# Programming Large Language Models with Algebraic Effect Handlers and the Selection Monad

Shangyin Tan, ***Guannan Wei***, Koushik Sen, Matei Zaharia

UC Berkeley

Tufts UNIVERSITY

# Prompting vs. Programming

A common pattern to interact with the LLMs is through prompting:

```
messages = [
    {
        "role": "system",
        "content": 'Return ONLY JSON: {"sentiment":"positive|negative|neutral"}',
    },
    {
        "role": "user",
        "content": "Classify the sentiment of: I love this model!",
    },
]

chat = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=messages,
    temperature=0,
)
```

# Prompting vs. Programming

Can we *program* with LLMs? An example with DSPy [ICLR '24]:

```python
# 1) Configure LLM
dspy.settings.configure(lm=dspy.OpenAI(model="gpt-4o-mini", temperature=0))

# 2) Define the typed signature
class SentimentSig(dspy.Signature):
    """Return ONLY a sentiment label for the input sentence."""
    sentence = dspy.InputField()
    sentiment = dspy.OutputField(
        desc="one of: positive, negative, or neutral",
    )

# 3) One-line predict call using the signature
dspy.Predict(SentimentSig)(sentence="I love this model!").sentiment
```

# What is missing with DSPy?

DSPy and other similar libraries provide good abstractions of prompting, but ...

```python
# 1) Configure LLM
dspy.settings.configure(lm=dspy.OpenAI(model="gpt-4o-mini", temperature=0))

# 2) Define the typed signature
class SentimentSig(dspy.Signature):
    """Return ONLY a sentiment label for the input sentence."""
    sentence = dspy.InputField()
    sentiment = dspy.OutputField(
        desc="one of: positive, negative, or neutral",
    )

# 3) One-line predict call using the signature
dspy.Predict(SentimentSig)(sentence="I love this model!").sentiment
```

# What is missing with DSPy?

DSPy and other similar libraries provide good abstractions of prompting, but …

**Not enough to specify "how" the program interacts with LLMs**

```python
# 1) Configure LLM
dspy.settings.configure(lm=dspy.OpenAI(model="gpt-4o-mini", temperature=0))

# 2) Define the typed signature
class SentimentSig(dspy.Signature):
    """Return ONLY a sentiment label for the input sentence."""
    sentence = dspy.InputField()
    sentiment = dspy.OutputField(
        desc="one of: positive, negative, or neutral",
    )

# 3) One-line predict call using the signature
dspy.Predict(SentimentSig)(sentence="I love this model!").sentiment
```

# Pangolin: Programming LLMs with Algebraic Effects and Handlers

- Algebraic effects + handlers:
  - Effect operation declares what can happen
  - Handler defines how it happens

# Pangolin: Programming LLMs with Algebraic Effects and Handlers

- Algebraic effects + handlers:
  - Effect operation declares what can happen
  - Handler defines how it happens

- **Pangolin** core idea:
  - represent LM interaction as “effects” and abstract operations
  - separate handlers provides interpretation for these “effects”

# Programming with Algebraic Effects

```
let emotion_classifier = λ sentence.
  let emotion_spec = specification {
    input = { sentence: str },
    output = { sentiment: str }
  };
  let input = { sentence = sentence };
  let result = LM(emotion_spec, input);
  result.sentiment
let result = emotion_classifier("I love you!")
```

# Programming with Algebraic Effects

```
1  let emotion_classifier = λ sentence.
2    let emotion_spec = specification {
3      input = { sentence: str },
4      output = { sentiment: str }
5    };
6    let input = { sentence = sentence };
7    let result = LM(emotion_spec, input);
8    result.sentiment
9  let result = emotion_classifier("I love you!")
```

# Programming with Algebraic Effects

```
1  let emotion_classifier = λ sentence.
2    let emotion_spec = specification {
3      input = { sentence: str },
4      output = { sentiment: str }
5    };
6    let input = { sentence = sentence };
7    let result = LM(emotion_spec, input);
8    result.sentiment
9  let result = emotion_classifier("I love you!")
```

# Defining a Naive Handler

```
let naive_lm_handler = {
  | LM(spec, input; k) ↦
    let prompt = write[spec](input);
    let raw_result = primlm(prompt);
    let output = read[spec.output](raw_result);
    k(output)
};

handle emotion_classifier("I love you!")
  with naive_lm_handler
```

