MuPy

A First Language Client for Mu Micro Virtual Machine

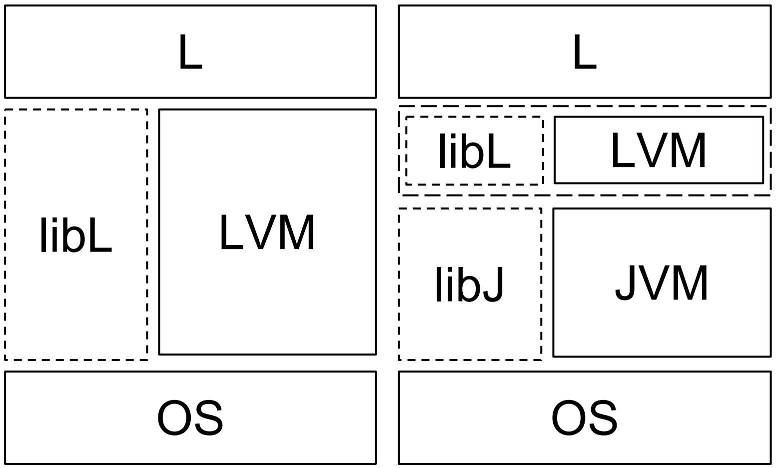
By: John Zhang
Supervised by: Prof. Steve Blackburn
Received help from: Kunshan Wang

Overview

- Motivation & Objective
- Outline of Contributions
- MuPy Translation Process
  - Overview
  - Graph Transform
  - MuType
  - Launcher
- Conclusion & Future Work

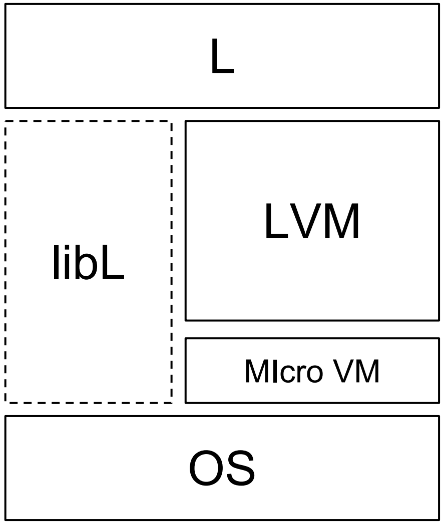
Language Implementation Strategies

- Monolithic
  - everything from ground up;
  - many difficult challenges.
- Virtual Machine Based
  - difficult elements already taken care of;
  - heavy weight (JVM, .Net);
  - highly catered towards its support language.



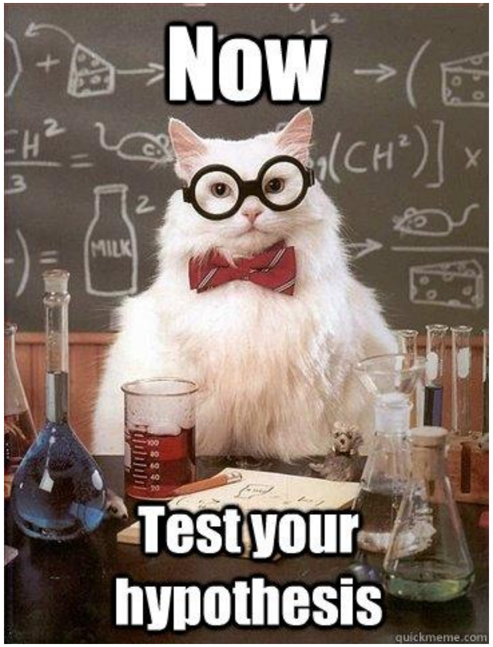
Micro Virtual Machine (μVM)

- A lightweight abstraction over main concerns:
  - concurrency (threads),
  - memory (GC),
  - hardware (JIT, back-ends).
- Key design features:
  - native GC and JIT,
  - cross-language reuse,
  - minimal (lightweight, verifiable).
- Mu -- a concrete μVM instance
  - online spec available (https://github.com/microvm/microvm-spec/wiki).



The need for testing

- Claims needs to be tested.
  - Can Mu actually support a non-trivial language?
  - How good is the design?
- Test by implementation
  - a reference implementation of Mu in Scala is available;
  - build a real language client.

