Bring Your Own Datatypes: Enabling Custom Datatype Exploration in Deep Learning

Gus Smith, Qualifying Exam
February 24th, 2020

sampl

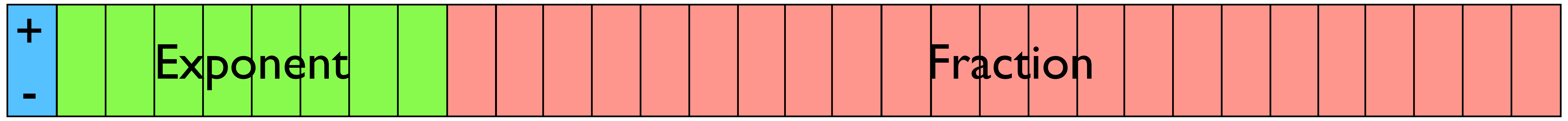
PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

tvm.ai

By “datatypes”, I mean **numerical datatypes:** how the hardware represents and operates on real numbers.

IEEE 754 Floating Point

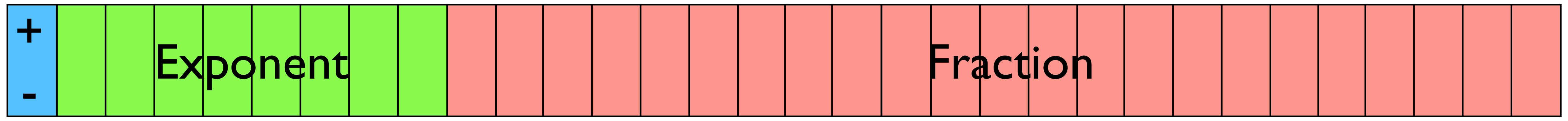
Single precision (32 bit)



$$\text{value} \approx \text{sign} * 2^{\text{exponent}} * 1.\text{fraction}$$

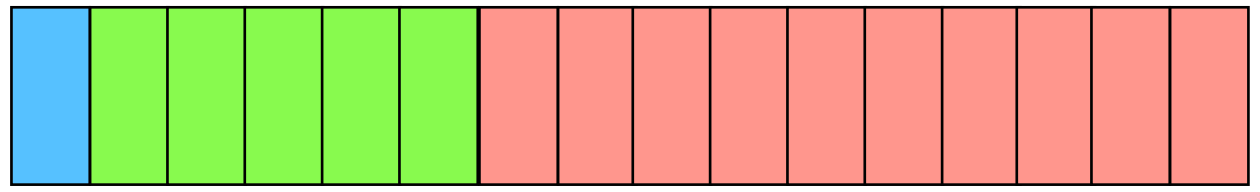
IEEE 754 Floating Point

Single precision (32 bit)

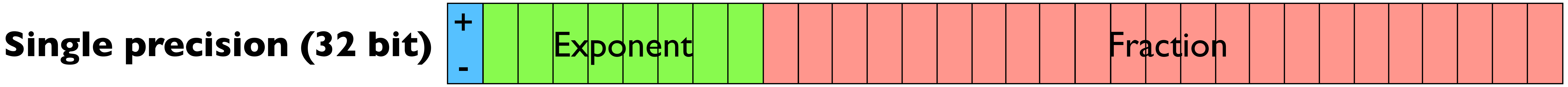


$$\text{value} \approx \text{sign} * 2^{\text{exponent}} * 1.\text{fraction}$$

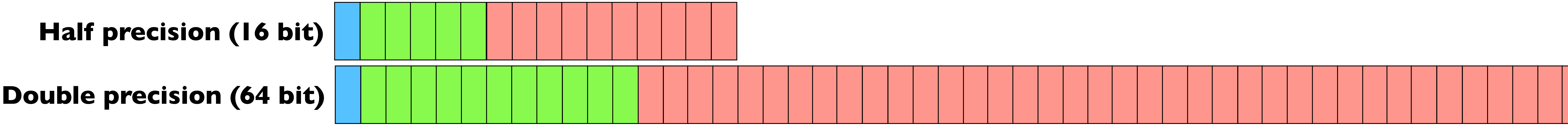
Half precision (16 bit)



IEEE 754 Floating Point

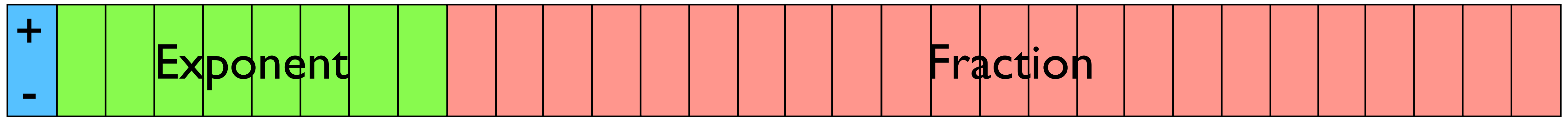


$$\text{value} \approx \text{sign} * 2^{\text{exponent}} * 1.\text{fraction}$$



IEEE 754 Floating Point

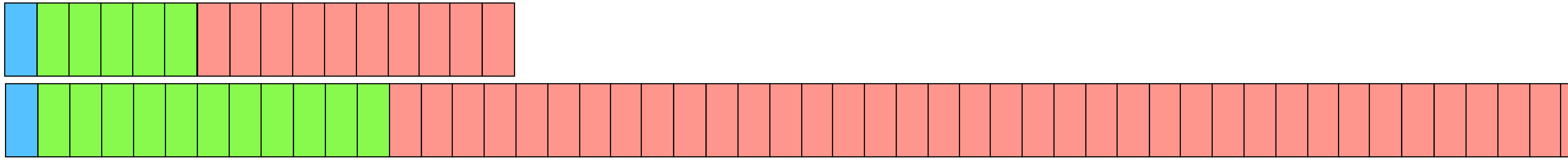
Single precision (32 bit)



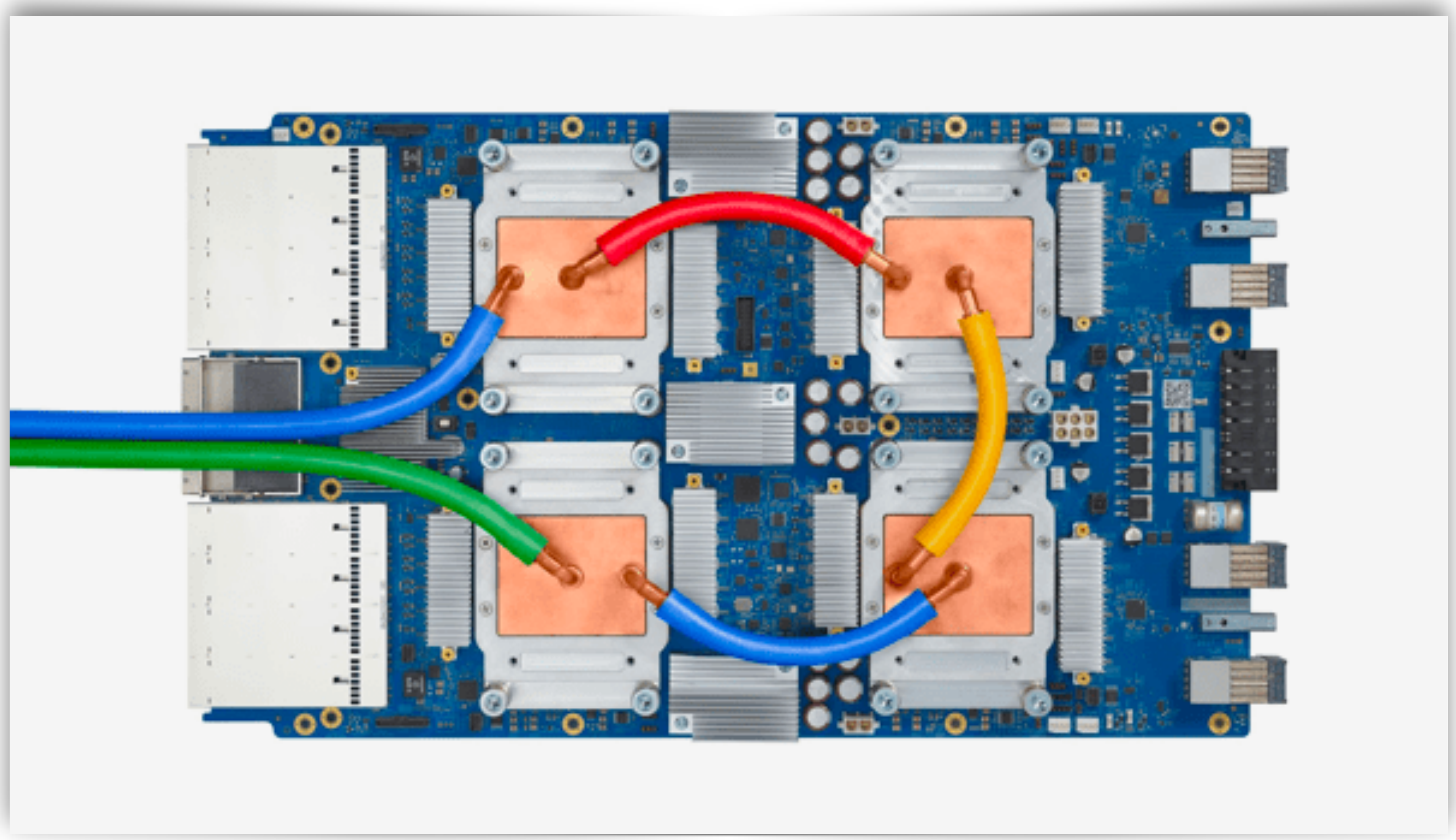
$$\text{value} \approx \text{sign} * 2^{\text{exponent}} * 1.\text{fraction}$$

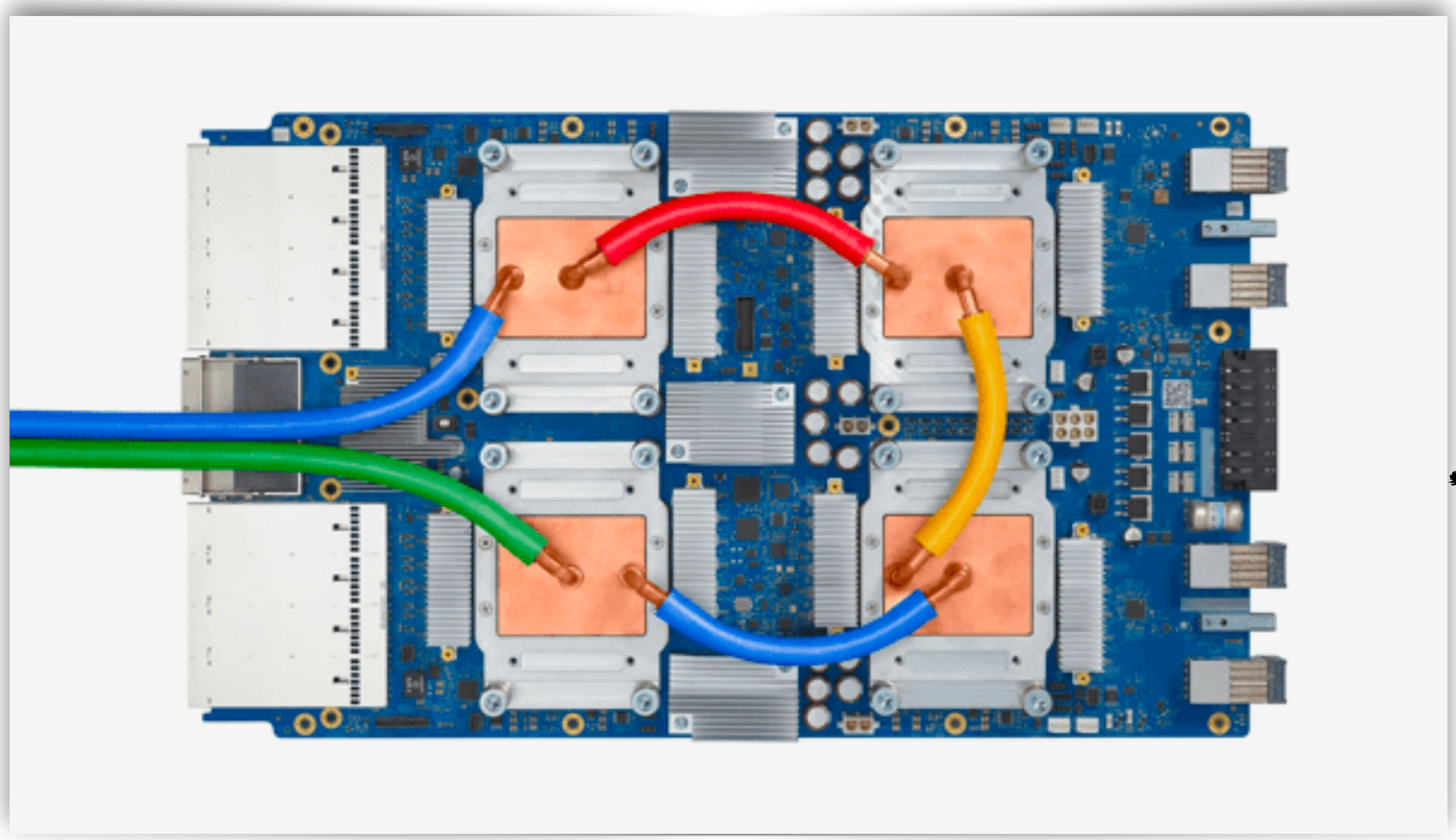
Half precision (16 bit)

Double precision (64 bit)

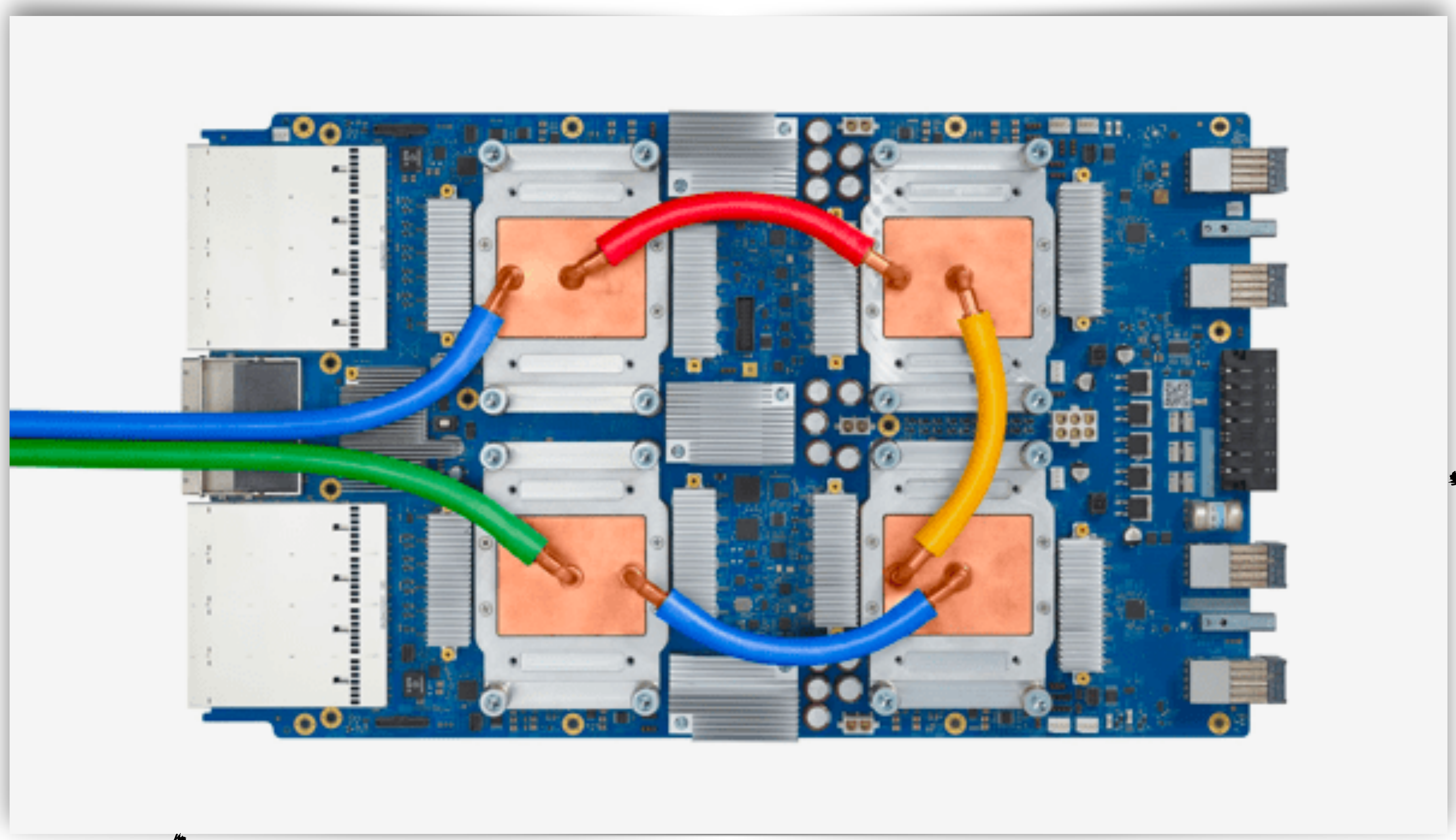


Has remained an industry standard for more than thirty years!





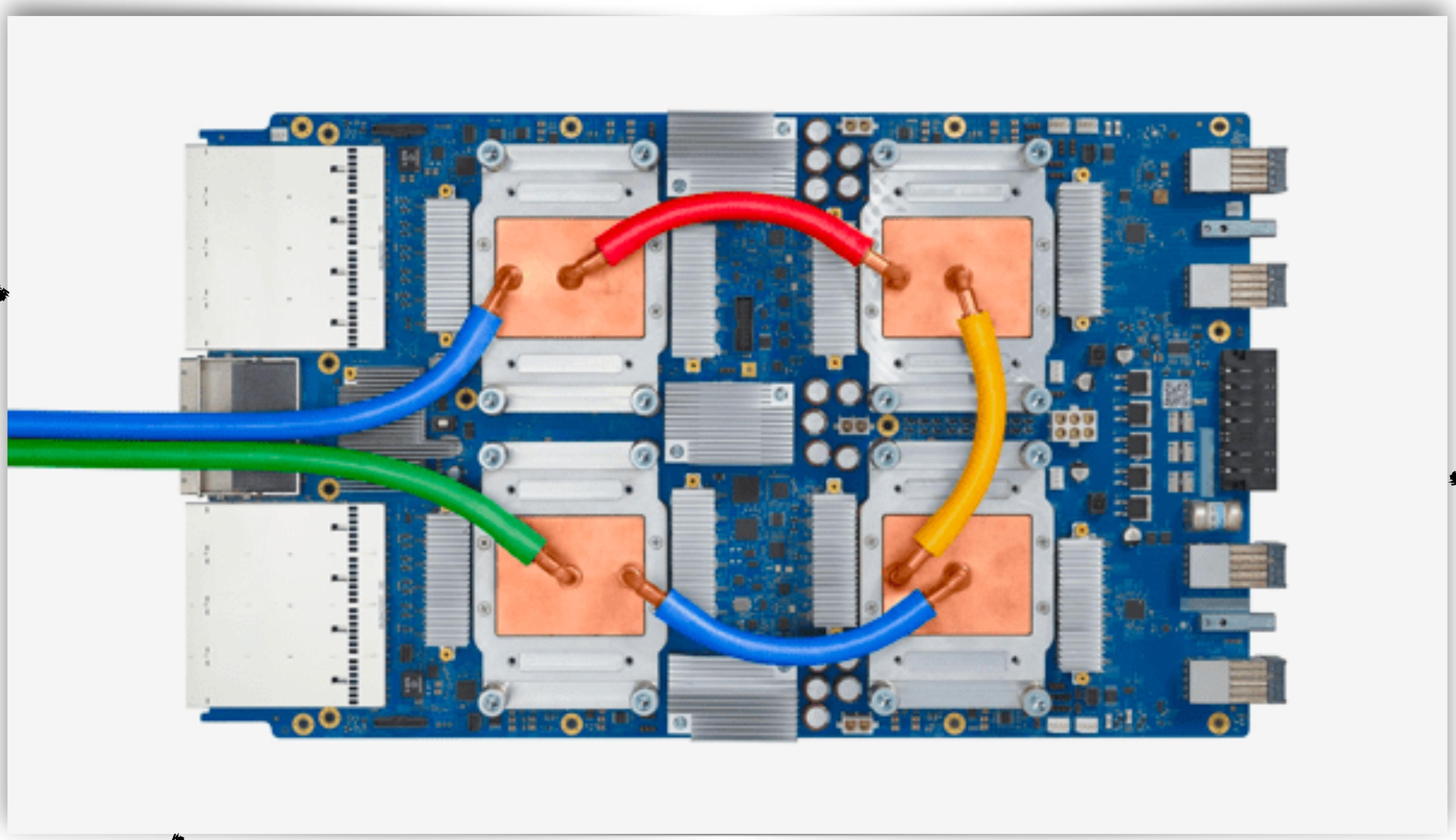
Should be fast and power-efficient



Should be fast and power-efficient

Needs small weights and activations
to maximize usage of chip area

Only needs to represent a specific range of values:
Weights and activations cluster (e.g. around [-1, 1])

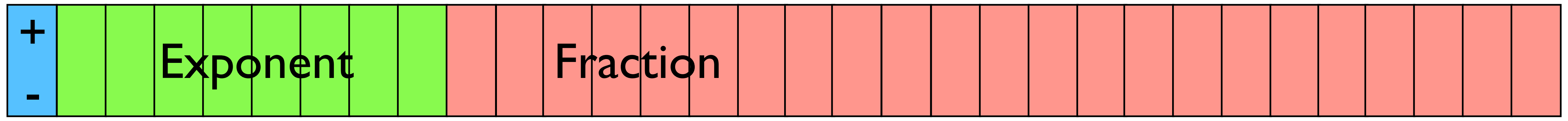


Should be fast and power-efficient

Needs small weights and activations
to maximize usage of chip area

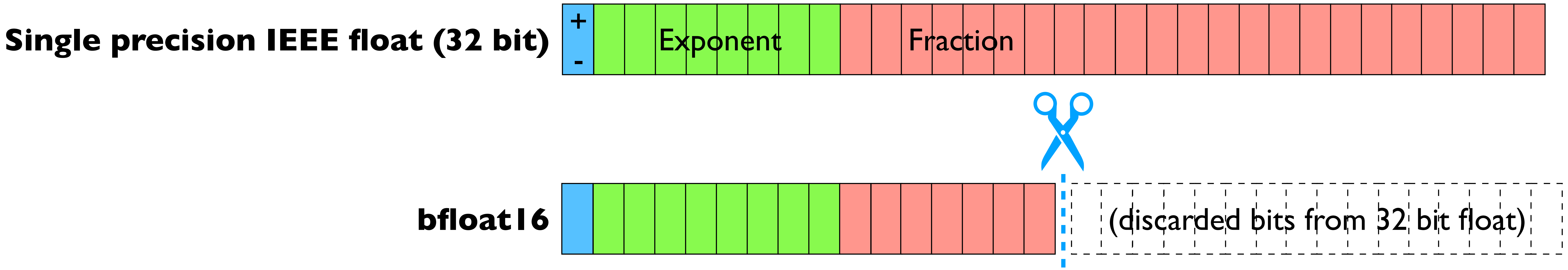
bfloat16

Single precision IEEE float (32 bit)



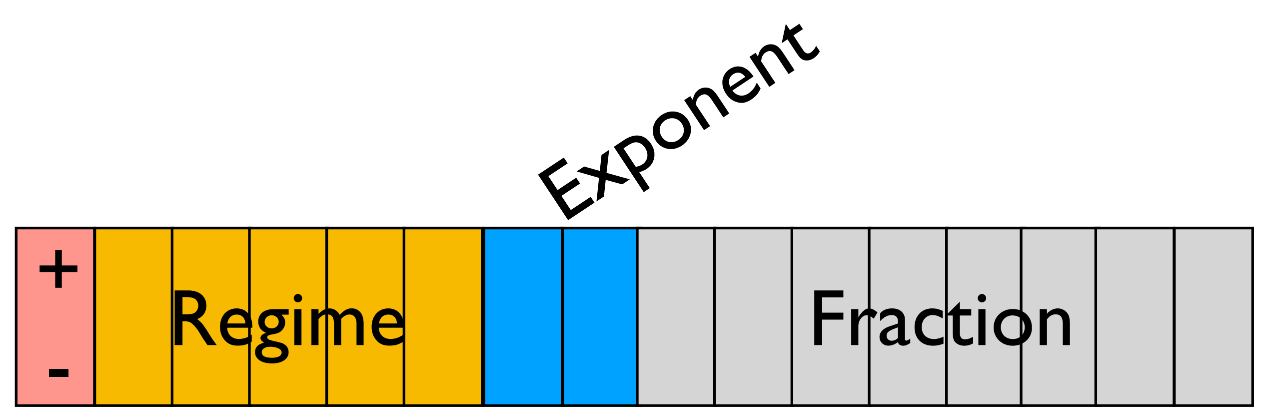
See https://cloud.google.com/tpu/docs/bfloat16

bfloat16



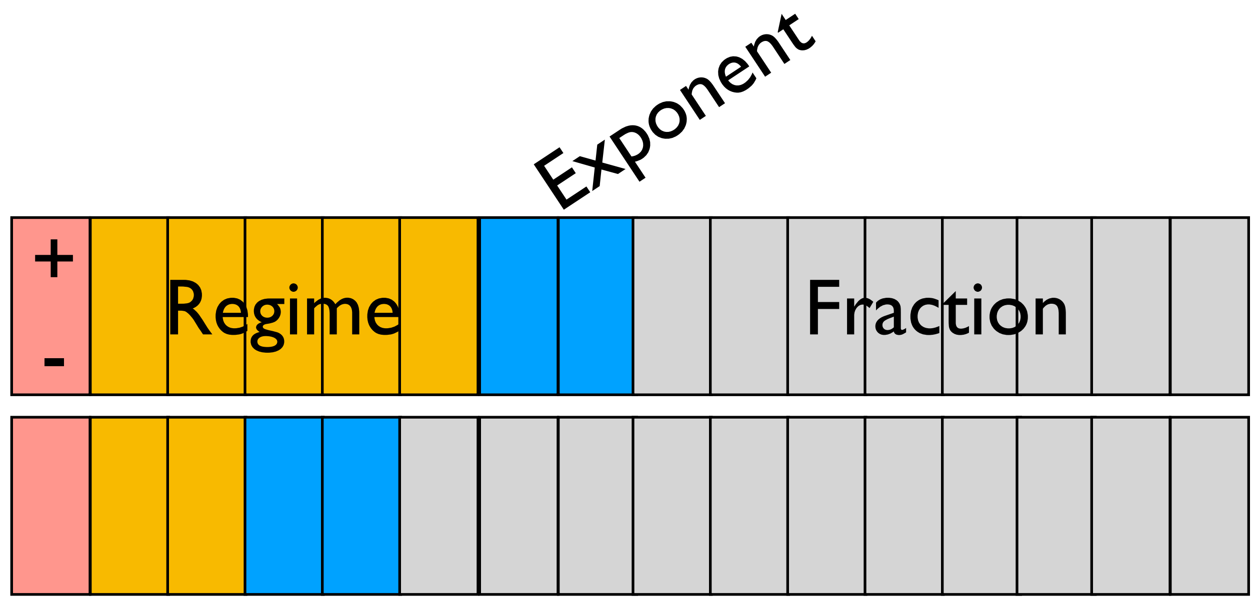
See https://cloud.google.com/tpu/docs/bfloat16

Posits



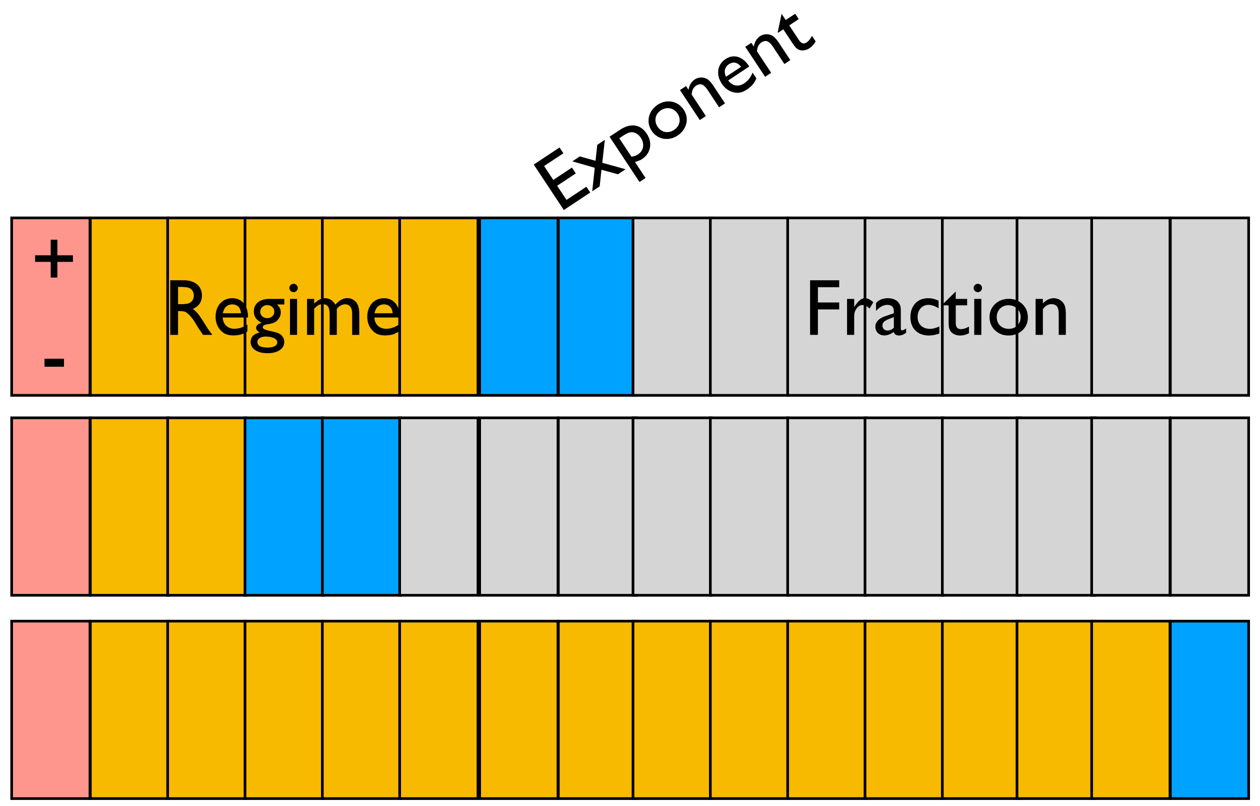
See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Posits



