Contents

Advanced Machine Learning

First Lecture

A *well-posed learning problem* occurs when a computer program learns from experience E, w.r.t some task T and some performance measure P. learning occurs if its performance on T, as measured by P, improves with experience E.

Stages of intelligent systems:

1. Expert systems (1960s–1980s)
2. Unsupervised learning (1980s–1990s)
3. Data-driven learning (2000s)

This course will focus on the third stage. Increases in flexibility and model expressiveness correspond to exponential increases in the number of parameters and computational complexity, in terms of FLOPs and dataset size.

Second Lecture

The second lecture recaps a number of basic ML concepts such as SGD, Losses, Regularization, etc…

We skip these notes since they are already covered in so many previous courses, they are to be considered trivial at this point

Backpropagation

We take notes on backpropagation since it is usually always expressed in a different formalism in every course.

Big Data Computing

First Lecture

Classroom: phdlyrl

Second Lecture

What's Big Data?

It is sometimes used as a buzzword, but it describes an actual phenomenon.

5 V's:

1. Value: Extracting knowledge from data.
2. Volume: The amount of data. (Measuring in terabytes, petabytes, exabytes)
3. Variety: Different formats, structured (SQL tables) and Unstructured (text, images, videos).
4. Velocity: The speed at which data is generated. (usually in real-time, you can't parse it all)
5. Veracity: The quality of the data. (Is it reliable?)

Scaling

When scaling, you have two options:

- Scaling *up*, you buy better hardware for your monolithic machine, however, moore's law is slowing down, and you will get diminishing returns, in general, you will not be able to keep up with increases in Volume/Velocity.
- Scaling *out*, you buy more machines, and distribute the workload, this is the way to go for big data, since this allows for exponential growth.

Scaling *out*, or *horizontal scaling*, also allows you for more flexibility, as once you deal with the orchestration of the workload, you can dynamically increase/reduce the number of allocated machines depending on a rolling cost/benefit analysis.

Network Bottlenecks

A bottleneck during a request is generated when:

- There is a skew in the ratio of communication/computation
- There is a non-overlappable part of communication/computation

Third Lecture

In this lecture, we will take a look at a use case of Big Data, *Distributed Deep Learning*.

Naturally, Deep Learning performs better with more data, this is actually an empirically derived law that has come up in recent OpenAI research: exponentially more data leads to a linear increase in model performance, irrespective of the model architecture (assuming a reasonable architecture).

How does a DNN work

To put it simply, we want to build a function estimator over a dataset D:

$$\hat{f} : D \to Y$$

Such that:

$$\hat{f}(x) \approx y$$

Where x is the input data, and y is the output data.

To do this, we use iterative optimization, where we start with a random function, and we iteratively update it to minimize the error between the output of the function and the actual output. We use the data from the dataset to *fit* our model, under an *inductive* bias, which is an assumption that the *test* data is similarly distributed (so that our fit will generalize).

Forward and Backward passes

Our iterative fitting process can be split in two passes:

1. Forward pass, where we compute the output of the model given the input.
2. Backward pass, where we compute the gradient of the error with respect to the model parameters.

We use the obtained gradient to refine the model parameters, and we repeat the process until we reach a stopping criterion (which indicates that the model has converged).

In general, most of these operations are matrix operations, which can be parallelized on the GPU in an efficient manner.

Parallelism Models

There are three orthogonal parallelism directions in which we can distribute the workload:

1. Data Parallelism
2. Pipeline Parallelism
3. Model Parallelism

The last two together are also known as *Model Parallelism.*

Data Parallelism

The simplest one is *Data Parallelism.* We split the data in n subsets, and we train n models in parallel, each on a different subset. We then perform *gradient aggregation* across these sets, and we update the model parameters.

In this way all the models are updated in parallel, and the only bottleneck is the gradient aggregation.

Gradient aggregation is *quite* costly, it contributes to a significant portion (20% to 50%) of total training time, with the portion increasing as the available bandwidth decreases.

