImportOnDemandDecln* ::= **import** *PackageName Or TypeName . ** ;

(171) *TypeName* ::= *Id*
 | *TypeName* . *Id*

(172) *TypeParam* ::= *Id*

(173) *TypeParamIList* ::= *TypeParam*
 | *TypeParamIList* , *TypeParam*
 | *TypeParamIList* ,

(174) *TypeParamList* ::= *TypeParam*
 | *TypeParamList* , *TypeParam*

(175) *TypeParams* ::= [*TypeParamList*]

(176) *TypeParamsI* ::= [*TypeParamIList*]

(177) *UnannotatedUnaryExp* ::= *PreIncrementExp*
 | *PreDecrementExp*
 | + *UnaryExpNotPlusMinus*
 | - *UnaryExpNotPlusMinus*
 | *UnaryExpNotPlusMinus*

(178) *UnaryExp* ::= *UnannotatedUnaryExp*
 | *Annotations UnannotatedUnaryExp*

(179) *UnaryExpNotPlusMinus* ::= *PostfixExp*
 | ~ *UnaryExp*
 | ! *UnaryExp*
 | ^ *UnaryExp*
 | | *UnaryExp*
 | & *UnaryExp*
 | * *UnaryExp*
 | / *UnaryExp*
 | % *UnaryExp*

(180) *UserAsyncStmt* ::= **async** *TypeArgs?* *Arguments?* *ClockedClause?* *ClosureBodyBlock*

(181) *UserAtEachStmt* ::= **ateach** *TypeArgs?* (*FormalList* **in** *ArgumentList?*) *ClosureBodyBlock*
 | **ateach** *TypeArgs?* (*ArgumentList?*) *ClosureBodyBlock*

(182) *UserAtomicStmt* ::= **atomic** *TypeArgs?* *Arguments?* *ClosureBodyBlock*

- (183) *UserAtStmt* ::= **at** *TypeArgs*? (*ArgumentList*?) *ClosureBodyBlock*
- (184) *UserBreakStmt* ::= **break** *TypeArgs*? *Exp*? ;
- (185) *UserCatchClause* ::= **catch** (*FormalList*?) *ClosureBodyBlock*
- (186) *UserCatches* ::= *UserCatchClause*
| *UserCatches UserCatchClause*
- (187) *UserContinueStmt* ::= **continue** *TypeArgs*? *Exp*? ;
- (188) *UserDoStmt* ::= **do** *TypeArgs*? *ClosureBodyBlock* **while** (*ArgumentList*?) ;
- (189) *UserEnhancedForStmt* ::= **for** *TypeArgs*? (*FormalList* **in** *ArgumentList*?) *ClosureBodyBlock*
| **for** *TypeArgs*? (*ArgumentList*?) *ClosureBodyBlock*
- (190) *UserFinallyBlock* ::= **finally** *ClosureBodyBlock*
- (191) *UserFinishStmt* ::= **finish** *TypeArgs*? *Arguments*? *ClosureBodyBlock*
- (192) *UserIfThenStmt* ::= **if** *TypeArgs*? (*ArgumentList*?) *ClosureBodyBlock*
| **if** *TypeArgs*? (*ArgumentList*?) *ClosureBodyBlock* **else** *ClosureBodyBlock*
- (193) *UserStmt* ::= *UserEnhancedForStmt*
| *UserIfThenStmt*
| *UserTryStmt*
| *UserThrowStmt*
| *UserAsyncStmt*
| *UserAtomicStmt*
| *UserWhenStmt*
| *UserFinishStmt*
| *UserAtStmt*
| *UserContinueStmt*
| *UserBreakStmt*
| *UserAtEachStmt*
| *UserWhileStmt*
| *UserDoStmt*