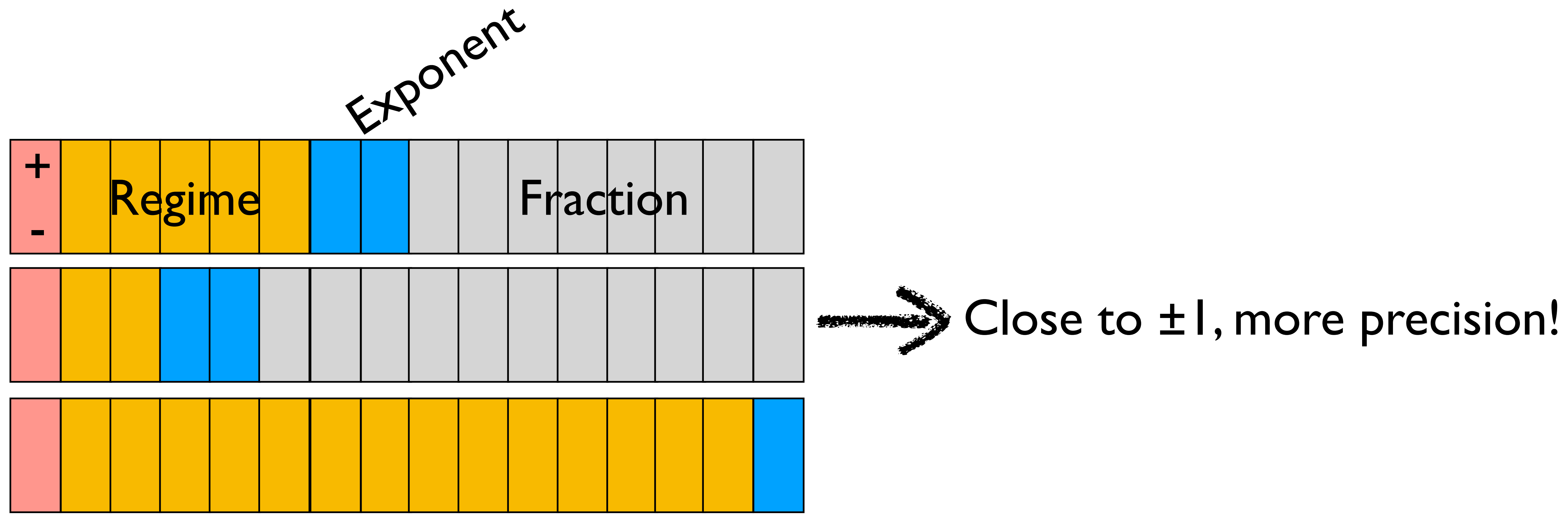
See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Posits



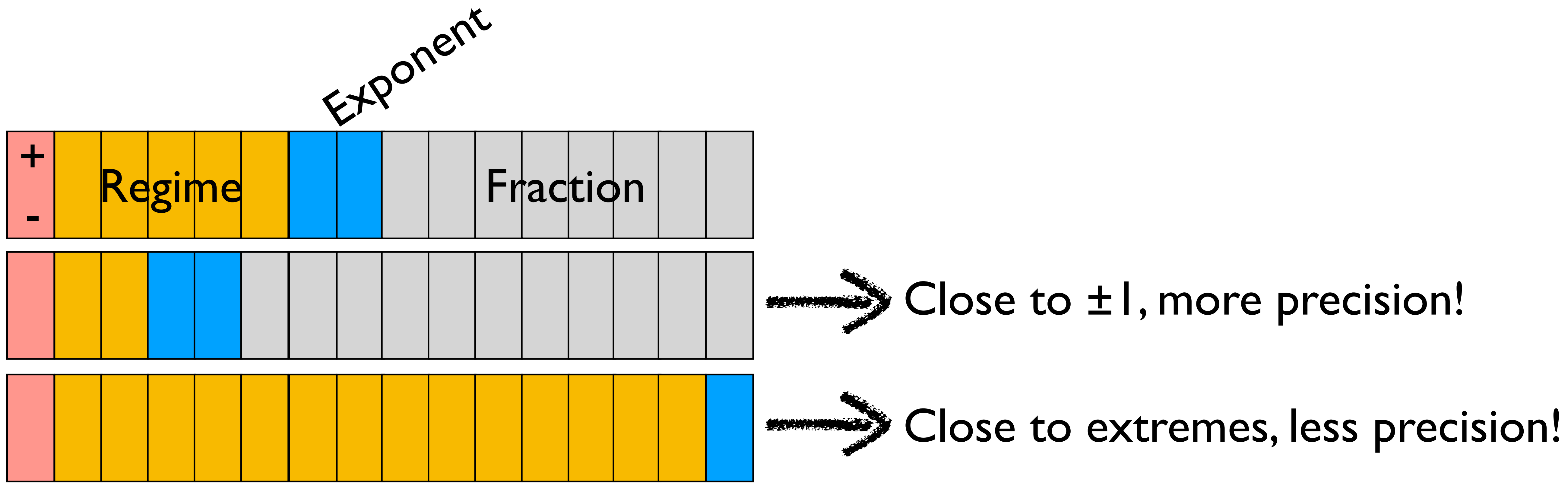
See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Posits



See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

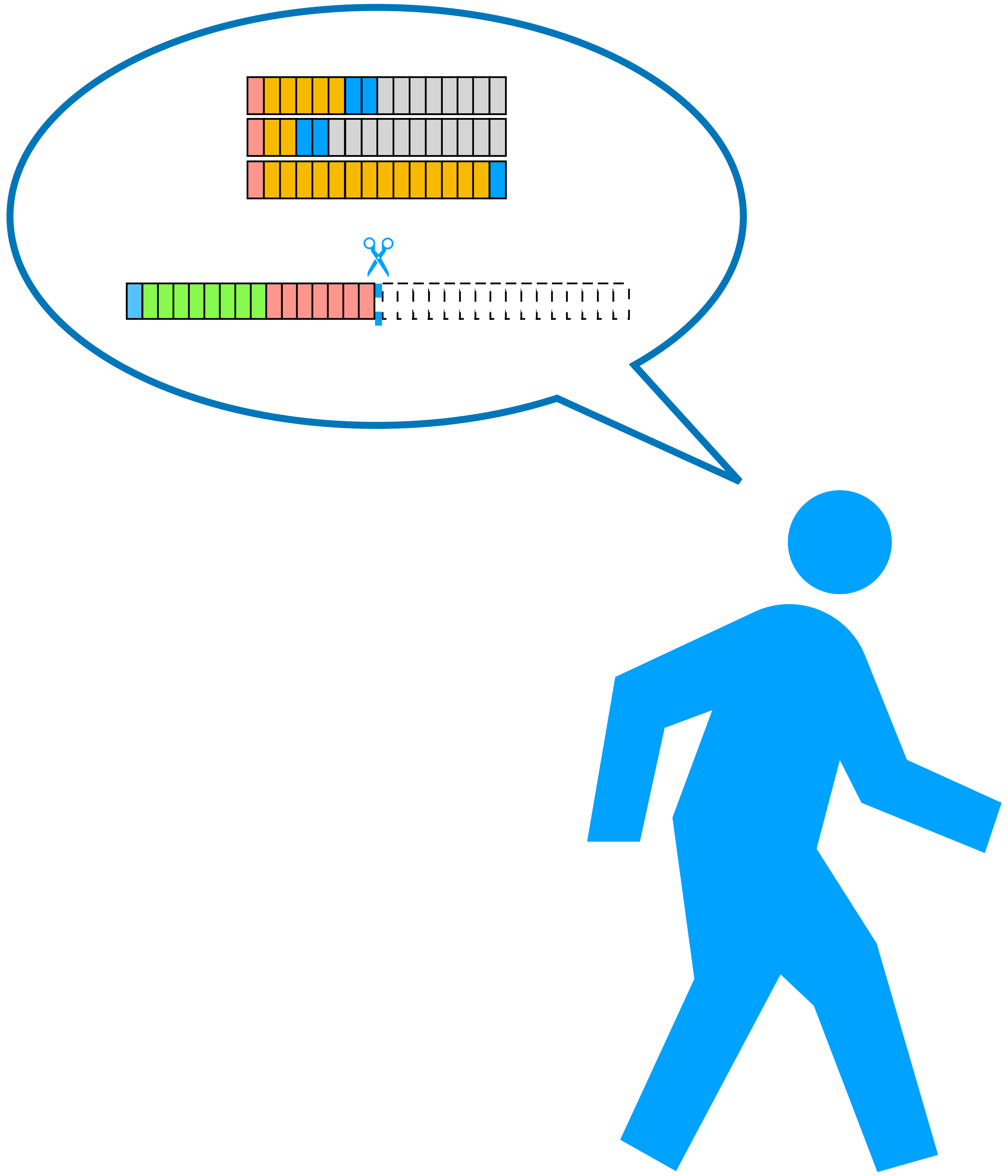
Posits

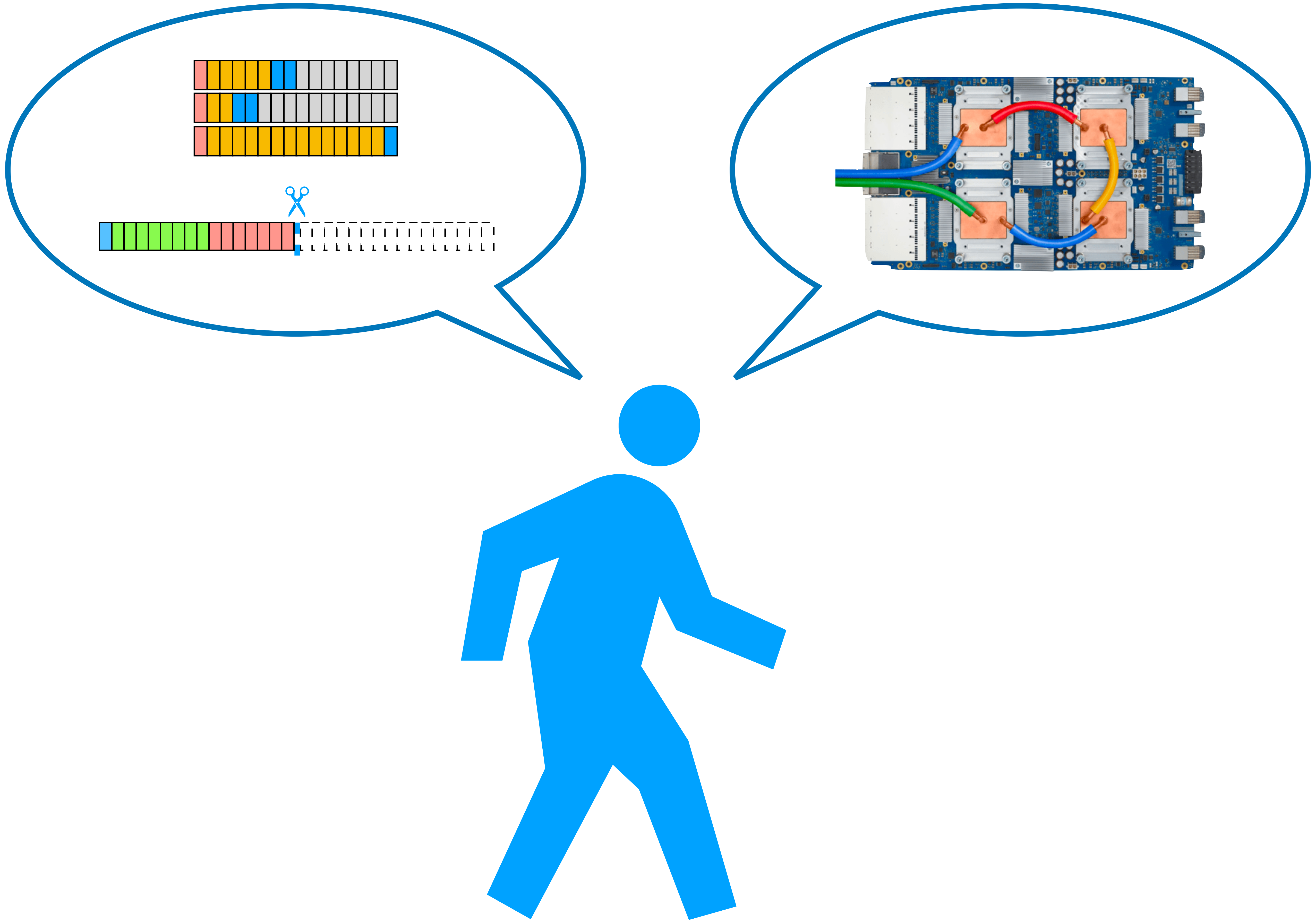


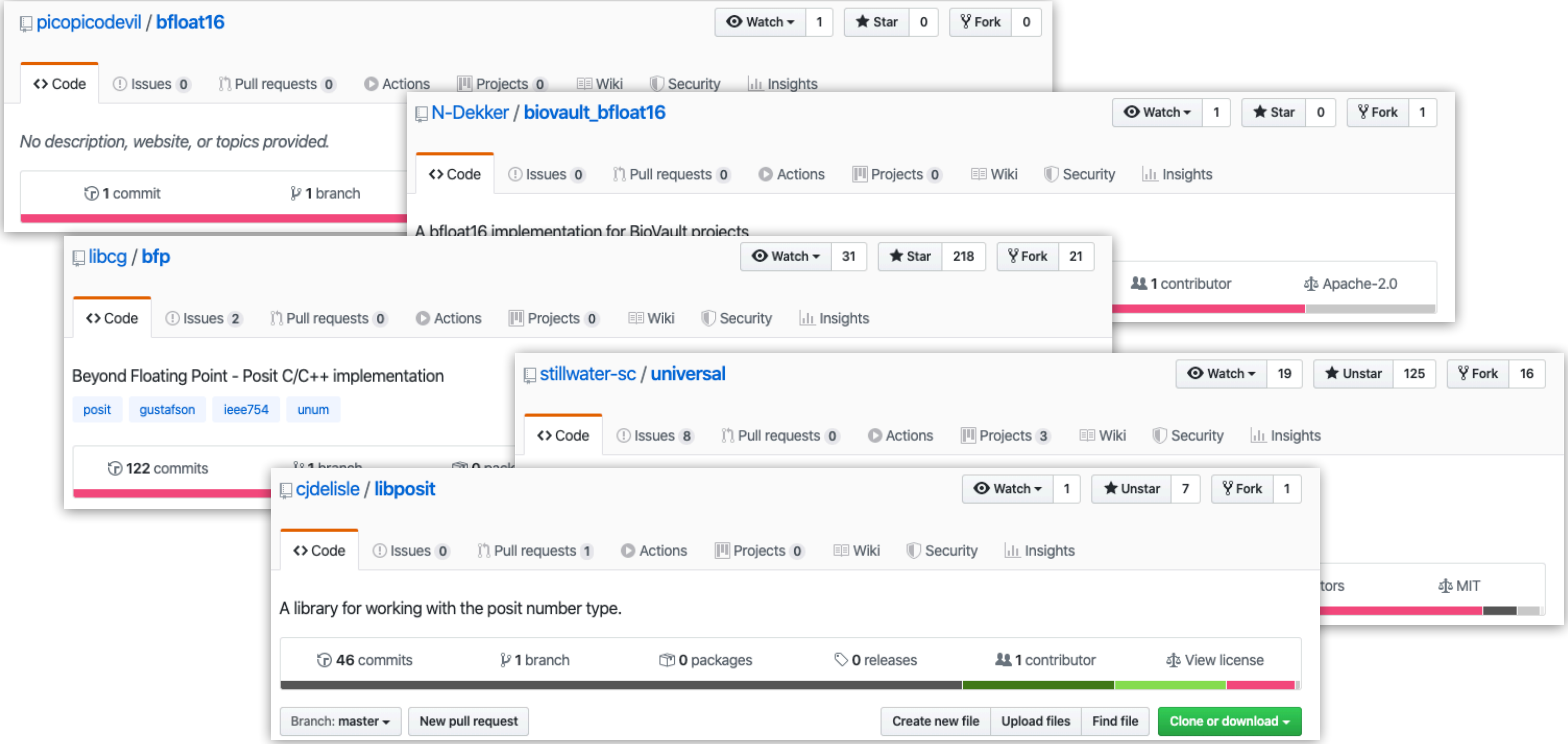
See www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf

Deep learning needs hardware specialization…

Deep learning needs hardware specialization…
…and hardware specialization needs new datatypes!






picopicodevil / bfloat16
Watch 1
Star 0
Fork 0
Code
Issues 0
Pull requests 0
Actions
Projects 0
Wiki
Security
Insights
No description, website, or topics provided.
1 commit
1 branch
N-Dekker / biovault_bfloat16
Watch 1
Star 0
Fork 1
Code
Issues 0
Pull requests 0
Actions
Projects 0
Wiki
Security
Insights
A bfloat16 implementation for BioVault projects
1 contributor
Apache-2.0
libcg / bfp
Watch 31
Star 218
Fork 21
Code
Issues 2
Pull requests 0
Actions
Projects 0
Wiki
Security
Insights
Beyond Floating Point - Posit C/C++ implementation
posit
gustafson
ieee754
unum
122 commits
stillwater-sc / universal
Watch 19
Unstar 125
Fork 16
Code
Issues 8
Pull requests 0
Actions
Projects 3
Wiki
Security
Insights
MIT
cjdelisle / libposit
Watch 1
Unstar 7
Fork 1
Code
Issues 0
Pull requests 1
Actions
Projects 0
Wiki
Security
Insights
A library for working with the posit number type.
46 commits
1 branch
0 packages
0 releases
1 contributor
View license
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download

https://github.com/cjdelisle/libposit

Types

libposit defines the following types:

- `posit8_t` 8 bit posit
  - `posit8x2_t` pair of 8 bit posits, result of `*_exact` functions.
- `posit16_t` 16 bit posit
  - `posit16x2_t` pair of 16 bit posits, result of `*_exact` functions.
- `posit32_t` 32 bit posit
  - `posit32x2_t` pair of 32 bit posits, result of `*_exact` functions.
- `posit64_t` 64 bit posit
  - `posit64x2_t` pair of 64 bit posits, result of `*_exact` functions.
- `posit128_t` 128 bit posit, result of posit64 `*_promote` functions. Only functions which work on a `posit128_t` are `posit128_iexp()` and `posit128_fract()`.

https://github.com/cjdelisle/libposit

Types

libposit defines the

- `posit8_t` 8
  - `posit8x`
- `posit16_t` 1
  - `posit16`
- `posit32_t` 3
  - `posit32`
- `posit64_t` 6
  - `posit64`
- `posit128_t`

  `posit128_ie`

Operations

For each of these types, libposit provides the following functions:

- `posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y)` : Add two posits together, output a rounded result.
- `posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y)` : Subtract one posit from another, output a rounded result.
- `posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y)` : Output 2 results, `sum` and `remainder` which when added will result in the exact same value as the inputs `x` and `y` but where the absolute value of `remainder` is as small as possible.
- `posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y)` : Exactly the same as `posit<X>_add_exact(x, y)` but with the sign of `y` flipped.
- `posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y)` : Multiply two posits, round the result to nearest
- `posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y)` : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- `posit8_t` 8 |
  - `posit8x`
- `posit16_t` 1
  - `posit16`
- `posit32_t` 3
  - `posit32`
- `posit64_t` 6
  - `posit64`
- `posit128_t`

  `posit128_ie`

Operations

For each of these types, libposit provides the following functions:

- `posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y)` : Add two posits together, output a rounded result.
- `posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y)` : Subtract one posit from another, output a rounded result.
- `posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y)` : Output 2 results, `sum` and `remainder` which when added will result in the exact same value as the inputs `x` and `y` but where the absolute value of `remainder` is as small as possible.
- `posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y)` : Exactly the same as `posit<X>_add_exact(x, y)` but with the sign of `y` flipped.
- `posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y)` : Multiply two posits, round the result to nearest
- `posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y)` : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- `posit8_t` 8
  - `posit8x`
- `posit16_t` 1
  - `posit16`
- `posit32_t` 3
  - `posit32`
- `posit64_t` 6
  - `posit64`
- `posit128_t`

  `posit128_ie`

Operations

For each of these types, libposit provides the following functions:

- `posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y)` : Add two posits together, output a rounded result.
- `posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y)` : Subtract one posit from another, output a rounded result.
- `posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y)` : Output 2 results, `sum` and `remainder` which when added will result in the exact same value as the inputs `x` and `y` but where the absolute value of `remainder` is as small as possible.
- `posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y)` : Exactly the same as `posit<X>_add_exact(x, y)` but with the sign of `y` flipped.
- `posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y)` : Multiply two posits, round the result to nearest
- `posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y)` : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- posit8_t 8
  - posit8x
- posit16_t 1
  - posit16
- posit32_t 3
  - posit32
- posit64_t 6
  - posit64
- posit128_t

  posit128_ie

Operations

For each of these types, libposit provides the following functions:

- posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y) : Add two posits together, output a rounded result.
- posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y) : Subtract one posit from another, output a rounded result.
- posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y) : Output 2 results, sum and remainder which when added will result in the exact same value as the inputs x and y but where the absolute value of remainder is as small as possible.
- posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y) : Exactly the same as posit<X>_add_exact(x, y) but with the sign of y flipped.
- posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y) : Multiply two posits, round the result to nearest
- posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y) : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- posit8_t 8
  - posit8x
- posit16_t 1
  - posit16
- posit32_t 3
  - posit32
- posit64_t 6
  - posit64
- posit128_t

  posit128_ie

Operations

For each of these types, libposit provides the following functions:

- posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y) : Add two posits together, output a rounded result.
- posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y) : Subtract one posit from another, output a rounded result.
- posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y) : Output 2 results, sum and remainder which when added will result in the exact same value as the inputs x and y but where the absolute value of remainder is as small as possible.
- posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y) : Exactly the same as posit<X>_add_exact(x, y) but with the sign of y flipped.
- posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y) : Multiply two posits, round the result to nearest
- posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y) : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

- posit8_t 8
  - posit8x
- posit16_t 1
  - posit16
- posit32_t 3
  - posit32
- posit64_t 6
  - posit64
- posit128_t

  posit128_ie

Operations

For each of these types, libposit provides the following functions:

- posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y) : Add two posits together, output a rounded result.
- posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y) : Subtract one posit from another, output a rounded result.
- posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y) : Output 2 results, sum and remainder which when added will result in the exact same value as the inputs x and y but where the absolute value of remainder is as small as possible.
- posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y) : Exactly the same as posit<X>_add_exact(x, y) but with the sign of y flipped.
- posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y) : Multiply two posits, round the result to nearest
- posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y) : Divide two posits, round the result to nearest

https://github.com/cjdelisle/libposit

Types

libposit defines the

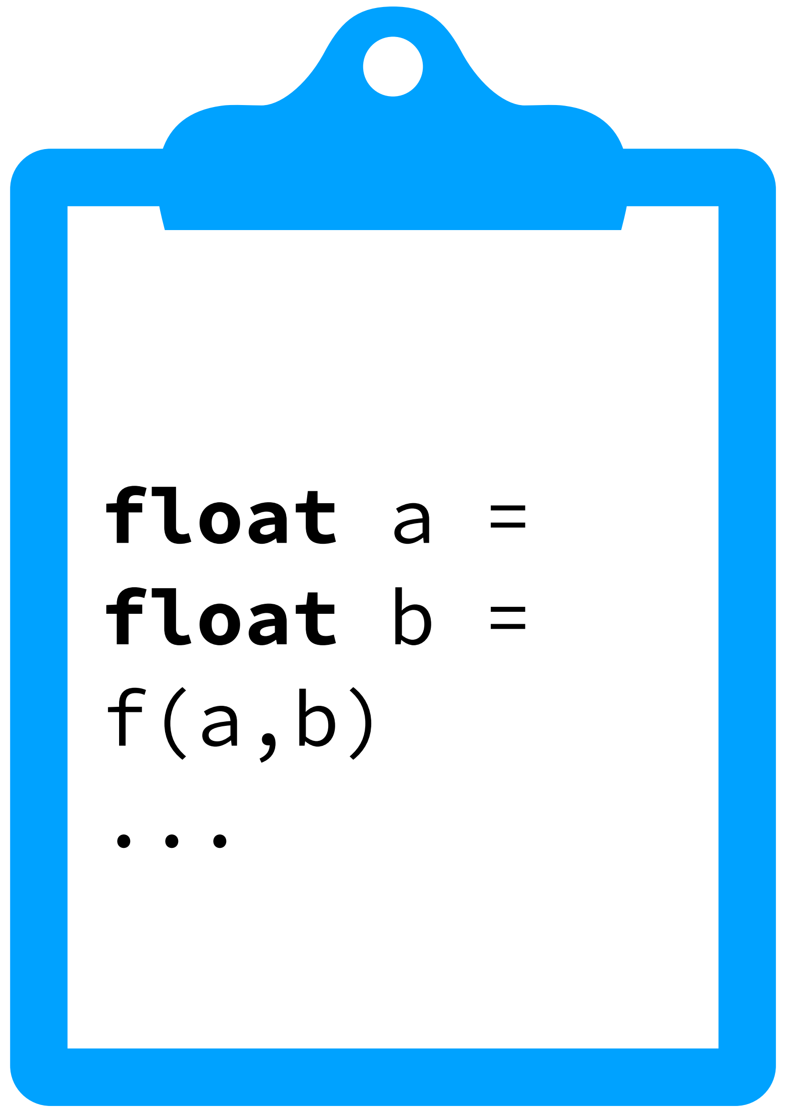
- `posit8_t` 8
  - `posit8x`
- `posit16_t` 1
  - `posit16`
- `posit32_t` 3
  - `posit32`
- `posit64_t` 6
  - `posit64`
- `posit128_t`
  `posit128_ie`

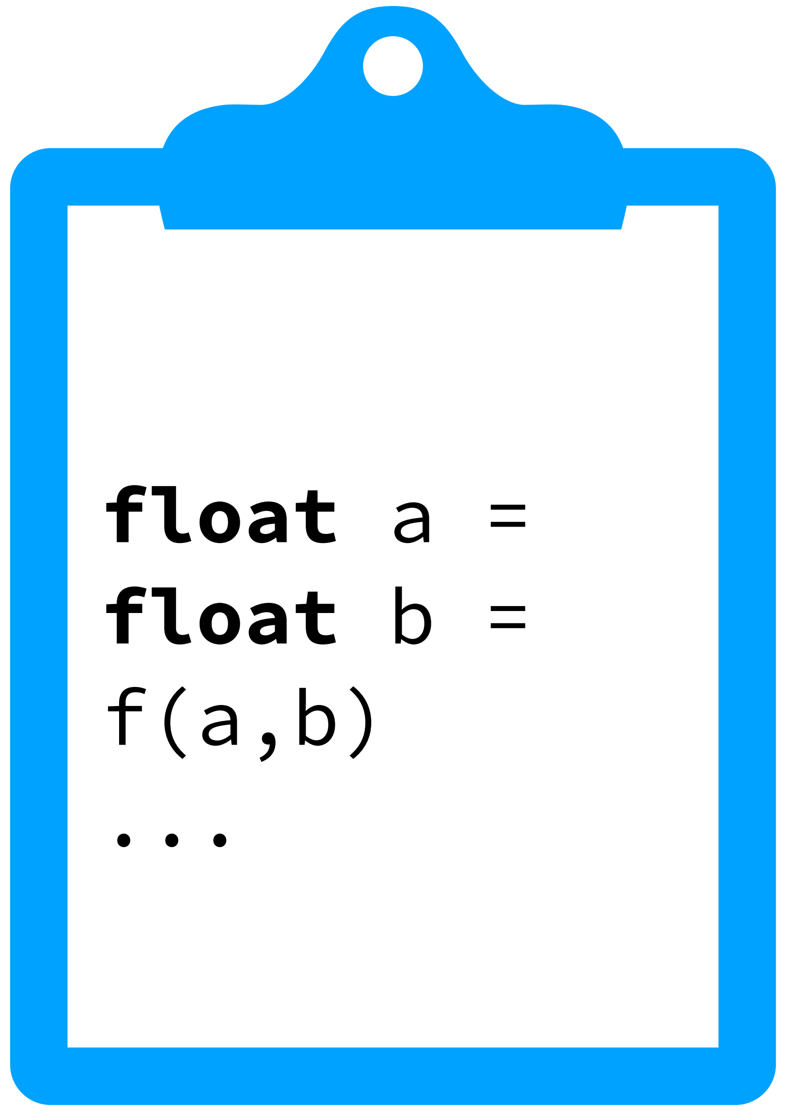
Operations

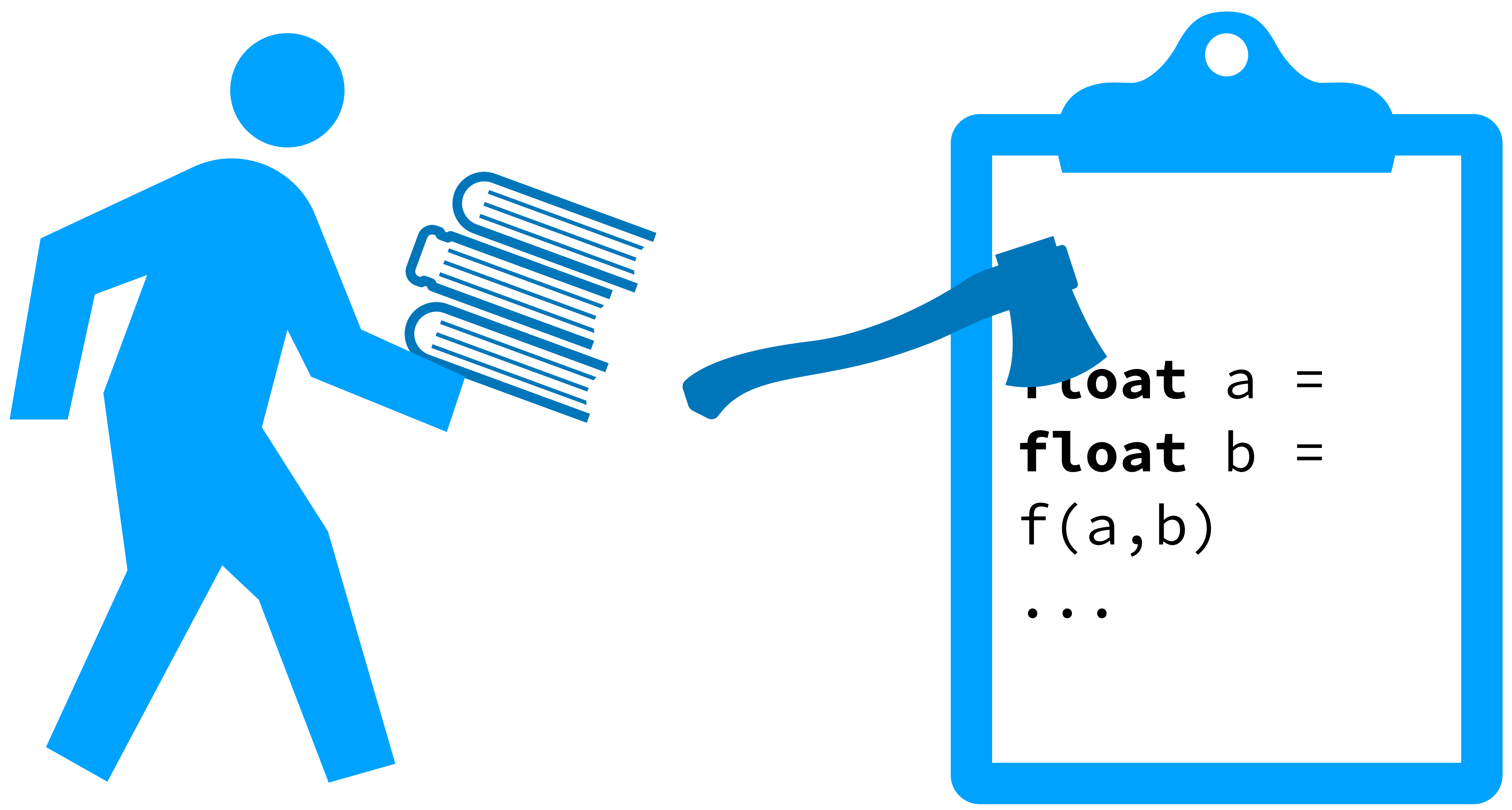
For each of these types, libposit provides the following functions:

