Python: The Even Fuller Monty

Version 5.90.0.4

August 29, 2013

```
E ::= []
    | (fetch E) | (set! E e) | (set! val E)
    | (alloc E)
    | (module E e)
    | ⟨E, mval⟩
    | ref := E | x := E
    | E[e := e] | v[E := e] | v[v := E]
    | E e
    | if E e e
    | let x = E in e
    | list⟨E, [e ...]⟩ | list⟨val, Es⟩
    | tuple⟨E, (e ...)⟩ | tuple⟨val, Es⟩
    | set⟨E, (e ...)⟩ | set⟨val, Es⟩
    | E[e] | val[E]
    | (builtin-prim op Es)
    | (raise E)
    | (return E)
    | (tryexcept E x e e) | (tryfinally E e)
    | (loop E e) | (frame E)
    | E (e ...) | val Es
    | E (e ...)*e | val Es*e
    | val (val ...)*E
    | (construct-module E) | (in-module E ε)
Es ::= (val ... E e ...)
```

Figure 1: Evaluation contexts

Appendix 1: The Rest of λ_π

The figures in this section show the rest of the λ_π semantics. We proceed to briefly present each of them in turn.

1 Contexts and Control Flow

Figures 1, 2, 3, and 4 show the different *contexts* we use to capture left-to-right, eager order of operations in λ_π. E is the usual *evaluation context* that enforces left-to-right, eager evaluation of expressions. T is a context for the first expression of a `tryexcept` block, used to catch instances of `raise`. Similarly, H defines contexts for loops, detecting `continue` and `break`, and R defines contexts for `return` statements inside functions. Each interacts with a few expression forms to handle non-local control flow.

```
T ::= []
    | (fetch T) | (set! T e) | (set! val T)
    | (alloc T)
    | ⟨T, mval⟩
    | ref := T | x := T
    | T[e := e] | v[T := e] | v[v := T]
    | T e
    | if T e e
    | let x = T in e
    | list⟨T, e⟩ | list⟨val, Ts⟩
    | tuple⟨T, e⟩ | tuple⟨val, Ts⟩
    | set⟨T, e⟩ | set⟨val, Ts⟩
    | T[e] | val[T]
    | (builtin-prim op Ts)
    | (raise T)
    | (loop T e) | (frame T)
    | T (e ...) | val Ts
    | T (e ...)*e | val Ts*e
    | val (val ...)*T
    | (construct-module T)
Ts ::= (val ... T e ...)
```

Figure 2: Contexts for try-except

```
H ::= []
    | (fetch H) | (set! H e) | (set! val H)
    | (alloc H)
    | ⟨H, mval⟩
    | ref := H | x := H
    | H[e := e] | v[H := e] | v[v := H]
    | H e
    | if H e e
    | let x = H in e
    | list⟨H, e⟩ | list⟨val, Hs⟩
    | tuple⟨H, e⟩ | tuple⟨val, Hs⟩
    | set⟨H, e⟩ | set⟨val, Hs⟩
    | H[e] | val[H]
    | (builtin-prim op Hs)
    | (raise H)
    | (tryexcept H x e e)
    | H (e ...) | val Hs
    | H (e ...)*e | val Hs*e
    | val (val ...)*H
    | (construct-module E)
Hs ::= (val ... H e ...)
```

Figure 3: Contexts for loops

```
R ::= []
    | (fetch R) | (set! R e) | (set! val R)
    | (alloc R)
    | ⟨R, mval⟩
    | ref := R | x := R
    | R[e := e] | v[R := e] | v[v := R]
    | R e
    | if R e e
    | let x = R in e
    | list⟨R, e⟩ | list⟨val, Rs⟩
    | tuple⟨R, e⟩ | tuple⟨val, Rs⟩
    | set⟨R, e⟩ | set⟨val, Rs⟩
    | R[e] | val[R]
    | (builtin-prim op Rs)
    | (raise R)
    | (loop R e)
    | (tryexcept R x e e)
    | R (e ...) | val Rs
    | R (e ...)*e | val Rs*e
    | val (val ...)*R
    | (construct-module E)
Rs ::= (val ... R e ...)
```

Figure 4: Contexts for return statements

Figure 5 shows how these contexts interact with expressions. For example, in first few rules, we see how we handle `break` and `continue` in `while` statements. When `while` takes a step, it yields a `loop` form that serves as the marker for where internal `break` and `continue` statements should collapse to. It is for this reason that H does *not* descend into nested `loop` forms; it would be incorrect for a `break` in a nested loop to `break` the outer loop.