- (194) *UserStmtPrefix* ::= *FullyQualifiedNamespace* .
 | *Primary* .
 | super .
 | *ClassName* . super .
- (195) *UserThrowStmt* ::= throw *TypeArgs*? *Exp*? ;
- (196) *UserTryStmt* ::= try *TypeArgs*? *Arguments*? *ClosureBodyBlock* *UserCatches*?
 UserFinallyBlock?
- (197) *UserWhenStmt* ::= when *TypeArgs*? (*ArgumentList*?) *ClosureBodyBlock*
- (198) *UserWhileStmt* ::= while *TypeArgs*? (*ArgumentList*?) *ClosureBodyBlock*
- (199) *VarDeclWType* ::= *Id* *HasResultType* = *VariableInitializer*
 | [*IdList*] *HasResultType* = *VariableInitializer*
 | *Id* [*IdList*] *HasResultType* = *VariableInitializer*
- (200) *VarDeclsWType* ::= *VarDeclWType*
 | *VarDeclsWType* , *VarDeclWType*
- (201) *VarKeyword* ::= val
 | var
- (202) *VariableDeclr* ::= *Id* *HasResultType*? = *VariableInitializer*
 | [*IdList*] *HasResultType*? = *VariableInitializer*
 | *Id* [*IdList*] *HasResultType*? = *VariableInitializer*
- (203) *VariableDecls* ::= *VariableDeclr*
 | *VariableDecls* , *VariableDeclr*
- (204) *VariableInitializer* ::= *Exp*
- (205) *Void* ::= void
- (206) *WhenStmt* ::= when (*Exp*) *Stmt*
- (207) *WhileStmt* ::= while (*Exp*) *Stmt*

References

- [1] David Bacon. Kava: A Java dialect with a uniform object model for lightweight classes. *Concurrency – Practice and Experience*, 15:185–206, 2003.
- [2] Joseph A. Bank, Barbara Liskov, and Andrew C. Myers. Parameterized types and Java. In *Proceedings of the 24th Annual ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145, 1997.
- [3] William Carlson, Tarek El-Ghazawi, Bob Numrich, and Kathy Yellick. Programming in the Partitioned Global Address Space Model, 2003. Presentation at SC 2003, <http://www.gwu.edu/upc/tutorials.html>.
- [4] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. In *Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.
- [5] J. Gosling, W. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2000.
- [6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.
- [7] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [8] Kiyokuni Kawachiya, Mikio Takeuchi, Salikh Zakirov, and Tamiya Onodera. Distributed garbage collection for managed X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 5:1–5:11, New York, NY, USA, 2012. ACM.
- [9] Jose E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Marc Snir, and Richard D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–, 2000.
- [10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc, 2 edition, January 2011.

- [11] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased performance for in-memory Hadoop jobs. In *Proceedings of VLDB Conference*, VLDB '12, 2012.
- [12] A. Skjellum, E. Lusk, and W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1999.
- [13] Mikio Takeuchi, David Cunningham, David Grove, and Vijay Saraswat. Java interoperability in managed X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, X10 '13, pages 39–46, New York, NY, USA, 2013. ACM.
- [14] Mikio Takeuchi, Yuki Makino, Kiyokuni Kawachiya, Hiroshi Horii, Toyotaro Suzumura, Toshio Suganuma, and Tamiya Onodera. Compiling X10 to Java. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 3:1–3:10, New York, NY, USA, 2011. ACM.
- [15] Mikio Takeuchi, Salikh Zakirov, Kiyokuni Kawachiya, and Tamiya Onodera. Fast method dispatch and effective use of primitives for reified generics in managed X10. In *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, X10 '12, pages 4:1–4:7, New York, NY, USA, 2012. ACM.
- [16] K. A. Yellick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11–13):825–836, 1998.

(), 107
()=, 107
++, 164
--, 164
:>, 180
<:, 68, 180
==, 168
? :, 167
DistArray, 232
 creation, 232
Object, 33
as, 176
ateach, 207
instanceof, 179
this, 158

acc, 289
activity, 204
 blocked, 204
 creating, 205
 initial, 207
 running, 204
 terminated, 204
allocation, 171
annotations, 236
 type annotations, 50
anonymous class, 145
Any
 structs, 149
application operator, 107
array, 222, 229
 access, 233
 constant promotion, 233
 constructor, 229
 distributed, 232
 operations on, 230
 pointwise operations, 234
 reductions, 234
 restriction, 233
 scans, 235
assert, 192
assignment, 163
 definite, 251

assignment operator, 107
async, 205
 clocked, 220
at, 197
 blocking copying, 202
 copying, 200
 GlobalRef, 202
 transient fields and, 201
ateach, 207
atomic, 208
 conditional, 210
auto-boxing
 struct to interface, 149

block, 185
Boolean, 150
 literal, 24
Boolean operations, 167
break, 187
Byte, 150

call, 161
 function, 161
 method, 161
 super, 162
cast, 172, 176
 to generic type, 45
catch, 194
Char, 150
char
 literal, 25
class, 33, 83
 anonymous, 145
 construction, 171
 field, 69
 inner, 141
 instantiation, 171
 invariant, 120
 nested, 141
 static nested, 141
class declaration, 33
class invariant, 120
class invariants, 120
clock, 215
 advanceAll, 218
 clocked statements, 217