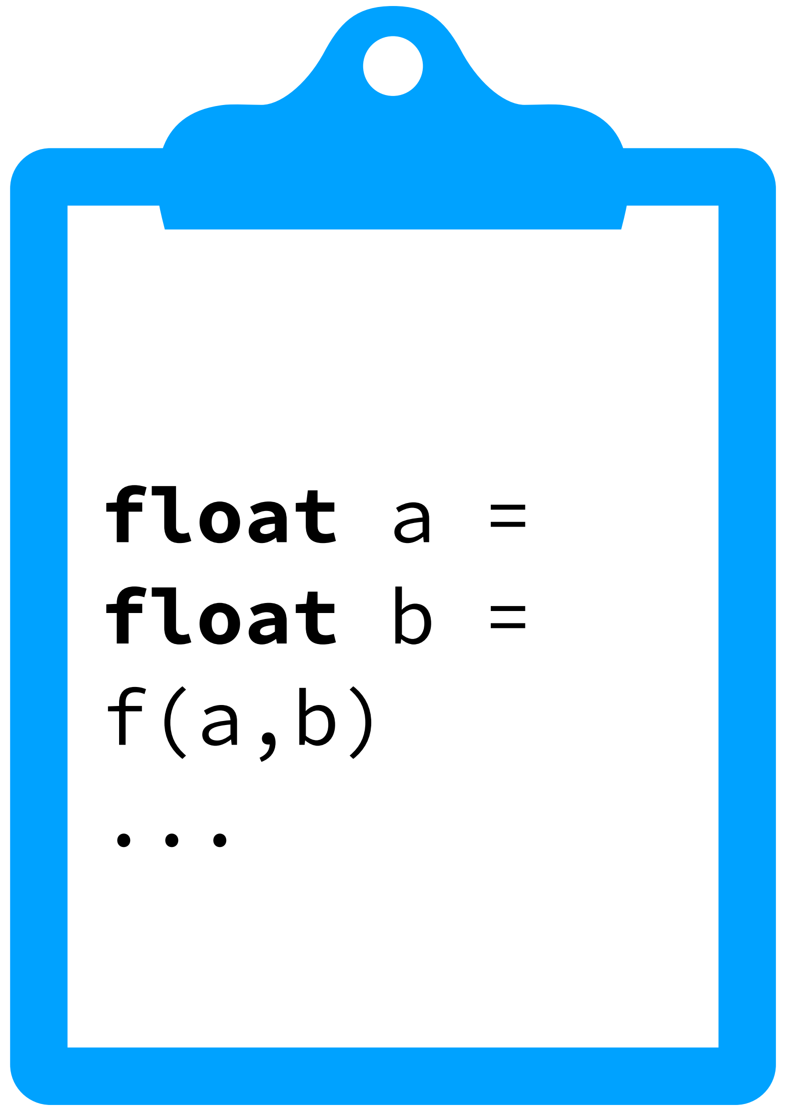
- `posit<X>_t posit<X>_add(posit<X>_t x, posit<X>_t y)`: Add two posits together, output a rounded result.
- `posit<X>_t posit<X>_sub(posit<X>_t x, posit<X>_t y)`: Subtract one posit from another, output a rounded result.
- `posit<X>x2_t posit<X>_add_exact(posit<X>_t x, posit<X>_t y)`: Output 2 results, `sum` and `remainder` which when added will result in the exact same value as the inputs `x` and `y` but where the absolute value of `remainder` is as small as possible.
- `posit<X>x2_t posit<X>_sub_exact(posit<X>_t x, posit<X>_t y)`: Exactly the same as `posit<X>_add_exact(x, y)` but with the sign of `y` flipped.
- `posit<X>_t posit<X>_mul(posit<X>_t x,posit<X>_t y)`: Multiply two posits, round the result to nearest
- `posit<X>_t posit<X>_div(posit<X>_t x, posit<X>_t y)`: Divide two posits, round the result to nearest

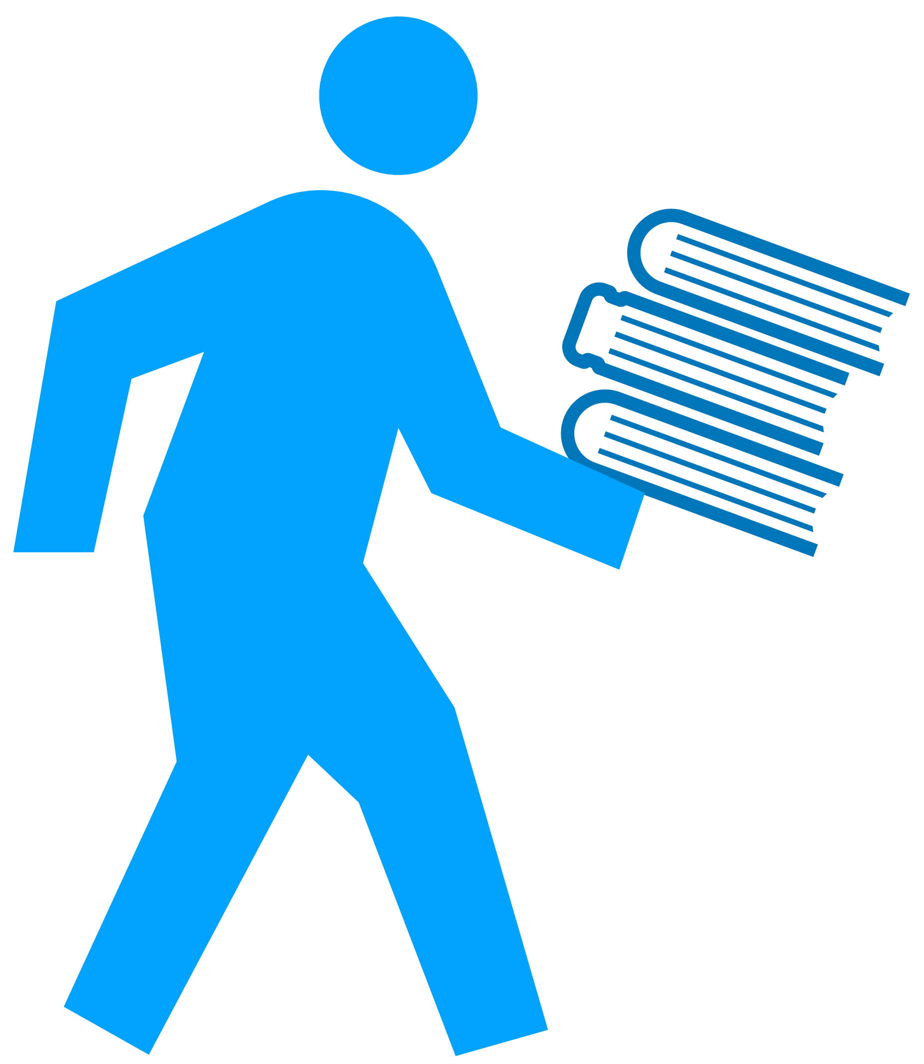
https://github.com/cjdelisle/libposit

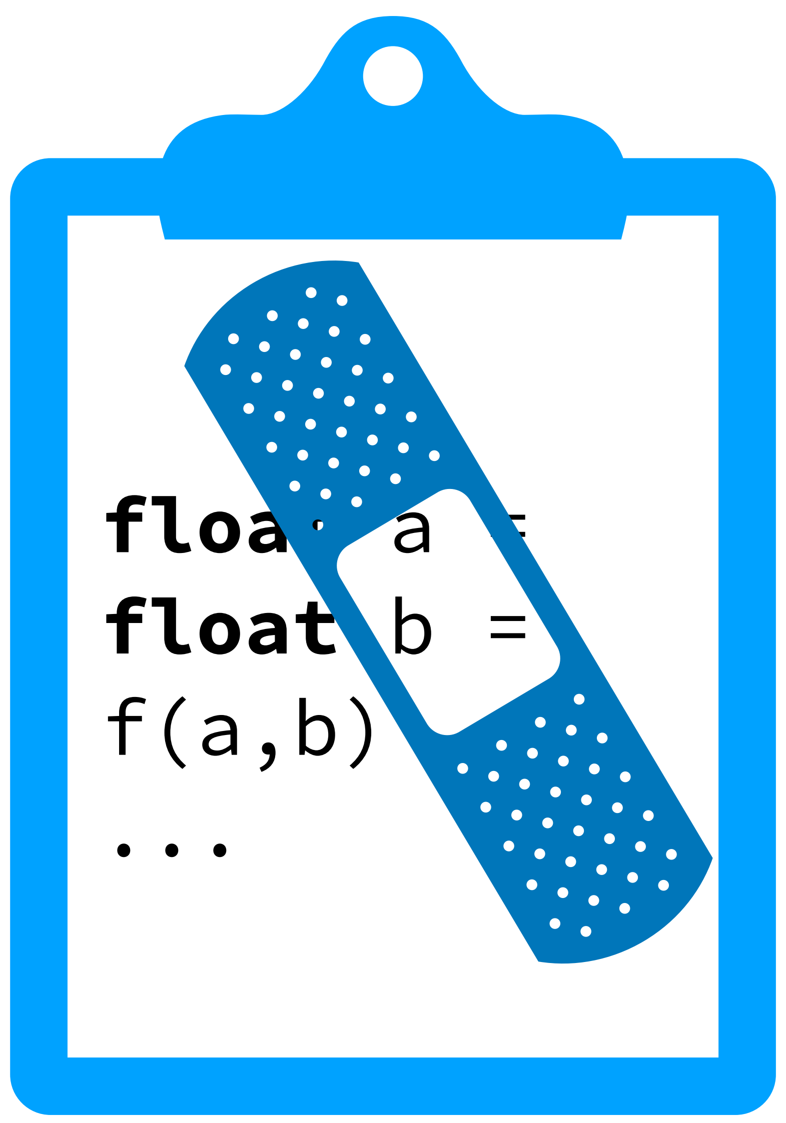

float a =
float b =
f(a,b)
...

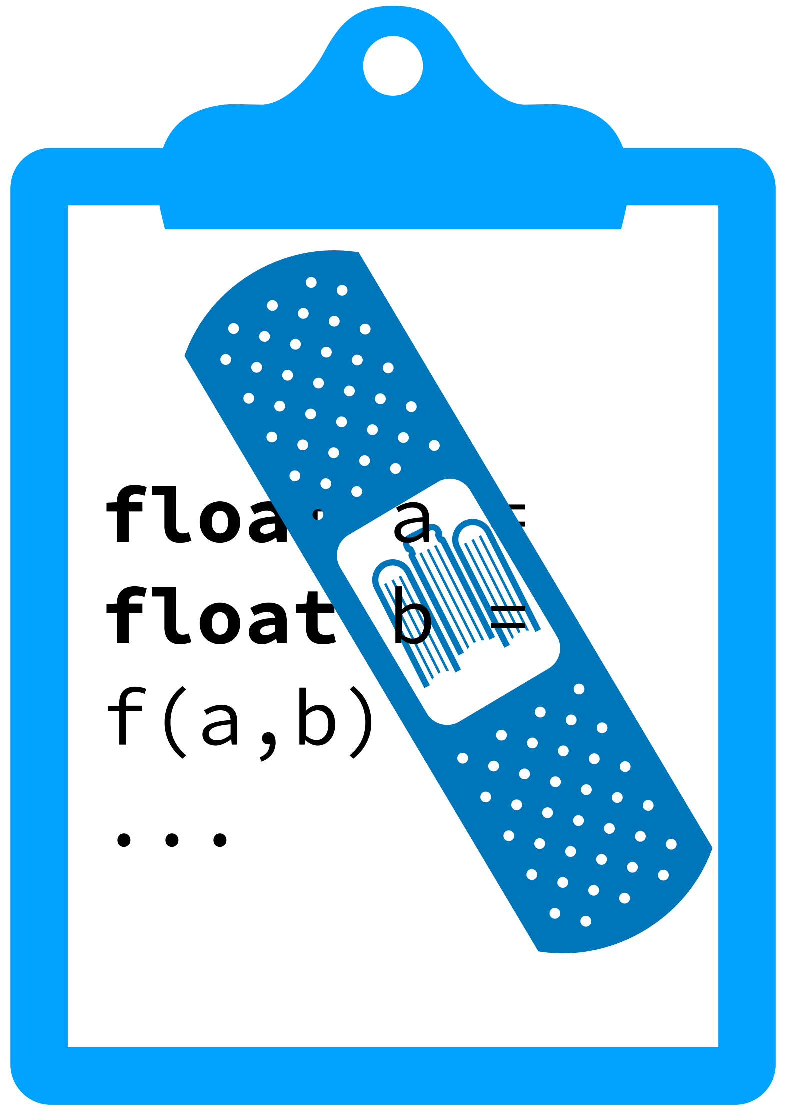

float a =
float b =
f(a,b)
...

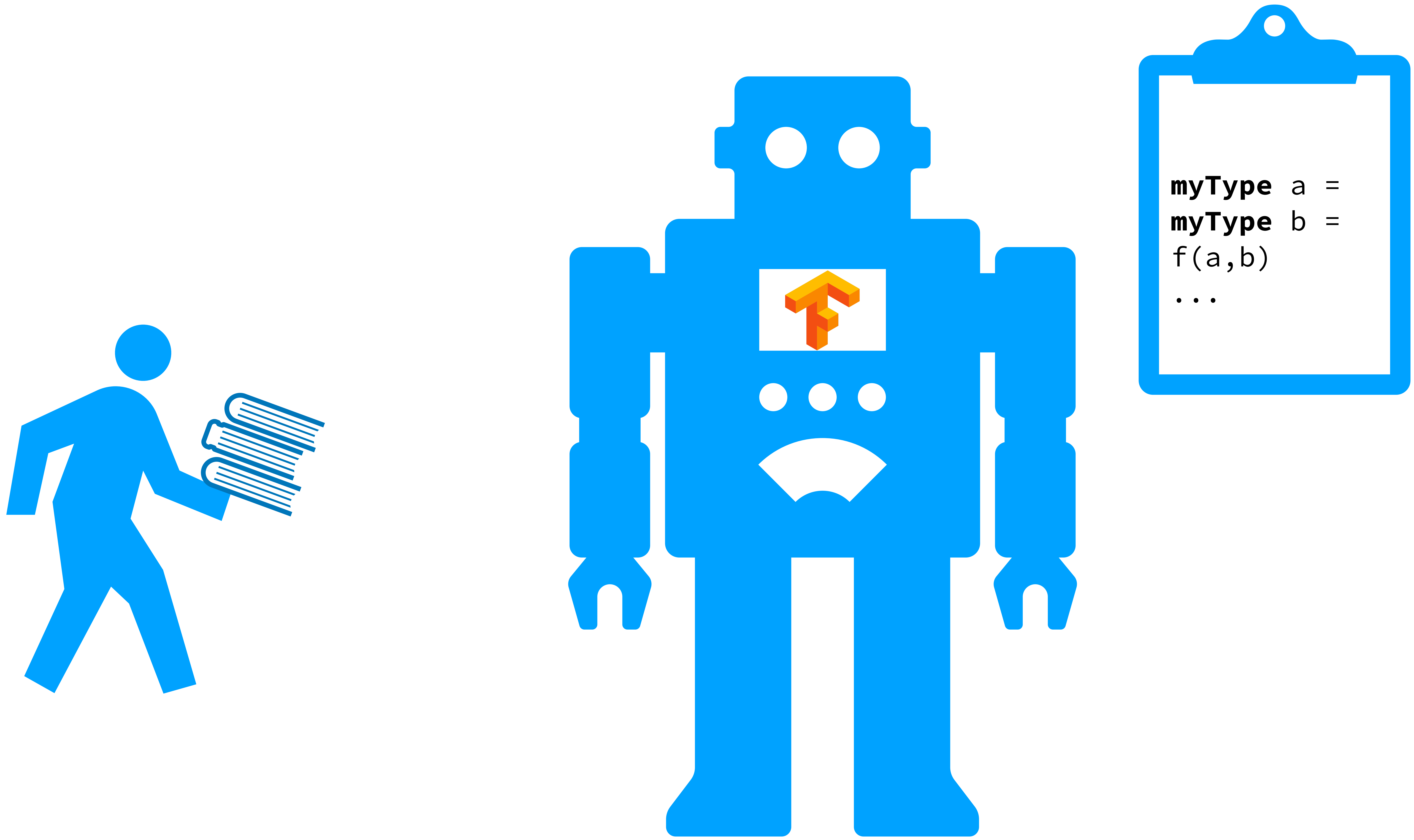

loat a =
float b =
f(a,b)
...

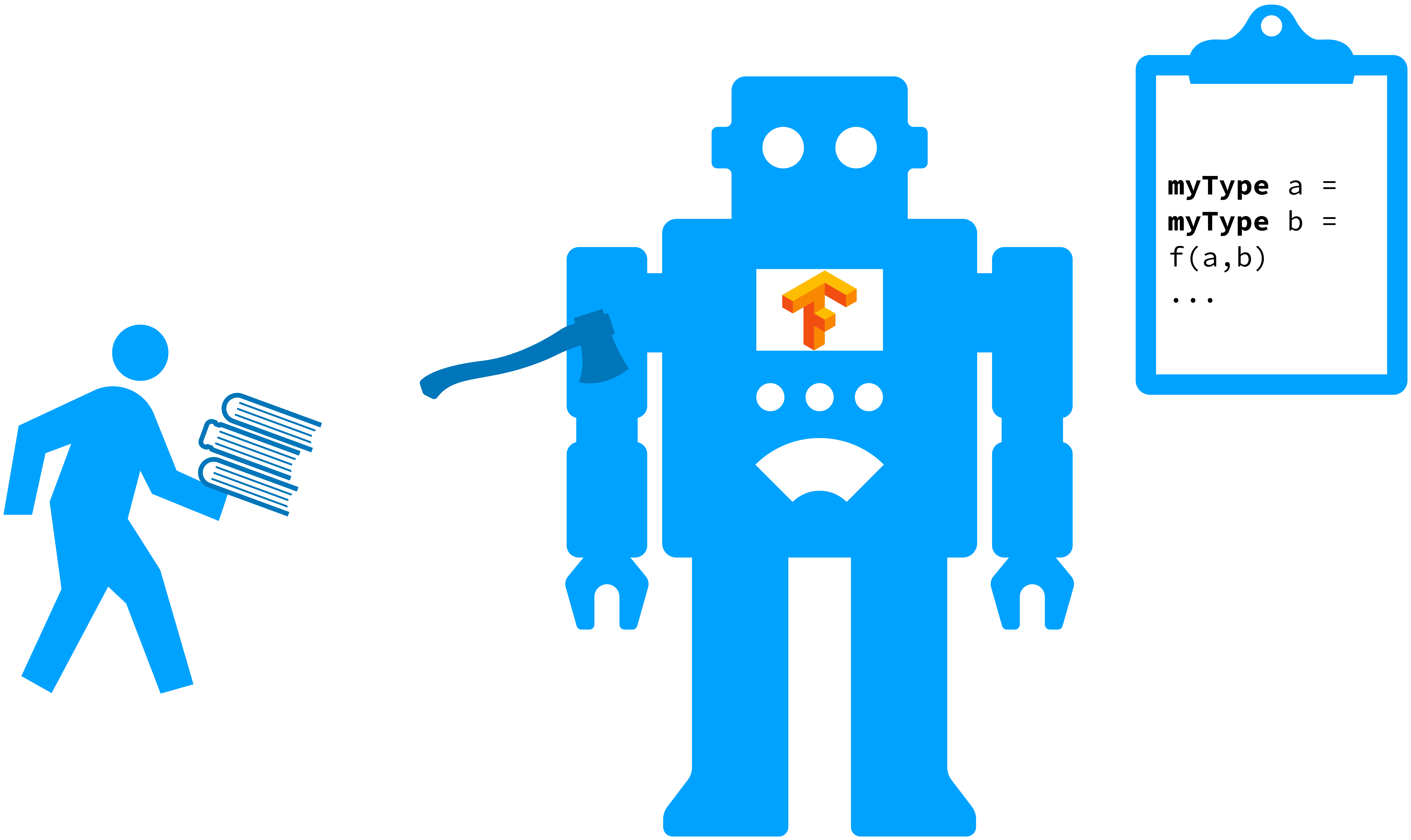

float a =
float b =
f(a,b)
...

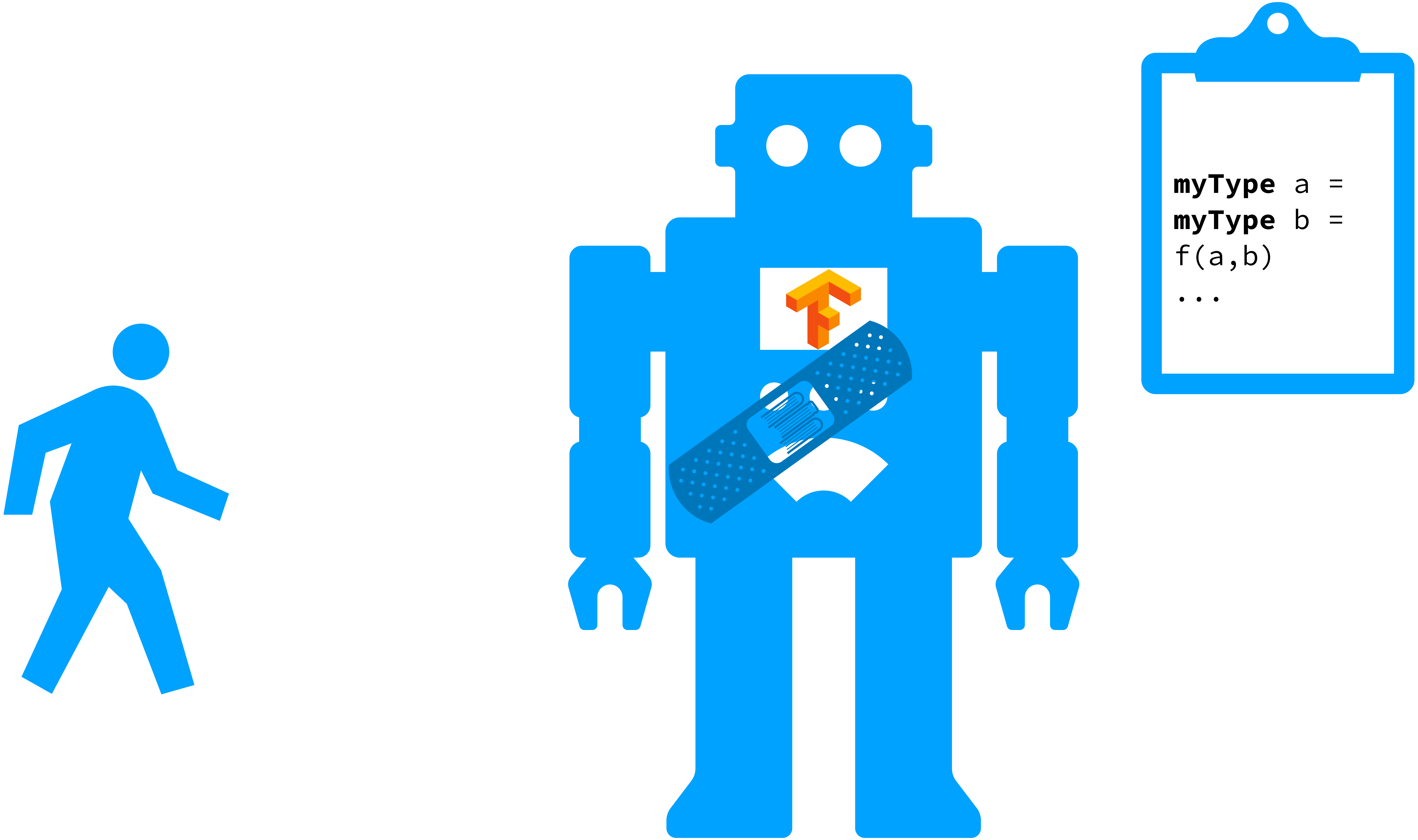


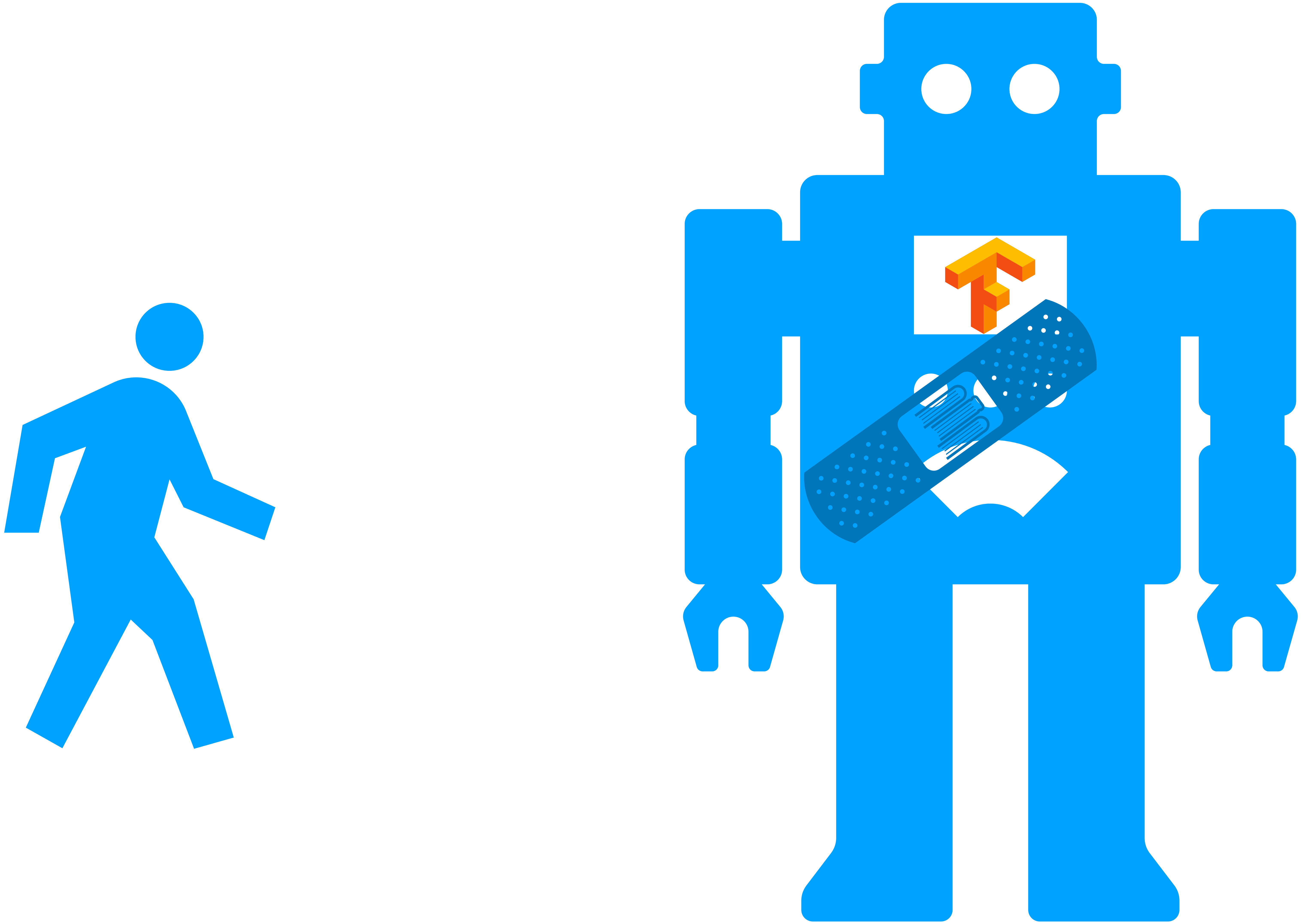

float b =
f(a,b)
...

