Verified reversible programming for verified lossless compression

LAFI Workshop

15th January 2023

Jan-Willem van de Meent

James Townsend

AMLAB
Amsterdam
Machine Learning Lab

Why reversible programming?

```python
def encode(code, symbols):
    [symbol_1, symbol_2] = symbols  # (1)
    code = Categorical(             # (2)
       {'f': 3/8,
        'h': 5/8}).encode(code, symbol_2)
    code = Categorical(             # (3)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).encode(code, symbol_1)
    return code
```

```python
def decode(code):  # Inverse of encode
    code, symbol_1 = Categorical(   # (3 inverse)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).decode(code)
    code, symbol_2 = Categorical(   # (2 inverse)
       {'f': 2/8,
        'h': 6/8}).decode(code)
    symbols = [symbol_1, symbol_2]  # (1 inverse)
    return code, symbols
```

Why reversible programming?

```
def encode(code, symbols):
    [symbol_1, symbol_2] = symbols   # (1)
    code = Categorical(              # (2)
       {'f': 3/8,
        'h': 5/8}).encode(code, symbol_2)
    code = Categorical(              # (3)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).encode(code, symbol_1)
    return code
```

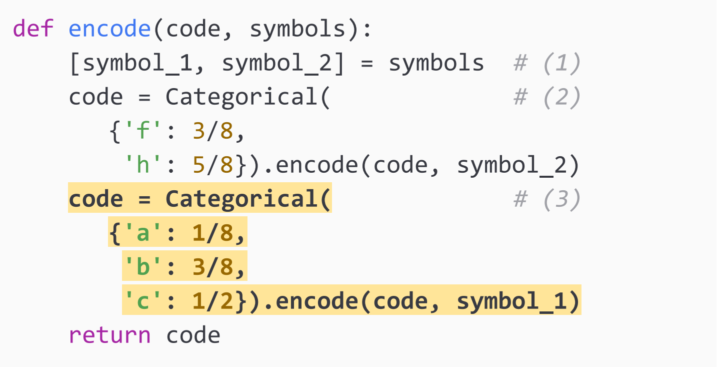
```
def decode(code):  # Inverse of encode
    code, symbol_1 = Categorical(   # (3 inverse)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).decode(code)
    code, symbol_2 = Categorical(   # (2 inverse)
       {'f': 2/8,
        'h': 6/8}).decode(code)
    symbols = [symbol_1, symbol_2]  # (1 inverse)
    return code, symbols
```

Why reversible programming?

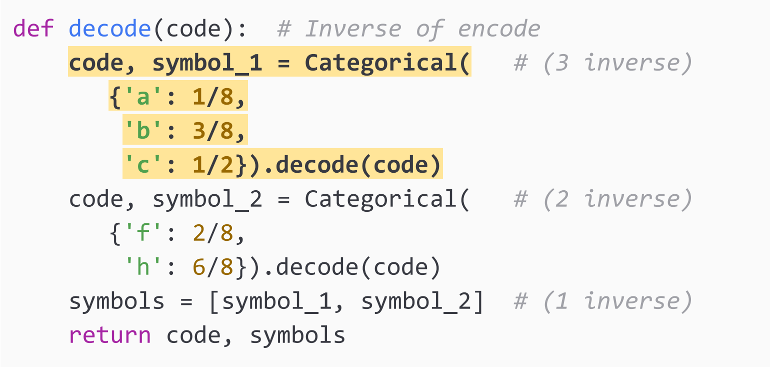
```
def encode(code, symbols):
    [symbol_1, symbol_2] = symbols  # (1)
    code = Categorical(             # (2)
       {'f': 3/8,
        'h': 5/8}).encode(code, symbol_2)
    code = Categorical(             # (3)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).encode(code, symbol_1)
    return code
```

```
def decode(code):  # Inverse of encode
    code, symbol_1 = Categorical(   # (3 inverse)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).decode(code)
    code, symbol_2 = Categorical(   # (2 inverse)
       {'f': 2/8,
        'h': 6/8}).decode(code)
    symbols = [symbol_1, symbol_2]  # (1 inverse)
    return code, symbols
```

Why reversible programming?



```
def encode(code, symbols):
    [symbol_1, symbol_2] = symbols  # (1)
    code = Categorical(              # (2)
       {'f': 3/8,
        'h': 5/8}).encode(code, symbol_2)
    code = Categorical(              # (3)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).encode(code, symbol_1)
    return code
```