ClockUseException, 216–218
creation, 217
drop, 219
operations on, 217
resume, 218
clocked
 async, 220
 finish, 220
clocked finish
 nested, 221
closure, 153
 parametrized, 35
coercion, 138, 175
 explicit, 176
 subsumption, 175
 user-defined, 177
comment, 23
concrete type, 35
conditional expression, 167
constrained type, 39
constraint, 41
 entailment, 44
 permitted, 41
 semantics, 43
 subtyping, 44
 syntax, 41
constraint solver
 incompleteness, 44
constructor, 98, 124
 and invariant, 121
 closure in, 130
 generated, 98
 inner classes in, 130
 parametrized, 35
container, 29
continue, 188
conversion, 175, 178
 numeric, 178
 string, 178
 user-defined, 178
widening, 178

declaration
 class declaration, 33
 interface declaration, 34
 reference class declaration, 33

type, 36
decrement, 164
default value, 49
definite assignment, 251
definitely assigned, 251
definitely not assigned, 251
dependent type, 39
destructuring, 66
DistArray, 232
 creation, 232
distributed array, 232
 creation, 232
distribution, 230
 block, 231
 constant, 231
 operations, 231
restriction
 range, 232
 region, 232
 unique, 231
do, 190
documentation type declaration, 68
Double, 150
double
 literal, 25
dynamic checks, 299

equality, 168
 function, 157
Exception, 193
 unchecked, 153
exception, 193, 194, 205
 model, 205
 rooted, 205
expression, 158
 allowed in constraint, 41
 conditional, 167
 constraint, 41
extends, 121

field, 69, 85
 access to, 159
 hiding, 85
 initialization, 85
 qualifier, 86
 static, 86

transient, 86, 199, 201
final, 92
finally, 194
finish, 206
 clocked, 220
 nested clocked, 221
FIRST_PLACE, 196
Float, 150
float
 literal, 25
for, 190
formal parameter, 67
function, 153
 ==, 157
 application, 154
 at(Place), 157
 equality, 157
 equals, 157
 hashCode, 157
 home, 157
 literal, 153
 outer variables in, 156
 toString, 157
 typeName, 157
types, 47

generic type, 39
guard, 120
 on method, 92

here, 196
hiding, 71

identifier, 23
if, 188
immutable variable, 64
implements, 121
implicit coercion, 138
implicitly non-escaping, 126
import, 75
import,type definitions, 37
increment, 164
initial activity, 207
initial value, 65
initialization, 65, 124
 of field, 85

- static, 100
- inner class, 141
 - constructor, 143
 - extending, 142
- instanceof, 179
- instantiation, 171
- Int, 150
 - literal, 25
- integers
 - unsigned, 150
- interface, 34, 77
 - field definition in, 80
- interface declaration, 34
- interoperability, 239
- invariant
 - and constructor, 121
 - checked, 121
 - class, 120
 - type, 120
- invocation, 161
 - function, 161
 - method, 161
- keywords, 24
- label, 186
- literal, 24, 158
 - Boolean, 24
 - char, 25
 - double, 25
 - float, 25
 - function, 155
 - integer, 25
 - string, 26
- local variable, 68
- Long, 150
- method, 89
 - calling, 161
 - final, 92
 - generic instance, 92
 - guard, 92
 - implicitly non-escaping, 126
 - instance, 89
 - invoking, 161
 - non-escaping, 126

NonEscaping, 126
overloading, 95
parametrized, 35
property, 93
resolution, 133
signature, 89
static, 89
which one will get called, 133
method resolution
 implicit coercions and, 138

name, 70
namespace, 70
native code, 239
new, 171
non-escaping, 126
 implicitly, 126
NonEscaping, 126
null, 25, 33
nullary constructor, 65
numeric operations, 164
numeric promotion, 164

object, 83
 constructor, 124
 field, 69, 85
 literal, 25
obscuring, 71
Offers, 289
offers, 289, 297
operation
 numeric, 164
operator, 26, 102
 user-defined, 102
overloading, 89

package, 70
parameter, 67
 val, 90
 var, 90
place, 196
 changing, 197
 FIRST_PLACE, 196
point, 224
 syntax, 224
polymorphism, 89