One interesting feature of `while` and `tryexcept` in Python is that they have distinguished "else" clauses. For `while` loops, these else clauses run when the condition is `False`, but *not* when the loop is broken out of. For `tryexcept`, the else clause is only visited if *no* exception was thrown while evaluating the body. This is reflected in E-TryDone and the else branch of the `if` statement produced by E-While.

We handle one feature of Python's exception raising imperfectly. If a programmer uses `raise` without providing an explicit value to throw, *the exception bound in the most recent active catch block* is thrown instead. We have a limited solution that involves raising a special designated "reraise" value, but this fails to capture some subtle behavior of nested catch blocks. We believe a more sophisticated desugaring that uses a global stack to keep track of entrances and exits to catch blocks will work, but have yet to verify it. We still pass a number of tests that use `raise` with no argument.

2 Mutation

There are *three* separate mutability operators in λ_π, `(set! e e)`, which mutates the value stored in a reference value, `e := e`, which mutates variables, and `(set-field e e e)`, which updates and adds fields to objects.

Figure 6 shows the several operators that allocate and manipulate references in different ways. We briefly categorize the purpose for each type of mutation here:

- We use `(set! e e)`, `(fetch e)` and `(alloc e)` to handle the update and creation of objects via the δ function, which reads but does not modify the store. Thus, even the lowly `+` operation needs to have its result re-allocated, since programmers only see references to numbers, not object values themselves. We leave the pieces of object values immutable and use this general strategy for updating them, rather than defining separate mutability for each type (e.g., lists).
- We use `e := e` for assignment to both local and global variables. We discuss global variables more in the next section. Local variables are handled at binding time by allocating references and substituting the new references wherever the variable appears. Local variable accesses and assignments thus work over references directly, since the variables have been substituted away by the time the actual assignment or access is reached. Note also that E-AssignLocal can override potential ☠ store entries.

$$(\texttt{while}\ e_1\ e_2\ e_3) \qquad \text{[E-While]}$$
$$\Longrightarrow\ \texttt{if}\ e_1\ (\texttt{loop}\ e_2\ (\texttt{while}\ e_1\ e_2\ e_3))\ e_3$$

$$(\texttt{loop}\ H[\texttt{continue}]\ e) \Longrightarrow e \qquad \text{[E-LoopContinue]}$$

$$(\texttt{loop}\ H[\texttt{break}]\ e) \Longrightarrow \texttt{vnone} \qquad \text{[E-LoopBreak]}$$

$$(\texttt{loop}\ \mathit{val}\ e) \Longrightarrow e \qquad \text{[E-LoopNext]}$$

$$(\texttt{tryexcept}\ \mathit{val}\ x\ e_{catch}\ e_{else}) \Longrightarrow e_{else} \qquad \text{[E-TryDone]}$$

$$(\texttt{tryexcept}\ T[(\texttt{raise}\ \mathit{val})]\ x\ e_{catch}\ e_{else}) \qquad \text{[E-TryCatch]}$$
$$\Longrightarrow\ \texttt{let}\ \ x = \mathit{val}\ \ \texttt{in}\ e_{catch}$$

$$(\texttt{frame}\ R[(\texttt{return}\ \mathit{val})]) \Longrightarrow \mathit{val} \qquad \text{[E-Return]}$$

$$(\texttt{frame}\ \mathit{val}) \Longrightarrow \mathit{val} \qquad \text{[E-FramePop]}$$

$$(\texttt{tryfinally}\ R[(\texttt{return}\ \mathit{val})]\ e) \qquad \text{[E-FinallyReturn]}$$
$$\Longrightarrow e\ (\texttt{return}\ \mathit{val})$$

$$(\texttt{tryfinally}\ H[\texttt{break}]\ e) \qquad \text{[E-FinallyBreak]}$$
$$\Longrightarrow e\ \texttt{break}$$