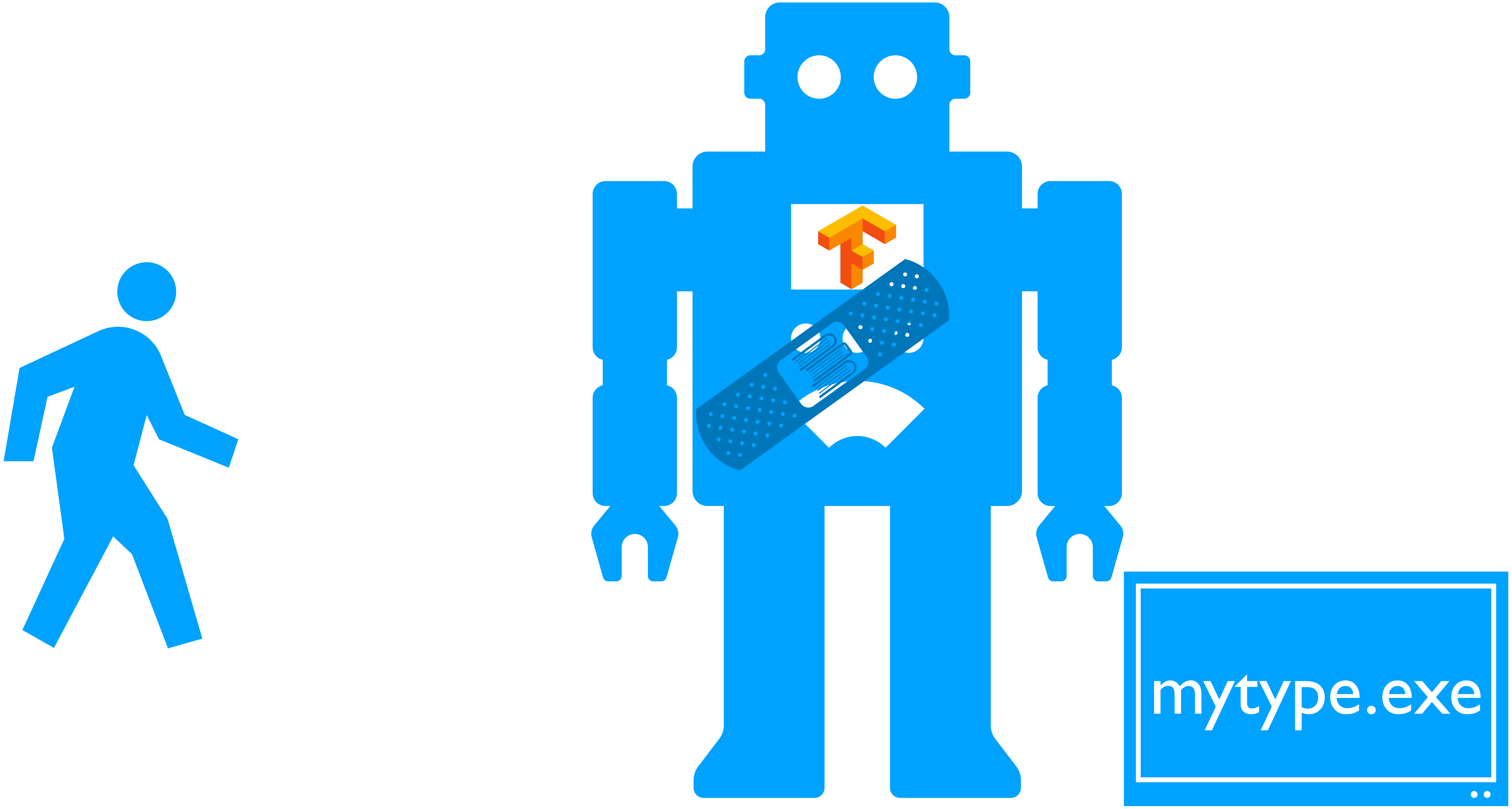

float
f(a,b)
...


myType a =
myType b =
f(a,b)
...


myType a =
myType b =
f(a,b)
...

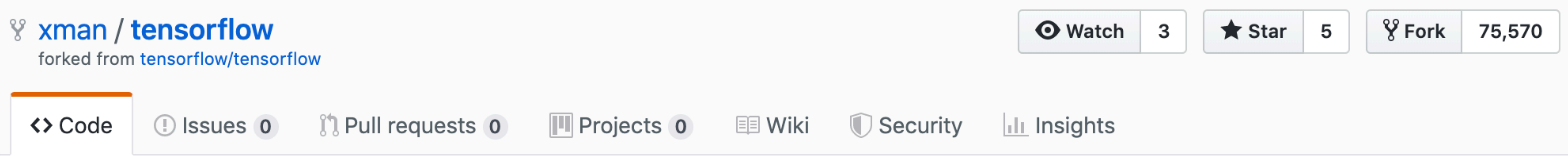

myType a =
myType b =
f(a,b)
...




mytype.exe

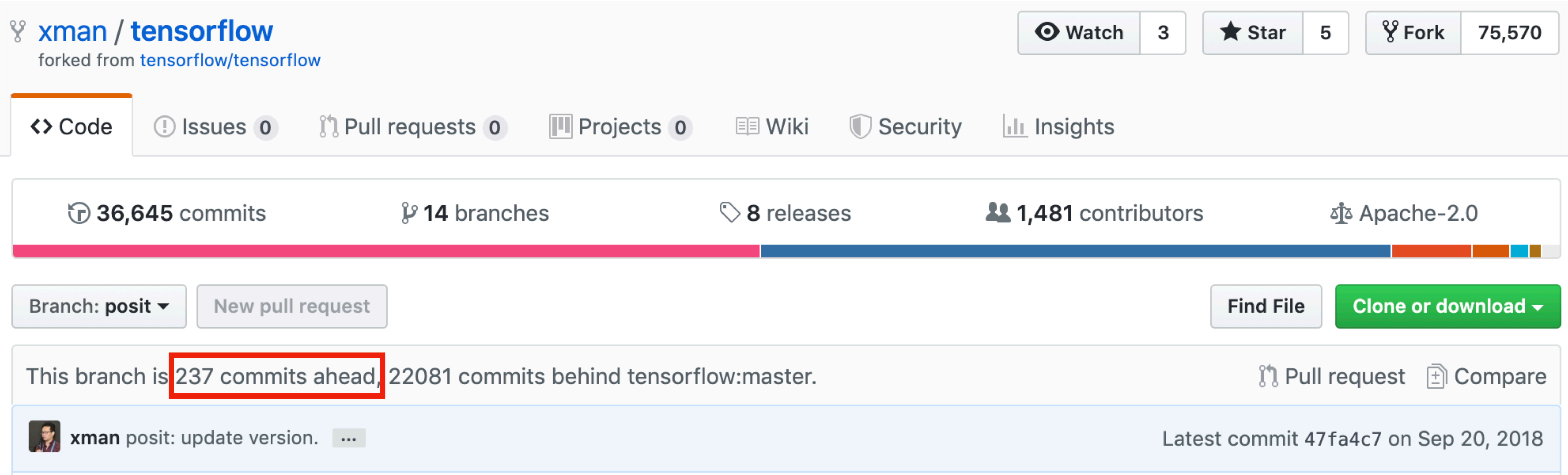
An Example in the Wild!

An Example in the Wild!



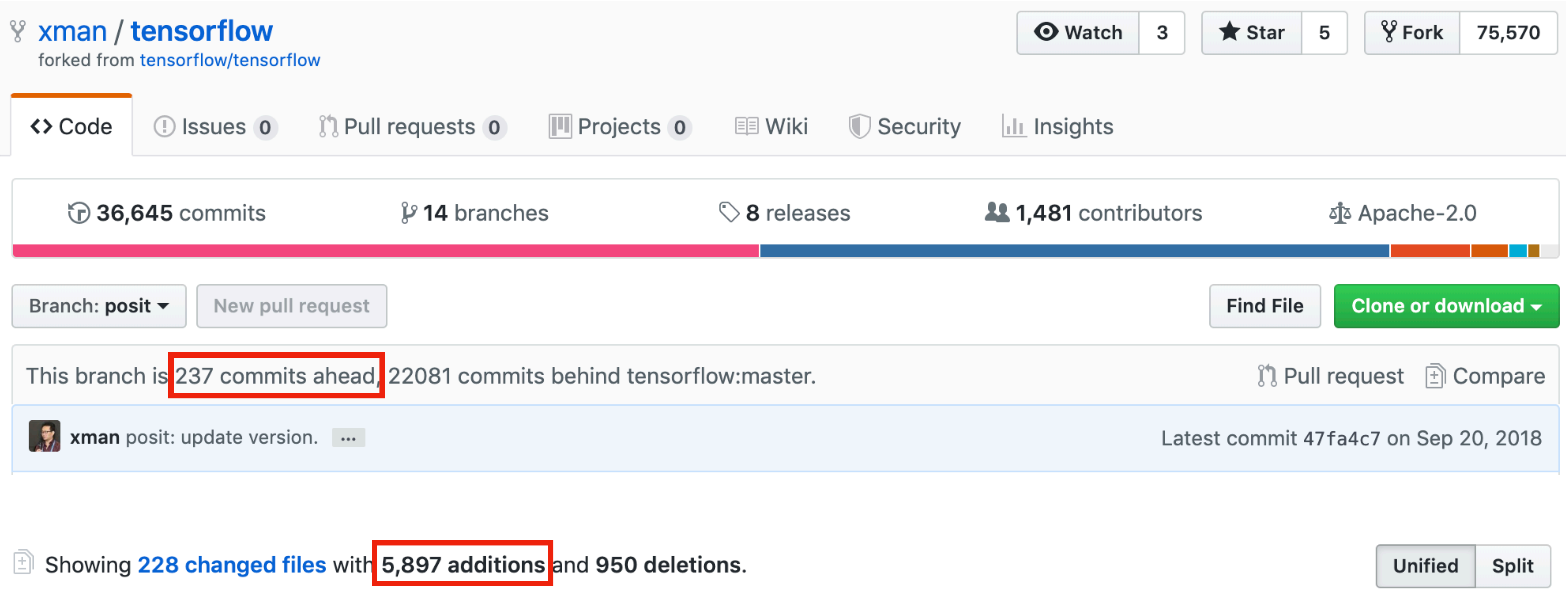
https://github.com/xman/tensorflow/tree/posit

An Example in the Wild!



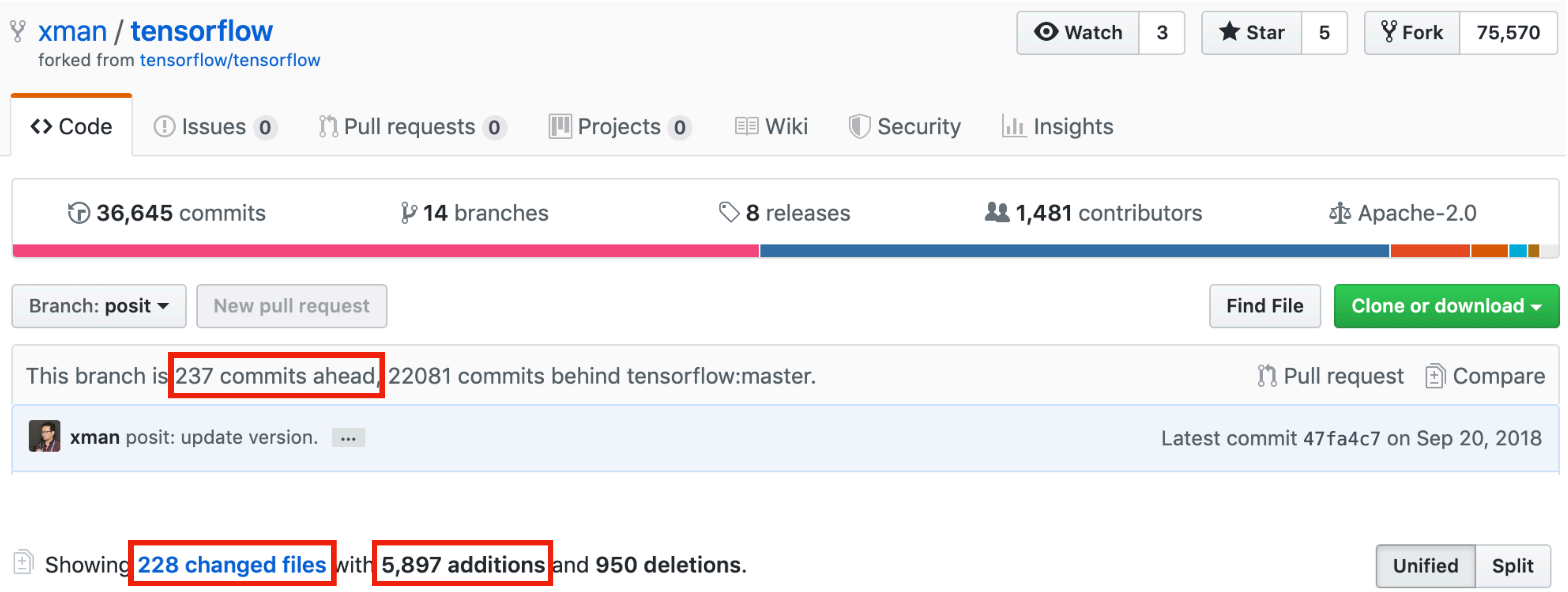
https://github.com/xman/tensorflow/tree/posit

An Example in the Wild!



https://github.com/xman/tensorflow/tree/posit

An Example in the Wild!



https://github.com/xman/tensorflow/tree/posit

The computational demands of deep learning require new datatypes…

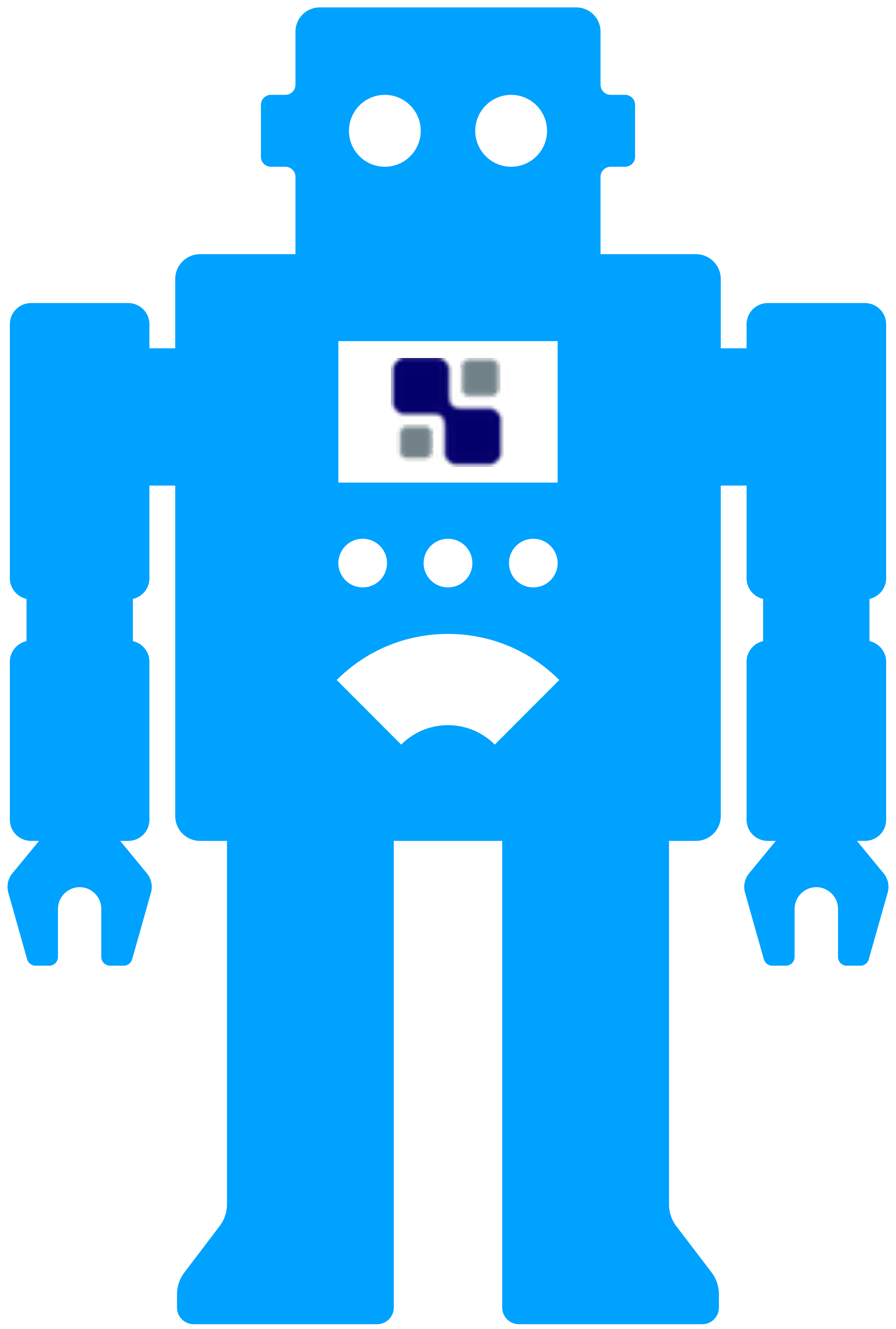
The computational demands of deep learning require new datatypes…

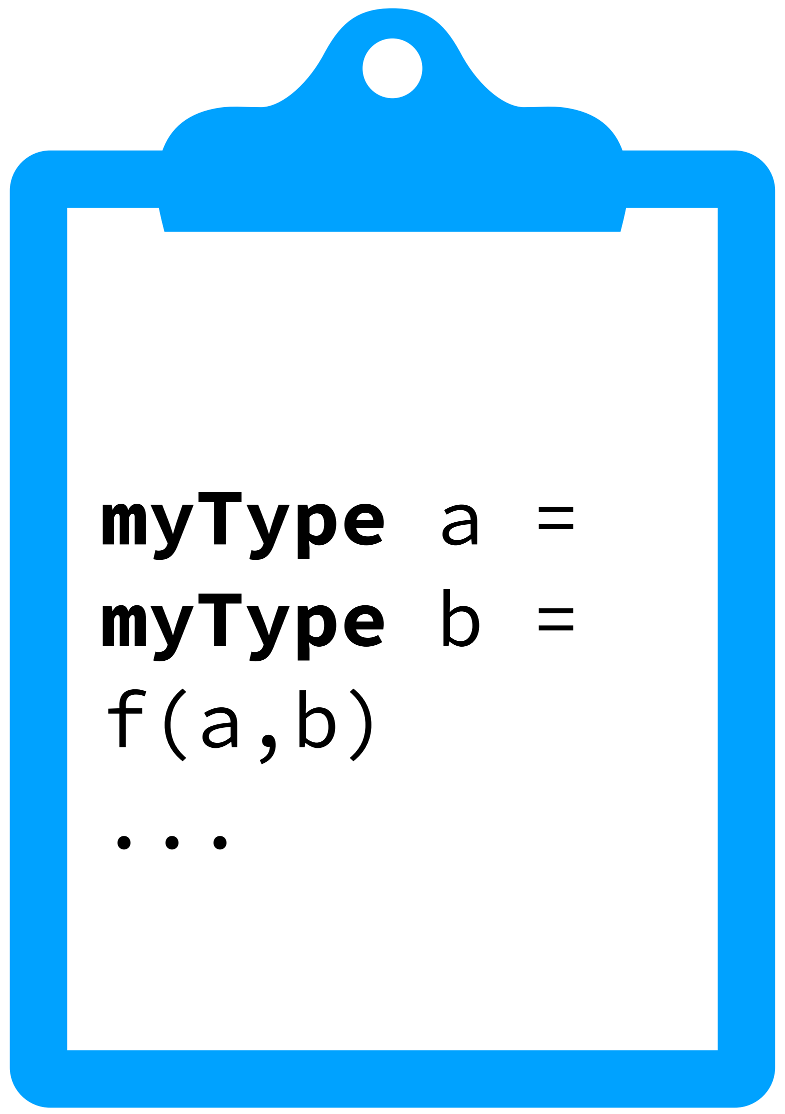
…but datatype research is difficult!

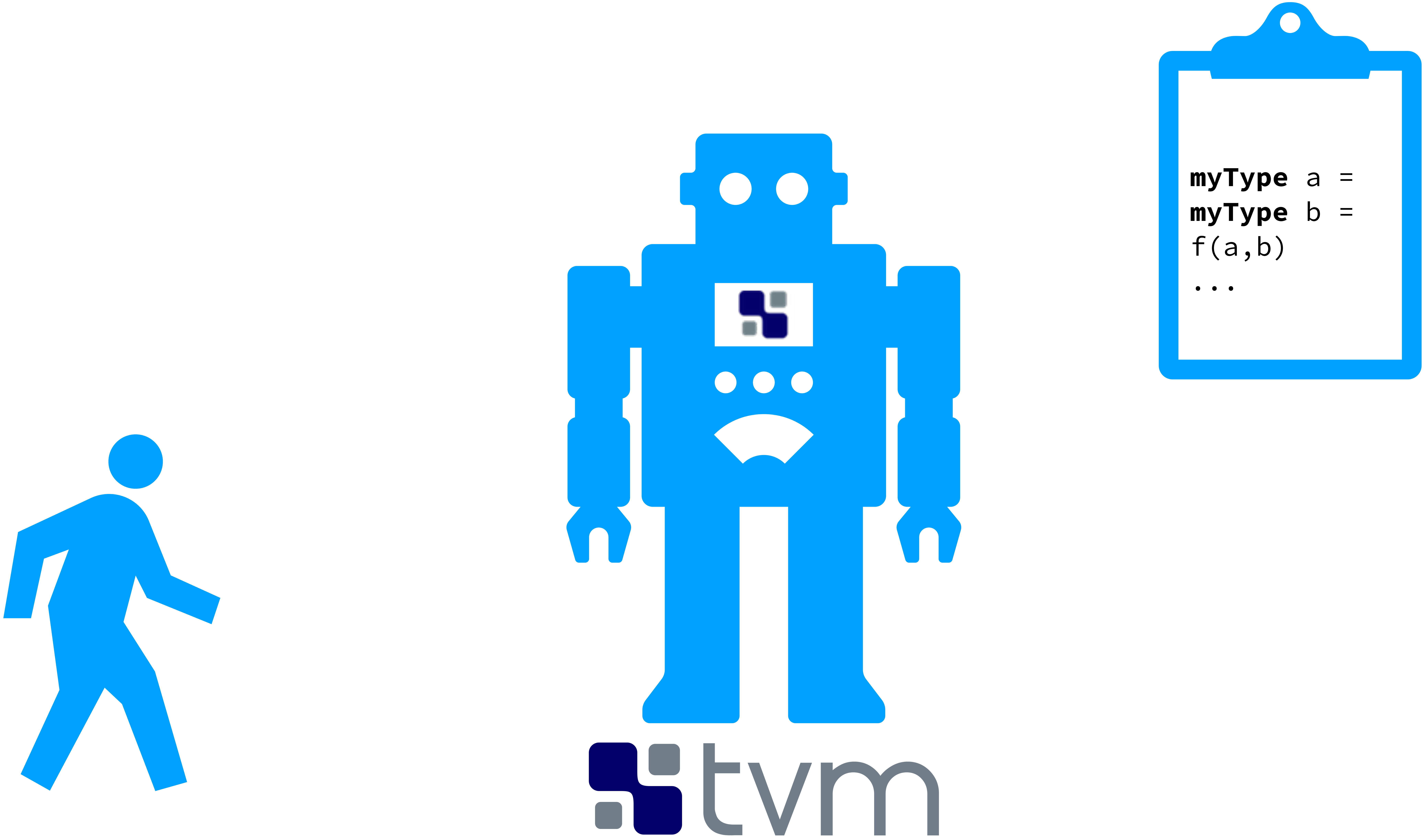
The computational demands of deep learning require new datatypes…

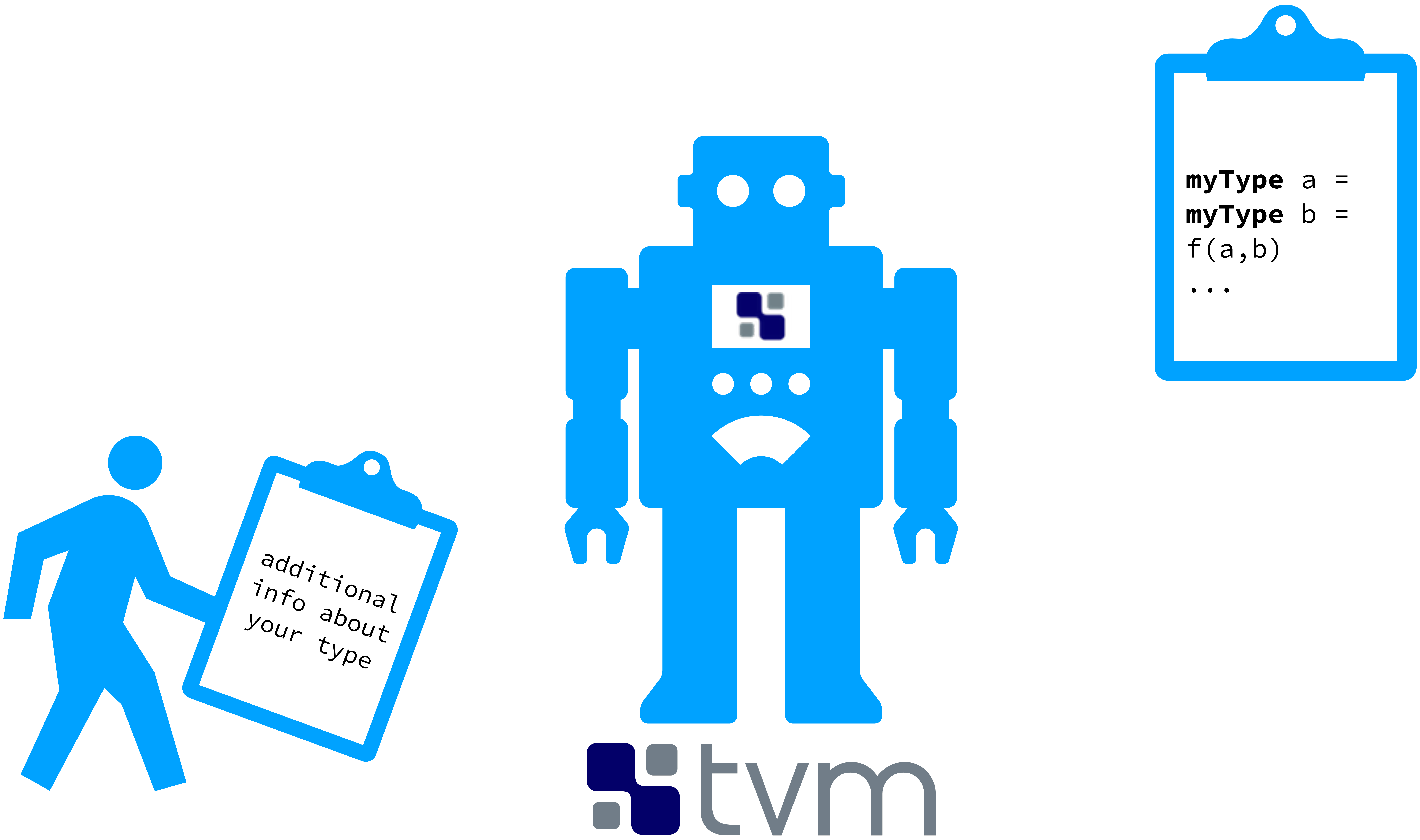
…but datatype research is difficult!

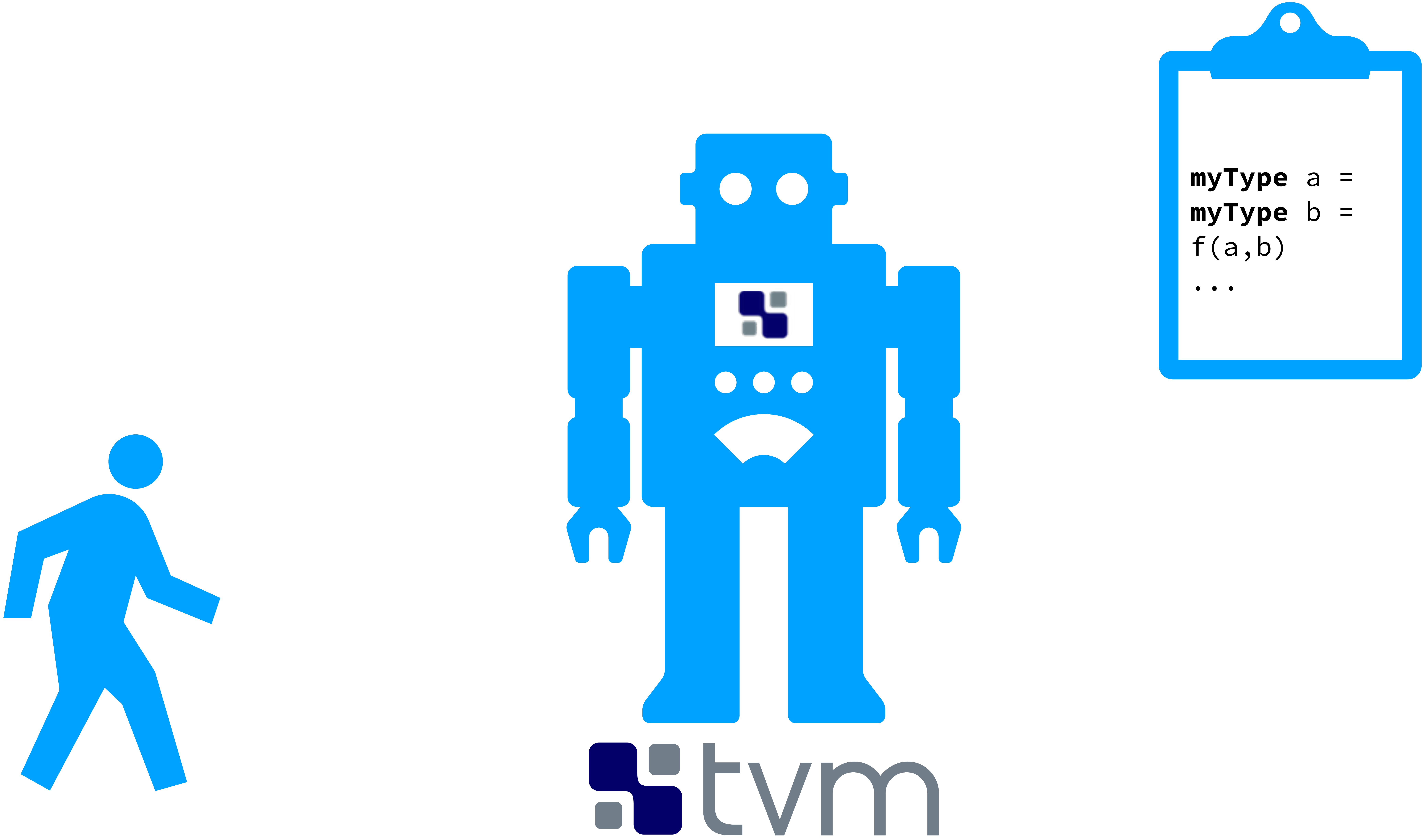
Our solution: the Bring Your Own Datatypes framework.