primitive types, 150
private, 73
promotion, 164
properties
 acyclic, 89
property, 34, 87
 initialization, 87
property method, 93
protected, 73
public, 73

qualifier
 field, 86

Rail, 222
rail
 construction, 181
 literal, 181
region, 227
 convex, 228
 intersection, 228
 operations, 228
 product, 228
 sub-region, 228
 syntax, 227
return, 192
root activity, 204

self, 39
shadowing, 70
Short, 150
signature, 89
statement, 183
statement label, 186
static nested class, 141
static nested struct, 151
STATIC_CHECKS, 299
string
 concatenation, 166
 literal, 26
struct, 147
 auto-boxing, 149
 casting to interface, 149
 construction, 171
 constructor, 124
 declaration, 148

field, 69
instantiation, 171
static nested, 151
subtype
 test, 180
subtyping, 50
supercall, 162
switch, 189

termination, 204
 abrupt, 193
 global, 204
 local, 204
 normal, 193
this, 158
throw, 193
transient, 86, 199, 201
try, 194
type
 annotated, 50
 class, 33
 coercion, 175
 concrete, 35
 constrained, 39
 conversion, 175, 178
 default value, 49
 definitions, 36
 dependent, 39
 function, 47
 generic, 35, 39
 inference, 54
 interface, 34
 parameter, 35
type conversion, 172
 implicit, 106
 user-defined, 106
type equivalence, 50
type inference, 54
type invariants, 120
type system, 30
type-checking
 extends clause, 121
 implements clause, 121
types, 29
 primitive, 150
 unsigned, 150

UByte, 150
UInt, 150
ULong, 150
unit type, 32
unsigned, 150
UShort, 150

val, 64, 184
var, 184
variable, 63
 declaration, 184
 immutable, 64
 local, 68
 val, 64
variable declaration, 64
variable declarator
 destructuring, 66
variable name, 23
VERBOSE_CHECKS, 299
void, 55

when, 210
 timing, 212
while, 189
white space, 23

A Deprecations

X10 version 2.4 has a few relics of previous versions, code that is being used by libraries but is not intended for general programming. They should be ignored.

These are:

- `acc` variables.
- The `offers` clause for use with collecting finish.
- The grammar allows covariant and contravariant type parameters, marked by `+` and `-`:

```
class Variant[X, +Y, -Z] {}
```

X10 does not support these in any other way.

- The syntax allows for a few Java-isms, such as `c.class` and `super.class`, which are not used.

B Change Log

B.1 Changes from X10 v2.5

To resolve parsing ambiguities, the property keyword is now mandatory to introduce property methods.

The X10 language now supports trailing closures. A trailing closure is a closure block that is written and after the parentheses of a function call. This closure block is passed as a closure without argument to the function.

B.2 Changes from X10 v2.4

Although there were no backwards incompatible language changes between X10 v2.4 and X10 v2.5, a few core class library APIs did have backwards incompatible changes. These changes were driven by our experience with Resilient and Elastic X10 and are designed to better support X10 computations over a dynamically varying number of Places. In summary,

1. Static constants such as `PlaceGroup.WORLD`, and `Place.MAX_PLACES` were removed. They are replaced by `Place.places()` and `Place.numPlaces()` which return values that represent the current view on the dynamically changeable set of Places available to the computation.
2. The removal of iteration functionality (`next` and `prev`) from `Place`. This functionality is now provided only through `PlaceGroup`.
3. The addition of `PlaceTopology` to provide a more flexible set of APIs describing the topological relationships of Places.

In addition, the `put` and `get` methods of `x10.util.Map` were changed to no longer wrap their return value in an instance of `x10.util.Box`. This improves the common case efficiency of map usage, but does require that the Values stored in Map satisfy the `haszero` constraint.

B.3 Changes from X10 v2.3

X10 v2.4 is not backwards compatible with X10 v2.3. The motivation for making backwards incompatible language changes with this release of X10 is to significantly improve the ability of the X10 programmer to exploit the expanded memory capabilities of modern computer systems. In particular, X10 v2.4 includes an extensive redesign of arrays and a change of the default type of unqualified integral literals (*e.g.* 2) from `Int` to `Long`. Taken together these two changes enable natural exploitation of large memories via 64-bit addressing and `Long`-based indexing of arrays and similar data structures.

B.3.1 Integral Literals

The default type of unqualified integral literals was changed from `Int` to `Long`.

The qualifying suffix `n` and `un` are used to indicate `Int` and `UIInt` literals respectively. The suffix `u` is now interpreted as indicating a `ULong` literal.