# Defining a Naive Handler

```
1  let naive_lm_handler = {
2    | LM(spec, input; k) ↦
3      let prompt = write[spec](input);
4      let raw_result = primlm(prompt);
5      let output = read[spec.output](raw_result);
6      k(output)
7  };
8
9  handle emotion_classifier("I love you!")
10   with naive_lm_handler
```

# Defining a Naive Handler

```
1  let naive_lm_handler = {
2    | LM(spec, input; k) ↦
3      let prompt = write[spec](input);
4      let raw_result = primlm(prompt);
5      let output = read[spec.output](raw_result);
6      k(output)
7  };
8
9  handle emotion_classifier("I love you!")
10   with naive_lm_handler
```

# Defining a Naive Handler

```
1  let naive_lm_handler = {
2    | LM(spec, input; k) ↦
3      let prompt = write[spec](input);
4      let raw_result = primlm(prompt);
5      let output = read[spec.output](raw_result);
6      k(output)
7  };
8
9  handle emotion_classifier("I love you!")
10   with naive_lm_handler
```

# Defining a Naive Handler

```
1  let naive_lm_handler = {
2    | LM(spec, input; k) ↦
3      let prompt = write[spec](input);
4      let raw_result = primlm(prompt);
5      let output = read[spec.output](raw_result);
6      k(output)
7  };
8
9  handle emotion_classifier("I love you!")
10   with naive_lm_handler
```

# Test-time Scaling: Handler with Multi-shot Continuations

```
let parallel_lm_handler(n) = {
  | return x ↦ [x]
  | LM(spec, input; k) ↦
    let prompt = write[spec](input);
    let raw_results = map(
      λ i. primlm(prompt), [1..n]
    );
    let parsed_outputs = map(
      λ result. read[spec.output](result),
      raw_results
    );
    fold(++, [], map(k, parsed_outputs))
};

handle emotion_classifier("I love you!")
  with parallel_lm_handler(16)
```

# Test-time Scaling: Handler with Multi-shot Continuations

```
let parallel_lm_handler(n) = {
  | return x ↦ [x]
  | LM(spec, input; k) ↦
    let prompt = write[spec](input);
    let raw_results = map(
      λ i. primlm(prompt), [1..n]
    );
    let parsed_outputs = map(
      λ result. read[spec.output](result),
      raw_results
    );
    fold(++, [], map(k, parsed_outputs))
};

handle emotion_classifier("I love you!")
  with parallel_lm_handler(16)
```

# Test-time Scaling: Handler with Multi-shot Continuations

```
let parallel_lm_handler(n) = {
  | return x ↦ [x]
  | LM(spec, input; k) ↦
    let prompt = write[spec](input);
    let raw_results = map(
      λ i. primlm(prompt), [1..n]
    );
    let parsed_outputs = map(
      λ result. read[spec.output](result),
      raw_results
    );
    fold(++, [], map(k, parsed_outputs))
};

handle emotion_classifier("I love you!")
  with parallel_lm_handler(16)
```

# Test-time Scaling: Handler with Multi-shot Continuations

```
let parallel_lm_handler(n) = {
  | return x ↦ [x]
  | LM(spec, input; k) ↦
    let prompt = write[spec](input);
    let raw_results = map(
      λ i. primlm(prompt), [1..n]
    );
    let parsed_outputs = map(
      λ result. read[spec.output](result),
      raw_results
    );
    fold(++, [], map(k, parsed_outputs))
};

handle emotion_classifier("I love you!")
  with parallel_lm_handler(16)
```

# Selection Monad

- Formulate *loss* inside effectful programs, and search objects through *choose* operations
- Handling selection monad can be combined with effect handling, with an additional *loss continuation*
- Particularly helpful for specifying retry with rewards and scaling behaviors in LLM interactions, and has been applied to other ML/RL tasks.[1]

[1] H*andling the Selection Monad.* Gordon Plotkin, Ningning Xie
*Effect Handlers for Choice-Based Learning*. Gordon Plotkin, Ningning Xie
*Choix: Choice-based Learning in Jax.* Shangyin Tan, Dan Zheng, Gordon Plotkin, Ningning Xie