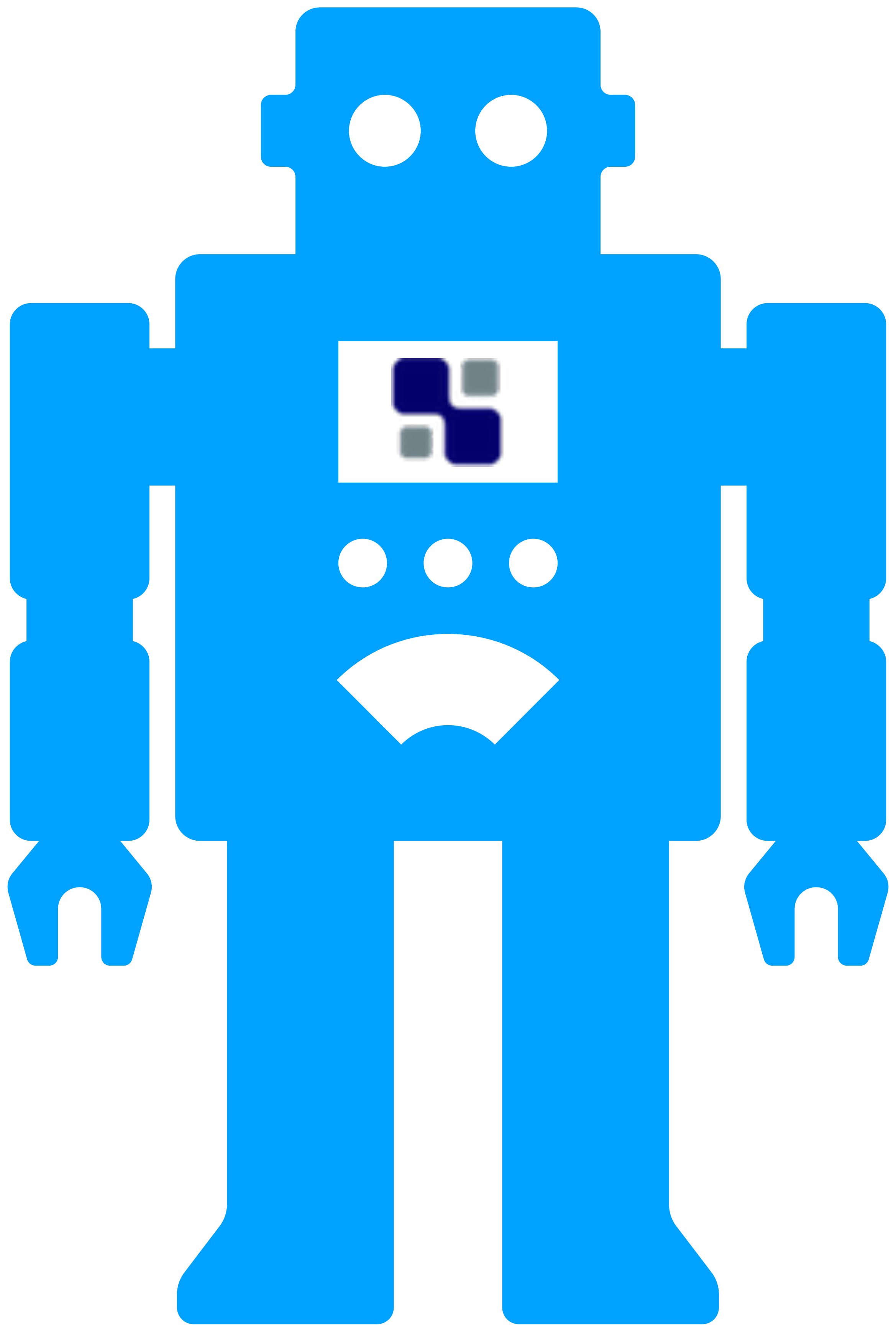



myType a =
myType b =
f(a,b)
...

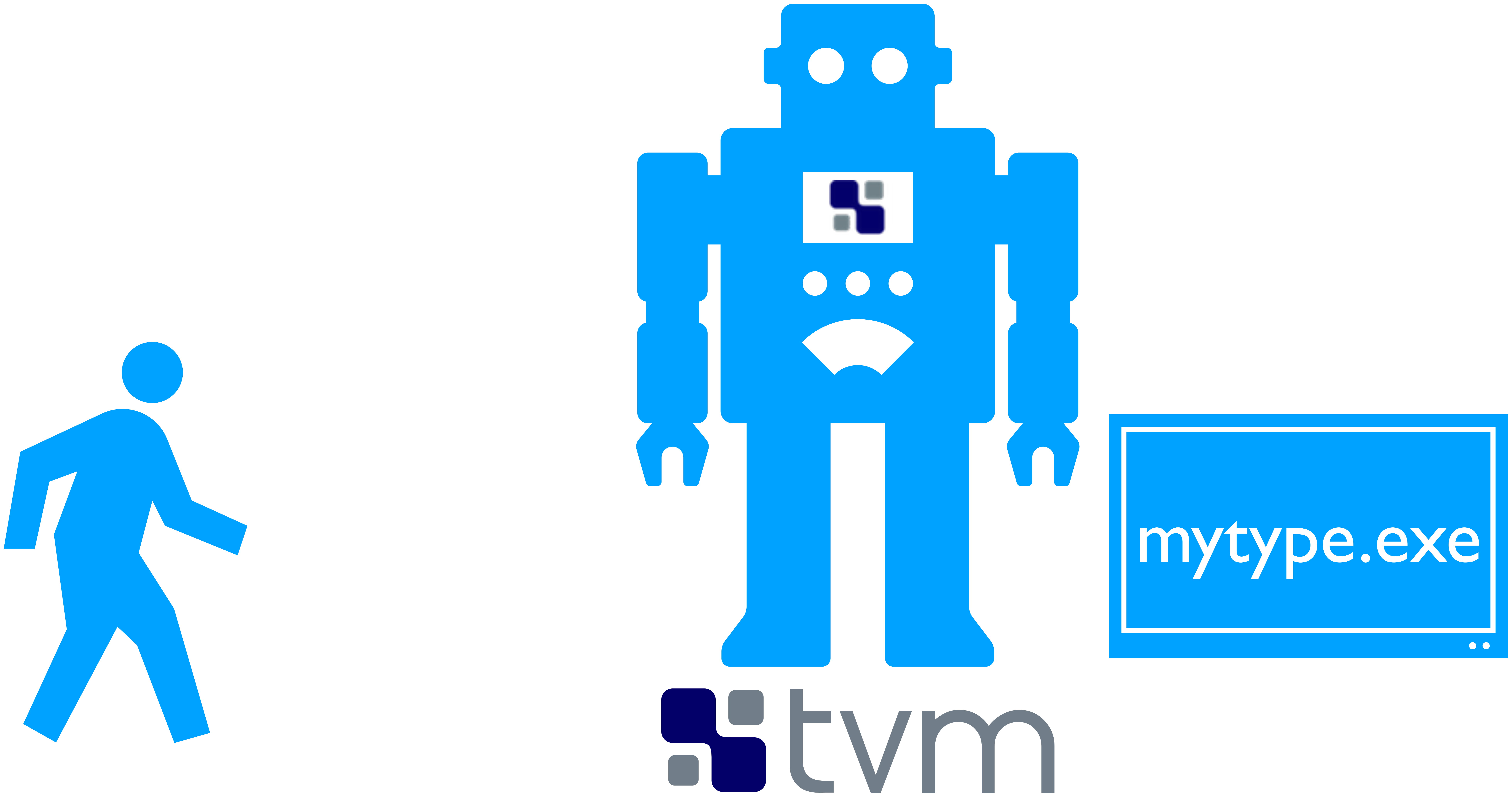

myType a =
myType b =
f(a,b)
...
tvm


additional info about your type
myType a =
myType b =
f(a,b)
...
tvm


myType a =
myType b =
f(a,b)
...
tvm



tvm


tvm
mytype.exe

Bring Your Own Datatypes

What do we want?

What do we want?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

What do we want?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads
2. User **gives TVM some information** about the library

What do we want?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

2. User **gives TVM some information** about the library

3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

But first…what is TVM?

But first…what is TVM?

tvm

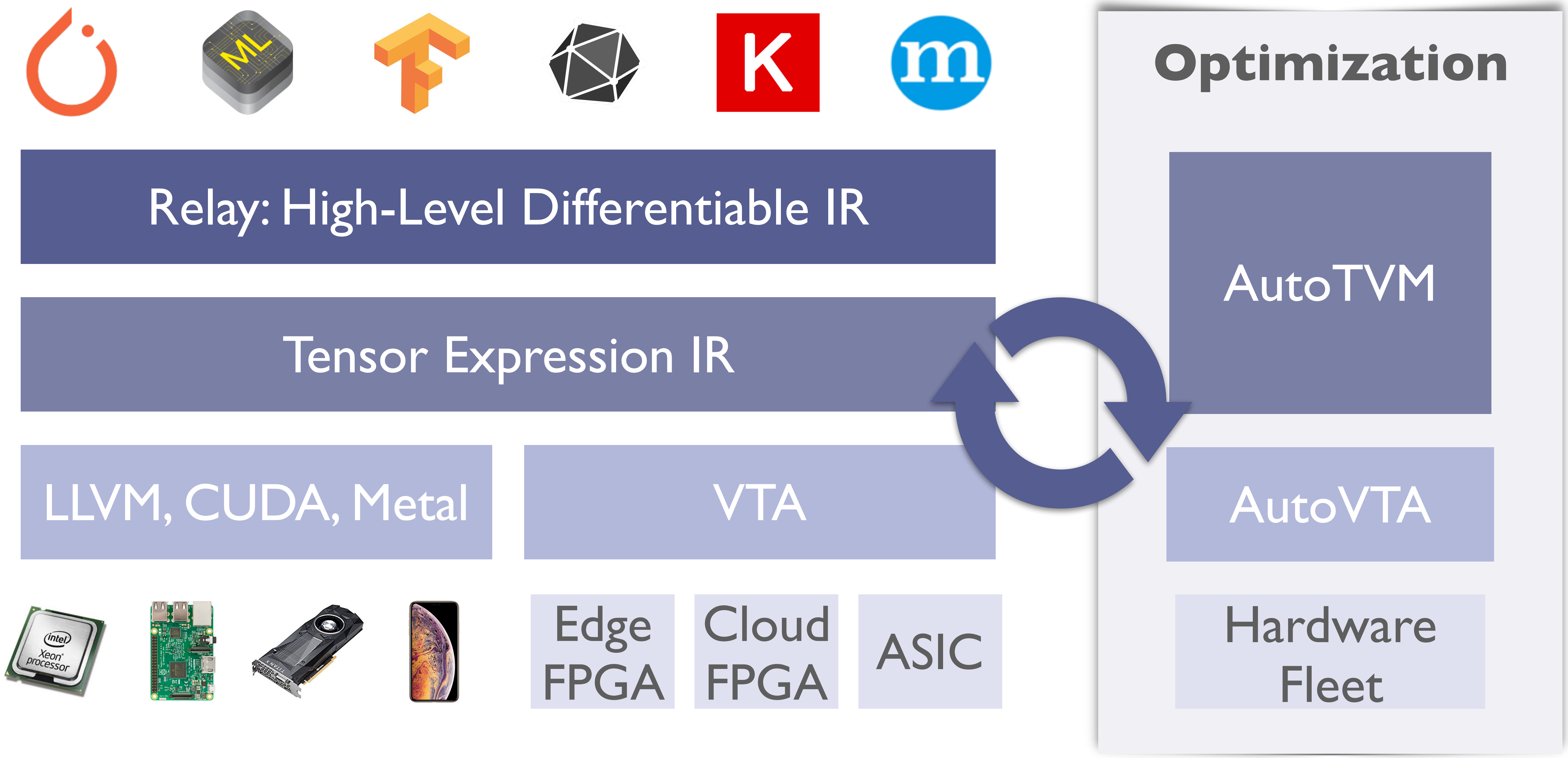
An **extensible, optimizing compiler** for deep learning.

But first…what is TVM?

An **extensible, optimizing compiler** for deep learning.

See tvm.ai for more info!

The TVM Stack

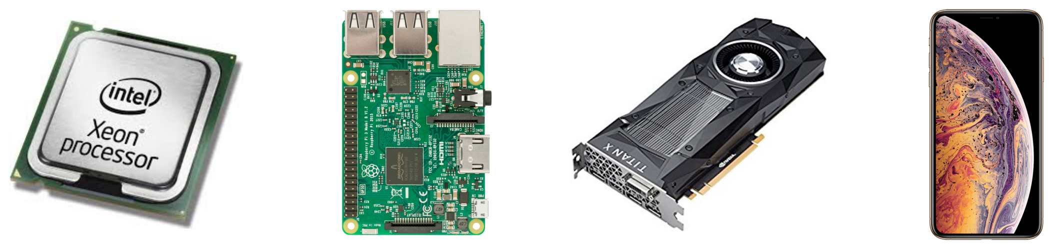


The TVM Stack

TVM IR

LLVM, CUDA, Metal

VTA

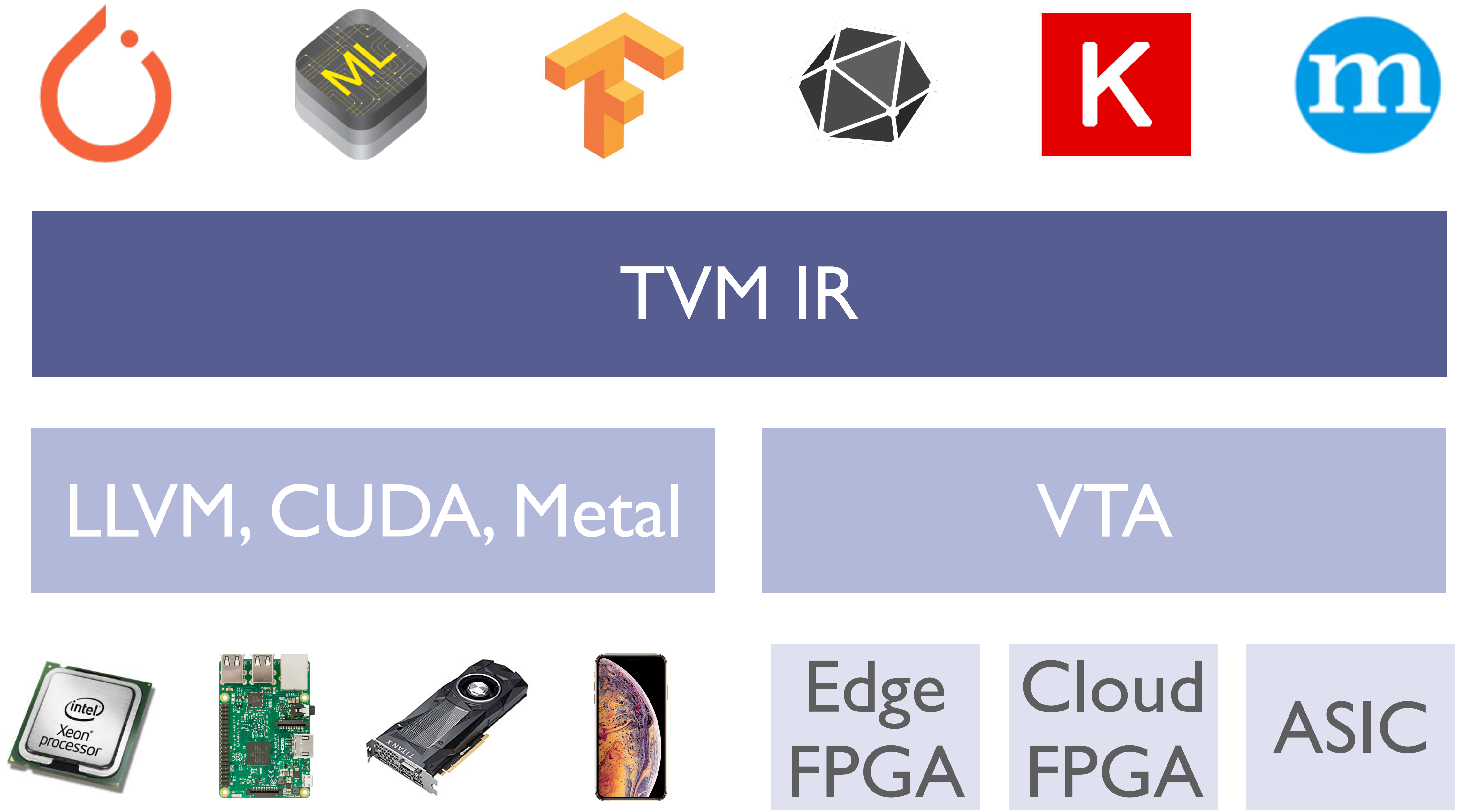


Edge FPGA

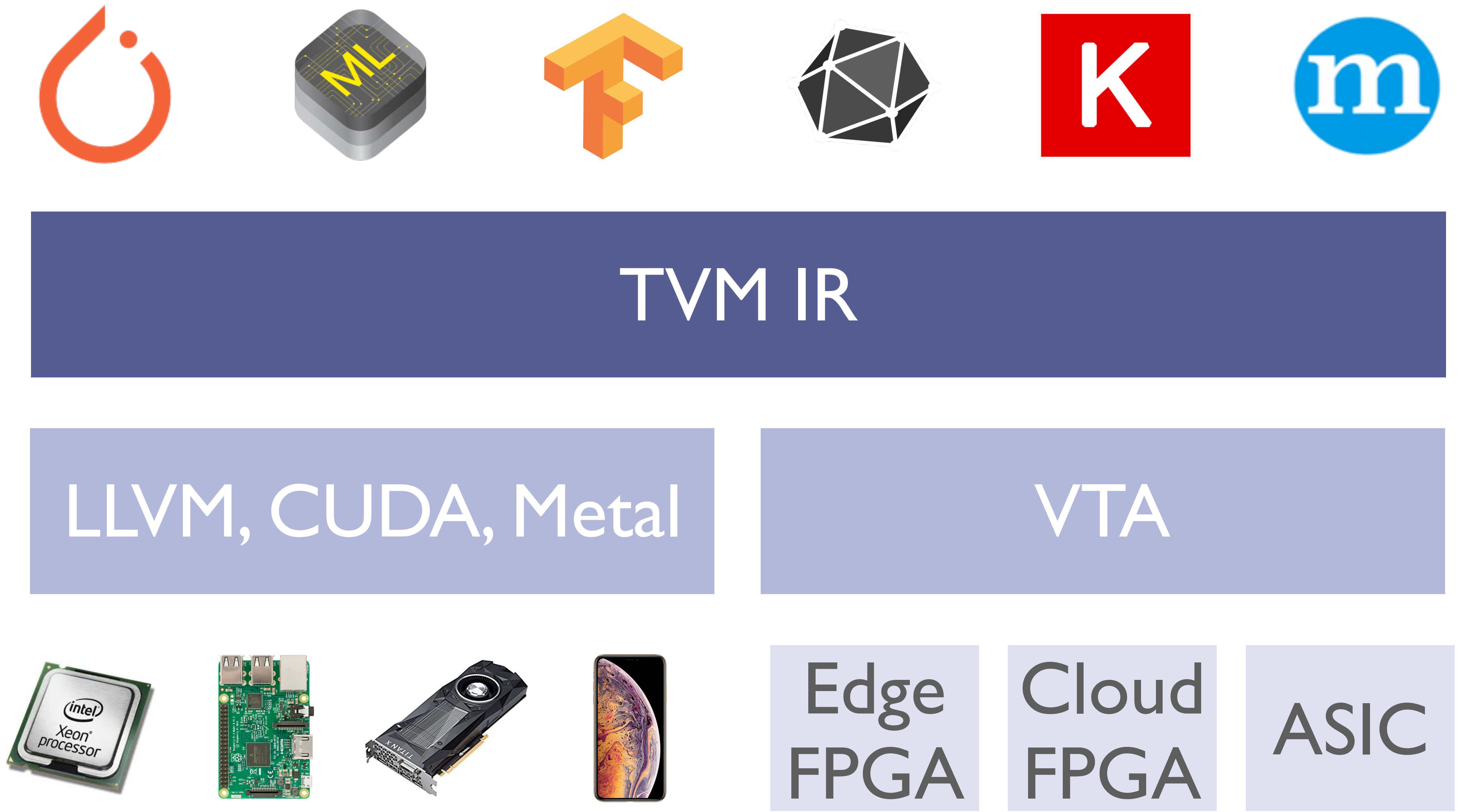
Cloud FPGA

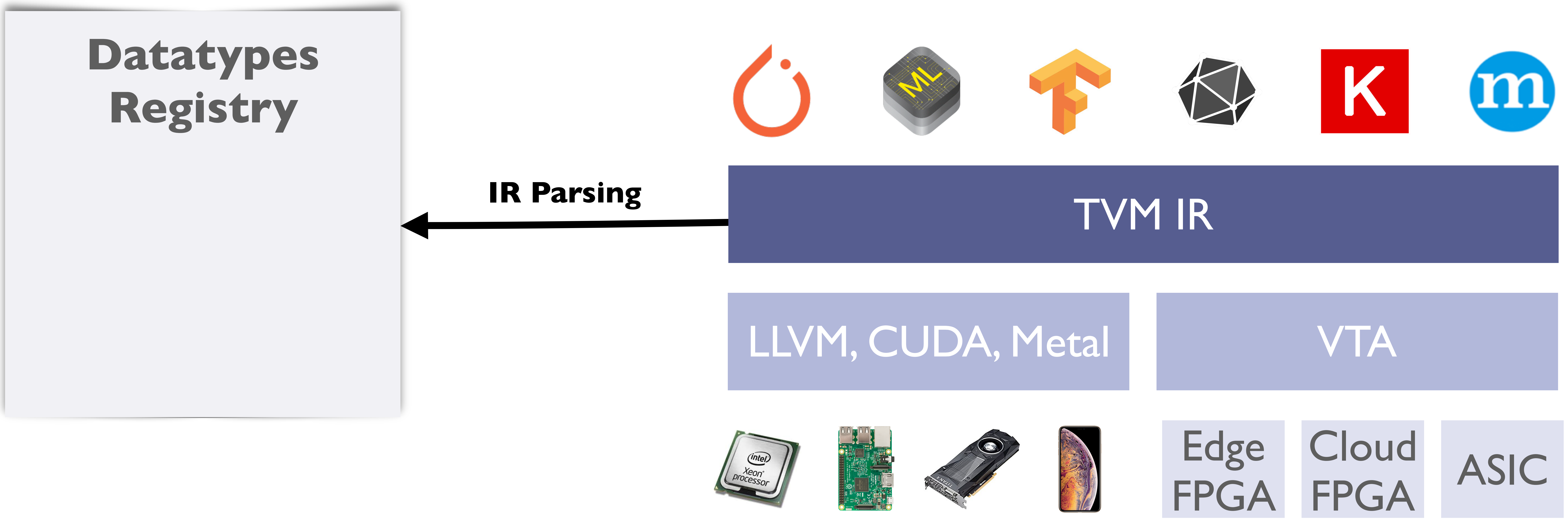
ASIC

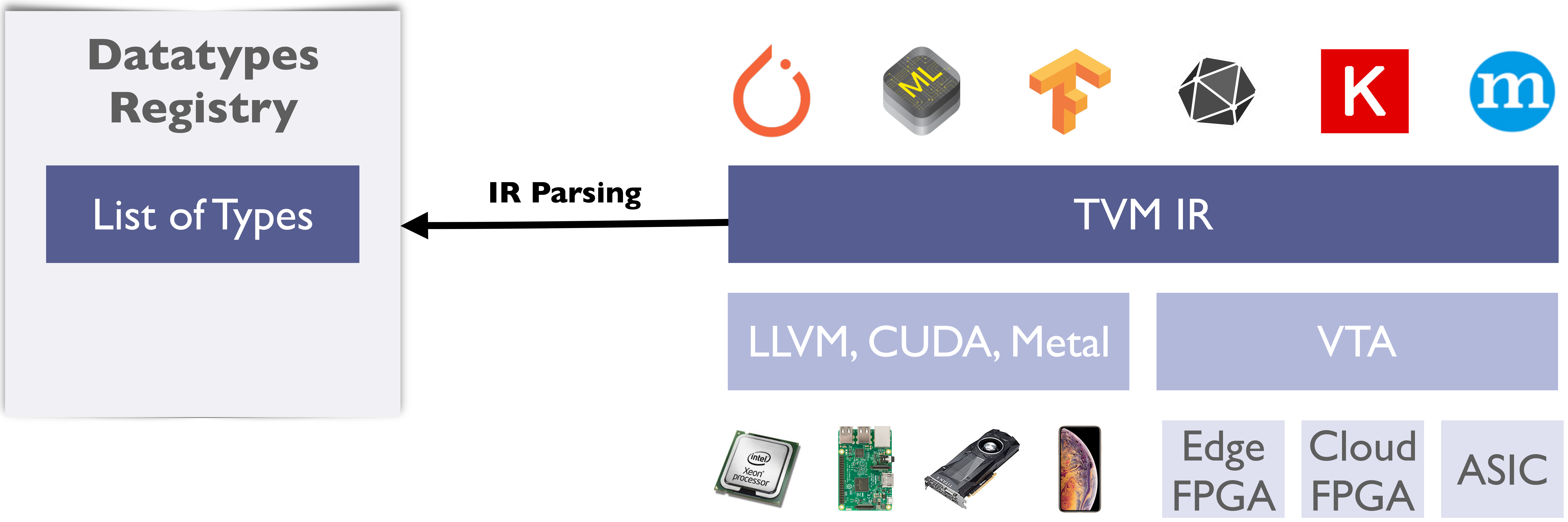
Datatypes Registry

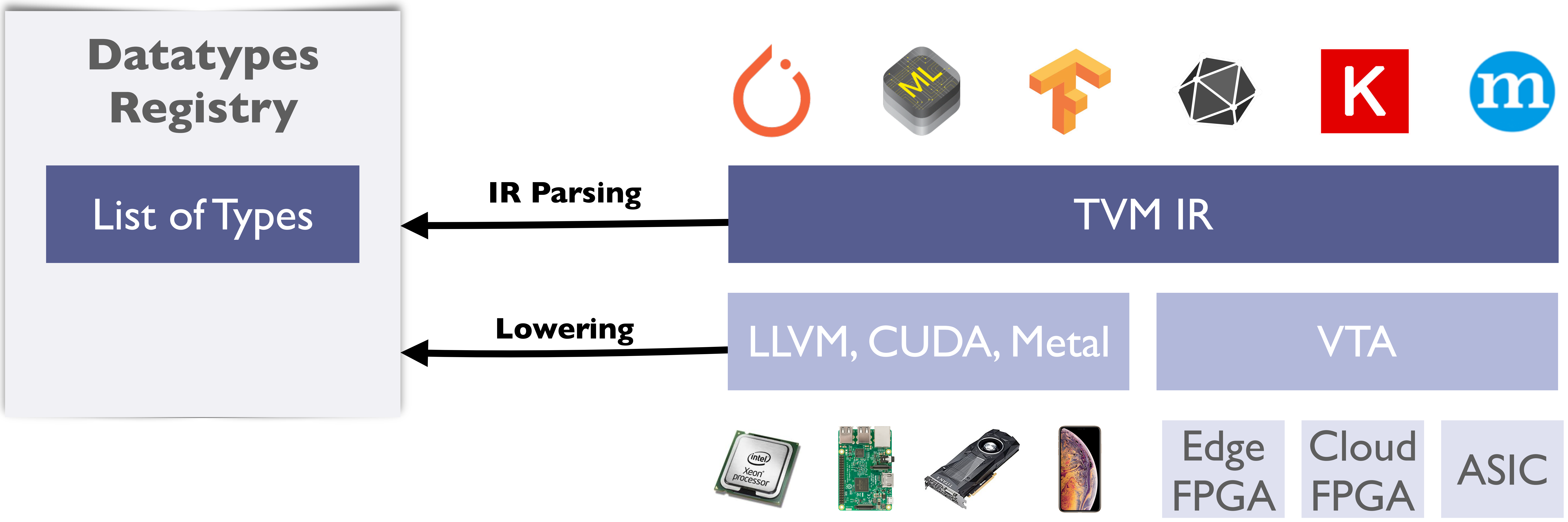


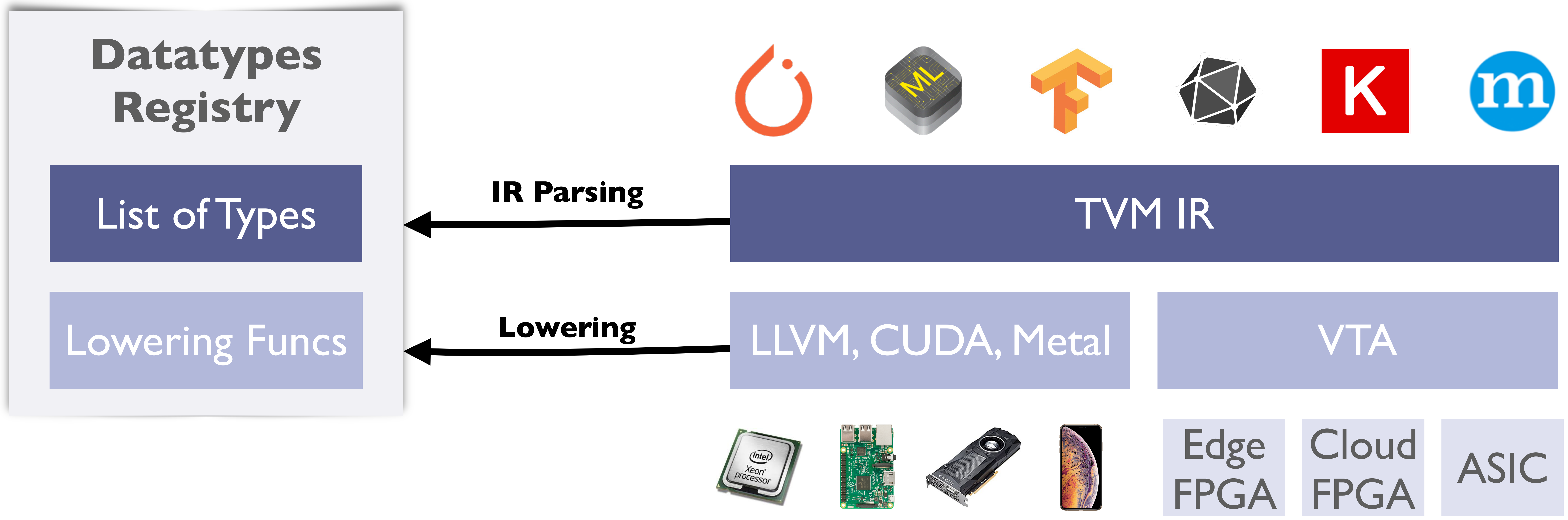
Datatypes Registry




Datatypes Registry
IR Parsing
TVM IR
LLVM, CUDA, Metal
VTA
Edge FPGA
Cloud FPGA
ASIC