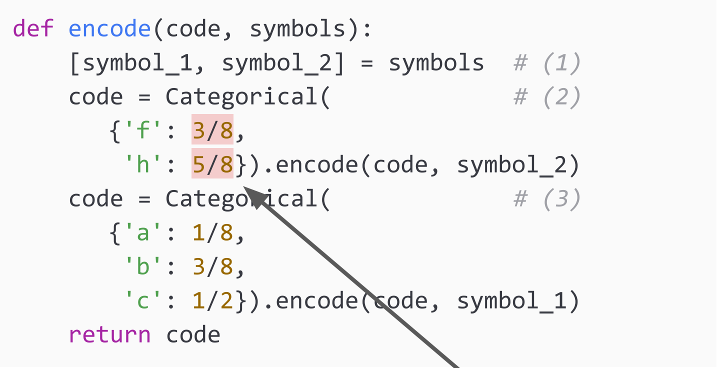
```
def decode(code):  # Inverse of encode
    code, symbol_1 = Categorical(   # (3 inverse)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).decode(code)
    code, symbol_2 = Categorical(   # (2 inverse)
       {'f': 2/8,
        'h': 6/8}).decode(code)
    symbols = [symbol_1, symbol_2]  # (1 inverse)
    return code, symbols
```

Why reversible programming?

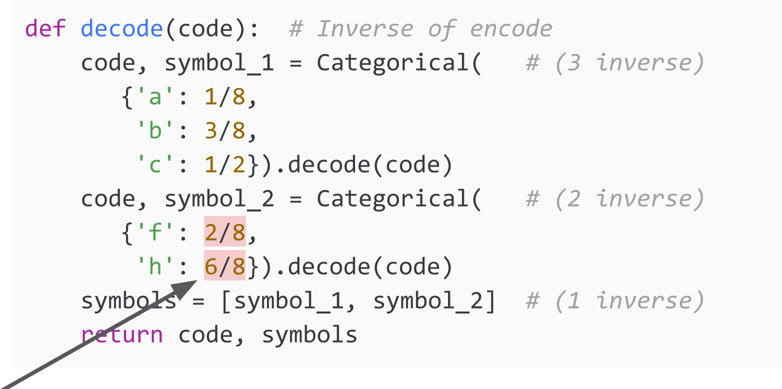
```
def encode(code, symbols):
    [symbol_1, symbol_2] = symbols  # (1)
    code = Categorical(             # (2)
       {'f': 3/8,
        'h': 5/8}).encode(code, symbol_2)
    code = Categorical(             # (3)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).encode(code, symbol_1)
    return code
```

```
def decode(code):  # Inverse of encode
    code, symbol_1 = Categorical(   # (3 inverse)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).decode(code)
    code, symbol_2 = Categorical(   # (2 inverse)
       {'f': 2/8,
        'h': 6/8}).decode(code)
    symbols = [symbol_1, symbol_2]  # (1 inverse)
    return code, symbols
```

Why reversible programming?



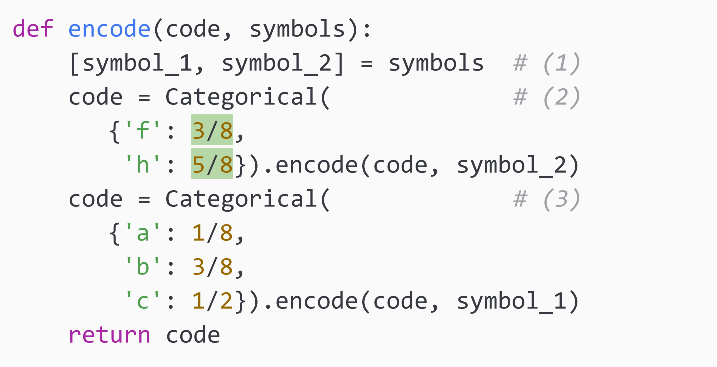
```python
def encode(code, symbols):
    [symbol_1, symbol_2] = symbols  # (1)
    code = Categorical(             # (2)
       {'f': 3/8,
        'h': 5/8}).encode(code, symbol_2)
    code = Categorical(             # (3)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).encode(code, symbol_1)
    return code
```