Naturally, this means that this operation has been the subject of heavy optimization, algorithmically and infrastructurally.

Parameter Servers

An initial solution was to use a *Parameter Server*, which is a server that holds the model parameters, and is responsible for the aggregation of the gradients. This does *not* scale well, as it introduces a single point of failure, and a bottleneck in the network.

Splitting the Aggregation

We can split the aggregation into K servers, which will each aggregate a subset of the gradients, and then we will aggregate the results of these servers. This is a more scalable solution, but it still has a bottleneck in the aggregation.

Naturally, as long as K is small, this is a good solution, but as K grows, the aggregation time will grow linearly, and the bottleneck will be reached.

Cost Model

A cost model on a K-PS (Parameter Server) system is:

$$\max\left(\frac{n}{\beta}, \frac{pn}{k\beta}\right)$$

Where n is the number of bytes in the gradient vector, p is the number of GPUs/workers, k is the number of servers, and β is the bandwidth of the network.

Data Parallelism Recap

In data parallelism, each worker must keep a copy of the model, therefore you can also encounter out-of-memory errors, as the model grows in size.

Data parallelism is simple and easy to implement, but has limited potential to be scalable.

Replication of the model is also a waste of resources, as we are just keeping copies around, and we are not using them. It should only be used as a last resort if we have more resources than we can use after applying the other parallelism models, and we still want idle time to be minimized.

Model Parallelism

In order to train models that do *not* fit into a single GPU, we can use *Model Parallelism*. in essence, model parallelism is the process of splitting the model into k parts, and training each part on a different GPU.

Now, the bottleneck is the GPU-to-GPU communication, since we have a sequential dependency between the parts of the model (each part depends on the output of the previous part). This happens in both the forward and backward passes.

Pipelining

Naturally, you can reduce the *bubble overhead* by pipelining the computation, so that the forward pass of the next part can start before the backward pass of the previous part has finished. The same concept can be applied on subsections of the forward/backward passes that use layers that are distributed across a network.

Operator Parallelism

In the case of *Operator Parallelism*, we split the model into operators, and we distribute the operators across the network. This is a more fine-grained approach, and it is usually used in conjunction with the other two parallelism models.

Here, the bottleneck is the communication between the operators, and the synchronization of the operators, however, we reduce the idle time of GPUs drastically.

Hierarchical Parallelism

Naturally, *Operator Parallelism* is applied on GPUs that are on the same node, so that the available bandwidth is maximized. Whereas *Model Parallelism* is applied on GPUs that are on different nodes, where communication is more costly.

This allows us to maximize the utilization of the available resources, and minimize the communication overhead.

Hybrid Parallelism

Hybrid parallelism (also called 3D-Parallelism), combines all three techniques, and offers maximum scalability and potential for resource utilization.

In general, one must be working with a sufficient quantity of data to *justify* the use of these techniques, as the overhead of the parallelism models can be significant, and can outweigh the benefits of the parallelism.

Challenges

Compute-Communication Overlap

When employing hybrid-parallelism, we perform computations and communications in a very fine-grained manner. The reason is that we want to perform gradient aggregation after each layer, so that we minimize the *bubbling* in our pipeline (we don't want to keep anyone waiting).

We want to maximize the overlap in windows of computation and communication, so that we can keep the GPUs and network as busy as possible.

Gradient Compression

Since DL is a stochastic process, one can afford to have some noise in the gradients, and still converge to a good solution. This allows us to compress the gradients, and reduce the communication overhead.

We have various techniques:

1. Quantization: Reducing the precision of the gradients.
2. Rank Decomposition: Decomposing the gradients into a low-rank matrix.
3. Sparsification: Sample the top k gradients, and send only those.

In sparsification, we can use various techniques to select the top k gradients, such as:

1. Top-k: Select the top k gradients by magnitude.
2. Thresholding: Select the gradients that are above a certain threshold λ.

There are theoretical proofs that show that asymptotically, model convergence is not affected by gradient sparsification, as long as the sparsification is done in a *smart* way.