Datatypes Registry
List of Types
IR Parsing
TVM IR
LLVM, CUDA, Metal
VTA
Edge FPGA
Cloud FPGA
ASIC

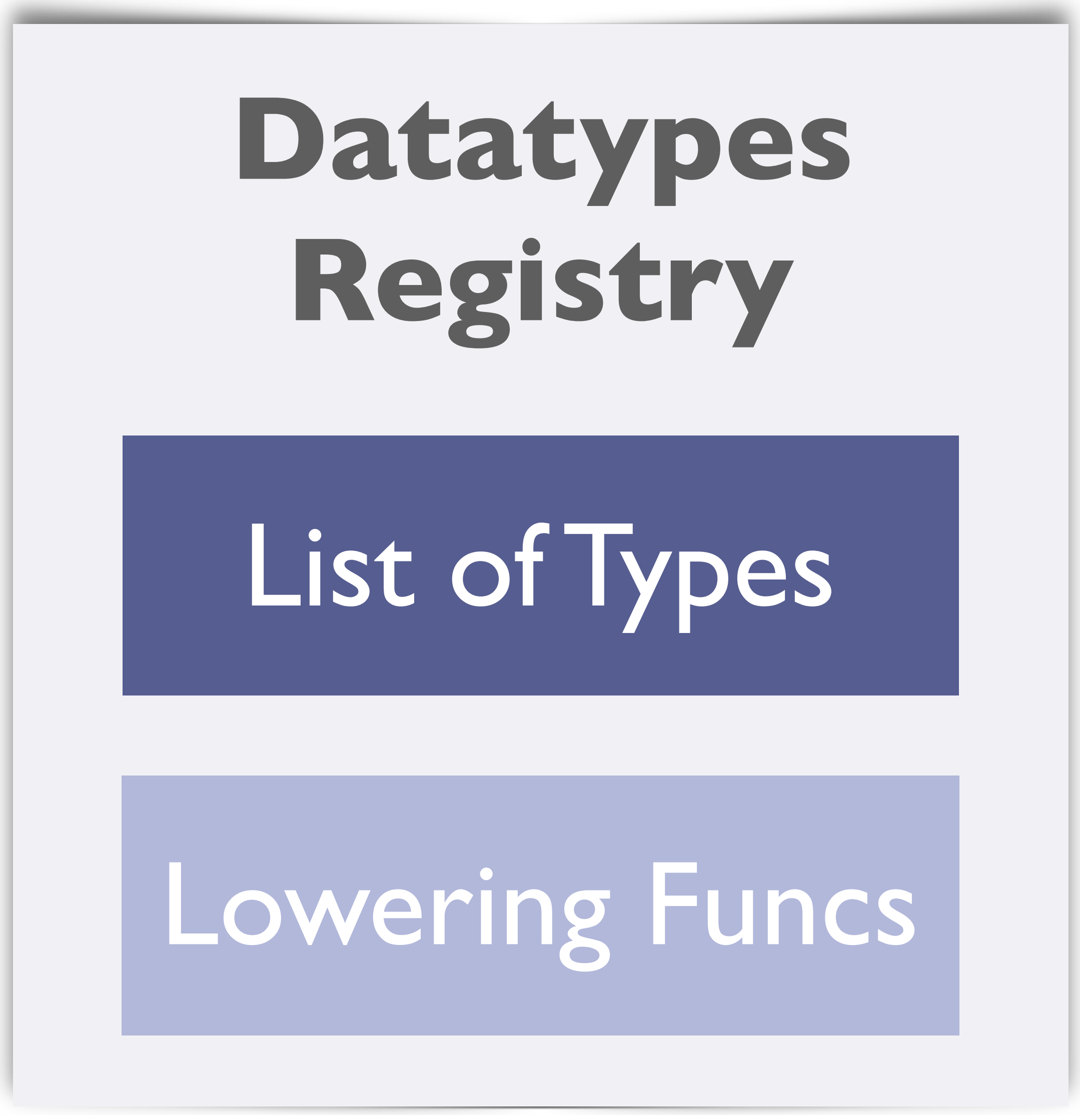

Datatypes Registry
List of Types
IR Parsing
Lowering
TVM IR
LLVM, CUDA, Metal
VTA
Edge FPGA
Cloud FPGA
ASIC


Datatypes Registry
List of Types
Lowering Funcs
IR Parsing
Lowering
TVM IR
LLVM, CUDA, Metal
VTA
Edge FPGA
Cloud FPGA
ASIC

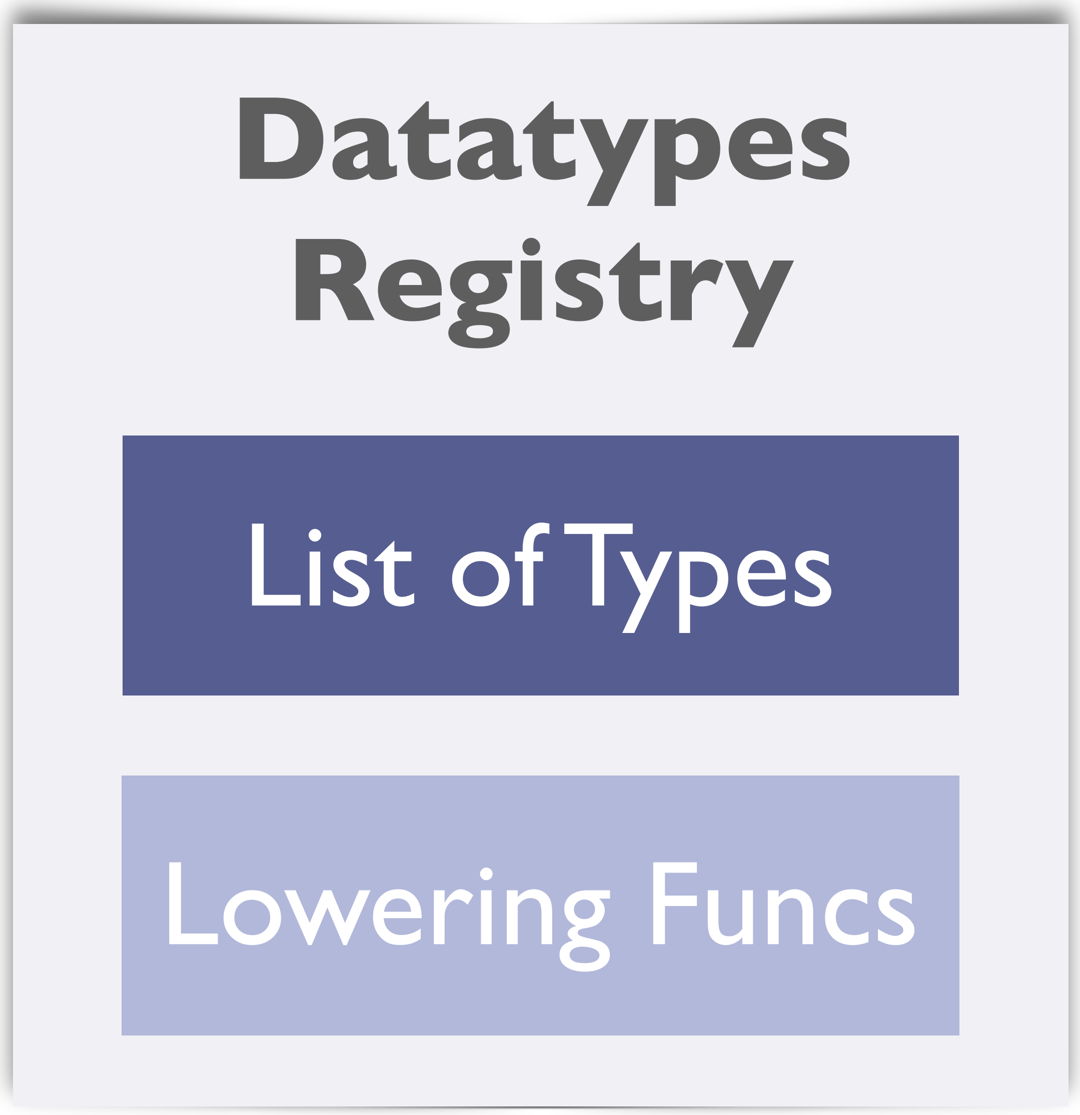
Datatypes Registry

List of Types

Lowering Funcs

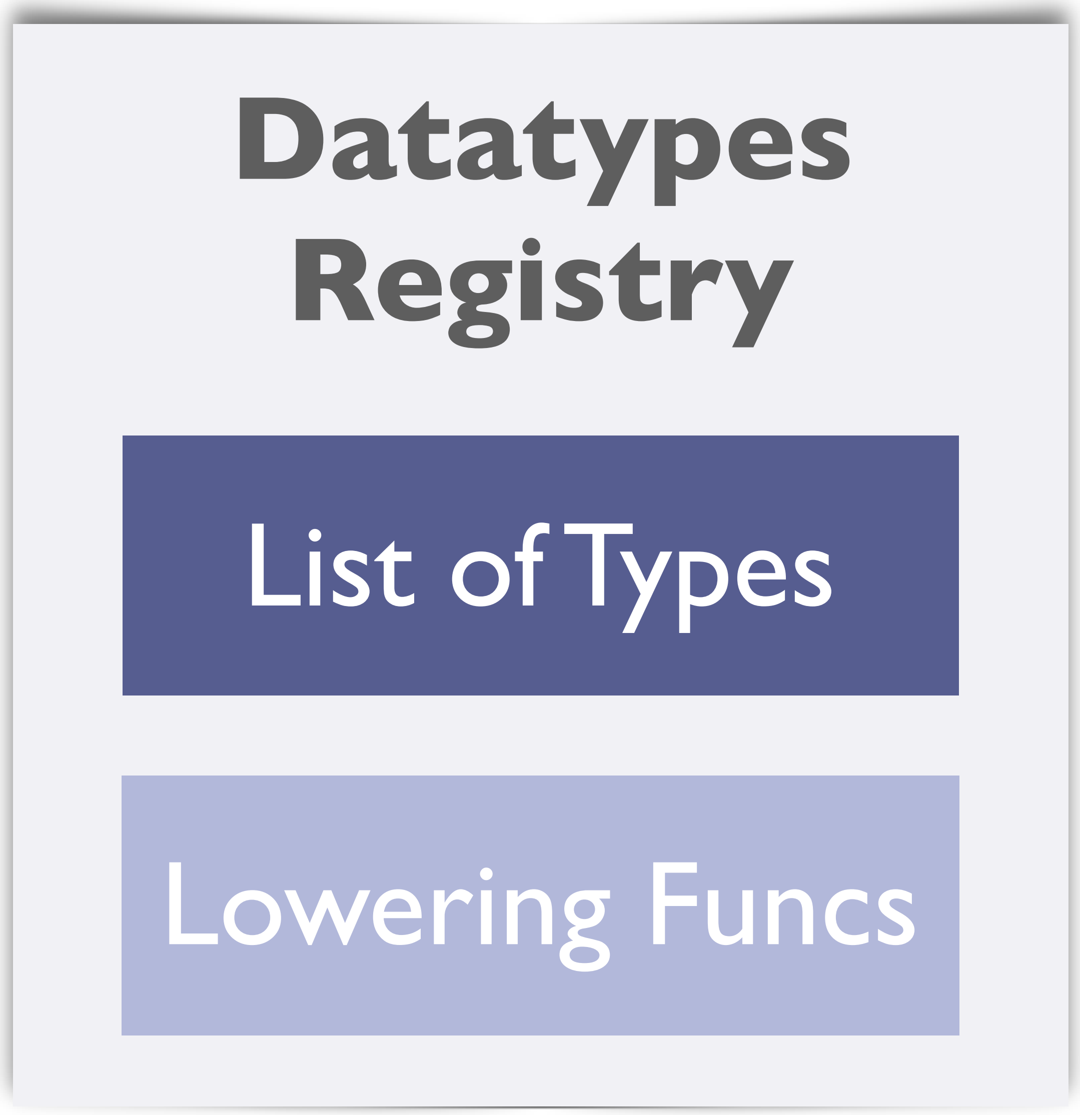


To register their datatype, the user provides a name and type code:



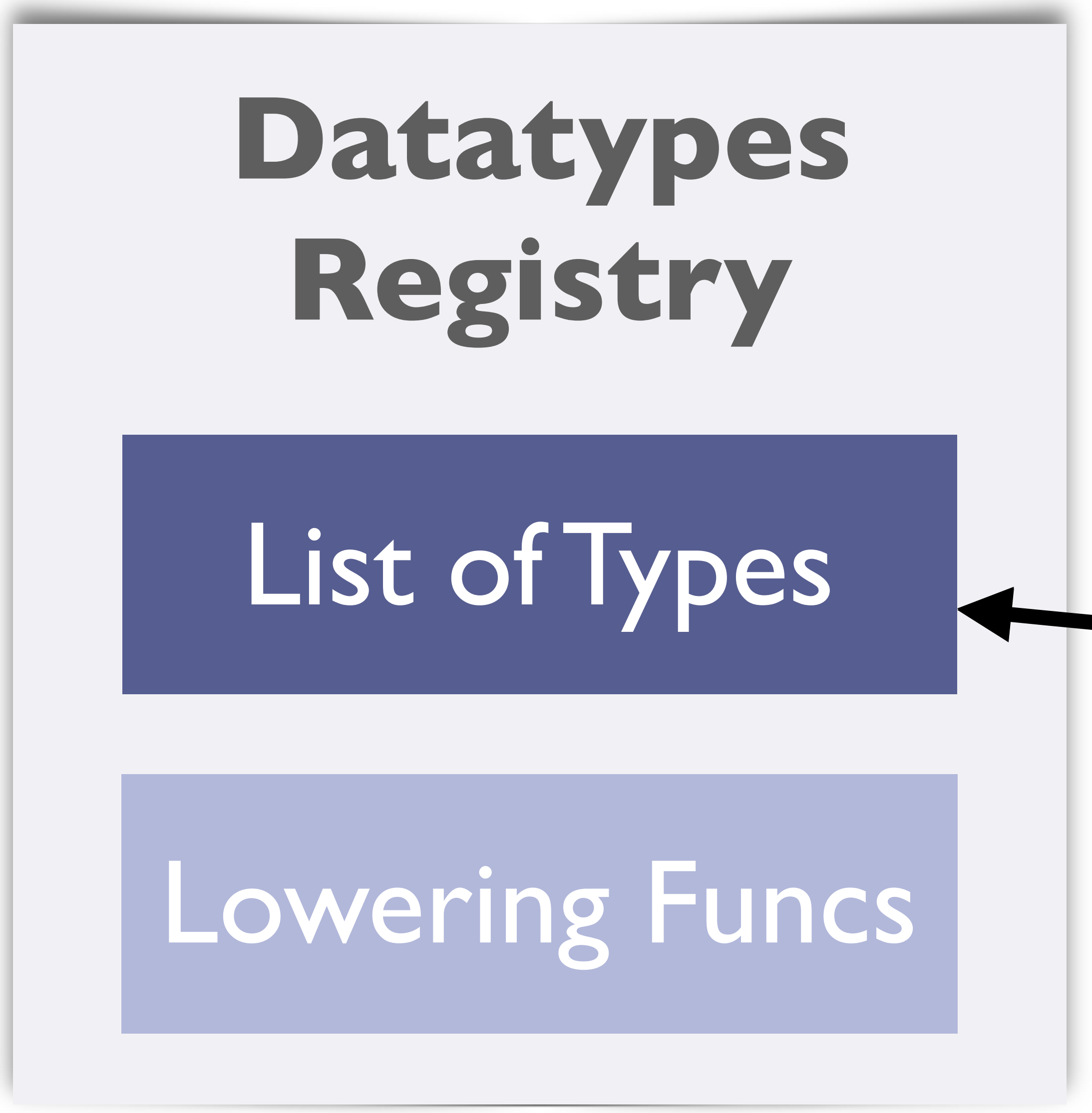
To register their datatype, the user provides a name and type code:

```
tvm.datatype.register("bfloat", 129)
```



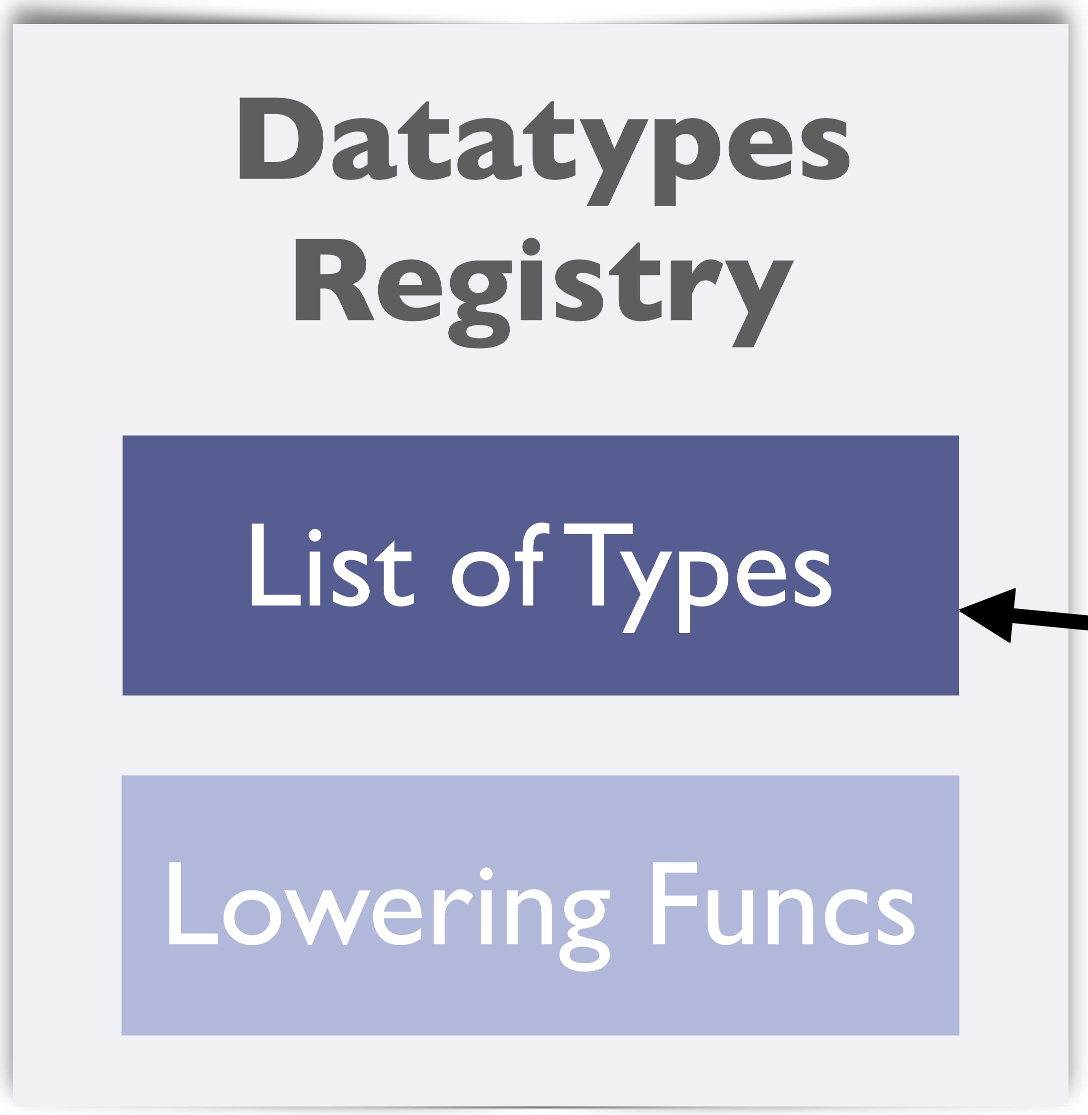
To register their datatype, the user provides a name and type code:

```
tvm.datatype.register("bfloat", 129)
```



To register their datatype, the user provides a name and type code:

```
tvm.datatype.register("bfloat", 129)
```



To register their datatype, the user provides a name and type code:

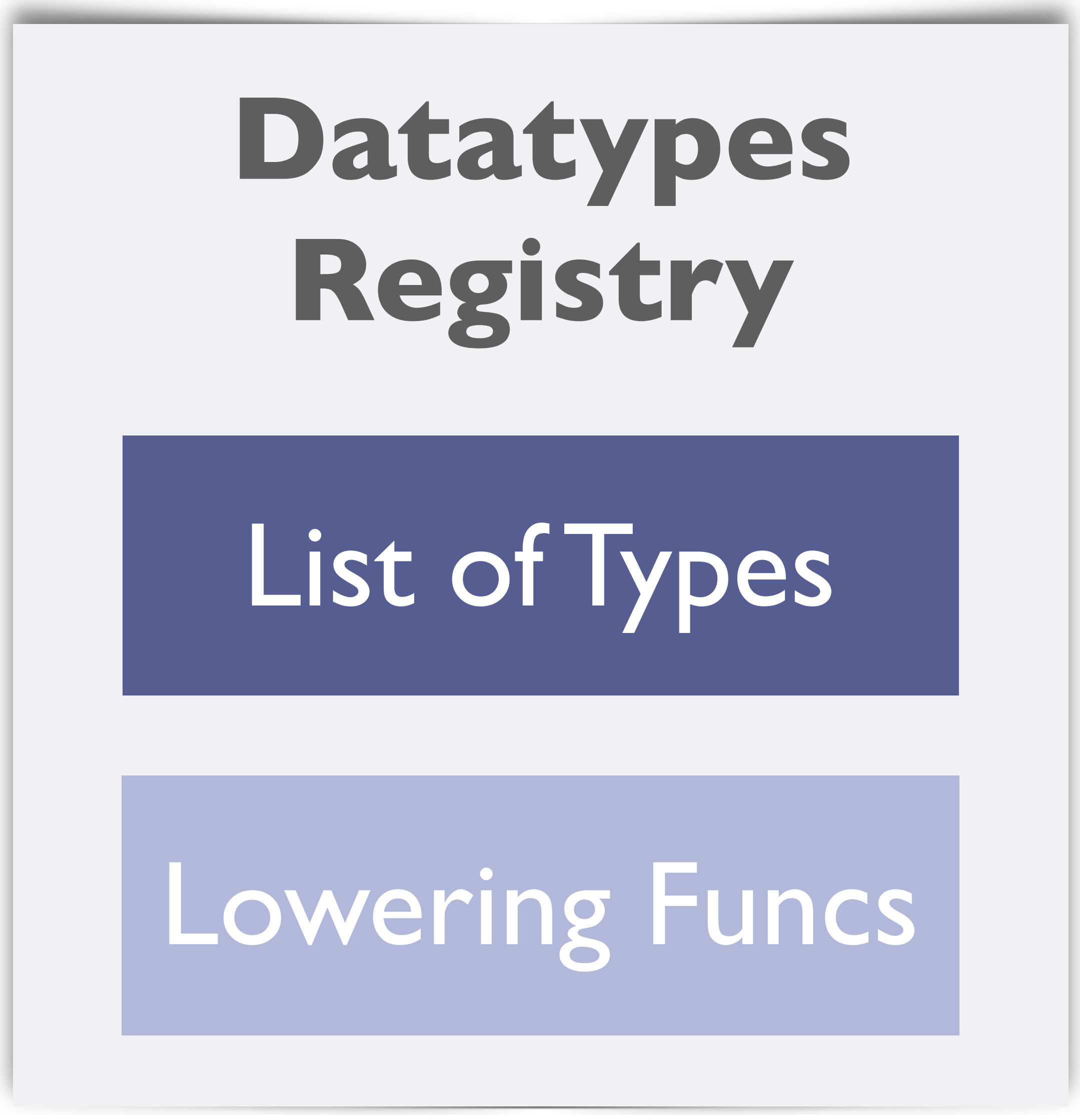
```
tvm.datatype.register("bfloat", 129)
```

This allows TVM to parse programs which use the datatype!

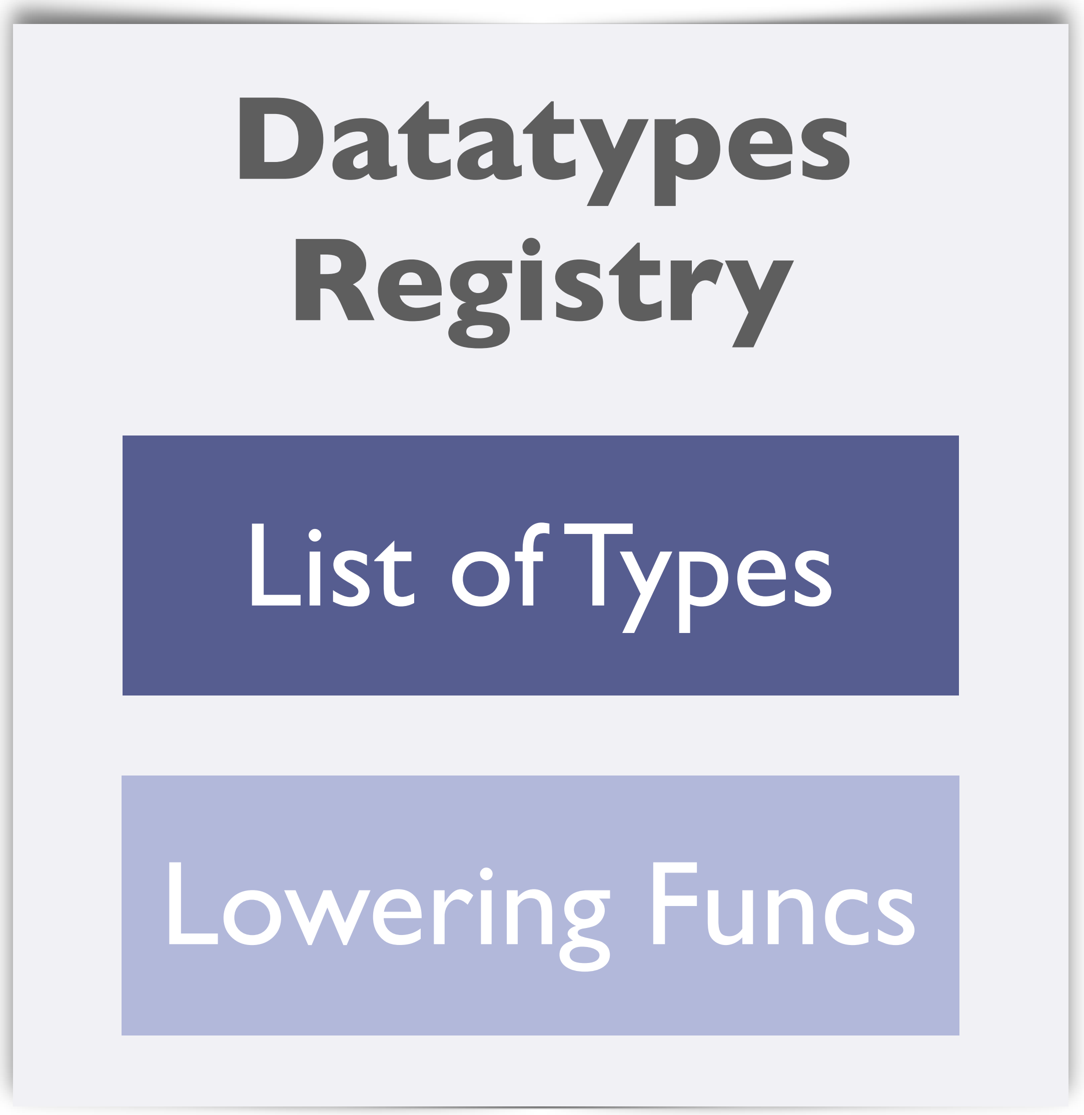
Datatypes Registry

List of Types

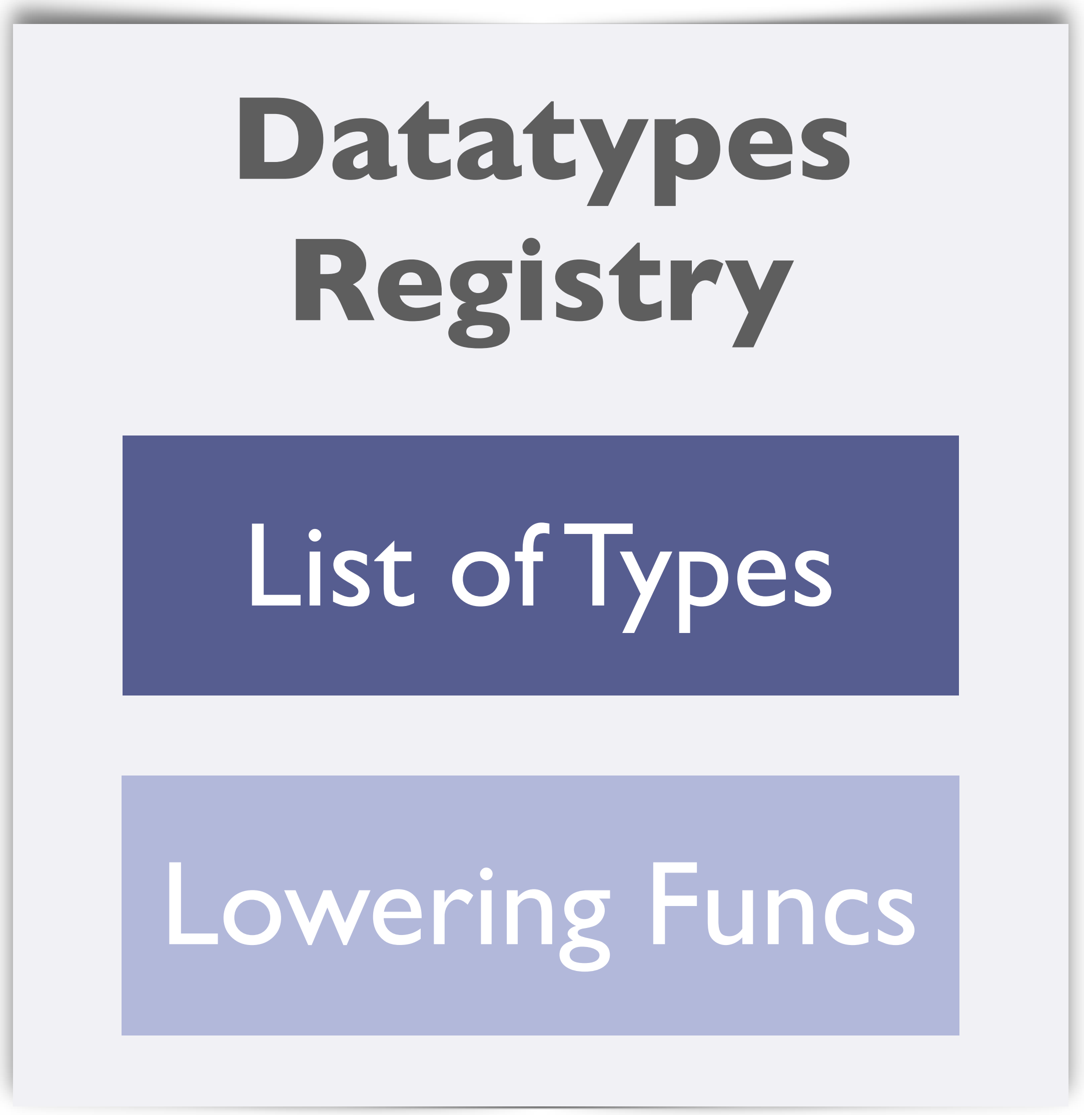
Lowering Funcs



Users register lowering functions with a similar API:

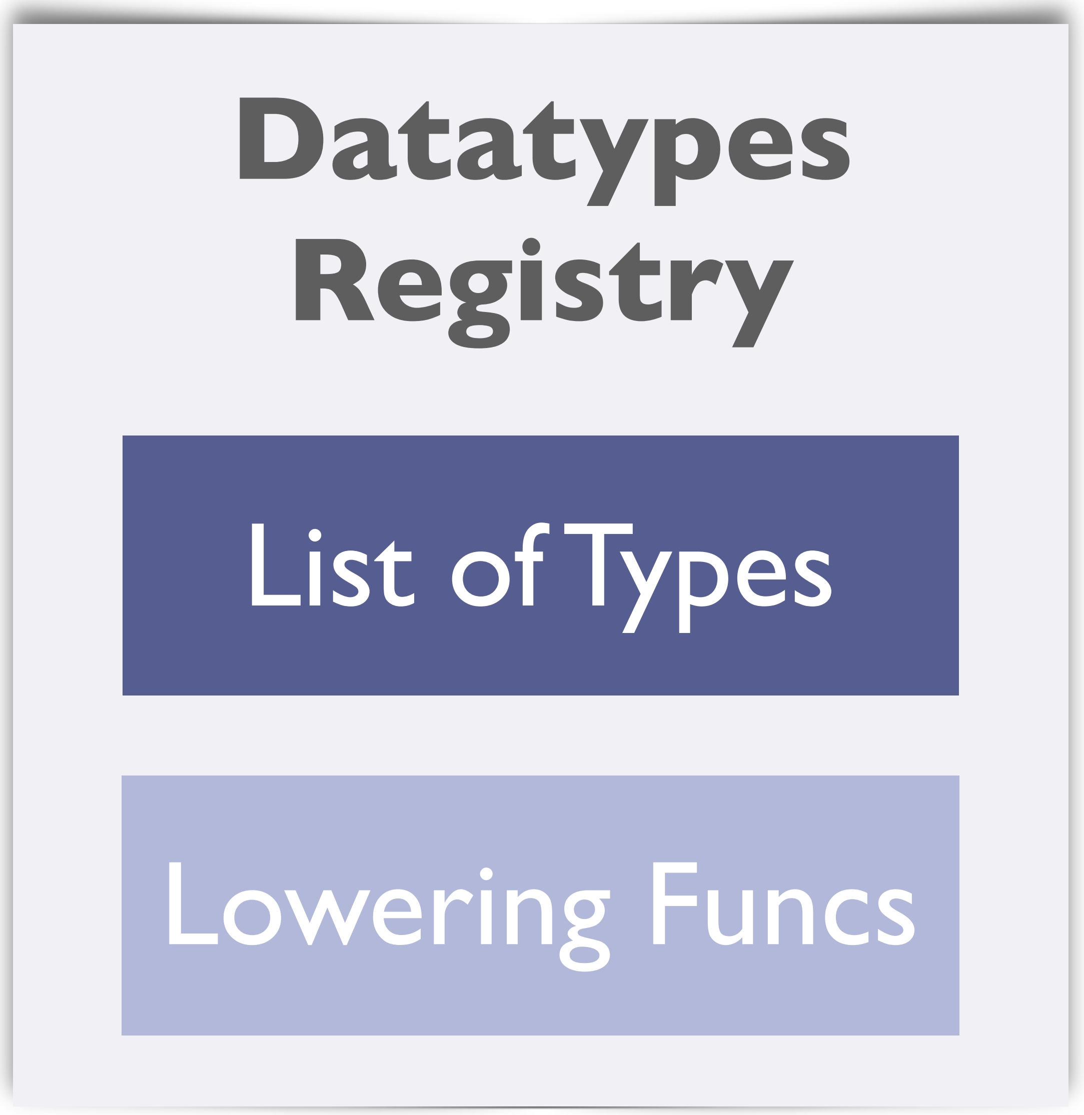


Users register lowering functions with a similar API:



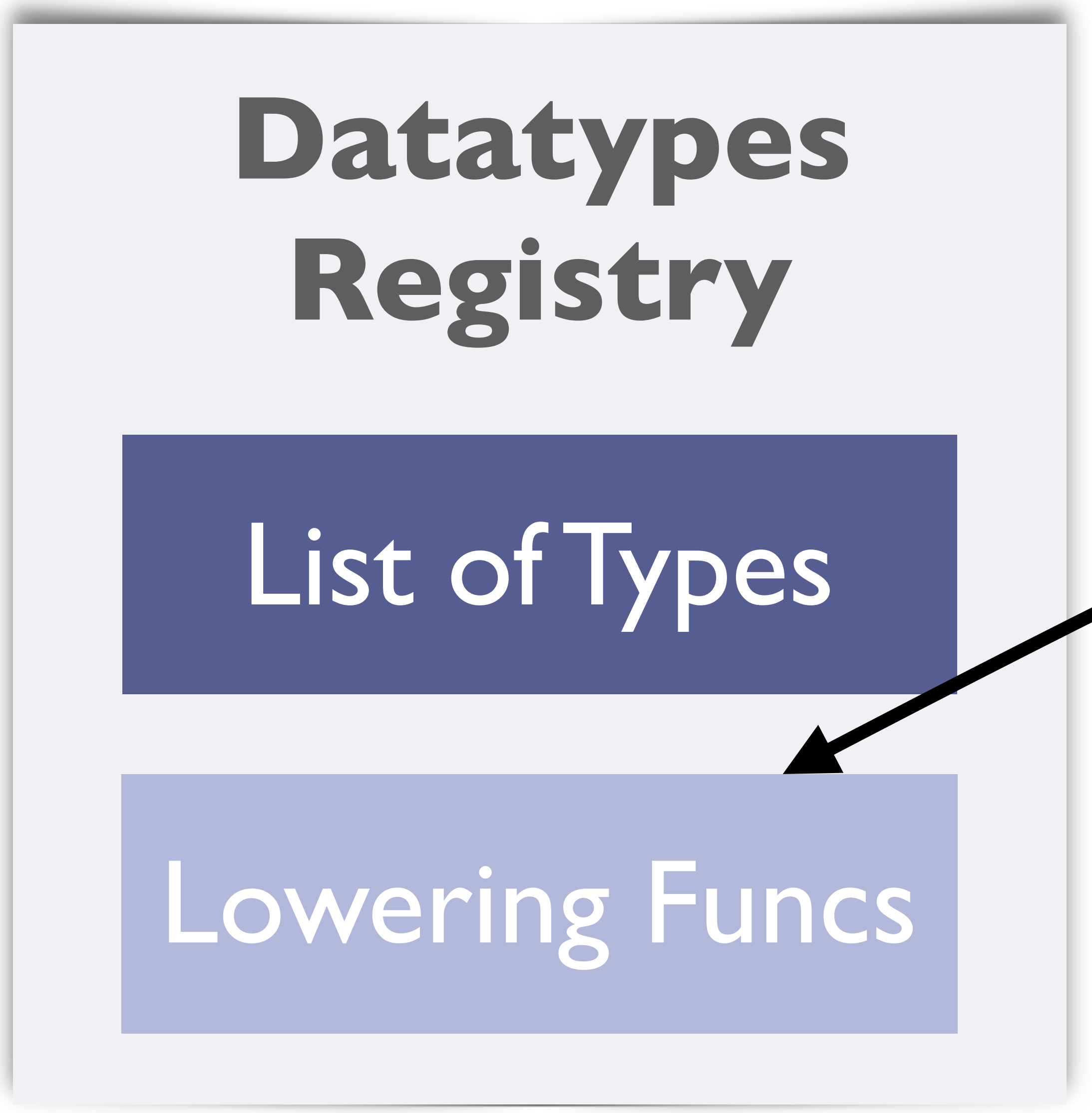
Users register lowering functions with a similar API:

```
tvm.datatype.register_op(
```



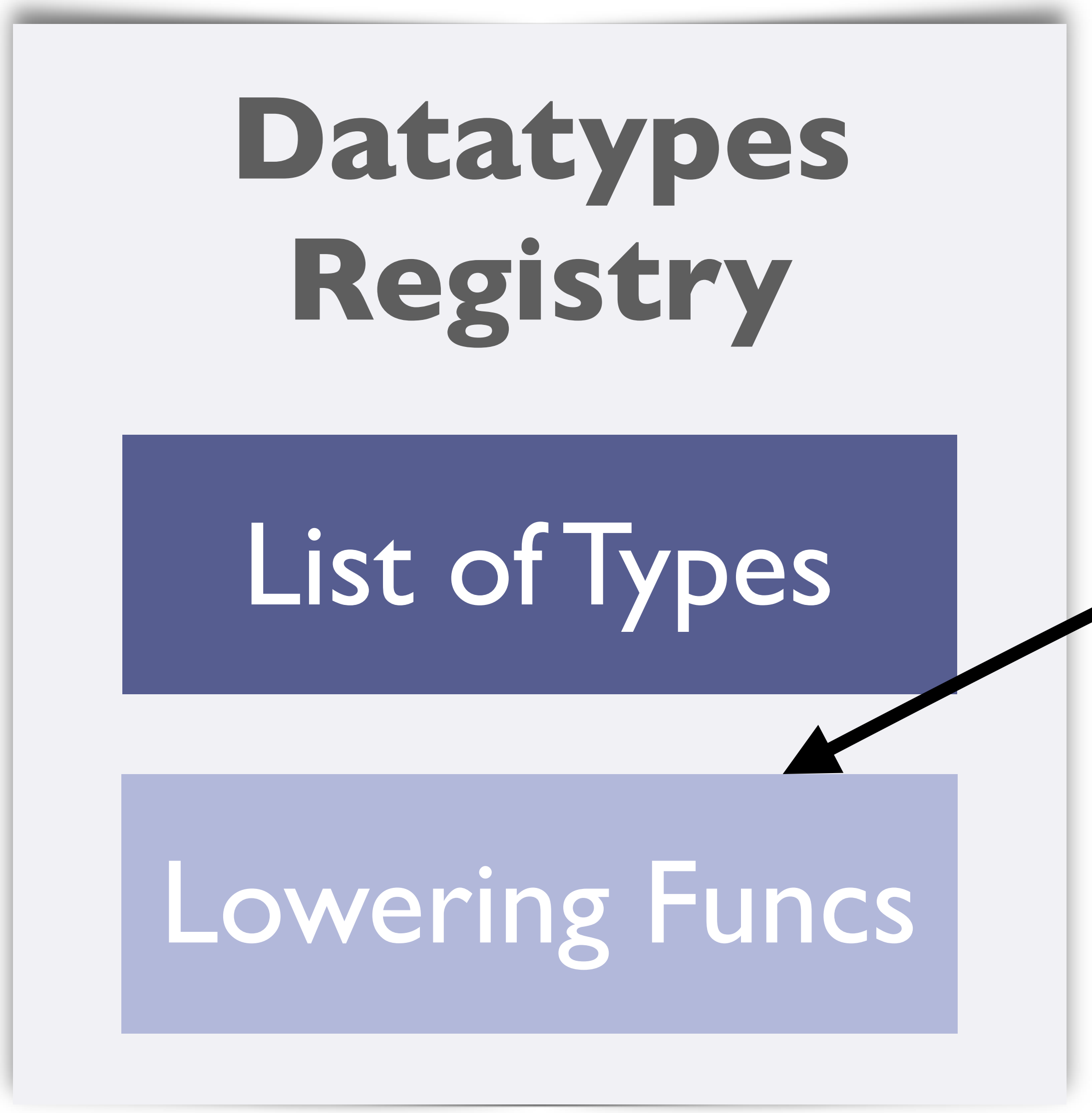
Users register lowering functions with a similar API:

```
tvm.datatype.register_op(
        lower_func, "Add", “llvm", “bfloat”)
```



Users register lowering functions with a similar API:

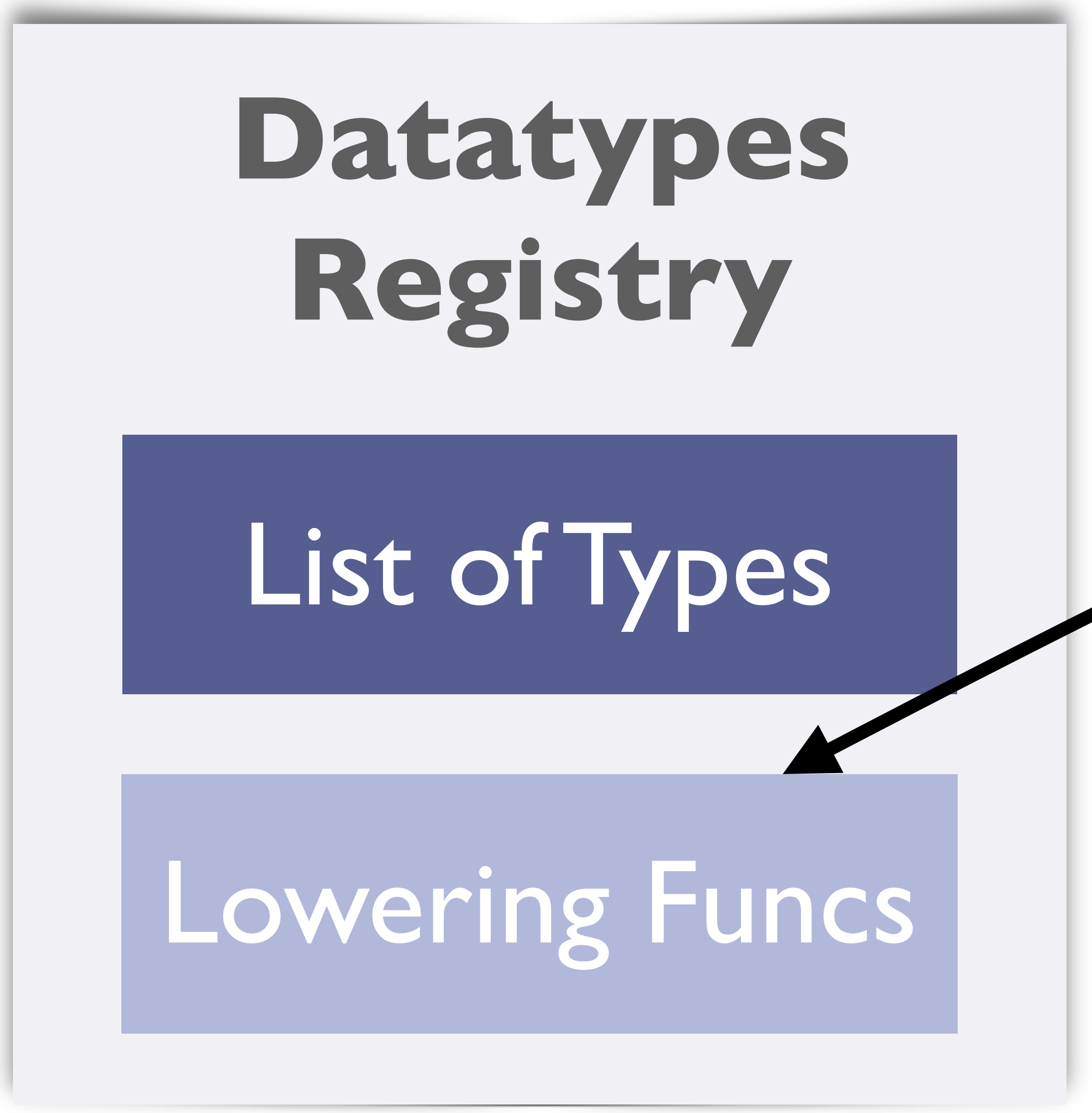
```
tvm.datatype.register_op(
        lower_func, "Add", “llvm", “bfloat”)
```



Users register lowering functions with a similar API:

```
tvm.datatype.register_op(
        lower_func, "Add", “llvm", “bfloat”)
```

This registers a lowering function which lowers bfloat Adds when compiling to LLVM.



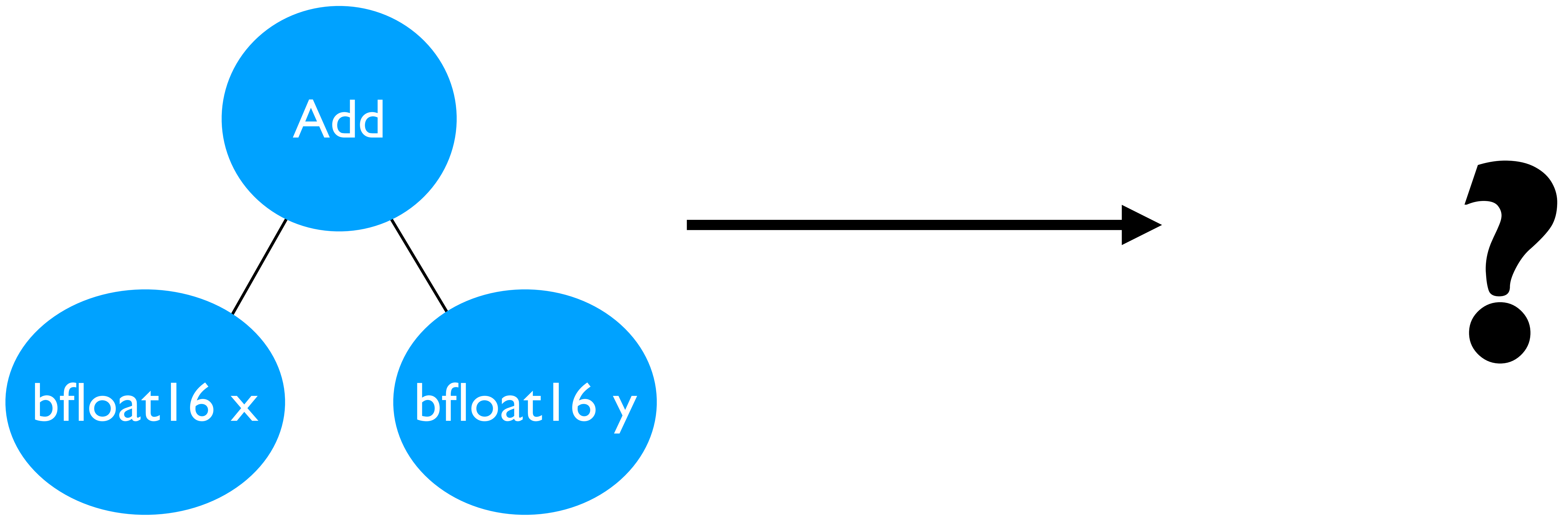
Users register lowering functions with a similar API:

```
tvm.datatype.register_op(
        lower_func, "Add", “llvm", “bfloat”)
```

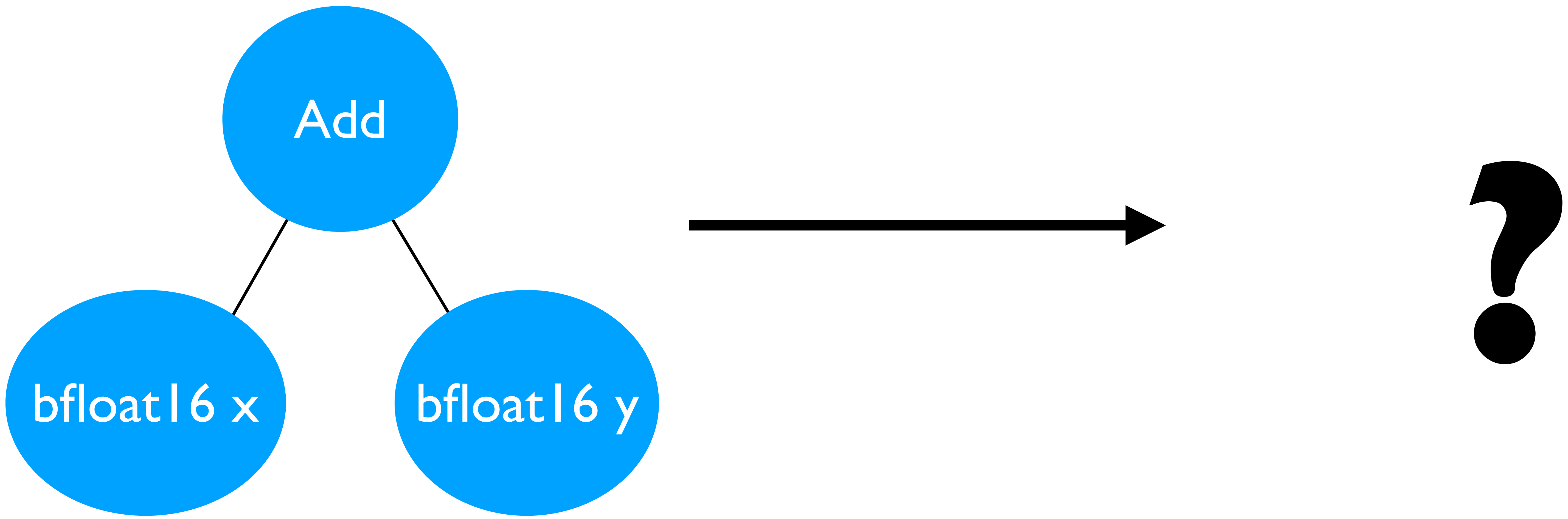
This registers a lowering function which lowers bfloat Adds when compiling to LLVM.

TVM will later use this lowering function when it spits out code!

What do lowering functions look like?

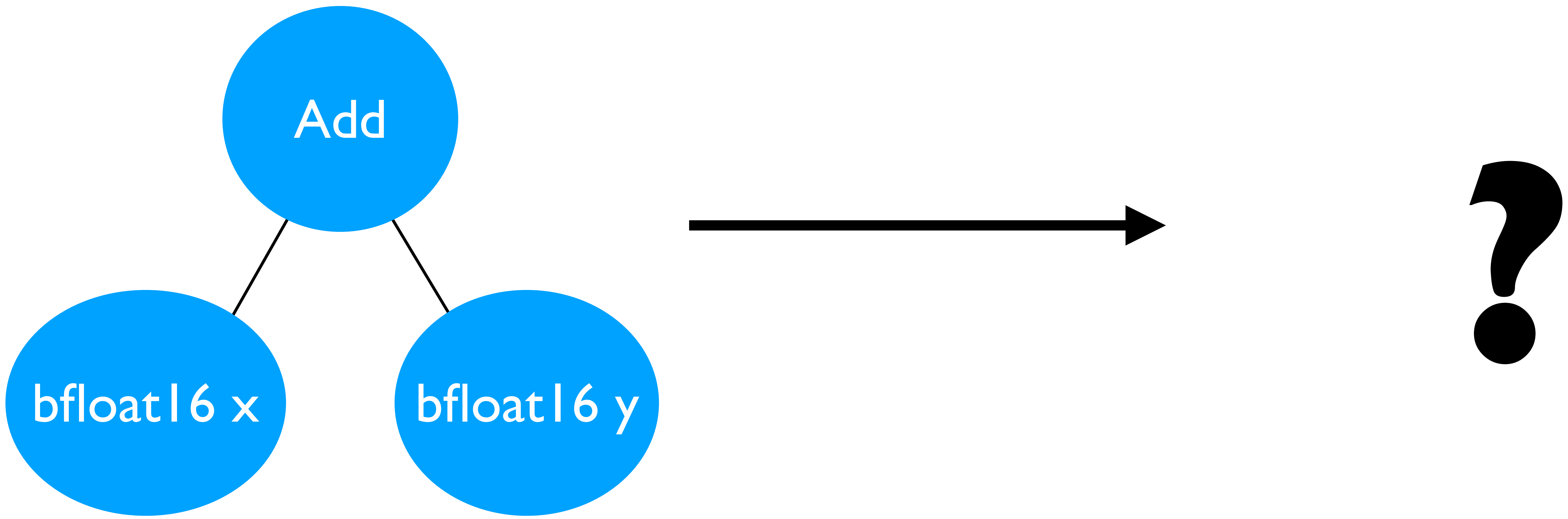


What do lowering functions look like?



Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

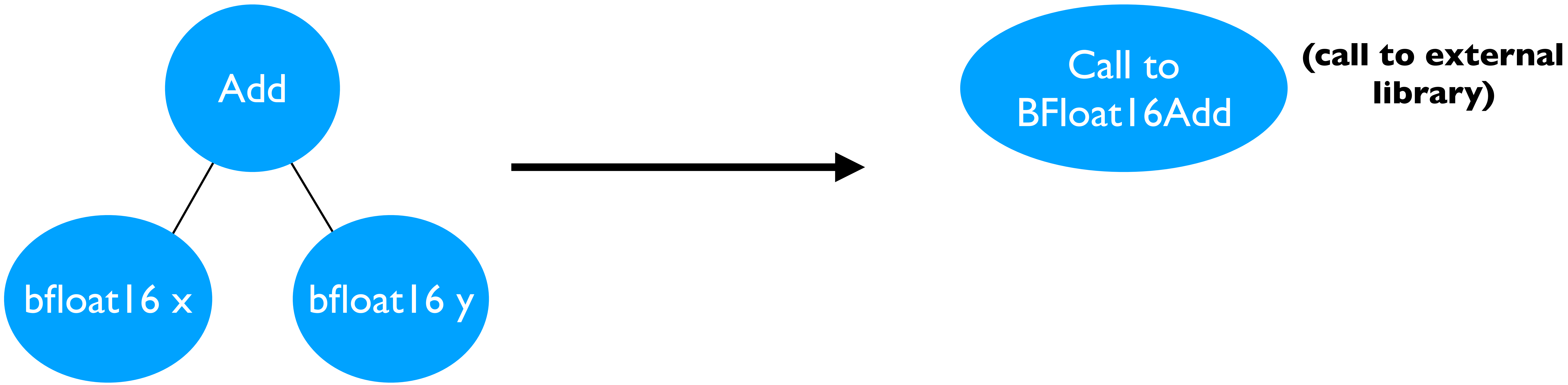
What do lowering functions look like?



Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

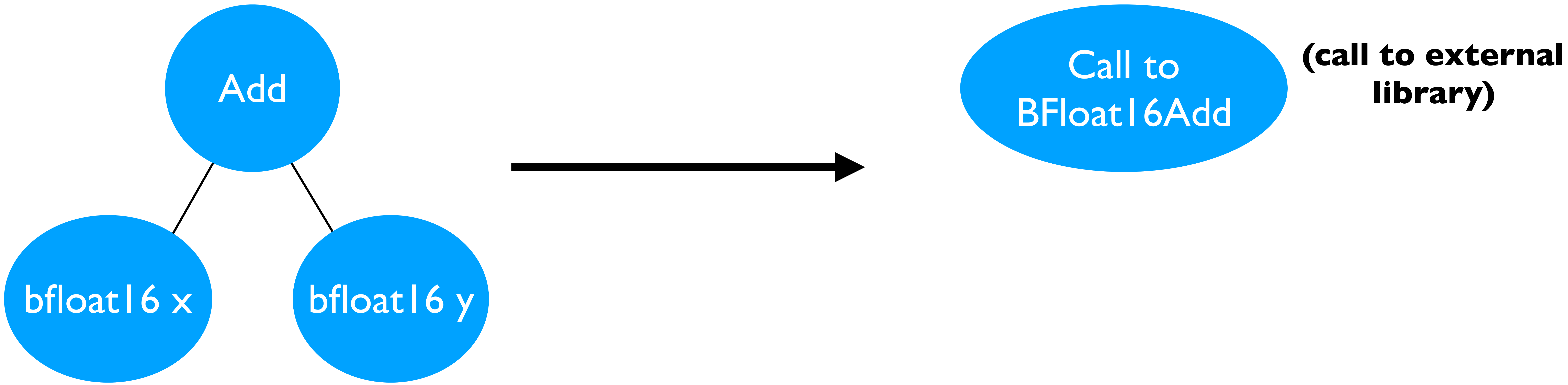
What do lowering functions look like?



Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

What do lowering functions look like?

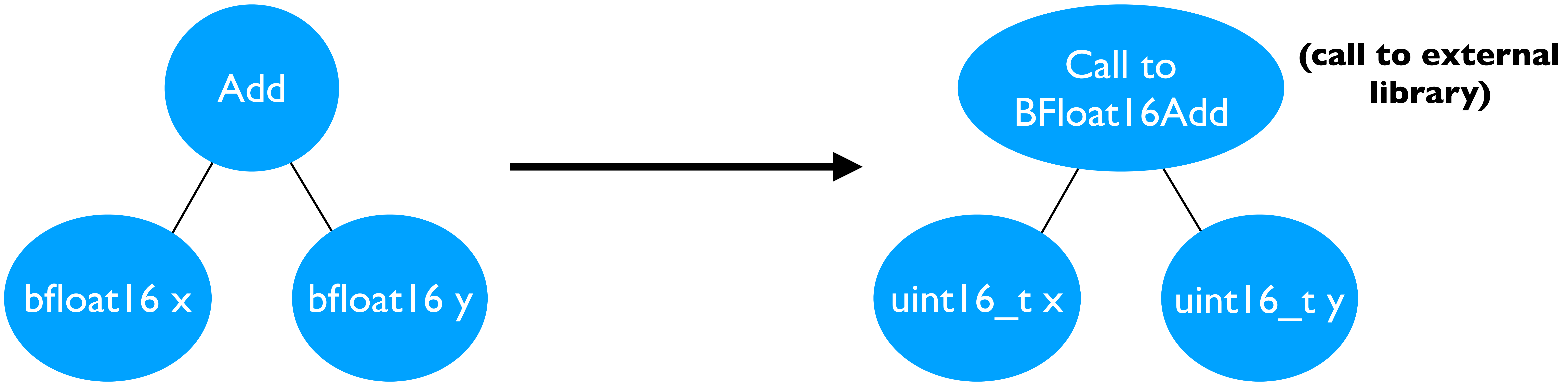


Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

And the inputs? They can just be hidden in opaque unsigned integer types!