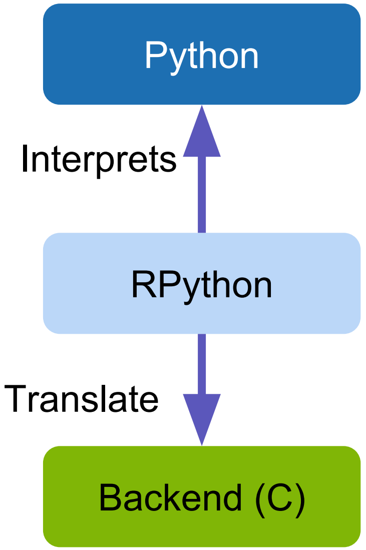


Outline of Contributions

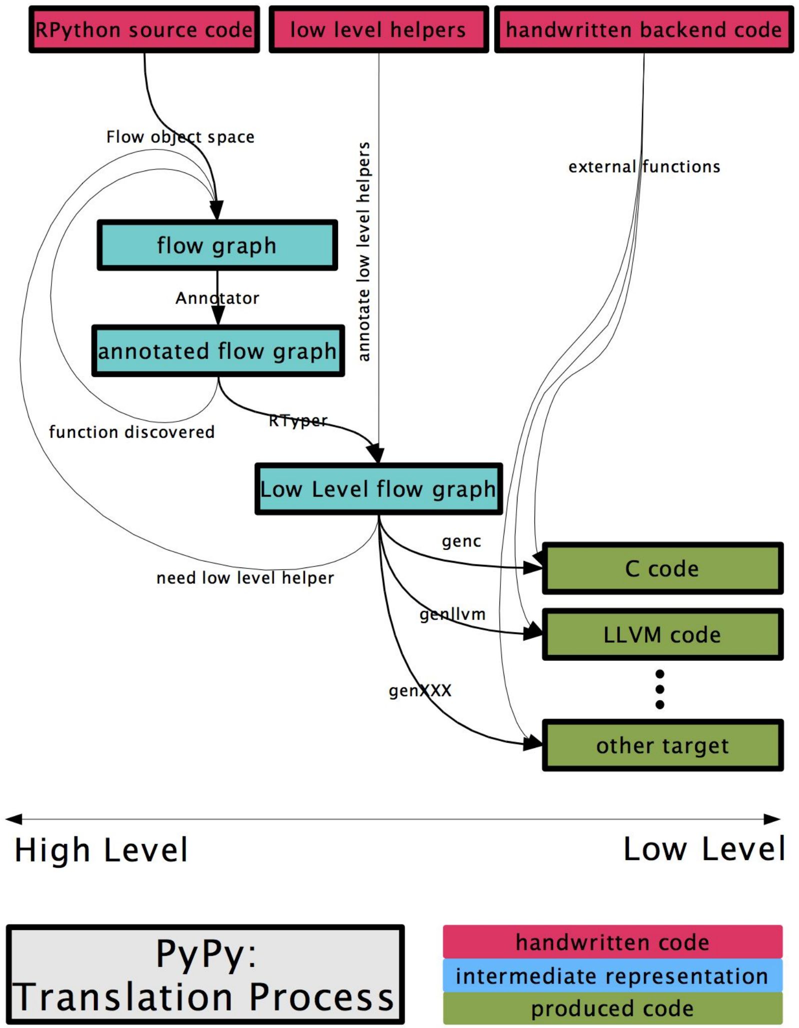
- Artefact
  - a back-end that can translate many essential features of RPython.
    - strings, classes, exceptions etc.
    - small scale programs such as GC Bench.
  - a launcher to run the MuPy code bundle.
- Assistance in Mu research
  - encountered and raised issues, stretched the design of Mu.
  - motivated many additional features
    - Heap Allocation Initialisation Language (HAIL).
    - Mu Native Interface (MuNI).

Language of Choice -- RPython

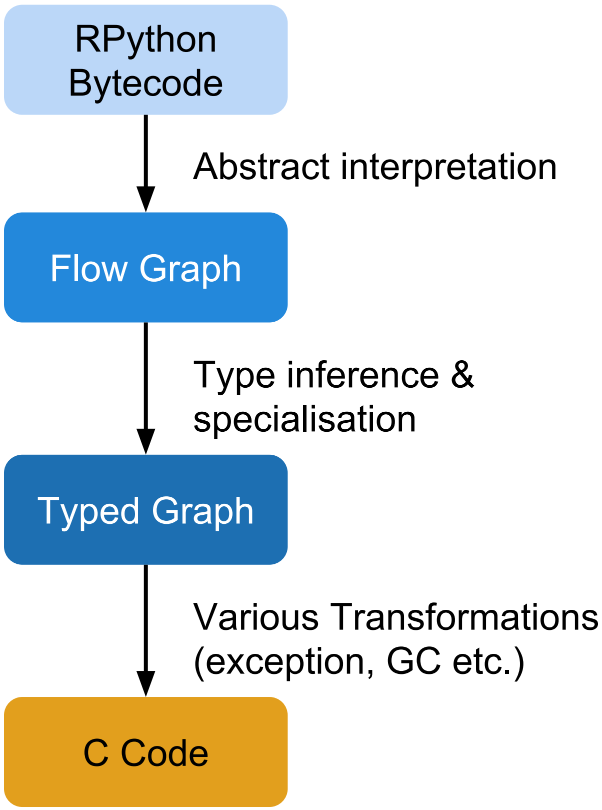
- Restricted Python (RPython)
  - a compiler framework for implementing interpreters of managed languages.
  - generates a meta-tracing JIT, and handles GC.
- Strategically critical
  - used to produce high performance implementation of other languages.
    - (Python, PHP, Erlang, JavaScript, Haskell etc.)
- Objective
  - a Mu back-end and language client for RPython.