Compression Tradeoff By compressing gradients, we lower the cost of a single iteration, but we increase the number of iterations needed to converge. This is a tradeoff that must be taken into account when designing a distributed deep learning system.

We have a real speed up if:

$$T_{comp} \cdot I_{comp} < T_{orig} \cdot I_{orig}$$

Aggregation Consistency

We observe that gradient accumulation acts as a *barrier*, which means that if a single GPU is, for some reason, considerably slower than the others, it will slow down the whole process. The probability of this happening is significant in large-scale systems, and it is a problem that must be addressed.

Possible solutions include: * Gradient Accumulation: Accumulate the gradients over multiple iterations, and then send them. * Ignoring Slow Workers: Ignore the gradients from slow workers, and only aggregate the gradients that you have currently available after a certain time.

Reduce Operations

We can use smart reduction techniques common in IPC (Inter-Process Communication) to reduce the number of operations needed to aggregate the gradients.

We keep a table of operations and total volume of communication:

Operation	Volume	Steps
Ideal	n	1
Parameter Server	$max(n, \frac{pn}{k})$	1
Näive AllReduce	$(p-1) \cdot n$	1
Ring AllReduce	$2 \cdot N$	$2(p-1)$
Bandwidth-optimal recursive doubling	$2 \cdot N$	$2\log_2(p)$

We explain each:

- Ideal: The ideal case, where we have a perfect system that can aggregate the gradients in a single operation.
- Parameter Server: The case where we have a parameter server, and we have to send the gradients to the server, and then the server sends the aggregated gradients back.
- Näive AllReduce: The case where we have a simple all-reduce operation, where each worker sends the gradients to all the other workers, and then each worker aggregates the gradients.
- AllReduce (ReduceScatter-AllGather): The case where we have a more sophisticated all-reduce operation, we scatter p subsets of the gradients to p workers, each worker aggregates the subset, and then we gather the results across all workers.

Ring AllReduce

We go in depth into the AllReduce operation. First, we build a logical ring topology of N workers, such that each one is connected to the next one, and the last one is connected to the first one.

We take our vector g, we divide it into N parts, and we send each part to the next worker. Each worker then aggregates the received part with its own part, and then sends the result to the next worker.

This results in a full aggregation of the gradients in $(p-1)$ steps, another full rotation is then performed to gather the results, and we have our aggregated gradients.

The communication volume is:

$$2 \cdot \frac{pn}{k} \approx 2n$$

Which is a 2-approximation of the ideal case.

We use a Ring AllReduce instead of a Tree AllReduce, since the Ring AllReduce is capable of only sending parts of the gradients, and not the full gradients, which can be problematic in the case of large models.

Bandwidth-optimal recursive doubling

A variation of the Ring AllReduce is the Bandwidth-optimal recursive doubling, this allows us to perform the aggregation in a logarithmic number of steps, and it is optimal in terms of bandwidth.

Latency-optimal recursive doubling

Using the same binary-tree like topology as the Bandwidth-optimal recursive doubling, we can build a latency-optimal recursive doubling, which is optimal in terms of latency, by sending the entire gradient vector in a single step.

Distributed Systems

First Lecture

DA RECUPERARE !!!

A Space-Time diagram shows the evolution of a distributed system over time, it is composed of a number of timelines, each associated to a process p_i. Events, occur across this timelines, and are represented by points on the diagram. If an event e_1 is causally related to another event e_2, then e_1 is to the left of e_2 and there is a line connecting them.

Second Lecture

Definitions

We now give a set of formal definitions for very natural concepts in distributed systems.

We define the ***history*** of a process p_i as the sequence of all events that occur in the timeline of p_i.

$$h_i = \left\langle e_i^1, e_i^2, \ldots, e_i^n \right\rangle$$

We can then have the history of the computation, which is a collection of all the histories of all the processes.

$$H = \langle h_1, h_2, \ldots, h_n \rangle$$

A ***prefix history*** is a sequence of events that occur in the timeline of a process up to a certain point.

$$h_i^k = \left\langle e_i^1, e_i^2, \ldots, e_i^k \right\rangle$$

A *local state* of process i after event e_i^k is the result of the computation up to that point, and is denoted as σ_i^k.