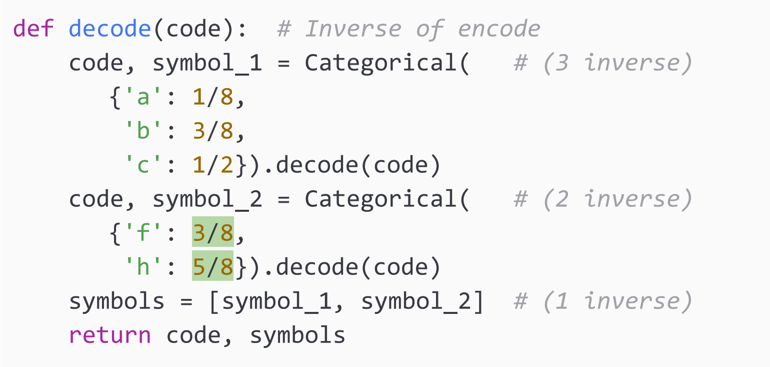
```python
def decode(code):  # Inverse of encode
    code, symbol_1 = Categorical(   # (3 inverse)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).decode(code)
    code, symbol_2 = Categorical(   # (2 inverse)
       {'f': 2/8,
        'h': 6/8}).decode(code)
    symbols = [symbol_1, symbol_2]  # (1 inverse)
    return code, symbols
```

😧 Bug!!

Why reversible programming?



```
def encode(code, symbols):
    [symbol_1, symbol_2] = symbols  # (1)
    code = Categorical(             # (2)
       {'f': 3/8,
        'h': 5/8}).encode(code, symbol_2)
    code = Categorical(             # (3)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).encode(code, symbol_1)
    return code
```



```
def decode(code):  # Inverse of encode
    code, symbol_1 = Categorical(    # (3 inverse)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).decode(code)
    code, symbol_2 = Categorical(    # (2 inverse)
       {'f': 3/8,
        'h': 5/8}).decode(code)
    symbols = [symbol_1, symbol_2]  # (1 inverse)
    return code, symbols
```

Why reversible programming?

```
def encode(code, symbols):
    [symbol_1, symbol_2] = symbols  # (1)
    code = Categorical(             # (2)
       {'f': 3/8,
        'h': 5/8}).encode(code, symbol_2)
    code = Categorical(             # (3)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).encode(code, symbol_1)
    return code
```

```
def decode(code):  # Inverse of encode
    code, symbol_1 = Categorical(   # (3 inverse)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).decode(code)
    code, symbol_2 = Categorical(   # (2 inverse)
       {'f': 3/8,
        'h': 5/8}).decode(code)
    symbols = [symbol_1, symbol_2]  # (1 inverse)
    return code, symbols
```

Question: can we prevent such bugs automatically?

Why reversible programming?

```
def encode(code, symbols):
    [symbol_1, symbol_2] = symbols  # (1)
    code = Categorical(              # (2)
       {'f': 3/8,
        'h': 5/8}).encode(code, symbol_2)
    code = Categorical(              # (3)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).encode(code, symbol_1)
    return code
```

```
def decode(code):  # Inverse of encode
    code, symbol_1 = Categorical(   # (3 inverse)
       {'a': 1/8,
        'b': 3/8,
        'c': 1/2}).decode(code)
    code, symbol_2 = Categorical(   # (2 inverse)
       {'f': 3/8,
        'h': 5/8}).decode(code)
    symbols = [symbol_1, symbol_2]  # (1 inverse)
    return code, symbols
```

Question: can we prevent such bugs automatically?

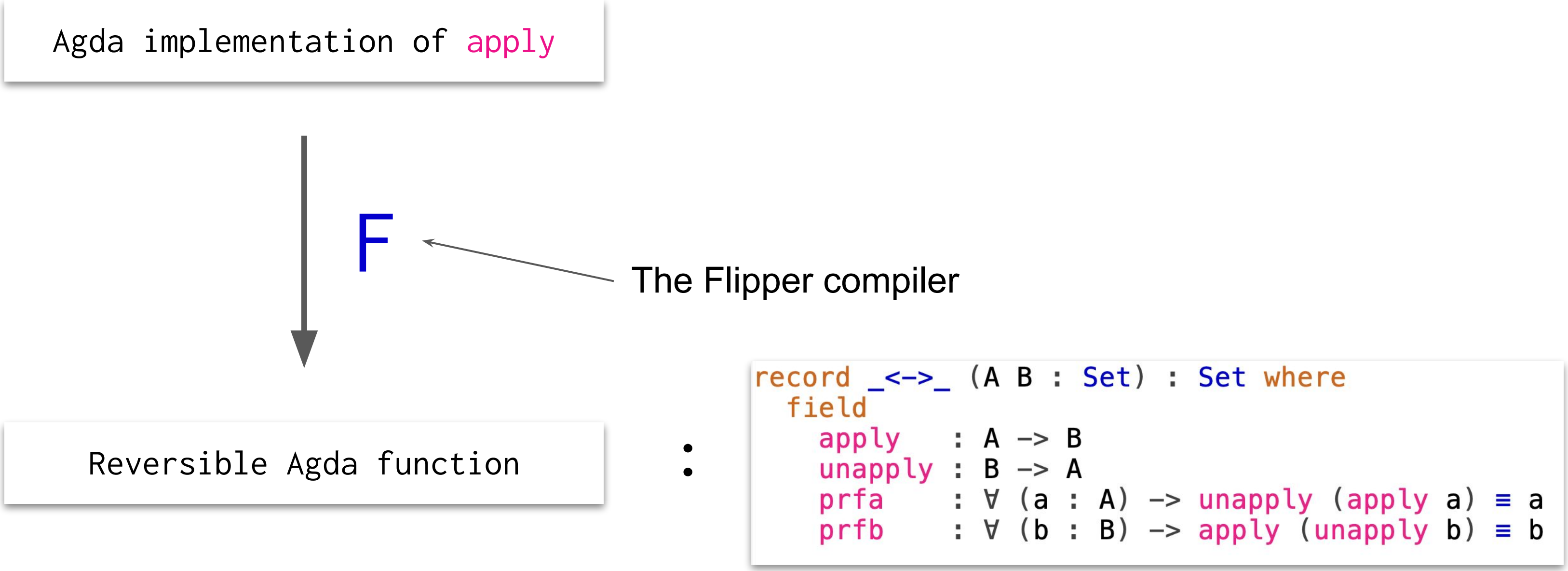
Idea: a *reversible* language, in which programs can be *done* and *undone.*

