Semantics for $\mathbf{DYNJIT}^{\mathbf{source}} \rightarrow \mathbf{DYNJIT}^{\mathbf{target}}$

September 14, 2020

CoreCPY can be always converted to $\textbf{DYNJIT}^{\textbf{source}}$ in a straightforward approach because, **the stack usage of the bytecode generated by valid Python source code** is, **finite**.

Regarding the perspective of partial evaluation, we know **CoreCPY** is a flowchart language with a finite stack(a stack S_N has a size$N$).

By assigning the abstract value **Dyn** s to a named variable s, and assigning the abstract value **Slot** i to a datum stored in the i-th element from the bottom of stack(BOS) if initialized, we will have finite configurations:

$$\textbf{Configurations} \subset \textbf{Labels} \times [\textbf{Dyn}\ s_1, \textbf{Dyn}\ s_2, \cdots, \textbf{Dyn}\ s_k]^k \times \mathcal{P}(\{\textbf{Slot}\ 1, \cdots, \textbf{Slot}\ N\})$$

where $\mathcal{P}$ means getting the power set, k is the number of named variables not matter what it is(cell, free, etc.) and, N, as we've mentioned above, is the size of the stack, which is finite for any given code.

Note that

$$\left|\textbf{Labels} \times [\textbf{Dyn}\ s_1, \textbf{Dyn}\ s_2, \cdots, \textbf{Dyn}\ s_k]^k \times \mathcal{P}(\{\textbf{Slot}\ 1, \cdots, \textbf{Slot}\ N\}\right|$$

is simply finite, this leads to finite **Configurations**, and also, finite basic blocks when translating the use of stack to unnamed registers.

DYNJIT$^{\text{source}}$ and DYNJIT$^{\text{target}}$

DYNJIT$^{\text{source}}$ Syntax

$$
\begin{array}{rcl}
\langle\text{repr}\rangle & ::= & S\ \textbf{constant} \\
& & |\ D\ \textbf{var} \\
\langle\text{instr}\rangle & ::= & \langle\text{var}\rangle = \textit{call}\ \langle\text{repr}\rangle\ (\langle\text{repr}\rangle\ *) \\
& & |\ \langle\text{var}\rangle = \langle\text{repr}\rangle \\
& & |\ \textit{return}\ \langle\text{repr}\rangle \\
& & |\ \textit{goto}\ \textbf{label} \\
& & |\ \textit{if}\ \langle\text{repr}\rangle\ \textit{goto}\ \textbf{label}\ \textit{goto}\ \textbf{label} \\
\langle\text{move}\rangle & ::= & \langle\text{var}\rangle \leftarrow \langle\text{var}\rangle \\
\langle\Phi\rangle & ::= & \textbf{label}\ \textit{:}\ \langle\text{move}\rangle\ * \\
\langle\text{basicblock}\rangle & ::= & \textit{label}\ \textbf{label}\ \textit{:}\ \Phi\ \textit{[}\ \langle\Phi\rangle\ *\ \textit{]}\ \langle\text{instr}\rangle\ + \\
\langle\text{entryblock}\rangle & ::= & \textit{label entry :}\ \Phi\ \textit{[}\ \langle\Phi\rangle\ *\ \textit{]}\ \langle\text{instr}\rangle\ + \\
\langle\text{basicblocks}\rangle & ::= & \langle\text{entryblock}\rangle\ \langle\text{basicblock}\rangle\ *
\end{array}
$$