# Selection Monad: Best-of-N Pattern

```
let best_of_n_handler(n) = {
  | LM(spec, input; k) ↦
    // same as before
    ...
    let parsed_outputs = ...
    let best = choose(parsed_outputs);
    k(best)
  | choose(xs; k; k_s) ↦
    let scores = map k_s xs;
    let best_idx = argmax(scores);
    k(xs[best_idx])
};

handle
  let res = emotion_classifier("I love you!");
  score confidence_score(res)
with best_of_n_handler(16)
```

# Selection Monad: Best-of-N Pattern

```
let best_of_n_handler(n) = {
  | LM(spec, input; k) ↦
    // same as before
    ...
    let parsed_outputs = ...
    let best = choose(parsed_outputs);
    k(best)
  | choose(xs; k; k_s) ↦
    let scores = map k_s xs;
    let best_idx = argmax(scores);
    k(xs[best_idx])
};

handle
  let res = emotion_classifier("I love you!");
  score confidence_score(res)
with best_of_n_handler(16)
```

# Selection Monad: Best-of-N Pattern

```
let best_of_n_handler(n) = {
  | LM(spec, input; k) ↦
    // same as before
    ...
    let parsed_outputs = ...
    let best = choose(parsed_outputs);
    k(best)
  | choose(xs; k; k_s) ↦
    let scores = map k_s xs;
    let best_idx = argmax(scores);
    k(xs[best_idx])
};

handle
  let res = emotion_classifier("I love you!");
  score confidence_score(res)
with best_of_n_handler(16)
```

# Selection Monad: Best-of-N Pattern

```
let best_of_n_handler(n) = {
  | LM(spec, input; k) ↦
    // same as before
    ...
    let parsed_outputs = ...
    let best = choose(parsed_outputs);
    k(best)
  | choose(xs; k; k_s) ↦
    let scores = map k_s xs;
    let best_idx = argmax(scores);
    k(xs[best_idx])
};

handle
  let res = emotion_classifier("I love you!");
  score confidence_score(res)
with best_of_n_handler(16)
```

# Selection Monad: Assertions as Feedback/Loss

```
let choose_handler = {
  | assert(pred, feedback; k; retry_ctx) ↦
    score(pred);
    k()
  | choose(xs; k; k_s) ↦
    let scores = map k_s xs;
    let best_idx = argmax(scores);
    k(xs[best_idx])
};

handle retry(feedback = "", n_retry = 0) {
  let res_1 = choose(emotion_classifier("I love
      you!"));
  assert(
    res ∈ ["positive", "negative", "neutral"],
    "Expected positive, negative, or neutral"
  );
  assert(len(res) < 10;
    "Sentiment should be short.")
} with choose_handler | parallel_lm_handler(5)
```

# Selection Monad: Assertions as Feedback/Loss

```
let choose_handler = {
  | assert(pred, feedback; k; retry_ctx) ↦
    score(pred);
    k()
  | choose(xs; k; k_s) ↦
    let scores = map k_s xs;
    let best_idx = argmax(scores);
    k(xs[best_idx])
};

handle retry(feedback = "", n_retry = 0) {
  let res_1 = choose(emotion_classifier("I love
      you!"));
  assert(
    res ∈ ["positive", "negative", "neutral"],
    "Expected positive, negative, or neutral"
  );
  assert(len(res) < 10;
    "Sentiment should be short.")
} with choose_handler | parallel_lm_handler(5)
```

# Selection Monad: Assertions as Feedback/Loss

```
let choose_handler = {
  | assert(pred, feedback; k; retry_ctx) ↦
    score(pred);
    k()
  | choose(xs; k; k_s) ↦
    let scores = map k_s xs;
    let best_idx = argmax(scores);
    k(xs[best_idx])
};

handle retry(feedback = "", n_retry = 0) {
  let res_1 = choose(emotion_classifier("I love
      you!"));
  assert(
    res ∈ ["positive", "negative", "neutral"],
    "Expected positive, negative, or neutral"
  );
  assert(len(res) < 10;
    "Sentiment should be short.")
} with choose_handler | parallel_lm_handler(5)
```

# Conclusion and Future Work

- What is a good abstraction for programming with LLMs?

- We have proposed LM programming language **Pangolin** and sketched its design
  - Algebraic effects, selection monad, and handlers provide a user-customizable way to interaction with LLMs
  - Good to express many LLM programming patterns

- Future work
  - Full implementation and formal semantics remains
  - Better abstraction for building agents (involving loops and tool invocations)