Flipper

Flipper is embedded in the (pure functional) language Agda.

Flipper

Flipper is embedded in the (pure functional) language Agda.



Agda implementation of apply

F

The Flipper compiler

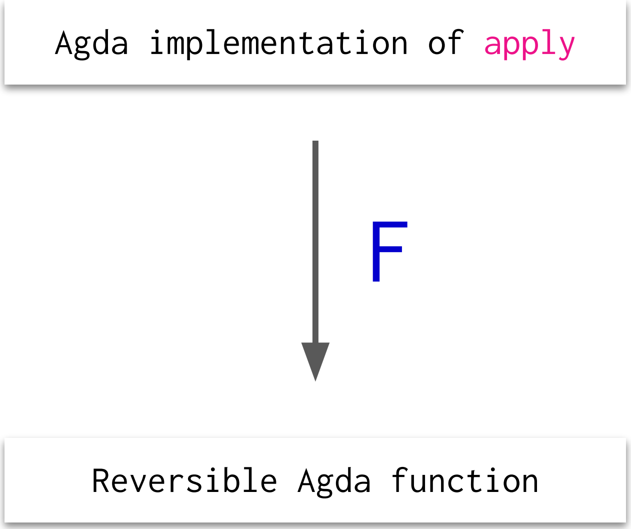
Reversible Agda function

:

```
record _<->_ (A B : Set) : Set where
  field
    apply   : A -> B
    unapply : B -> A
    prfa    : ∀ (a : A) -> unapply (apply a) ≡ a
    prfb    : ∀ (b : B) -> apply (unapply b) ≡ b
```

Flipper

Flipper is embedded in the (pure functional) language Agda.

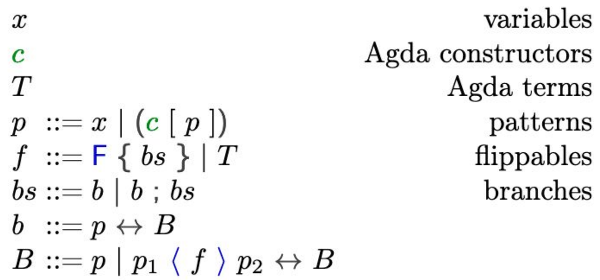


Agda implementation of apply

F

Reversible Agda function

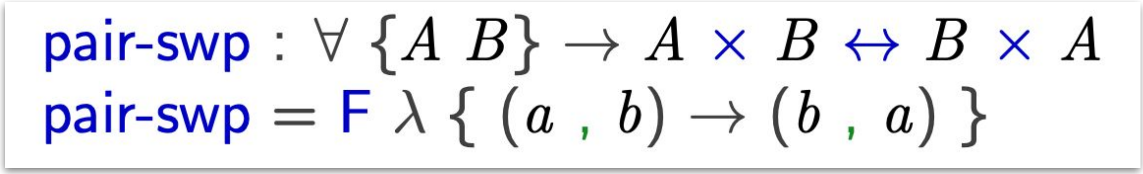
Grammar for apply



x		variables
c		Agda constructors
T		Agda terms
p	$::= x \mid (c\ [\ p\ ])$	patterns
f	$::= \mathsf{F}\ \{\ bs\ \} \mid T$	flippables
bs	$::= b \mid b\ ;\ bs$	branches
b	$::= p \leftrightarrow B$	
B	$::= p \mid p_1\ \langle\ f\ \rangle\ p_2 \leftrightarrow B$	