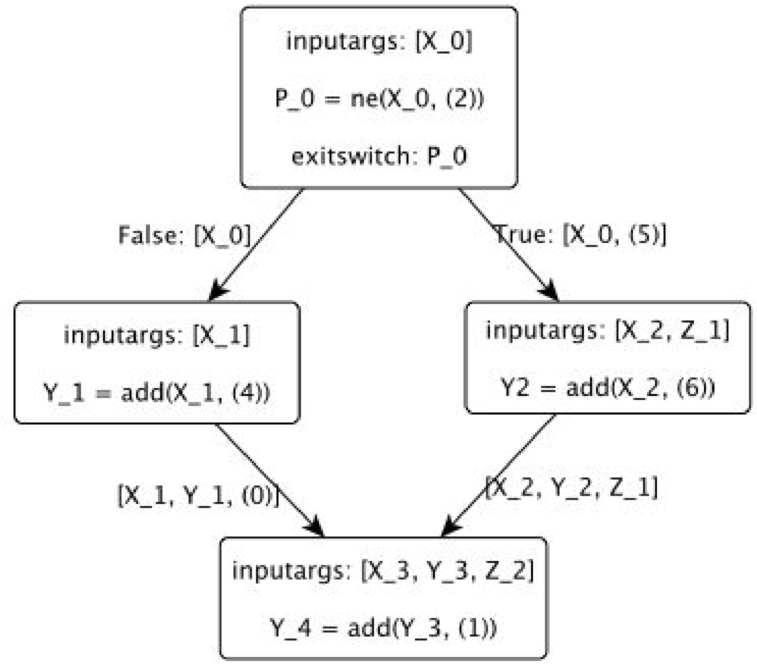


Overview of Translation Process

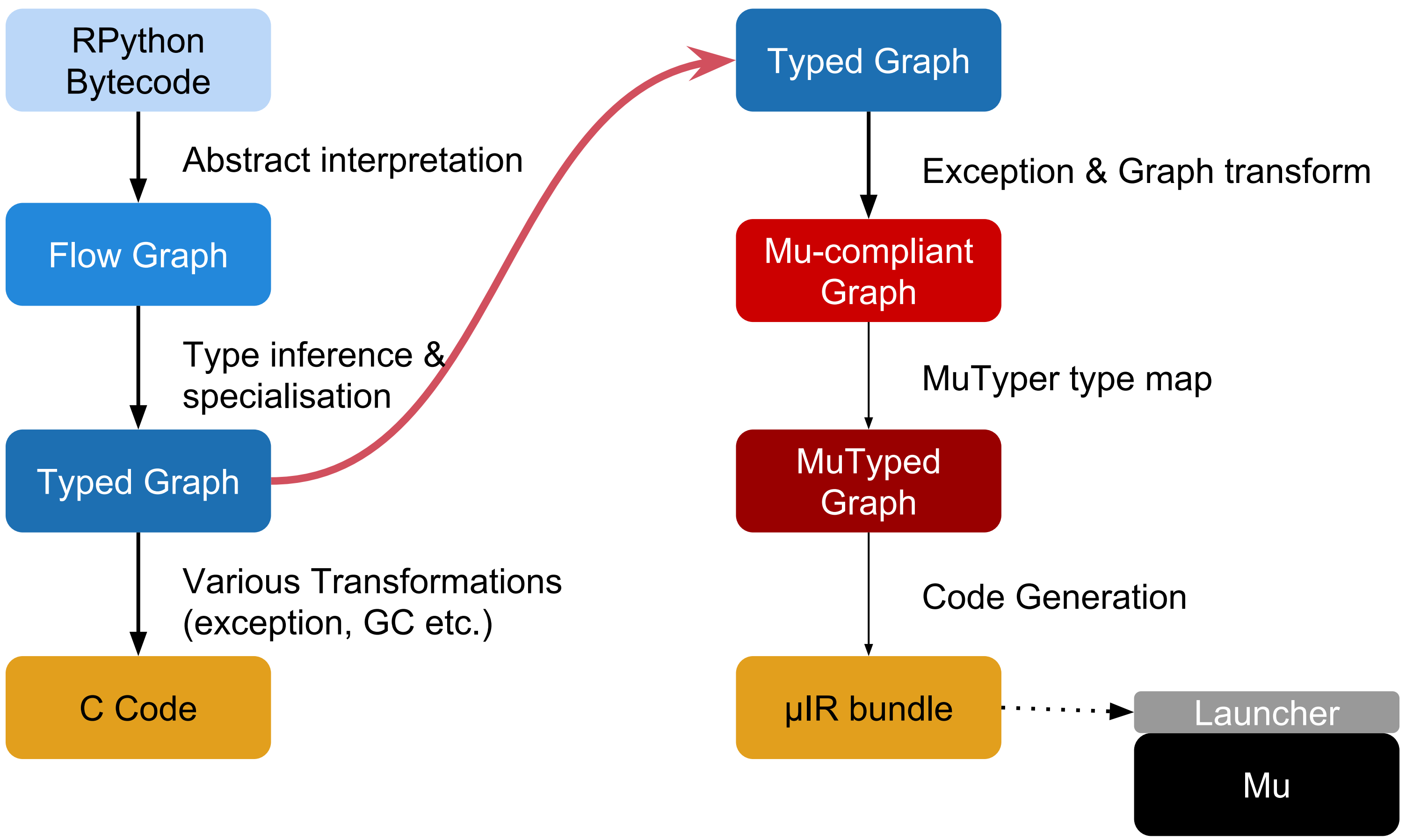


Overview of Translation Process

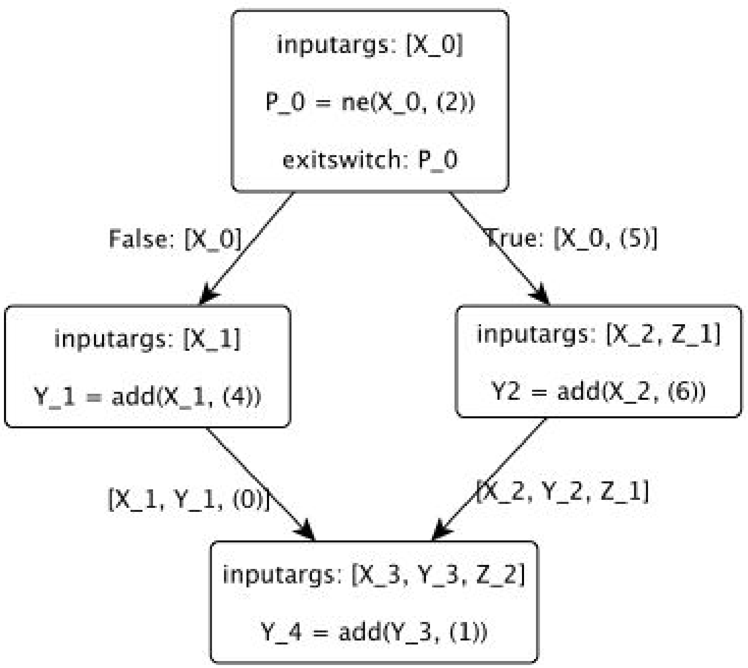




Overview of Translation Process



Overview of Translation Process




```
.funcdef @f VERSION @f_v1 <@f_sig> (%x0) {
  %blk0:
      %p0 = NE <@i64> %x0 @i64_2
      BRANCH2 %blk2 %blk1
  %blk1:
      %y1 = ADD <@i64> %x0 @i64_4
      BRANCH %blk3
  %blk2:
      %y2 = ADD <@i64> %x0 @i64_6
      BRANCH %blk3
  %blk3:
      %y3 = PHI {%blk1: %y1, %blk2: %y2}
      %z2 = PHI {%blk1: @i64_0, %blk2: @i64_5}
      %y4 = ADD <@i64> %y3 @i64_1
      ...
}
```

Graph Transform

- Goal: Transform the structure of the graph to be in compliant to Mu.
- Necessary RPython back-end optimisations:
  - no-op removal
  - exception raising operations to function call
  - SSI to SSA conversion
- Tasks for MuPy
  - graph cleaning
  - adding transitional blocks
  - exception transform

Graph Transform -- Exception Transform

- Mu supports exception handling
  - execution interruption
  - throw and catch exception objects (`THROW`, `LANDINGPAD`);
  - `EXC` clause for instructions;
  - efficient stack unwinding.
- Transform strategy:
  - pack exception type and value into a single heap object and throw it;
  - define `EXC` clause and catching block;
  - use `ll_issubclass` to check matching exception type.