A *global state* is the collection of all local states of all processes at a certain point in time.

$$\sigma^k = \left\langle \sigma_1^k, \sigma_2^k, \ldots, \sigma_n^k \right\rangle$$

A *run* k' is a total re-ordering of the events such that the internal order of every process is preserved. This means that a *run* allows the interleaving of the events of different processes, but does not break the order of events in the history of a single process.

A *cut* C is a collection of the prefixes of every processes' history up to a certain point.

Consistency

A cut C is said to be *consistent* iff:

$$e \rightarrow e' \wedge e' \in C \Rightarrow e \in C$$

Or, in other words, if an event is in the cut, then all the events that causally precede it are also in the cut.

By sequentializing the events in a run and extracting a prefix, we can trivially obtain a consistent cut.

Consistency of Intersection

A possible course exercise is, given two cuts C_1 and C_2, to prove that their intersection is also a consistent cut.

Given the definition of consistency accomodated for the intersection of two cuts:

$$e \rightarrow e' \wedge e' \in C_1 \cap C_2 \Rightarrow e \in C_1 \cap C_2$$

We know that if e' is in the intersection, then it is in both C_1 and C_2. By the definition of consistency, we know that e must be in both C_1 and C_2, and therefore in their intersection.

Formally:

$$e \rightarrow e' \wedge e' \in C_1 \rightarrow e \in C_1$$

$$e \rightarrow e' \wedge e' \in C_2 \rightarrow e \in C_2$$

Therefore:

$$e \rightarrow e' \land e' \in C_1 \cap C_2 \rightarrow e \in C_1 \land e \in C_2 \rightarrow e \in C_1 \cap C_2$$

Consistency of Union

Given two consistent cuts C_1 and C_2, we can prove that their union is also a consistent cut.

Given the definition of consistency accomodated for the union of two cuts:

$$e \rightarrow e' \land e' \in C_1 \cup C_2 \Rightarrow e \in C_1 \cup C_2$$

Then, formally:

$$e \rightarrow e' \land e' \in C_1 \rightarrow e \in C_1$$

$$e \rightarrow e' \land e' \in C_2 \rightarrow e \in C_2$$

Therefore:

$$e \rightarrow e' \land e' \in C_1 \cup C_2 \rightarrow e \in C_1 \lor e \in C_2 \rightarrow e \in C_1 \cup C_2$$

Run Reconstruction

Assume that we have a set of processes $P = \{p_1, p_2, p_3, \}$, and a monitor process p_0 that is responsible for detecting deadlocks.

Everytime that an event is generated, its responsible process sends a message to the monitor, which then updates its local state.

The monitor process can then reconstruct the run by aggregating the local process states into a global state.

Assuming asynchrony and unbounded message delays, the monitor process does not reconstruct a run, since the order of the individual processes can be inverted.

If we assume a FIFO message delivery, then the monitor process can reconstruct a run. This can be trivially achieved by having a sequence number, in a system such as TCP.

This does *not* recover consistency, since the order of separate processes can still be inverted. However, this level of *local consistency* is sufficient for deadlock detection.

Simplifying Assumptions for Consistency

Assume that I have:

1. A finite δ_i notification delay on each process p_i.
2. A global clock C that is synchronized with all processes. We call an idealized version of this clock *Real Clock* (RC).

Then, the monitor process can offline-reconstruct a run by using the global clock to order the events.

Online-reconstruction can be achieved (and is preferred) by exploiting the additional δ_i information. Once we receive a notification, we await time $\max \delta_i$ before committing the event to the global state, in the meantime, we store the event in a buffer that is ordered by the global clock.