B.3.2 Arrays

An extensive redesign of the X10 abstractions for arrays is the major new feature of the X10 v2.4 release. Although this redesign only involved very minor changes to the actual X10 language specification, the core class libraries did change significantly. As mentioned above, the driving motivation for the change was a long-contemplated strategic decision to shift to from `Int`-based (32-bit) to `Long`-based (64-bit) indexing for all X10 arrays. This change enables X10 to better utilize the rapidly expanding memory capacity and 64-bit address space found on modern machines. For consistency, the `id` field of `x10.lang.Place` and the `size` and indexing-related APIs of the `x10.util` collection hierarchy were also changed from `Int` to `Long`.

Once this inherently backwards-incompatible decision was made, the X10 team decided to do a larger rethinking of all of X10's array implementations to introduce a new time and space optimal implementation of zero-based, dense, rectangular multi-dimensional arrays. This new implementation, in the `x10.array` package, is intended to provide the best possible performance for the common-case it supports. The previous, more general array implementation is still available, but has been relocated to a new package `x10.array.regionarray`. In addition, the `x10.lang.Rail` class was re-introduced as a separate class in its own right and provides the intrinsic indexed storage abstraction on which both array packages are built. The intent is that the combination of `Rail`, `x10.array` and `x10.regionarray` provide a spectrum of array abstractions that capture common usage patterns and enable appropriate trade-offs between performance and flexibility.

In more detail the major array-related changes made in the X10 v2.4 release are

1. The class `x10.lang.Rail` was introduced. It provides an efficient one-dimensional, zero-based, densely indexed array implementation. `Rail` will provide the best performance and is the preferred implementation of this basic abstraction.
2. The array literal syntax `[1, 2, 3]` is now defined to create a `Rail` instead of an `Array`.
3. The main method signature is changed from `Array[String]` to `Rail[String]`.
4. `x10.util.IndexedMemoryChunk` has removed from the X10 standard library.
5. To enable usage of classes from both `x10.array` and `x10.regionarray`, the package `x10.array` is no longer auto-imported by the X10 compiler.
6. Most classes in the `x10.array` package in the X10 v2.3 release were relocated to the `x10.regionarray` package in v2.4. A few classes like `Point` and `PlaceGroup` were moved to the `x10.lang` package instead.
7. `Point`, `Region`, `Dist`, etc. were all updated to support long-based indexing by consistently changing indexing related fields and methods from `Int` to `Long`.

B.3.3 Other Changes from X10 v2.3

1. The custom serialization protocol was changed to operate in terms of new user-level classes `x10.io.Serializer` and `x10.io.Deserializer`. The `serialize` method of the `xcdx10.io.CustomSerialization` interface now takes a `Serializer` as an argument. The custom deserialization constructor for a class takes a `Deserializer`. The `x10.io.SerialData` class used by the X10 v2.3 custom serialization protocol has been removed from the class library.
2. A constraint was added to `PlaceLocalHandle` that types used to instantiate a `PlaceLocalHandle` must satisfy both the `isref` and `haszero` constraints.
3. The `x10.util.Team` API was revised by (a) removing the `endpoint` argument from all API calls and (b) to operate on `Rail` and `xcd'Long`` where appropriate.

B.4 Changes from X10 v2.2

1. In previous versions of X10 static fields were eagerly initialized in `Place 0` and the resulting values were serialized to all other places before execution of the user main function was started. Starting with X10 v2.2.3, static fields are lazily initialized on a per-Place basis when the field is first read by an activity executing in a given Place.
2. The new syntax `T isref` for some type `T` will hold if `T` is represented by a pointer at runtime. This is similar to the type constraint `T haszero`. `T isref` is true for `T` that are function types, classes, and all values that have been cast to

interfaces (including boxed structs). `T isref` is used in the standard library, e.g. for the `GlobalRef[T]` and `PlaceLocalHandle[T]` APIs.

