Towards abductive functional programming

Koko Muroya
Steven Cheung & Dan R. Ghica
(University of Birmingham)
Parameter tuning via targeted abduction

Koko Muroya
Steven Cheung & Dan R. Ghica
(University of Birmingham)
A programming idiom for optimisation & ML

Model: \( f_{a,b}(x) = a \times x + b \)

**Input** \( x_0 \)

**Use model**

**Output** \( f_{a,b}(x_0) \)

**Data** \( \{(x_i, y_i)\}_{i=1}^N \)

**Train model**

**Updated model** \( f_{a',b'} \)
# Build inference graph.
# Create and initialise variables W and b.

W = tf.Variable(...)  
b = tf.Variable(...)  

y = W * x_data + b  

#NOTE: Nothing actually computed here!

Model: \( f_{a,b}(x) = ax + b \)
Example: parameter optimisation in TensorFlow

**Model** \( f_{a,b}(x) = a \times x + b \)

```python
# Create a session.
sess = tf.Session()
sess.run(init)
y_initial_values = sess.run(y)
# Compute some y values
```

\( W = \text{tf.Variable}(\ldots) \)
\( b = \text{tf.Variable}(\ldots) \)
\( y = W \times x_{\text{data}} + b \)

**Input** \( x_0 \)

**Output** \( f_{a,b}(x_0) \)
# Build training graph.
loss = tf.some_loss_function(y, y_data)
# Create an operation that calculates loss.
tf.train.some_optimiser.minimize(loss)
# Create an operation that minimizes loss.
init = tf.initialize_all_variables()
# Create an operation initializes variables.
sess = tf.Session()
sess.run(init)

# Perform training:
for step in range(201):
    sess.run(train)
TensorFlow

● shallow embedded DSL
  ○ lack of integration with host language
  ○ cannot use libraries in graphs
  ○ difficult to debug / type graphs

● imperative “variable” update
TensorFlow

- shallow embedded DSL
  - lack of integration with host language
  - cannot use libraries in graphs
  - difficult to debug / type graphs
- imperative parameter ("variable") update

Proper *functional* language?

- simple & uniform programming language
  - full integration with base language
  - typed in ML-style
  - well-defined operational semantics
- funcional parameter update
Key idea:
Abductive reasoning
Abductive inference: background

- Logical inference
  - Deduction (specialisation)
  - Induction (generalisation)
  - Abduction (explanation)

- Previous applications
  - Abductive logic programming
  - Program verification ([http://fbinfer.com/](http://fbinfer.com/))
Abductive inference: background

- logical inference
  - deduction (specialisation)
  - induction (generalisation)
  - abduction (explanation)

- previous applications
  - abductive logic programming
  - program verification (http://fbinfer.com/)
Abductive inference: background

- logical inference
  - deduction (specialisation)
  - induction (generalisation)
  - abduction (explanation)

- previous applications
  - abductive logic programming
  - program verification (http://fbinfer.com/)
Abductive inference: our use

- possible deductive rule for abduction

\[
\Gamma \vdash A \\
\Gamma \vdash (P \Rightarrow A) \land P
\]

“abduct” explanation $P$ of $A$ in “targeted” way
“Parameter tuning via targeted abduction”

model \( f_{a,b}(x) = a \times x + b \)

let \( m \ x = \{2\} \times x + \{3\};; \)

use

output \( f_{a,b}(x_0) \)

m \ 0;;

train

updated model \( f_{a',b'} \)

let \( f \ @ \ p = m \) in
let \( q = \text{optimise} \ p \) in
\( f \ q;; \)
“Parameter tuning via targeted abduction”

model \( f_{a,b}(x) = a \times x + b \)

let \( m \ x = \{2\} \times x + \{3\} \);

provisional constants ("targets")

cf. definitive constants \( 0,1,2,\ldots \)
Parameter tuning via targeted abduction

model $f_{a,b}(x) = a \times x + b$

let $m x = \{2\} \times x + \{3\};$

use

output $f_{a,b}(x_0)$

$m 0;;$

(* simply function application *)
“Parameter tuning via targeted abduction”

Model: \( f_{a,b}(x) = a \times x + b \)

let \( m \ x = \{2\} \times x + \{3\};; \)

provisional constants

abductive decoupling

Train

updated model: \( f_{a',b'} \)

let \( f @ p = m \) in (* “decouple” model \( f \) and parameters \( p \) *)
let \( q = \text{optimise} \ p \) in (* compute “better” parameter values *)
Let \( m' = f q \) in (* “improve” model using new parameters *)
Abductive decoupling: informal semantics

```
let m x = \{2\} * x + \{3\};;

let f @ p = m in
let q = optimise p in
f q;;
```

```
val m = fun x -> \{2\} * x + \{3\}

val f = fun (p1,p2) -> fun x -> p1 * x + p2

val p = (2,3)
```

---

model with provisional constants

parameterised model

parameter vector
Abductive decoupling: informal semantics

let m x = \{2\} \times x + \{3\};;

let f @ p = m in
let q = optimise p in
f q;;

val m = fun x -> \{2\} \times x + \{3\}

val f = fun (p1,p2) -> fun x -> p1 \times x + p2

val p = (2,3)

model with provisional constants

parameterised model

parameter vector

\[ \Gamma \vdash A \]
\[ \Gamma \vdash (P \Rightarrow A) \wedge P \]

abduction rule
Promoting provisional to definitive constants

```ocaml
let m x = \{2\} \times x + \{3\};;
let \(f @ p = m\) in
let q = p in
f q;;
```