Flipper

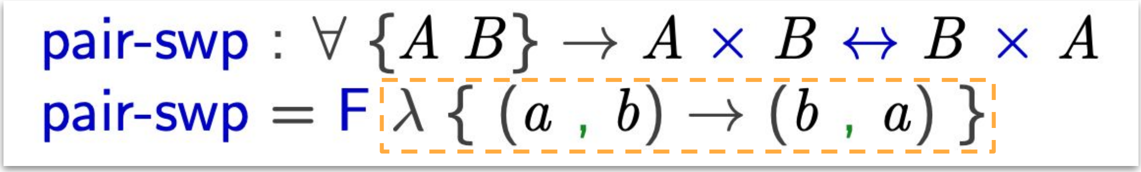
Example: swap the elements of a pair



```
pair-swp : ∀ {A B} → A × B ↔ B × A
pair-swp = F λ { (a , b) → (b , a) }
```

Flipper

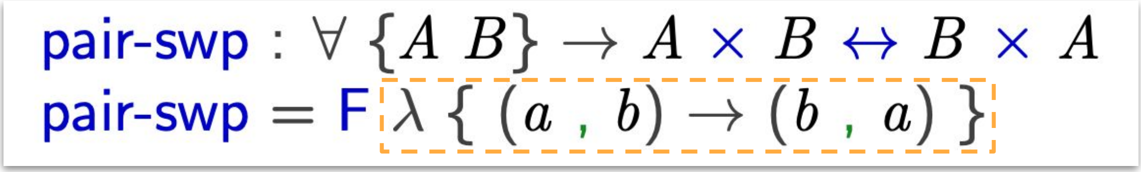
Example: swap the elements of a pair



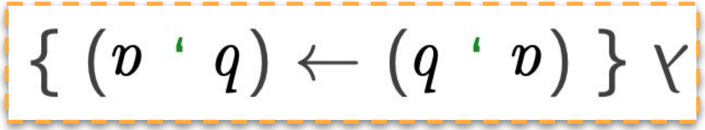
```
pair-swp : ∀ {A B} → A × B ↔ B × A
pair-swp = F λ { (a , b) → (b , a) }
```

Flipper

Example: swap the elements of a pair

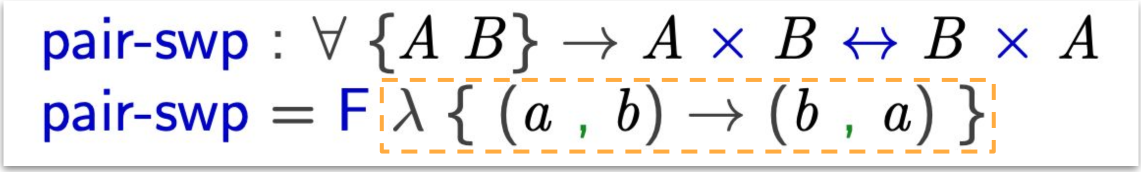


```
pair-swp : ∀ {A B} → A × B ↔ B × A
pair-swp = F λ { (a , b) → (b , a) }
```

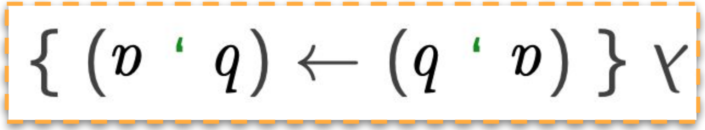


Flipper

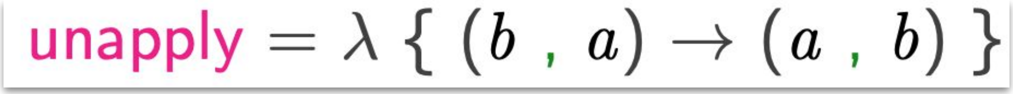
Example: swap the elements of a pair



```
pair-swp : ∀ {A B} → A × B ↔ B × A
pair-swp = F λ { (a , b) → (b , a) }
```



```
λ { (a , b) → (b , a) }
```



```
unapply = λ { (b , a) → (a , b) }
```

Next steps for Flipper

- Finish compression implementation (nearly done)
- Maybe make standalone
- Look out for other use cases

The End

Thanks for listening

P.S. Check out https://github.com/j-towns/flipper...