Real Clock Property

The clock property relates the ***happens before*** relation of messages to the real time of the system.

$$e \rightarrow e' \Rightarrow TS(e) < TS(e')$$

We can obtain a trivial clock by designing a system as such:

1. Each process has associates a sequence number $k \in \mathbb{N}$ to each event.
2. Each event i, which receives a message m from event j, has a sequence number given by the formula $\max(k_i, k_j) + 1$.

This gives us the clock property ***by construction***, since each event has a sequence number that is always greater than the sequence number of the event that caused it and of any event that it is casually related to.

This device is called a *Lamport Clock*, or a *Logical Clock*.

History of an event

Given an event e, we can define its history as the sequence of events that causally precede it. These are all the events that the monitor process must have received before the event e is committed to the global state.

$$\Theta(e) = \{e' \mid e' \rightarrow e\}$$

This is a *consistent* cut.

Opaque detection of consistent cuts

Given a notification about an event, for example, e_5^4, the monitor process can ensure that e_1^4 can be committed to global state by checking that for every process, we have received a notification with lamport clock ≥ 4.

We might have some processes that consistently lag behind, and therefore do not reach the lamport clock in a timely manner. We can trivially solve this by sending a liveness check to all processes for which a sufficiently high lamport number has not been received. If the process is alive but not working, it will re-adjust its lamport clock and send a notification.

This will not impact throughput, since the monitor process is not a bottleneck, and the liveness check is a very lightweight operation, that is performed only when necessary.

Building a Better Clock

We can build a *better clock*, that is, one with the enhanced property:

$$e \rightarrow e' \iff TS(e) < TS(e')$$

This is called a *Vector Clock*.

It works by communicating the history of the current event in a compact way, we do this by sending a vector of length n where n is the number of processes in the system. The vector's entries are updated to be the highest lamport number detected for that relative process, either from a message that has been received, or from the local lamport clock (in the case of the current process).

This allows us to check whether:

$$VC(e) \subseteq VC(e')$$

$$e \rightarrow e' \iff VC(e) \subseteq VC(e')$$

When messages are sent, the vector clock is attached to the message, and the receiving process updates its vector clock by taking the maximum of the two vectors, and summing 1 to the current process.

The monitor process p_0 keeps track of its own vector, where each entry is the highest lamport number that it has received from each process. When it receives a notification, if the vector clock is consistent with the monitor's vector clock, then the event is committed to the global state.

Since vectors are ***not*** sets, we have to use component-wise comparison to check for causality.

$$VC(e) < VC(e') \iff \forall i \in \{1, 2, \ldots, n\} \quad VC(e)_i < VC(e')_i$$

So we are back to the original definition of the happens before relation.

Third Lecture

We now discuss some more advanced vector clock properties.

Vector Clock Properties

Strong Clock Property

Assume we have two events e and e' . Then, we have that:

$$e \rightarrow e' \iff VC(e) \leq^{\star} VC(e')$$

Where $\leq^s tar$ is a ***partial order*** relation, such as:

$$\forall VC(e_i)[k] \leq VC(e_j)[k] \quad \wedge \quad \exists k' VC(e_i)[k'] < VC(e_j)[k']$$

Meaning that all the elements of the vector clock of e are less than or equal to the corresponding elements of the vector clock of e', and at least one element is strictly less, so that we have a *strict* partial order.

Retrieving History Cardinality

Given a vector clock $VC(e)$, we can retrieve the number of events that causally precede it by taking the sum of all the elements of the vector clock.

$$|\Theta(e)| = \sum_{i=1}^{n} VC(e)[i]$$

Retrieving Causality

Given two events e_i and e_j, we can check whether they are causally related by checking the vector clocks.

$$e_i \rightarrow e_j \iff VC(e_i)[i] \leq VC(e_j)[i]$$

Existance of a Causal Event

We want to check whether

$$\exists e_k : \neg(e_k \rightarrow e_i) \wedge \left(e_k \rightarrow e_j\right)$$

This is true if and only if:

$$\exists k : VC(e_i)[k] < VC(e_j)[k]$$

Meaning that there is a certain event that ticked the clock of e_j but not the clock of e_i.