DYNJIT$^{\text{target}}$ Syntax

$$
\begin{array}{rcl}
\langle\text{absvalue}\rangle & ::= & (\langle\text{repr}\rangle\ ,\ \langle\text{type}\rangle) \\
\langle\text{expr}\rangle & ::= & \langle\text{absvalue}\rangle \\
& & |\ \textit{call}\ \langle\text{expr}\rangle\ (\langle\text{expr}\rangle\ *) \\
\langle\text{stmt}\rangle & ::= & \langle\text{var}\rangle = \langle\text{expr}\rangle \\
& & |\ \textit{goto}\ \textbf{label} \\
& & |\ \textit{return}\ \langle\text{expr}\rangle \\
& & |\ \textit{if}\ \langle\text{expr}\rangle\ \textit{goto}\ \textbf{label}\ \textit{goto}\ \textbf{label} \\
& & |\ \textit{do}\ \langle\text{expr}\rangle \\
& & |\ \textit{label}\ \textbf{label} \\
& & |\ \{\ \langle\text{stmts}\rangle\ \} \\
& & |\ \textit{switch}\ \langle\text{expr}\rangle\ \langle\text{case}\rangle\ * \\
\langle\text{case}\rangle & ::= & |\ \langle\text{type}\rangle \rightarrow \langle\text{stmt}\rangle \\
\langle\text{stmts}\rangle & ::= & \langle\text{stmt}\rangle\ *
\end{array}
$$

Types

```
type fptr = int
type meth = int
type t =
| ⊥
| ⊤
| NoneT
| FPtrT   of fptr
| MethT   of meth
| NomT    of string * (string , t) map
| TupleT  of t list
| TypeT   of t list
| CellT   of t list
| UnionT  of t list
| IntrinT of intrinsic
```

A method is a specialised function.

Information of a function, including its definition body (a *basicblocks* in $\mathbf{DYNJIT}^{\mathbf{source}}$), variables, arugument information, will be able to lookup with **fptr**.

Information of a method, including its definition body, (a *stmts* in $\mathbf{DYNJIT}^{\mathbf{target}}$), specialised arugument information, will be able to lookup with **meth**.

Particularly, for the sake of convenience, we use $(t_1|t_2|\cdots|t_n)$ for $UnionT\ [t_1, \cdots, t_n]$. In the implementation part, we always lift up union types to the top level, hence there's no nested union type $UnionT\ [UnionT \cdots]$.

Also we use *bool* for boolean prmitive type $NomT$"$boolean$" $\cdots$, as well as int, $float$, etc.

Intrinsics and **Constants**

```
type intrinsic =
| isinstance
| typeof
| upcast
| downcast

type constant =
| NoneC
| UndefC
| TypeL    of t
| FPtrC    of fptr
| MethC    of meth
| IntC     of int
| FloatC   of float
| StrC     of string
| TupleC   of const list
| IntrinsicC  of intrinsic
```

The Operational Semantics for *source* $\Rightarrow$ *target*

We need the following auxiliary symbols for giving the semantics

- r, $r_1, r_2, \cdots$ are *repr*.
- t, $t_1, t_2, \cdots$ are *type*.
- a, $a_1, a_2, \cdots$ are *absvalue*.
- c, $c_1, c_2, \cdots$ are *const*.
- n, $n_1, n_2, \cdots$ are variable names, or *var*.
- l, $l_1, l_2, \cdots$ are *label*.
- L, $L_1, L_2, \cdots$ are a suite of *stmt* or *instr*, depending on the context.
- σ : an immutable array σ of variables' abstract values.
  $\sigma[n := a]$ returns a new array σ' having the same length. We use $\sigma(n)$ to access a pair of variable n's type representation and data representation.
- F: Given the index of a function, i, $F(i)$ gives the total information of the function.
- M: Given the index of a method, i, $M(i)$ gives the total information of the method.
- ct: Given the language for a constant, l, $ct(l)$ returns the constant value.
- $\preceq$: The relationship "more specific than", for instance, $t_1 \preceq t_1|t_2$

- $\emptyset$ indicates that no residual program is generated.
- $\cap_{type}$: type intersection.
  $t_1 \cap_{type} t_2$ gives the most general type t_{12} such that $t_{12} \preceq t_1 \wedge t_{12} \preceq t_2$.