3. `x10.lang.Object` is gone, there is now no single class that is the root of the X10 class hierarchy.
 - If, for some reason, you were explicitly extending `Object`, don't do that anymore.
 - If you were doing `new Object()` to get a fresh value, use `new Empty()` instead.
 - If you were using `Object` as a supertype, use `Any` (the one true supertype).
 - If you were using the type constraint `T <: Object` to disallow structs, use `T isref` instead.
4. The exception hierarchy has changed, and checked exceptions have been reintroduced. The 'throws' annotation is required on methods, as in Java. It is not supported on closures, so checked exceptions cannot be thrown from a closure. The exception hierarchy has been chosen to exist in a 1:1 relationship with Java's. However, unlike Java, we prefer using unchecked exceptions wherever possible, and this is reflected in the naming of the X10 classes. The following classes are all in the `x10.lang` package.
 - `CheckedThrowable` (mapped to `java.lang.Throwable`)
 - `CheckedException` extends `CheckedThrowable` (mapped to `java.lang.Exception`)
 - `Exception` extends `CheckedException` (mapped to `java.lang.RuntimeException`)
 - `Error` extends `CheckedThrowable` (mapped to `java.lang.Error`)

Anything under `CheckedThrowable` can be thrown using the `throw` statement. But anything that is not under `Exception` or `Error` can only be thrown if it is caught by an enclosing `try/catch`, or it is thrown from a method with an appropriate `throws` annotation, as in Java.

`RuntimeException` is gone from X10. Use `Exception` instead.

All the exceptions in the standard library are under `Exception`, except `AssertionError` and `OutOfMemoryException`, which are under `Error` (as in Java). This means all exceptions in the standard library remain unchecked.

B.5 Changes from X10 v2.1

1. Covariance and contravariance are gone.
2. Operator definitions are regularized. A number of new operator symbols are available.
3. The operator `in` is gone. `in` is now only a keyword.

4. Method functions and operator functions are gone.
5. `m..n` is now a type of struct called `IntRange`.
6. `for(i in m..n)` now works. The old forms, `for((i) in m..n)` and `for([i] in m..n)`, are no longer needed.
7. `(e as T)` now has type `T`. (It used to have an identity constraint conjoined in.)
8. `vars` can no longer be assigned in their place of origin. Use a `GlobalRef[Cell[T]]` instead. We'll have a new idiom for this in 2.3.
9. The `-STATIC_CALLS` command-line flag is now `-STATIC_CHECKS`.
10. Any string may be written in backquotes to make an identifier: '`while`'.
11. The `next` and `resume` keywords are gone; they have been replaced by static methods on `Clock`.
12. The typed array construction syntax `new Array[T][t1,t2]` is gone. Use `[t1 as T, t2]` (if just plain `[t1,t2]` doesn't work).

B.6 Changes from X10 v2.0.6

This document summarizes the main changes between X10 2.0.6 and X10 2.1. The descriptions are intended to be suggestive rather than definitive; see the language specification for full details.

B.6.1 Object Model

1. Objects are now local rather than global.
 - (a) The `home` property is gone.
 - (b) `at(P)S` produces deep copies of all objects reachable from lexically exposed variables in `S` when it executes `S`. (**Warning:** They are copied even in `at(here)S`.)
2. The `GlobalRef[T]` struct is the only way to produce or manipulate cross-place references.
 - (a) `GlobalRef`'s have a `home` property.
 - (b) Use `GlobalReffoo` to make a new global reference.
 - (c) Use `myGlobalRef()` to access the object referenced; this requires `here == myGlobalRef.home`.
3. The `!` type modifier is no longer needed or present.

4. `global` modifiers are now gone:
 - (a) `global` methods in *interfaces* are now the default.
 - (b) `global fields` are gone. In some cases object copying will produce the same effect as global fields. In other cases code must be rewritten. It may be desirable to mark nonglobal fields `transient` in many cases.
 - (c) `global methods` are now marked `@Global` instead. Methods intended to be non-global may be marked `@Pinned`.

B.6.2 Constructors

1. `proto` types are gone.
2. Constructors and the methods they call must satisfy a number of static checks.
 - (a) Constructors can only invoke `private` or `final` methods, or methods annotated `@NonEscaping`.
 - (b) Methods invoked by constructors cannot read fields before they are written.
 - (c) The compiler ensures this with a detailed protocol.
3. It is still impossible for X10 constructors to leak references to `this` or observe uninitialized fields of an object. Now, however, the mechanisms enforcing this are less obtrusive than in 2.0.6; the burden is largely on the compiler, not the programmer.

B.6.3 Implicit clocks for each finish

Most clock operations can be accomplished using the new implicit clocks.

1. A `finish` may be qualified with `clocked`, which gives it a clock.
2. An `async` in a `clocked finish` may be marked `clocked`. This registers it on the same clock as the enclosing `finish`.
3. `clocked async S` and `clocked finish S` may use `next` in the body of `S` to advance the clock.
4. When the body of a `clocked finish` completes, the `clocked finish` is dropped from the clock. It will still wait for spawned `asyncs` to terminate, but such `asyncs` need to wait for it.