MuTyper

- Goal: Maps the type system (LLTS -> MuTS)
  - Analogous to RPython Typer (RTyper)
  - Tasks:
    - maps the types of variables and constants in the graph;
    - maps the constant values in the graph;
    - converts global heap objects ‘constants’ to global cells;
    - maps the operations to Mu instructions.

MuTyper -- Heap Object Initialisation

- Not well supported by Mu (used to)
- 2 approaches:
  - compiler generates bundle entry initialisation routine.
  - specify heap object type and value before launch.
    - Motivated the HAIL language in recent Mu spec.
    - A new path to be explored in future.

```
(<* struct StdOutBuffer { super=..., inst_linebuf=... }>)
```

MuTyper -- Mapping Instructions

- Only convert numeric and pointer operations
    - Ignore GC and JIT
- Address operations
    - raw memory access and compiler intrinsics
      `raw_memcopy() -> memcpy()`
    - behavioural imitation has significant performance cost.
        - motivated MuNI in recent spec.

Launcher

- Part of the MuPy language client
- Load and run the bundle code
  - initialise the command-line arguments object (list of strings) in the heap.
  - sets up the trap handler
    - used for print output
  - various other initialisations
    - libraries
    - part of future work

Summary

- Milestones achieved in developing a Mu back-end for RPython
    - can correctly translate many essential features (class, exceptions, arithmetics etc.);
    - can translate small scale programs (GC Bench).
- Motivated the research of Mu and enriched its scope.
    - HAIL
    - MuNI

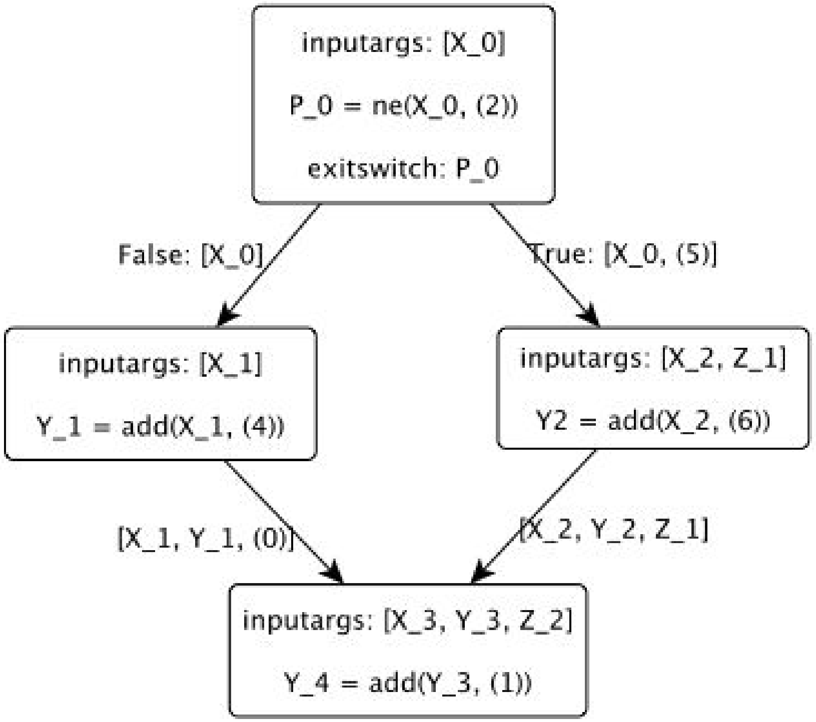
Future Goals

- Adapt to the most recent Mu spec
  - adopt HAIL and MuNI in translation
  - there has been significant changes in Mu spec.
- Fully port RPython
  - RPython standard library
- Test target: RPython Simple Object Machine (SOM).

Thank You!

Graph Transform -- Input

- RPython Control Flow Graph (CFG) representation
  - function as graphs.
  - blocks, operations & links.
  - Single Static Information (SSI) variables and constants.
    - Extension to SSA form.