Expression

$$\frac{}{\sigma \vdash_{expr} S\ c \Rightarrow (S\ a, ct(c))} \quad \frac{}{\sigma \vdash_{expr} D\ n \Rightarrow \sigma(n)}$$

Statement

$$\frac{\sigma \vdash_{expr} r \Rightarrow (S\ true, bool) \quad \sigma \vdash_{block} l_1 \Rightarrow l_1'}{\sigma \vdash_{stmt}\ if\ r\ goto\ l_1\ goto\ l_2\ \Rightarrow (goto\ l_1', \sigma)}\ (\textbf{CONST–IF–TRUE})$$

$$\frac{\sigma \vdash_{expr} r \Rightarrow (S\ false, bool) \quad \sigma \vdash_{block} l_2 \Rightarrow l_2'}{\sigma \vdash_{stmt}\ if\ r\ goto\ l_1\ goto\ l_2\ \Rightarrow (goto\ l_2', \sigma)}\ (\textbf{CONST–IF–FALSE})$$

$$\frac{\sigma \vdash_{expr} r \Rightarrow (D\ n_2, t) \quad a_2 = (D\ n_2, t) \quad a_1 = (D\ n_1, t)}{\sigma \vdash_{stmt}\ n_1 = r\ \Rightarrow (n_1 = a_2, \sigma[n_1 := a_1])}\ (\textbf{ASS–VAR})$$

$$\frac{\sigma \vdash_{expr} r \Rightarrow (S\ c, t) \quad a = (S\ c, t)}{\sigma \vdash_{stmt}\ n = r\ \Rightarrow (\emptyset\,, \sigma[n := a])}\ (\textbf{ASS–CONST–PROP})$$

$$\frac{\begin{array}{c}\sigma \vdash_{expr} r_1 \Rightarrow (r_1^*, t_1) \quad \sigma \vdash_{expr} r_2 \Rightarrow (r_2^*, TypeT\ t_2) \\ boolean = t_1 \preceq t_2 \neq No\,Partial\,Order\ Error\end{array}}{\sigma \vdash_{stmt}\ n = call\ isinstance(r_1, r_2)\ \Rightarrow (\emptyset\,, \sigma[n := (boolean, bool)])}(\textbf{STATIC–SPEC–INST})$$

$$\frac{\begin{array}{c}\sigma \vdash_{expr} r_1 \Rightarrow (D\ n', \top) \quad \sigma \vdash_{expr} r_2 \Rightarrow (r_2^*, TypeT\ t) \quad a_1 = (D\ n', \top) \quad a_2 = (r_2^*, TypeT\ t) \\ a_n = (D\ n, bool) \quad \sigma[n := (S\ true, bool)] \vdash_{stmts} L \Rightarrow L' \quad \sigma[n := (S\ false, bool)] \vdash_{stmts} L \Rightarrow L''\end{array}}{\sigma \vdash_{stmts}\ n = call\ isinstance(r_1, r_2); L\ \Rightarrow n = call\ isinstance(a_1, a_2); if\ a_n\ then\ L'\ else\ L'}$$

Statements

$$\frac{}{\sigma \vdash_{stmts} \emptyset \Rightarrow \emptyset}$$

$$\frac{\sigma \vdash_{stmt} instr \Rightarrow (stmt, \sigma') \quad \sigma' \vdash_{stmts} L \Rightarrow L'}{\sigma \vdash_{stmts} instr; L \Rightarrow stmt; L'}$$

Block

$$\frac{l' = \textbf{Visited}(\sigma, l)}{\sigma \vdash_{block} l \Rightarrow l'}$$

$$\frac{\begin{array}{c}\emptyset = \textbf{Visited}(\sigma, l) \quad L = \textbf{which\_block}(l) \quad l' = gensym() \\ \textbf{Visited}(\sigma, l) = l' \quad \sigma \vdash_{stmt} \Rightarrow L' \quad \textbf{Block}(l') := L'\end{array}}{\sigma \vdash_{block} l \Rightarrow l'}$$

The second rule involves more details. WIP.