B.6.4 Asynchronous initialization of val

`vals` can be initialized asynchronously. As always with `vals`, they can only be read after it is guaranteed that they have been initialized. For example, both of the `prints` below are good. However, the commented-out `print` in the `async` is bad, since it is possible that it will be executed before the initialization of `a`.

```
val a:Int;
finish {
    async {
        a = 1;
        print("a=" + a);
    }
    // WRONG: print("a=" + a);
}
print("a=" + a);
```

B.6.5 Main Method

The signature for the `main` method is now:

```
def main(Array[String]) {...}
```

or, if the arguments are actually used,

```
def main(argv: Array[String](1)) {...}
```

B.6.6 Assorted Changes

1. The syntax for destructuring a point now uses brackets rather than braces: `for([i] in 1..10)`, rather than the prior `(i)`.

B.6.7 Safety of atomic and when blocks

1. Static effect annotations (`safe`, `sequential`, `nonblocking`, `pinned`) are no longer used. They have been replaced by dynamic checks.
2. Using an inappropriate operation in the scope of an `atomic` or `when` construct will throw `IllegalOperationException`. The following are inappropriate:
 - `when`
 - `resume()` or `next` on clocks
 - `async`
 - `Future.make()`, or `Future.force()`.
 - `at`

B.6.8 Removed Topics

The following are gone:

1. `foreach` is gone.
2. All vars are effectively shared, so `shared` is gone.
3. The place clause on `async` is gone. `async (P) S` should be written `at(P) async S`.
4. Checked exceptions are gone.
5. `future` is gone.
6. `await ... or ...` is gone.
7. `const` is gone.

B.6.9 Deprecated

The following constructs are still available, but are likely to be replaced in a future version:

1. `ValRail`.
2. `Rail`.
3. `ateach`
4. `offers`. The `offers` concept was experimental in 2.1, but was determined inadequate. It has not been removed from the compiler yet, but it will be soon. In the meantime, traces of it are still visible in the grammar. They should not be used and can safely be ignored.

B.7 Changes from X10 v2.0

Some of these changes have been made obsolete in X10 2.2.

- Any is now the top of the type hierarchy (every object, struct and function has a type that is a subtype of Any). Any defines `home`, `at`, `toString`, `typeName`, `equals` and `hashCode`. Any also defines the methods of `Equals`, so `Equals` is not needed any more.
- Revised discussion of incomplete types.
- The manual has been revised and brought into line with the current implementation.

B.8 Changes from X10 v1.7

The language has changed in the following ways. Some of these changes have been made obsolete in X10 2.2.

- **Type system changes:** There are now three kinds of entities in an X10 computation: objects, structs and functions. Their associated types are class types, struct types and function types.

Class and struct types are called *container types* in that they specify a collection of fields and methods. Container types have a name and a signature (the collection of members accessible on that type). Collection types support primitive equality == and may support user-defined equality if they implement the `x10.lang.Equals` interface.

Container types (and interface types) may be further qualified with constraints.

A function type specifies a set of arguments and their type, the result type, and (optionally) a guard. A function application type-checks if the arguments are of the given type and the guard is satisfied, and the return value is of the given type. A function type does not permit == checks. Closure literals create instances of the corresponding function type.

Container types may implement interfaces and zero or more function types.

All types support a basic set of operations that return a string representation, a type name, and specify the home place of the entity.

The type system is not unitary. However, any type may be used to instantiate a generic type.

There is no longer any notion of value classes. value classes must be re-written into structs or (reference) classes.

- **Global object model:** Objects are instances of classes. Each object is associated with a globally unique identifier. Two objects are considered identical == if their ids are identical. Classes may specify global fields and methods. These can be accessed at any place. (global fields must be immutable.)
- **Proto types.** For the decidability of dependent type checking it is necessary that the property graph is acyclic. This is ensured by enforcing rules on the leakage of `this` in constructors. The rules are flexible enough to permit cycles to be created with normal fields, but not with properties.
- Place types. Place types are now implemented. This means that non-global methods can be invoked on a variable, only if the variable's type is either a struct type or a function type, or a class type whose constraint specifies that the object is located in the current place.

There is still no support for statically checking array access bounds, or performing place checks on array accesses.