Snapshots

Now we assume that the processes do ***not*** send notifications, rather, the monitoring process p_0 polls processes with a snapshot request.

This was already discussed in the first lecture, and we know that what we retrieve cannot retrieve a consistent cut.

We can change the protocol to improve this method. Now p_0 sends a snapshot request to all processes, however, when a process receives a snapshot request, it has to broadcast it to any other process.

Only requests from p_0 are broadcasted, so that we do not flood the network with snapshot requests.

When a process receives a snapshot request, it sends a snapshot of its local state to p_0.

With this behavior, under the reasonable FIFO-channel assumption, we can reconstruct a consistent cut.

Proof of Consistency

Assume that we have e_i and e_j such that $e_i \rightarrow e_j$, and that e_i is in the snapshot C, we want to prove that e_j is also in the snapshot.

By contradiction, assume that $e_i \notin C$, then p_i received the snapshot request before e_i, but then, since the channels are FIFO, and e_j was spawned by a received message from p_i, then the broadcasted snapshot request sent to p_j reached p_j before e_j was spawned, and therefore e_j is also ***not*** in the snapshot.

This protocol is known as the ***Chandy-Lamport*** protocol.

Models of Computation

First Lesson

Contact Information

Prof email: piperno@di.uniroma1.it

Actual lecture times:

- Wednsday 13:30 - 15:00
- Thursday 16:00 - 17:30, or 16:15 - 17:55

Course Contents

The course content will focus on **functional programming** and λ-calculus.

The main application of these languages is that of *function application*, these languages are devoid of *assignment* semantics, and are therefore called *pure*.

The main ingredients of our programs will be:

- Variables, which are usually denoted by lowercase letters
- Starting from these, you obtain the set of all programs, which are called λ-terms, this set is denoted as Λ.

Set Definition

We can provide an inductive definition of a set of λ-terms, Λ:

$$\frac{x \in V}{x \in \Lambda} \quad \text{(var)}$$

$$\frac{M \in \Lambda \quad N \in \Lambda}{(MN) \in \Lambda} \quad \text{(app)}$$

$$\frac{M \in \Lambda \quad x \in V}{\lambda x.M \in \Lambda} \quad \text{(abs)}$$

Applying these inductive rules (variables, application, abstraction).

Bactus Normal Form

Another way of describing a set of lambda terms is by using Bactus Normal Form (BNF).

$$\Lambda :: Var|\Lambda\Lambda|\lambda Var.\Lambda$$

Which is essentially a grammar for the set of lambda terms.

Lambda calculus is left-associative:

$$((xy)z) = xyz \neq x(yz)$$

All functions are unary (we assume currying). For example, the function $f(x, y)$ is represented as:

$$\lambda x.\lambda y.fxy$$

.

Functions in lambda calculus can be applied to other functions or themselves, they can also return functions.

Beta Reduction

The main operation in lambda calculus is ***beta reduction***, which is the application of a function to an argument.

$$\underbrace{(\lambda x.M)N}_{redex} \rightarrow_\beta M[N/x]$$

Where $M[N/x]$ is the result of substituting all occurrences of x in M with N.

Some other examples:

$$(\lambda x.x)y \rightarrow_\beta y$$

$$(\lambda x.xx)y \rightarrow_\beta yy$$

$$(\lambda xy.yx)(\lambda u.u) \rightarrow_\beta \lambda y.y(\lambda u.u)$$

$$(\lambda xy.yx)(\lambda t.y) \rightarrow_\beta \lambda y.y(\lambda t.y)$$

This rule can be applied in any context in which it appears.

Formal Substitution Definition

We give a formal definition of substitution, $M[N/x]$:

$$x[N/x] = N$$

$$y[N/x] = y$$

$$(M_1 M_2)[N/x] = M_1[N/x] M_2[N/x]$$

$$(\lambda t.P)[N/x] = \lambda t.(P[N/x])$$

As observed, substitution is always in place of *free* variables, therefore the abstraction is *not* replaced in the last rule.

