Purity

MinML (with products) is a **purely functional** language.

- No assignment, no I/O, no mutation of storage.
- No exception handlers.
- Only one form of error.

The language is **not observably sequential**.

---

Sequentiality

Contrast with full ML, which is **observably sequential**.

- `(print "hello", print "goodbye")`
- Evaluates to `((),())`
- But left-to-right is distinguishable from right-to-left!
- What would “simultaneous” mean?

---

Two Semantics

This suggests two different semantics for MinML with pairs.

- **Sequential**, which fixes an evaluation ordering.
  - Left-to-right, as before.
  - Right-to-left, also possible.

- **Parallel**, which evaluates pairs simultaneously.

---

Sequential Semantics

The **sequential semantics** of tuples specifies **left-to-right** evaluation.

\[
\begin{align*}
\epsilon_1 &\mapsto_{seq} \epsilon_1' \\
(e_1, e_2) &\mapsto_{seq} (e_1, e_2')
\end{align*}
\]

\[
\begin{align*}
\epsilon_1 \text{ value} &\mapsto_{seq} \epsilon_1' \text{ value} \\
(e_1, e_2) &\mapsto_{seq} (e_1, e_2') \text{ value}
\end{align*}
\]
**Parallel Semantics**

The parallel semantics of tuples specifies simultaneous evaluation.

\[
e_1 \rightsquigarrow_{par} e_1' \quad e_2 \rightsquigarrow_{par} e_2' \\
(v_1, v_2) \rightsquigarrow_{par} (v_1', v_2')
\]

v_1 value \( e_2 \rightsquigarrow_{par} e_2' \)

\[
(v_1, v_2) \rightsquigarrow_{par} (v_1', v_2')
\]

v_2 value \( e_1 \rightsquigarrow_{par} e_1' \)

\[
(v_1, v_2) \rightsquigarrow_{par} (v_1', v_2')
\]

**Implicit Parallelism**

Both semantics compute the same results!

**Theorem 1 (Implicit Parallelism)**

\[ e \rightsquigarrow_{seq} v \iff e \rightsquigarrow_{par} v \]

How to prove this?

**Implicit Parallelism**

Clearly, if \( v \) value, then \( v \rightsquigarrow_{seq} v \) iff \( v \rightsquigarrow_{par} v \).

Suffices to show

1. If \( e \rightsquigarrow_{seq} e' \rightsquigarrow_{par} v \), then \( e \rightsquigarrow_{par} v \).

2. If \( e \rightsquigarrow_{par} e' \rightsquigarrow_{seq} v \), then \( e \rightsquigarrow_{seq} v \).

Proof is by induction on (sequential or parallel) evaluation.

**Implicit Parallelism**

It may be shown that

- \( e_1' \rightsquigarrow_{par} v_1 \)
- \( e_2 \rightsquigarrow_{par} v_2 \)

By induction \( e_1 \rightsquigarrow_{par} v_1 \), from which the result follows.
Implicit Parallelism

It may be shown that

- \( e_1 \mapsto_{seq} v_1 \);
- \( e_2 \mapsto_{seq} v_2 \).

By induction

\[ e_1 \mapsto_{seq} v_1 \]
and

\[ e_2 \mapsto_{seq} v_2 \]

from which the result follows.

What’s The Difference?

In terms of behavior parallelism doesn’t matter.

- Same value in either semantics.
- Can’t observe parallelism.

In terms of efficiency, it obviously does matter!

Sequential Complexity

The sequential complexity of \( e \) is the number \( k \) (if any) such that

\[ e \mapsto_{seq}^k v. \]

That is: how many steps it takes to reach a value in the sequential semantics.

Parallel Complexity

The parallel complexity of \( e \) is the number \( k \) (if any) such that

\[ e \mapsto_{par}^k v. \]

That is: how many steps it takes to reach a value in the parallel semantics.

A Simple Example

The sequential complexity of \( n \) is \( O(2^n) \).

- Two recursive calls for each recursive call.
- Both calls must complete before return.

A Simple Example

Naïve Fibonacci:

```plaintext
fun plus (p:int*int):int is
  split p as (m:int,n:int) in m+n

fun fib (n:int):int is
  if m=0 then 1
  else if m=1 then 1
  else plus (fib (n-1), fib (n-2)) fi fi
```
A Simple Example

The parallel complexity of $\text{fib } n$ is $O(n)$.

- Evaluate recursive calls simultaneously.
- Wait for both before returning.