C Options

C.1 Compiler Options: Common

The X10 compilers have many useful options.

C.1.1 Optimization: -O or -optimize

This flag causes the compiler to generate optimized code.

C.1.2 Debugging: -DEBUG=boolean

This flag, if true, causes the compiler to generate debugging information. It is false by default.

C.1.3 Call Style: -STATIC_CHECKS, -VERBOSE_CHECKS

By default, if a method call *could* be correct but is not *necessarily* correct, the X10 compiler generates a dynamic check to ensure that it is correct before it is performed. For example, the following code:

```
def use(n:Int{self == 0}) {}
def test(x:Int) {
    use(x); // creates a dynamic cast
}
```

compiles even though it is possible that $x \neq 0$ when `use(x)` is called. In this case, the compiler inserts a cast, which has the effect of checking that the call is correct before it happens:

```
def use(n:Int{self == 0}) {}
def test(x:Int) {
    use(x as Int{self == 0});
}
```

The compiler produces a warning that it inserted some dynamic casts. If you then want to see what it did, use `-VERBOSE_CHECKS`.

You may also turn on strict static checking, with the `-STATIC_CHECKS` flag. With static checking, calls that cannot be proved correct statically will be marked as errors.

C.1.4 Help: `-help` and `--help`

These options cause the compiler to print a list of all command-line options.

C.1.5 Source Path: `-sourcepath path`

This option tells the compiler where to look for X10 source code.

C.1.6 Output Directory: `-d directory`

This option tells the compiler to produce its output files in the specified directory.

C.1.7 Executable File: `-o path`

This option tells the compiler what path to use for the executable file.

C.2 Compiler Option: C++

The C++ compilation command `x10c++` has the following option as well.

C.2.1 Runtime: `-x10rt impl`

This option tells which runtime implementation to use. The choices are `sockets`, `standalone`, `pami`, `mpi`, and `bgas_bgp`.

C.3 Compiler Option: Java

The Java compilation command `x10c` has the following option as well.

C.3.1 Class Path: `-classpath path`

This option is used in conjunction with the Java interoperability feature to tell the compiler where to look for Java .class files that may be used by the X10 code being compiled.

C.4 Execution Options: Java

The Java execution command `x10` has a number of options as well.

C.4.1 Class Path: `-classpath path`

This option specifies the search path for class files.

C.4.2 Library Path: `-libpath path`

This option specifies the search path for native libraries.

C.4.3 Heap Size: `-mssize` and `-mxsize`

Sets the minimum and maximum size of the heap.

C.4.4 Stack Size: `-sssize`

Sets the maximum size of the stack.

C.4.5 Places: `-np count`

Specify the number of places.

C.4.6 Hosts: `-host host1,host2,...` or `-hostfile file`

Specify the hosts either by the list of host names or the host file.

C.4.7 Runtime: `-x10rt impl`

This option tells which runtime implementation to use. The choices are `sockets`, `JavaSockets` (experimental), and `mpi` (experimental).

C.4.8 Help: `-h`

Prints a listing of all execution options.

C.5 Running X10

An X10 application is launched either by a direct invocation of the generated executable or using a launcher command. The specification of the number of places and the mapping from places to hosts is transport specific and discussed in §C.6 for Managed X10 (Java back end) and §C.7 for Native X10 (C++ back end). For distributed runs, the x10 distribution (libraries) and the compiled application code (binary or bytecode) are expected to be available at the same paths on all the nodes.

Detailed, up-to-date documentation may be found at <http://x10-lang.org/documentation/practical-x10-programming/x10rt-implementations.html>

C.6 Managed X10

Managed X10 applications are launched using the x10 script followed by the qualified name of the main class.

```
x10c HelloWholeWorld.x10
x10 HelloWholeWorld
```

The main purpose of the x10 script is to set the jvm classpath and the `java.library.path` system property to ensure the x10 libraries are on the path.

C.7 Native X10

On most platforms and for most transports, X10 applications can be launched by invoking the generated executable.

```
x10c++ -o HelloWholeWorld HelloWholeWorld.x10
./HelloWholeWorld
```

On cygwin, X10 applications must be launched using the runx10 script followed by the name of the generated executable.

```
x10c++ -o HelloWholeWorld HelloWholeWorld.x10
runx10 HelloWholeWorld
```

The purpose of the runx10 script is to ensure the x10 libraries are on the path.

D Acknowledgments and Trademarks

The X10 language has been developed as part of the IBM PERCS Project, which is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.