If we had an abstraction of type $\lambda x.P$ where $x \in P$, it would be best to rename x in order to avoid name clashes.

Types of Variables

We distinguish two kinds of variables:

- Free variables: variables that are not bound by an abstraction
- Bound variables: variables that are bound by an abstraction

For example, in the term $\lambda x.xy$, y is a free variable, while x is a bound variable.

Bound variables can be renamed, whereas for free variables the naming is relevant.

$$\begin{cases} FV(x) = \{x\} \\ FV(MN) = FV(M) \cup FV(N) \\ FV(\lambda x.M) = FV(M) - \{x\} \end{cases}$$

A set of lambda terms where $LM(\Lambda) =$ is called *closed.*

Extra Rules

$$(\mu) \quad \frac{M \to_\beta M'}{NM \to_\beta NM'}$$

$$(\nu) \quad \frac{M \to_\beta M'}{MN \to_\beta M'N}$$

$$(\xi) \quad \frac{M \to M'}{\lambda x.M \to \lambda x.M'}$$

These rules allow us to select redexes in a context-free manner in the middle of our lambda term. We can then choose the order of evaluation of our redexes, while still taking care of the left-associative order of precedence. Our calculus is therefore not *determinate* but is still *deterministic*, meaning that there may be multiple reduction strategies but they all lead to the same result.

This corollary is called the *Church Rosser Theorem*, discovered in 1936.

In general, a call-by-value-like semantic is preferrable when choosing evaluation paths, as it clears the most amount of terms as early as possible.

- Call By Value is *efficient*
- Call By Name is *complete*, **if** the lambda term is normalizable

Second Lecture

Alpha Reduction

Alpha reduction is the renaming of bound variables in a lambda term.

$$\lambda x.M \to_\alpha \lambda y.M[y/x]$$

This rule is used to avoid name clashes between bound variables.

Arithmetic Expressions

The set of all valid arithmetic expressions has a very precise syntax. In general, a syntax can be viewed either as a tool for checking validity or as a generator of valid expressions (a grammar).

We proceed to give a definition for arithmetic expressions

$$\frac{x \in \mathbb{N}}{x \in \text{Expr}} \quad (\text{num})$$

$$\frac{X \in \text{Expr} \quad Y \in \text{Expr}}{X + Y \in \text{Expr}} \quad (\text{add})$$

$$\frac{X \in \text{Expr} \quad Y \in \text{Expr}}{X \times Y \in \text{Expr}} \quad (\text{mul})$$

Etc, etc… for all the other binary operations.

From this, we can successfully decompose any arithmetic expression into a syntactic tree. With this set of rules, we have a slight problem: we can't represent negative numbers. We could solve this either by adding a rule for unary minus, or by specifying the num rule over $\mathbb{Z}$ instead of $\mathbb{N}$.

Combinators

We define three combinators:

$$S = \lambda xyz.xz(yz)$$

$$K = \lambda xy.x$$

$$I = \lambda u.u$$

We have that $SKy \to_\beta I$

Exercise 1.1: β-reduce $S(KS)S$

This reduces to $\lambda zbc.z(bc)$, which is the B combinator (composition).

Exercise 1.2: β-reduce $S(BBS)(KK)$.

This reduces to $\lambda zcd.zdc$, which is the C combinator (permutation).

For exercise this, we show a step-by-step reduction:

$$\begin{aligned}
&S(BBS)(KK)\\
\rightsquigarrow&\lambda z.(BBS)z((KK)z)\\
\rightsquigarrow&\lambda z.(\lambda c.B(Sc))z((KK)z)\\
\rightsquigarrow&\lambda z.(\lambda c.B(Sc))zK\\
\rightsquigarrow&\lambda z.B(Sz)K\\
\rightsquigarrow&\lambda z.(\lambda c.(Sz)(Kc)\\
\rightsquigarrow&\lambda zc.Sz(Kc)\\
\rightsquigarrow&\lambda zcd.zd(Kcd)\\
\rightsquigarrow&\lambda zcd.zdc \quad \square.
\end{aligned}$$