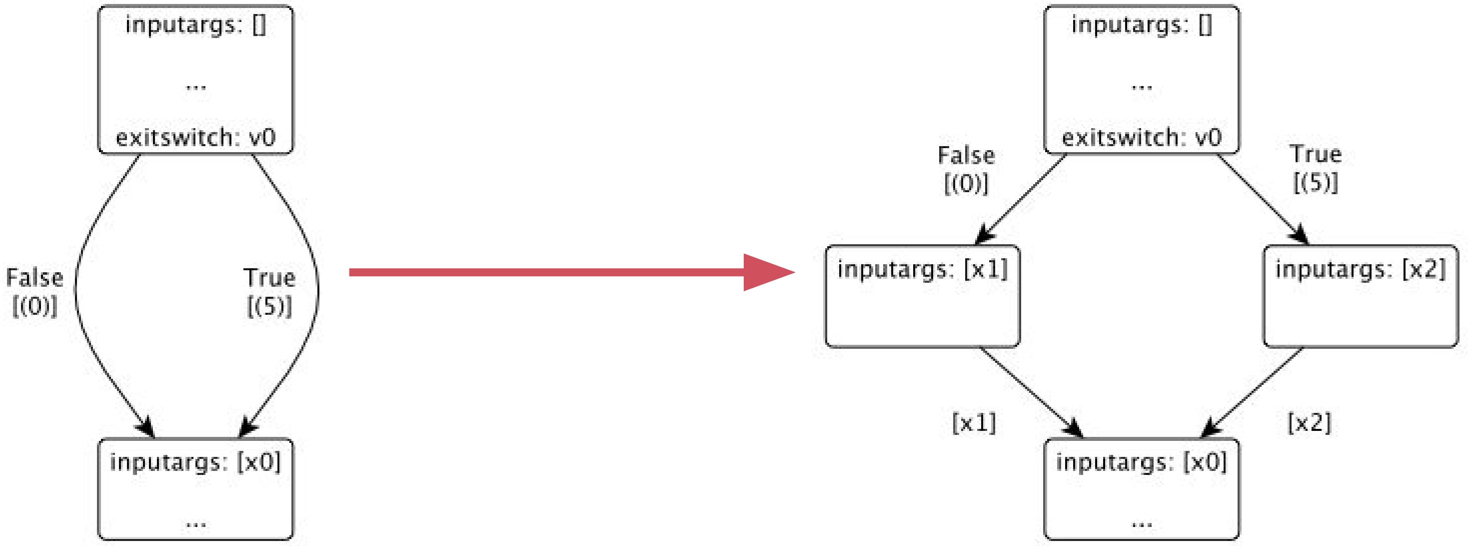
Graph Transform -- Desired Mu IR

- Similar structure
  - Functions contains instruction blocks
- Differences:
  - Explicit branching instructions
  - SSA instead of SSI
    - Can be handled by RPython back-end optimisations

```
.funcdef @f VERSION @f_v1 <@f_sig> (%x0) {
  %blk0:
      %p0 = NE <@i64> %x0 @i64_2
      BRANCH2 %blk2 %blk1
  %blk1:
      %y1 = ADD <@i64> %x0 @i64_4
      BRANCH %blk3
  %blk2:
      %y2 = ADD <@i64> %x0 @i64_6
  %blk3:
      %y3 = PHI {%blk1: %y1, %blk2: %y2}
      %z2 = PHI {%blk1: @i64_0, %blk2: @i64_5}
      %y4 = ADD <@i64> %y3 @i64_1
      ...
}
```

Graph Transform -- Transitional Blocks

- Two links carry the different arguments to the same block.
  - Problematic for `PHI` instruction.
  - Adding transitional blocks.