Thus, parallelism can vastly improve efficiency!

Work and Depth

The work of an expression is the total number of instructions required to evaluate it to a value.

The sequential complexity is equal to the work.

- Each evaluation rule has at most one premise.
- So each step corresponds to one instruction.

A Simple Example

This improvement relies on performing an unbounded number of instructions simultaneously!

- "Degree" of parallelism increases exponentially with depth of recursion!
- On an idealized computer you can always keep up, but not in practice!

(More on this later.)

Work and Depth

The sequential complexity bounds the parallel complexity.

- One transition step corresponds to many instructions.
- But must perform the same overall work!

Parallel Cost Semantics

A parallel cost semantics specifies the work and depth complexity.

$e \vdash w, d, v$

Informally,

- $e$ evaluates to $v$ with $\ldots$
- $\ldots$ work $w$ and $\ldots$
- $\ldots$ depth $d$. 

Work and Depth

The depth of an expression is the length of the longest sequential dependency chain in its evaluation.

- Only the components of a pair are independent (here).
- Others require sequential evaluation ordering.

The parallel complexity is equal to the depth.

- Precisely captures independence of components of a pair.
- Exploits unbounded parallelism to avoid false dependencies.
Parallel Cost Semantics

Cost semantics for pairing:

\[
e_1 \uparrow^{w_1,d_1} v_1, e_2 \uparrow^{w_2,d_2} v_2
\]

\[\overrightarrow{(e_1, e_2)} \uparrow^{w,d} (v_1, v_2)\]

where

\[d = \max(d_1, d_2), \quad w = w_1 + w_2.\]

Correctness of Cost Semantics

The cost semantics correctly specifies sequential and parallel complexity:

**Theorem 2**

\[
e \uparrow^{w,d} v \iff e \rightarrow^{w} v \text{ and } e \rightarrow^{d} v.
\]

From left-to-right, proceed by induction on evaluation.

From right-to-left, consider work and depth separately.

Vector Parallelism

Similar ideas apply to vectors.

- Dynamic generation of n-tuples of values of a type.
- Parallel operations such as mapping a function across a vector.

Supports a parallel formulation of graph algorithms and matrix computations.

Vector Parallelism

Cost semantics for vector expressions:

\[
\forall 0 \leq i < n \ (e_i \uparrow^{w_i,d_i} v_i) \quad \overrightarrow{[e_0, \ldots, e_{n-1}]} \uparrow^{w,d} [v_0, \ldots, v_{n-1}]
\]

where \(w = \sum_{i=0}^{n-1} w_i\) and \(d = \max_{i=0}^{n-1} d_i\).

Depth cost reflects parallel evaluation of elements.

Vector Parallelism

Vector expressions:

- Values \([v_0, \ldots, v_{n-1}]\).
- Evaluate components in parallel:

\[
\forall i \in I \ (e_i \rightarrow^{v'} e'_i) \quad \forall i \notin I \ (e'_i = e_i \text{ & } e_i \text{ value})
\]

\[\overrightarrow{[e_0, \ldots, e_{n-1}]} \rightarrow^{v, e}[e_0, \ldots, e_{n-1}]\]

where \(\emptyset \neq I \subseteq \{0, \ldots, n-1\}\).
Vector Parallelism

Parallel application of a function to a vector:

\[
\text{map}(v, [v_0, \ldots, v_{n-1}]) \mapsto \text{par} [\text{apply}(v, v_0), \ldots, \text{apply}(v, v_{n-1})]
\]

Cost semantics:

\[
\begin{align*}
& e_1 \Downarrow^{w_1, d_1} v \\
& e_2 \Downarrow^{w_2, d_2} [v_0, \ldots, v_{n-1}] \\
& \text{map}(e_1, e_2) \Downarrow^{w_1 + w_2 + \cdots + w_{n-1}, d_1 + d_2 + \cdots + d_{n-1}} [v_0, \ldots, v_{n-1}]
\end{align*}
\]

Create an index vector:

\[
\text{index}(n) \mapsto \text{par} [0, \ldots, n-1]
\]

Cost semantics:

\[
\begin{align*}
& e \Downarrow^{w, d} n \\
& \text{index}(e) \Downarrow^{w, d, n} [0, \ldots, n-1]
\end{align*}
\]

Summary

Pure languages support implicit parallelism.

- Simultaneous evaluation of components of pairs.
- Vector parallelism (see notes).

Parallelism affects complexity, not behavior.

- Purely a matter of cost.
- No worries about correctness.