<table>
<thead>
<tr>
<th>Code snippet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val m = fun x -&gt; \{2\} \times x + \{3\};;</code></td>
<td>Model with provisional constants</td>
</tr>
<tr>
<td><code>val f = fun (p1,p2) -&gt; fun x -&gt; p1 \times x + p2</code></td>
<td>Parameterised model</td>
</tr>
<tr>
<td><code>val p = (2,3)</code></td>
<td>Parameter vector</td>
</tr>
<tr>
<td><code>val q = (2,3)</code></td>
<td>(Trivially updated) parameter vector</td>
</tr>
<tr>
<td><code>let m x = \{2\} \times x + \{3\};;</code></td>
<td>Result: model with definitive constants</td>
</tr>
</tbody>
</table>

Muroya (U. B'ham.)
Parameter tuning via targeted abduction

\[ f_{a,b}(x) = a \times x + b \]

let \( m \ x = \{2\} \times x + \{3\} \);

\[
\begin{align*}
\text{output} & \quad f_{a,b}(x_0) \\
m & \quad \Theta \\
\text{use} & \\
\text{provisional} & \text{constants} \\
\text{train} & \\
\text{updated model} & \quad f_{a',b'} \\
\text{let } f @ p & = m \text{ in} \\
\text{let } q & = \text{optimise } p \text{ in} \\
f & q \\
\text{abductive} & \text{decoupling}
\end{align*}
\]
Targeted abduction: syntax & types

(fixed) field

provisional constant

opaque vector space, representing $\mathbb{F}^n$

\[
\Delta, a \mid \Gamma, f : V_a \rightarrow T, x : V_a \vdash t : T'
\]

\[
\frac{
}{\Delta \mid \Gamma \vdash \text{abd } f @ x \rightarrow t : T \rightarrow T'}
\]

let $f @ x = u$ in $t \equiv (\text{abd } f @ x \rightarrow t) u$
Targeted abduction: syntax & types

(provisional) constant

opaque vector space, representing \( \mathbb{F}^n \)

\[
\begin{align*}
\Delta, a & \vdash \Gamma, f : V_a \to T, x : V_a \vdash t : T' \\
\Delta & \vdash \text{abd } f \circ x \to t : T \to T'
\end{align*}
\]

(* abduction of open terms *)
let \( m \ x = \{2\} \times x + n \) in
let \( f \ @ p = m \) in
...

Muroya (U. B'ham.)
Targeted abduction: opaque vectors

- size determined **dynamically**
- order of coordinates **unknown**
  - ... yet we want deterministic programs
  - always point-free (no access to bases/coordinates)
  - only **symmetric** operations (invariant over permutation of bases/coordinates)

- possible in theory
  - symmetric tensors
- reasonable in practice
  - not all, *but most*, optimisation algorithms are symmetric
Targeted abduction: *symmetric vector operations*

**standard vector operations**

\[ +_a : V_a \rightarrow V_a \rightarrow V_a \]  
(vector addition)

\[ \times_a : \mathbb{F} \rightarrow V_a \rightarrow V_a \]  
(scalar multiplication)

\[ \bullet_a : V_a \rightarrow V_a \rightarrow \mathbb{F}, \]  
(dot product)

**iterated vector operations**

\[ +^L_a : (V_a \rightarrow V_a) \rightarrow V_a \rightarrow V_a \]  
(left-iterative vector addition)

\[ +^R_a : V_a \rightarrow (V_a \rightarrow V_a) \rightarrow V_a \]  
(right-iterative vector addition)

\[ \times^L_a : (V_a \rightarrow \mathbb{F}) \rightarrow V_a \rightarrow V_a \]  
(left-iterative scalar multiplication)
Targeted abduction: example use

numerical gradient descent

```ocaml
let m x = \{2\} * x + \{3\};;

let f @ p = m in
let q = grad_desc f p loss 0.001 in
f q;;
```

(* least square on some reference data *)
let loss f p = ...;;

(* numerical gradient descent *)
let grad_desc f p loss rate =
  let d = 0.001 in
  let g e =
    let old = loss f p in
    let new = loss f (p ⊕ (d ⊠ e)) in
    ((old - new) / d) * rate) ⊠ e in
  g ⊕ p;;

folding over standard basis
\[ f + \sum_a^L v_0 := \text{foldr} (\lambda e \lambda v. f(e) + v) E_a v_0 \]
Targeted abduction: syntax & types

(provisional constant)

\[ \Delta, a \mid \Gamma, f : V_a \to T, x : V_a \vdash t : T' \]
\[ \Delta \mid \Gamma \vdash \text{abd } f @ x \to t : T \to T' \]

opaque vector space, representing \( \mathbb{F}^n \)

\[ \text{let } f @ x = u \text{ in } t \equiv (\text{abd } f @ x \to t) u \]
Targeted abduction: operational semantics

- provisional constants are linear!

\[
\begin{align*}
\text{let } m \ x &= \{0\} \ast x + \{0\};; \\
\text{vs } \text{let } p &= \{0\} \text{ in } \\
\text{let } m \ x &= p \ast x + p;;
\end{align*}
\]

- graph rewriting semantics
  - … based on Geometry of Interaction
  - [http://www.cs.bham.ac.uk/~drg/goa/visualiser/](http://www.cs.bham.ac.uk/~drg/goa/visualiser/)
  - determinism
  - soundness of execution
  - safety of garbage-collection
  - call-by-value evaluation
Conclusions

● a fully-integrated language for parameter tuning
  ○ abductive decoupling “abd”
  ○ simply-typed + abduction rule
  ○ formal operational semantics
    ■ call-by-value
    ■ determinism
    ■ sound execution & safe garbage-collection

● open problems
  ○ actual ML compiler extension
    ■ abduction is dynamic & complex
    ■ … but not computationally dominant
  ○ stochastical machinery

http://www.cs.bham.ac.uk/~drg/goa/visualiser/