Graph Transform -- Exception Transform

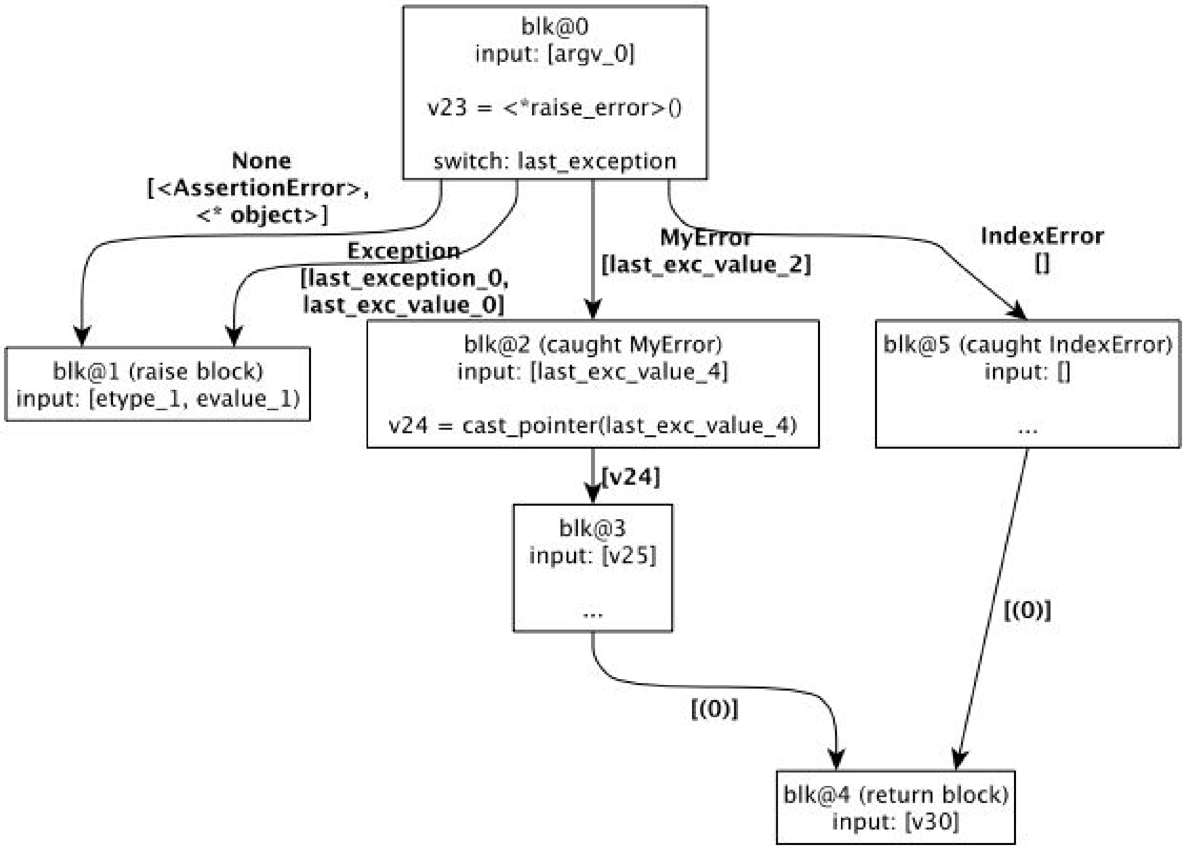
- Elements in RPython CFG
  - `last_exception` exit switch
  - Links having different Exception classes as exit cases.
  - Each graph has a unique exception raise block.
- Intended to be further specialised into concrete exception mechanisms.

Graph Transform -- Exception Transform

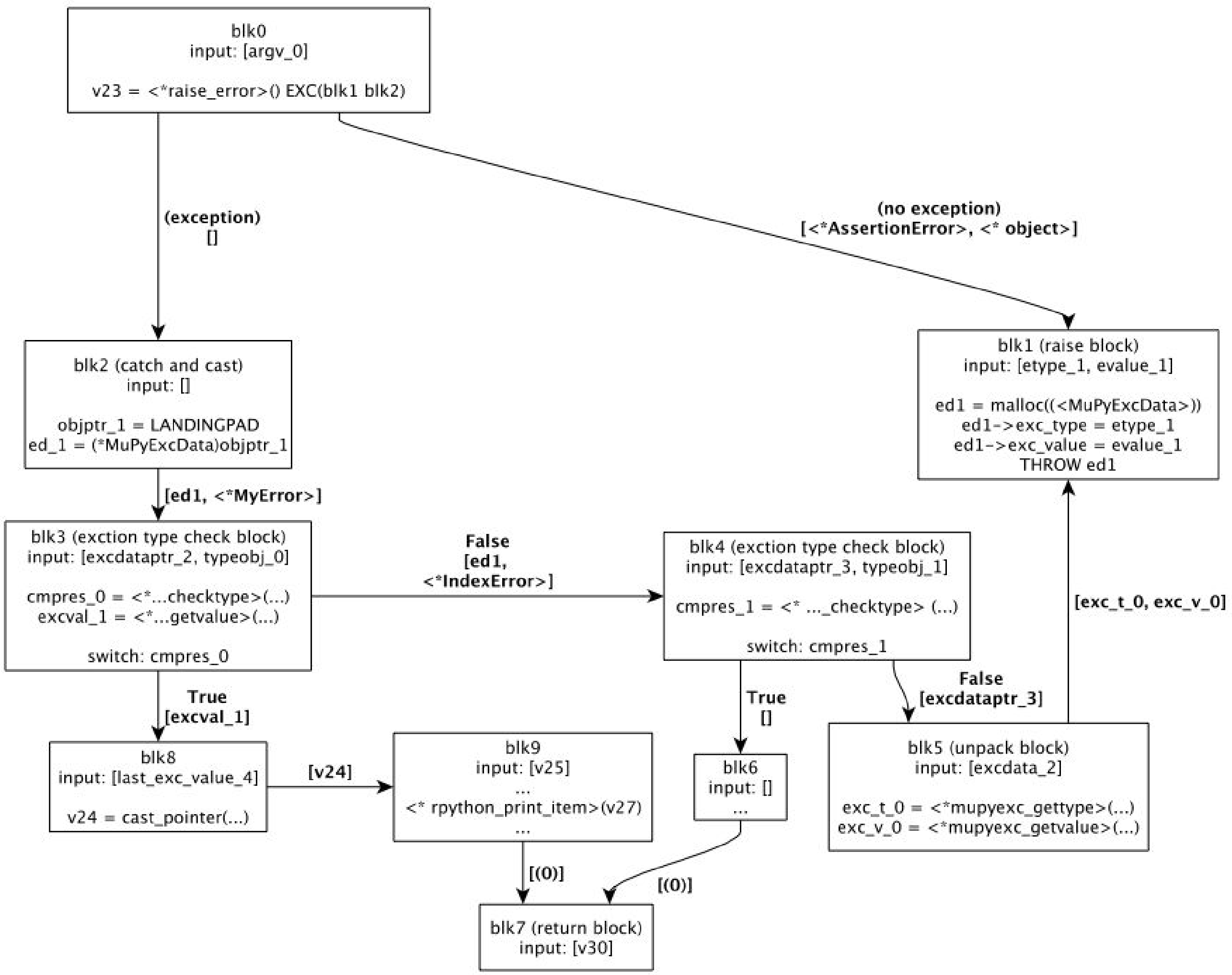
```
class MyError(Exception):
      def __init__(self, msg):
      self.message = msg

try:
      raise_error_1()
except MyError as e:
      print e.message
except IndexError:
      print "Caught"

def raise_error_1():
      raise MyError("1st msg")
```