What do lowering functions look like?

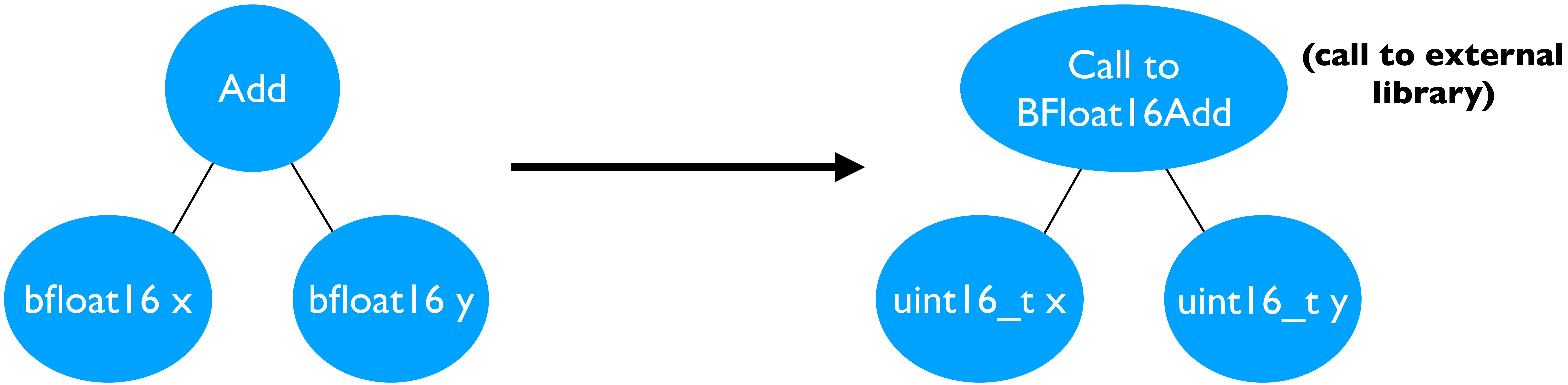


Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

And the inputs? They can just be hidden in opaque unsigned integer types!

What do lowering functions look like?



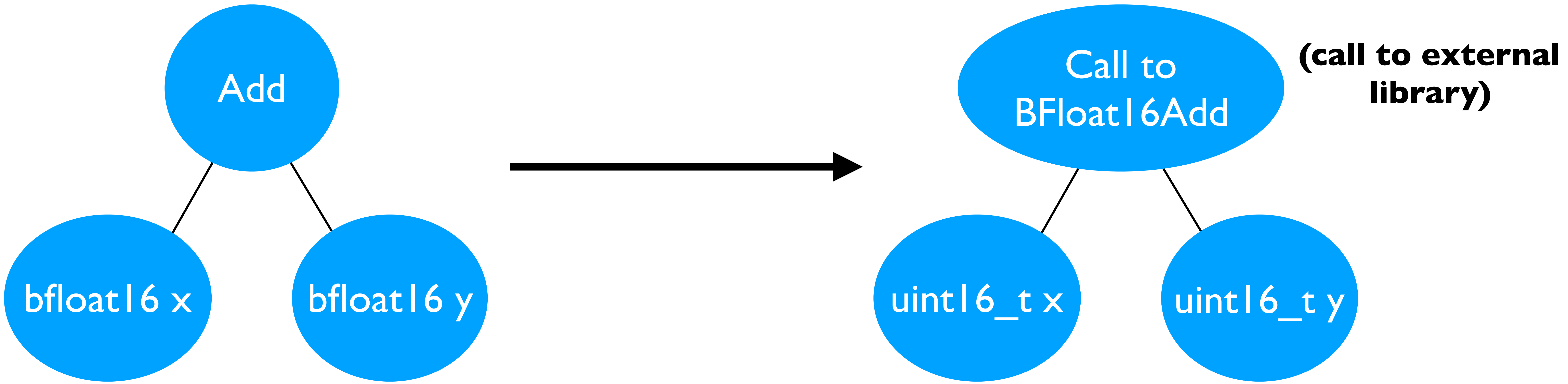
Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

And the inputs? They can just be hidden in opaque unsigned integer types!

We provide a helper function for creating this type of lowering function:

What do lowering functions look like?



Lowering functions convert programs using custom datatypes into programs that native TVM can understand and compile.

In our case, we know our Add over bfloats should become a call to our bfloat library!

And the inputs? They can just be hidden in opaque unsigned integer types!

We provide a helper function for creating this type of lowering function:

```
tvm.datatype.create_lower_func("BFloat16Add")
```

What do we want?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

2. User **gives TVM some information** about the library

3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

What do we have?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

2. User **gives TVM some information** about the library

3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

What do we have?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

2. User **gives TVM some information** about the library

    - Datatype name

3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

What do we have?

1. User **makes or finds** a custom datatype library which they'd like to use in deep learning workloads

2. User **gives TVM some information** about the library

   - Datatype name

   - Lowering functions—user just provides names of library functions!

3. TVM **compiles and runs programs** which use the custom datatype, handling the custom datatype by calling into the provided library

Evaluation

To exercise the framework, I decided to conduct a preliminary evaluation of how a model's trained accuracy changes as we change the datatype.

To exercise the framework, I decided to conduct a preliminary evaluation of how a model's trained accuracy changes as we change the datatype.

We will first discuss the experiment itself and its results—then, we will reflect on the utility of the framework.

Experiment Design

Experiment Design

1. Gathered a list of datatypes

Experiment Design

1. Gathered a list of datatypes
    - TVM-native

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made
   - From GitHub

Experiment Design

1. Gathered a list of datatypes
    - TVM-native
    - Hand-made
    - From GitHub

2. Pretrained models on the entire CIFAR-10 training set (50k images, 10 classes) in float32 using PyTorch

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made
   - From GitHub

2. Pretrained models on the entire CIFAR-10 training set (50k images, 10 classes) in float32 using PyTorch

3. Converted the pretrained weights to the custom datatypes.

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made
   - From GitHub

2. Pretrained models on the entire CIFAR-10 training set (50k images, 10 classes) in float32 using PyTorch

3. Converted the pretrained weights to the custom datatypes.
   - This was done without retraining

Experiment Design

1. Gathered a list of datatypes
   - TVM-native
   - Hand-made
   - From GitHub

2. Pretrained models on the entire CIFAR-10 training set (50k images, 10 classes) in float32 using PyTorch

3. Converted the pretrained weights to the custom datatypes.
   - This was done without retraining

4. Ran the models with the converted datatypes over a sample of the CIFAR-10 test set (100 images each) using TVM

Datatypes:

Datatypes:

- **TVM-native float32** (not using the framework)

Datatypes:

- **TVM-native float32** (not using the framework)
- **float32** (using the framework)

Datatypes:

- **TVM-native float32** (not using the framework)
- **float32** (using the framework)
- Two implementations of **posit8es0**, **posit16es1**, **posit32es2**:
  - Stillwater Supercomputing's Universal library
  - libposit

Datatypes:

- **TVM-native float32** (not using the framework)
- **float32** (using the framework)
- Two implementations of **posit8es0**, **posit16es1**, **posit32es2**:
  - Stillwater Supercomputing's Universal library
  - libposit
- Two implementations of **bfloat16**:
  - My own "naive" implementation
  - biovault-bfloat16: another implementation from GitHub

Datatypes:

- **TVM-native float32** (not using the framework)
- **float32** (using the framework)
- Two implementations of **posit8es0**, **posit16es1**, **posit32es2**:
  - Stillwater Supercomputing's Universal library
  - libposit
- Two implementations of **bfloat16**:
  - My own "naive" implementation
  - biovault-bfloat16: another implementation from GitHub
- "**noptype**": always returns 0

Models:

- MobilenetV1
- Resnet50

Experiment Results and Evaluation

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
✅ **Universal posit16**	0.77	0.71
✅ **Universal posit32**	0.77	0.71
libposit posit8	0.08	0.1
✅ **libposit posit16**	0.77	0.71
✅ **libposit posit32**	0.77	0.71

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
our bfloat16	0.08	0.11
biovault bfloat16	0.1	0.1
❌ **Universal posit8**	0.08	0.1
Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
❌ **libposit posit8**	0.08	0.1
libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

Same size!

	resnet accuracy	mobilenet accuracy
float32	0.77	0.71
❌ our bfloat16	0.08	0.11
❌ biovault bfloat16	0.1	0.1
Universal posit8	0.08	0.1
✅ Universal posit16	0.77	0.71
Universal posit32	0.77	0.71
libposit posit8	0.08	0.1
✅ libposit posit16	0.77	0.71
libposit posit32	0.77	0.71

Framework Evaluation

I wanted to evaluate three aspects of the framework:

I wanted to evaluate three aspects of the framework:

- Overhead

I wanted to evaluate three aspects of the framework:

- Overhead
- Ease of use

I wanted to evaluate three aspects of the framework:

- Overhead
- Ease of use
- Breadth of datatypes which can be used

Overhead

To measure overhead, I compared the inference time of three types:

To measure overhead, I compared the inference time of three types:

- TVM-native float32

To measure overhead, I compared the inference time of three types:

- TVM-native float32
- float32 implemented in the framework

To measure overhead, I compared the inference time of three types:

- TVM-native float32
- float32 implemented in the framework
- “noptype”: a custom type that does no work

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

Main source of overhead: not in added computation, but in the **compilation opportunity cost.**

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

Main source of overhead: not in added computation, but in the **compilation opportunity cost.**

- noptype is fairly low-overhead in both workloads → function calls don't add too much

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

Main source of overhead: not in added computation, but in the **compilation opportunity cost.**

- noptype is fairly low-overhead in both workloads → function calls don't add too much
- float32 is low-overhead in Mobilenet, which is optimized for low total float ops

mobilenet v1

	mean inference time (s)	framework overhead
native float32	0.10	1x
float32	0.12	1.11x
noptype	0.11	1.10x

resnet50

	mean inference time (s)	framework overhead
native float32	0.21	1x
float32	0.52	2.51x
noptype	0.28	1.35x

Main source of overhead: not in added computation, but in the **compilation opportunity cost.**

- noptype is fairly low-overhead in both workloads → function calls don't add too much
- float32 is low-overhead in Mobilenet, which is optimized for low total float ops
- …but float32 is high-overhead in ResNet → native float32 ResNet much more optimized!

Ease of Use

By the Numbers…

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
    - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
    - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.
- In the **150-line** Python script which runs ResNet50 with posit32,

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
    - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.
- In the **150-line** Python script which runs ResNet50 with posit32,
    - **57 lines** register the datatype and define lowering functions for the 12 operators,

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
  - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.
- In the **150-line** Python script which runs ResNet50 with posit32,
  - **57 lines** register the datatype and define lowering functions for the 12 operators,
  - **3 lines** convert the model to posit32,

By the Numbers…

To use libposit posit32 in Mobilenet and ResNet…

- A total of **12 operators** needed to be implemented, taking **about 70 lines of C++ in a wrapper library**.
  - Operators including **casts to/from posits**, **add/sub/mul/div**, more complex math operators like **exp**, and comparators like **max**.
- In the **150-line** Python script which runs ResNet50 with posit32,
  - **57 lines** register the datatype and define lowering functions for the 12 operators,
  - **3 lines** convert the model to posit32,
  - **3 lines** convert the input, run the model, and convert the output.

How could we improve?

How could we improve?

- Allow the user to specify their own calling convention, removing the need for a wrapper over the library

How could we improve?

- Allow the user to specify their own calling convention, removing the need for a wrapper over the library
- Implement cleaner registration functions in the TVM Python frontend

Breadth of Datatypes

Breadth of Datatypes

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Questionable whether current BYOD can represent…

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Questionable whether current BYOD can represent…

- Block floating point, or any other type with “external” state

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Questionable whether current BYOD can represent…

- Block floating point, or any other type with "external" state
- Datatypes with elements larger than 64 bits

Breadth of Datatypes

We were able to successfully represent a small sample of modern datatypes!

Questionable whether current BYOD can represent…

- Block floating point, or any other type with “external” state
- Datatypes with elements larger than 64 bits

Could potentially be implemented by allowing datatypes to attach metadata to each scalar

Future Work

Future Work

- Training in TVM—ramifications for custom datatypes?

Future Work

- Training in TVM—ramifications for custom datatypes?
- Improve performance

Future Work

- Training in TVM—ramifications for custom datatypes?
- Improve performance
  - Enable inlining of LLVM

Future Work

- Training in TVM—ramifications for custom datatypes?
- Improve performance
  - Enable inlining of LLVM
  - Allow users to supply optimized kernels for higher-level operators, e.g. posit conv2d

Future Work

- Training in TVM—ramifications for custom datatypes?
- Improve performance
  - Enable inlining of LLVM
  - Allow users to supply optimized kernels for higher-level operators, e.g. posit conv2d
- Tackle complex datatypes like block floating point

In conclusion, I have presented the Bring Your Own Datatypes framework.

In conclusion, I have presented the Bring Your Own Datatypes framework.

In conclusion, I have presented the Bring Your Own Datatypes framework.

I have shown how the framework can enable useful datatype research.

Thank You!

Extra Slides

Registering the Datatype

```
import tvm
from tvm import relay
from ctypes import CDLL, RTLD_GLOBAL

CDLL('libposit-wrapper.so', RTLD_GLOBAL)

tvm.datatype.register('posit32', 131)

tvm.datatype.register_op(
    tvm.datatype.create_lower_func('Posit32Add'),
    'Add', 'llvm', 'posit32')
```

Registering the Datatype

✅ **Any loaded C-linkage functions will work!**

```python
import tvm
from tvm import relay
from ctypes import CDLL, RTLD_GLOBAL

CDLL('libposit-wrapper.so', RTLD_GLOBAL)

tvm.datatype.register('posit32', 131)

tvm.datatype.register_op(
    tvm.datatype.create_lower_func('Posit32Add'),
    'Add', 'llvm', 'posit32')
```

Registering the Datatype

✅ **Any loaded C-linkage functions will work!**

```
import tvm
from tvm import relay
from ctypes import CDLL, RTLD_GLOBAL

CDLL('libposit-wrapper.so', RTLD_GLOBAL)

tvm.datatype.register('posit32', 131)

tvm.datatype.register_op(
    tvm.datatype.create_lower_func('Posit32Add'),
    'Add', 'llvm', 'posit32')
```

✅ **Helper functions for common-case**

Implementing the Datatype

```
#include <cstdint>
#include <posit.h>

posit32_t Uint32ToPosit32(uint32_t in) {
  return posit32_reinterpret(in);
}

uint32_t Posit32ToUint32(posit32_t in) {
  return posit32_bits(in);
}

extern "C" uint32_t Posit32Add(uint32_t a, uint32_t b) {
  auto a_p = Uint32ToPosit32(a);
  auto b_p = Uint32ToPosit32(b);
  auto sum_p = posit32_add(a_p, b_p);
  return Posit32ToUint32(sum_p);
}
```

Implementing the Datatype

```
#include <cstdint>
#include <posit.h>

posit32_t Uint32ToPosit32(uint32_t in) {
  return posit32_reinterpret(in);
}

uint32_t Posit32ToUint32(posit32_t in) {
  return posit32_bits(in);
}

extern "C" uint32_t Posit32Add(uint32_t a, uint32_t b) {
  auto a_p = Uint32ToPosit32(a);
  auto b_p = Uint32ToPosit32(b);
  auto sum_p = posit32_add(a_p, b_p);
  return Posit32ToUint32(sum_p);
}
```

❌ **Inflexible calling convention**

Converting the Program

```python
module, params = load_resnet50()
module, params = change_dtype('float32', 'custom[posit32]32',
                              module['main'], params)

ex = relay.create_executor(mod=module)
model = ex.evaluate()
```

Converting the Program

```
module, params = load_resnet50()
module, params = change_dtype('float32', 'custom[posit32]32',
                              module['main'], params)

ex = relay.create_executor(mod=module)
model = ex.evaluate()
```

✅ **Changing datatype of a program is easy!**

Running the Program

```python
image = ... # load image

image = convert_ndarray('custom[posit32]32', image)

with tvm.build_config(disable_vectorize=True):
    output = model(image, **params)

output = convert_ndarray('float32', output)
```

Running the Program

```
image = ... # load image

image = convert_ndarray('custom[posit32]32', image)

with tvm.build_config(disable_vectorize=True):
    output = model(image, **params)

output = convert_ndarray('float32', output)
```

✅ **Converting data is also easy!**

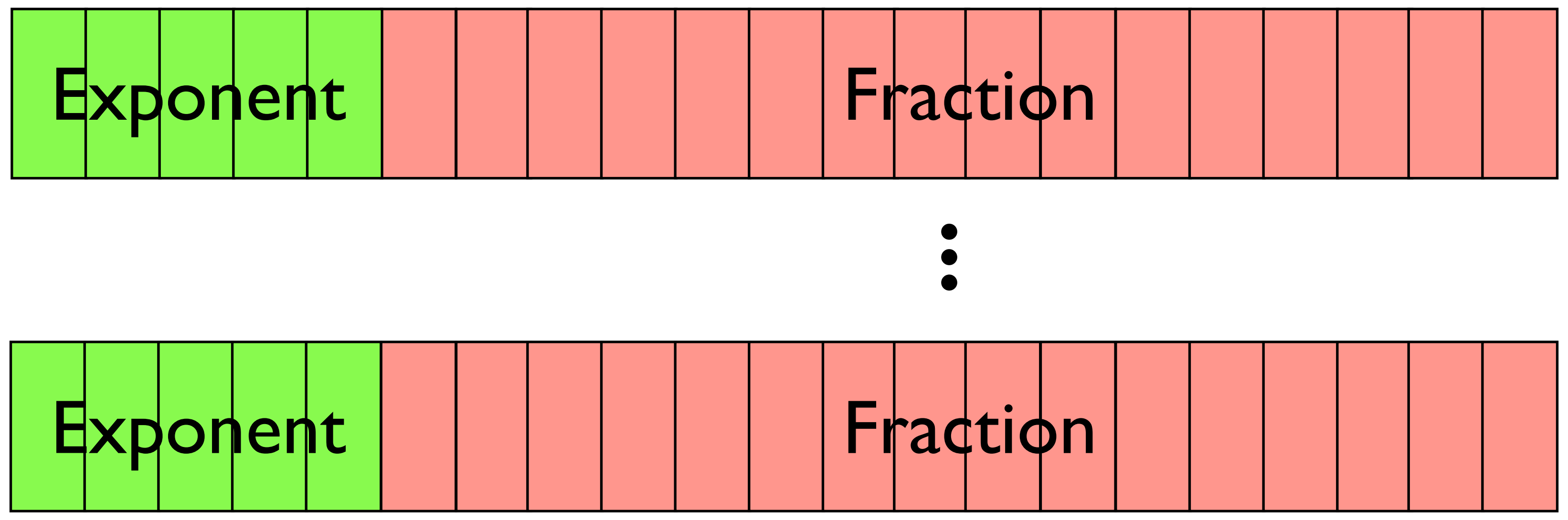
Running the Program

```
image = ... # load image

image = convert_ndarray('custom[posit32]32', image)

with tvm.build_config(disable_vectorize=True):
    output = model(image, **params)

output = convert_ndarray('float32', output)
```

✅ **Converting data is also easy!**

❌ **Have to manually disable vectorization**

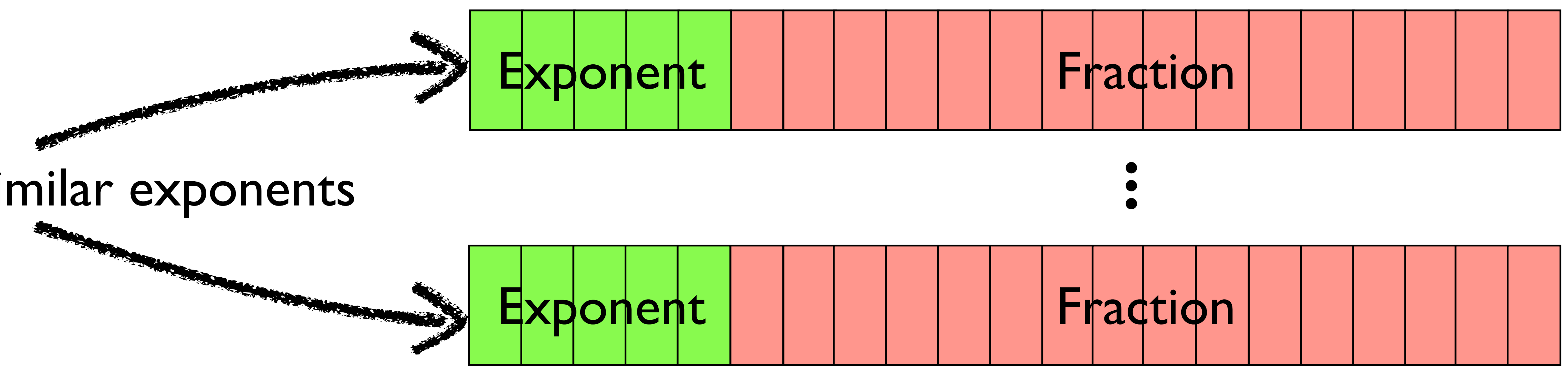
Flexpoint



See Köster et. al., “An Adaptive Numerical Format for Efficient Training of Deep Neural Networks”

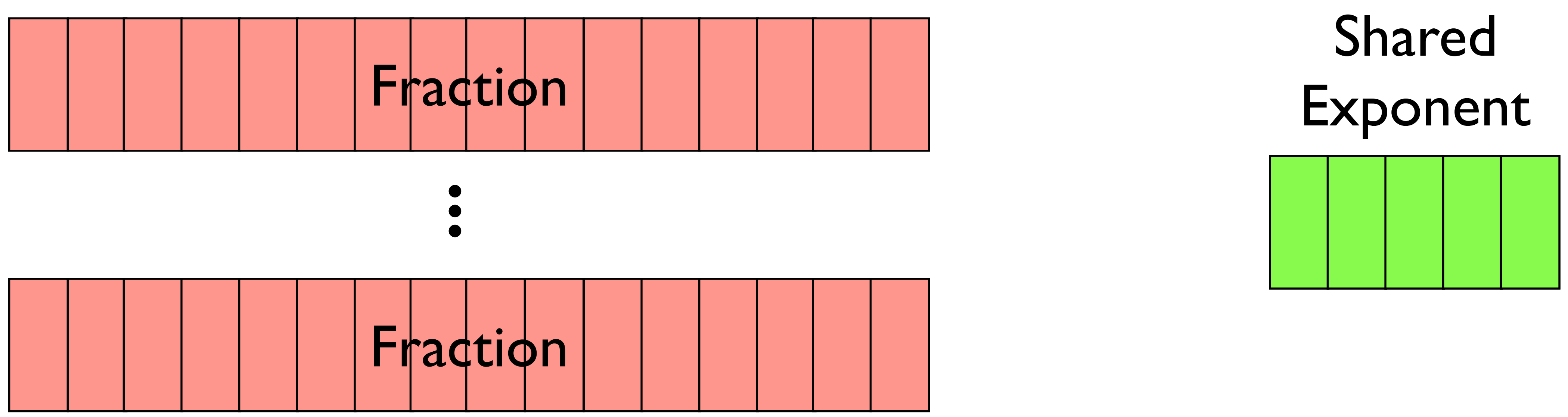
Flexpoint

Similar magnitude → similar exponents



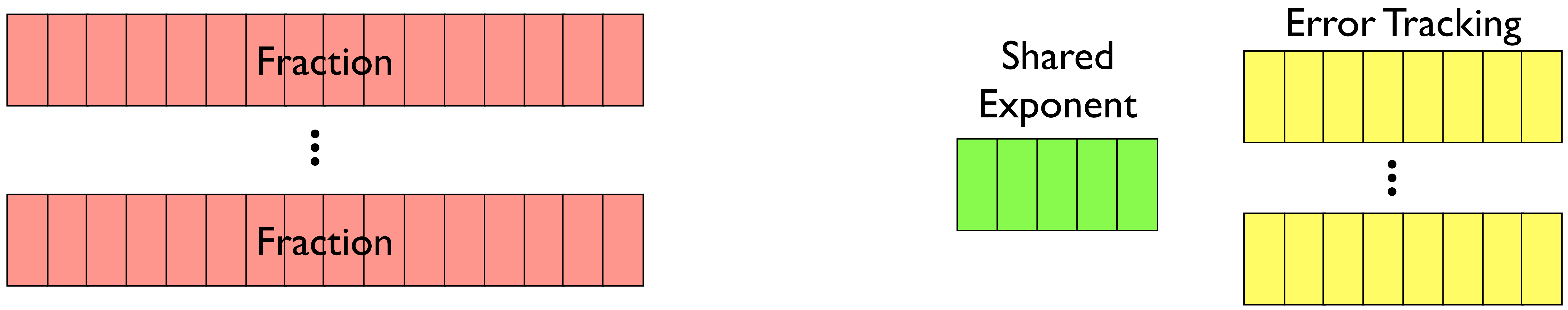
See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

Flexpoint



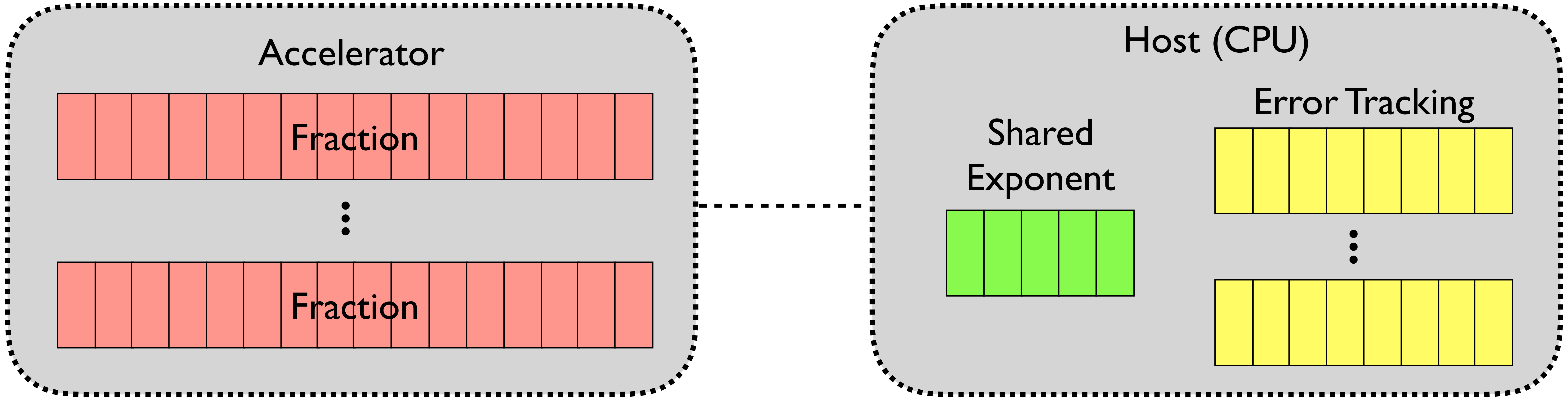
See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

Flexpoint



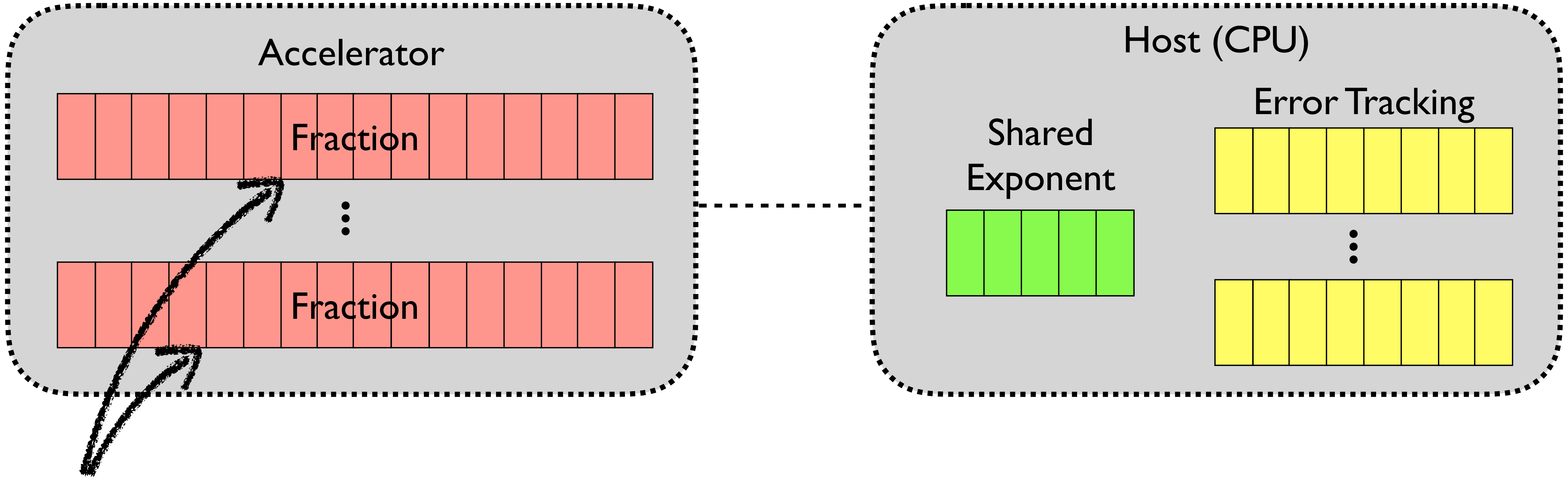
See Köster et. al., "An Adaptive Numerical Format for Efficient Training of Deep Neural Networks"

Flexpoint



See Köster et. al., “An Adaptive Numerical Format for Efficient Training of Deep Neural Networks”

Flexpoint



Just integers! We can use integer hardware!

See Köster et. al., “An Adaptive Numerical Format for Efficient Training of Deep Neural Networks”