Graph Transform -- Exception Transform



MuTyper -- Type Map

LLTS	MuTS
Signed	int<64>
Unsigned	int<64>
Char	int<8>
Bool	int<1>
Void	void
Struct	struct
FixedSizeArray	array
Array<T>	hybrid<int<64> T>
Ptr<T>	ref<T>
Ptr<FuncType>	funcref
Address	ptr

MuTyper -- Constant to Global Cells

- Initialised global heap objects as `Constants` in the graph
  - Does not correspond to constants (values) in Mu, but global cells (global memory).
- Strategy:
  - replace these elements in the graph with values loaded from Mu global cells.

Heap Object Initialisation Routine

- Implementation
  - "Loading constants as variables from global cell" strategy
  - Create an `__init__` function and fill it with initialisation code.
  - Insert a call to program entry point, and a `thread_exit` instruction.

```
.funcdef @__init__ VERSION @__init___v1 <@__init___sig> ()
{
%blk_0:
  %obj_3 = NEWHYBRID <@hyb_rpy_string_hdr_i8 @int_64> @i64_10
  %obj_4 = GETIREF <@hyb_rpy_string_hdr_i8> %obj_3
  %obj_5 = GETFIXEDPARTIREF <@hyb_rpy_string_hdr_i8> %obj_4
  %obj_6 = GETFIELDIREF <@stt_rpy_string_hdr 0> %obj_5
  STORE  <@int_64> %obj_6 @i64_-1 // Hash code field
  %obj_7 = GETFIELDIREF <@stt_rpy_string_hdr 1> %obj_5
  STORE <@int_64> %obj_7 @i64_10  // Length field
  %obj_8 = GETVARPARTIREF <@hyb_rpy_string_hdr_i8> %obj_4
  %obj_9 = SHIFTIREF <@int_8 @int_64> %obj_8 @i64_0
  STORE <@int_8> %obj_9 @i8_76 // 'L'
  %obj_10 = SHIFTIREF <@int_8 @int_64> %obj_8 @i64_1
  STORE <@int_8> %obj_10 @i8_97 // 'a'
  ...
  CALL <@9_main_sig> @9_main ()
  COMMONINST @uvm.thread_exit
}
```

Other Progress

- Print statements
  - Using a ‘magic’ function call that is translated to a `TRAP` instruction.
  - On the client side, handle the trap and print on terminal.

```
def rpython_print_newline():
  buf = stdoutbuffer.linebuf;
  s = buf + '\n';

  from ... import uir_fake_print;
  uir_fake_print(s);
```

```
.funcdef @117_rpython_print_newline VERSION ... <...> ()
{
%blk_0:
  ...
  %trap = TRAP <@void>  KEEPALIVE(%s_1)
  ...
}
```