$$(\texttt{tryfinally}\ T[(\texttt{raise}\ \mathit{val})]\ e) \qquad \text{[E-FinallyRaise]}$$
$$\Longrightarrow e\ (\texttt{raise}\ \mathit{val})$$

$$(\texttt{tryfinally}\ H[\texttt{continue}]\ e) \qquad \text{[E-FinallyContinue]}$$
$$\Longrightarrow e\ \texttt{continue}$$

$$(E[\ \texttt{if}\ \mathit{val}\ e_1\ e_2\ ]\ \varepsilon\ \Sigma) \qquad \text{[E-IfTrue]}$$
$$\longrightarrow (E[e_1]\ \varepsilon\ \Sigma)$$
$$\texttt{where (truthy?}\ \mathit{val}\ \Sigma)$$

$$(E[\ \texttt{if}\ \mathit{val}\ e_1\ e_2\ ]\ \varepsilon\ \Sigma) \qquad \text{[E-IfFalse]}$$
$$\longrightarrow (E[e_2]\ \varepsilon\ \Sigma)$$
$$\texttt{where (not (truthy?}\ \mathit{val}\ \Sigma))$$

$$\mathit{val}\ e \Longrightarrow e \qquad \text{[E-Seq]}$$

Figure 5: Control flow reductions

$(E[@ref_{fun}\ \ (val\ \ldots)]\ \varepsilon\ \Sigma)$ [E-App]

$\longrightarrow (E[(\text{frame subst}[\![(x\ \ldots),\ (ref_{arg}\ \ldots),\ e]\!])]\ \varepsilon\ \Sigma_1)$

where $\langle any_c, \lambda(x\ \ldots)\ (\text{no-var}).e\ \ , any_{dict}\rangle = \Sigma(ref_{fun})$,
$(\text{equal? (length } (val\ \ldots))\ \ (\text{length } (x\ \ldots)))$,
$(\Sigma_1\ (ref_{arg}\ \ldots_1)) = \text{extend-store/list}[\![\Sigma,\ (val\ \ldots)]\!]$

$(E[\ \text{let}\ \ x = v\text{+undef}\ \ \text{in}\ e\ ]\ \varepsilon\ \Sigma)$ [E-LetLocal]

$\longrightarrow (E[[x/ref]e\ ]\ \varepsilon\ \Sigma_1)$

where $(\Sigma_1\ ref) = \text{alloc}(\Sigma, v\text{+undef})$

$(E[\ \text{let}\ \ x = v\text{+undef}\ \ \text{in}\ e\ ]\ \varepsilon\ \Sigma)$ [E-LetGlobal]

$\longrightarrow (E[e]\ \text{extend-env}[\![\varepsilon,\ x,\ ref]\!]\ \Sigma_1)$

where $(\Sigma_1\ ref) = \text{alloc}(\Sigma, v\text{+undef})$

$(E[ref]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma)$ [E-GetVar]

where $\Sigma = ((ref_1\ v\text{+undef}_1)\ \ldots\ (ref\ val)\ (ref_n\ v\text{+undef}_n)\ \ldots)$

$(E[ref]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma)$ [E-GetVar]

where $\Sigma = ((ref_1\ v\text{+undef}_1)\ \ldots\ (ref\ val)\ (ref_n\ v\text{+undef}_n)\ \ldots)$

$(E[ref\ :=\ val\ ]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma[ref{:=}val])$ [E-AssignLocal]

$(E[x\ :=\ val\ ]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma[ref{:=}val])$ [E-AssignGlobal]

where $\varepsilon = ((x_2\ ref_2)\ \ldots\ (x\ ref)\ (x_3\ ref_3)\ \ldots)$

$(E[(\text{alloc } val)]\ \varepsilon\ \Sigma) \longrightarrow (E[@ref_{new}\ ]\ \varepsilon\ \Sigma_1)$ [E-Alloc]

where $(\Sigma_1\ ref_{new}) = \text{alloc}(\Sigma, val)$

$(E[(\text{fetch } @ref\ )]\ \varepsilon\ \Sigma) \longrightarrow (E[\Sigma(ref)]\ \varepsilon\ \Sigma)$ [E-Fetch]

$(E[(\text{set! } @ref\ \ val)]\ \varepsilon\ \Sigma) \longrightarrow (E[val]\ \varepsilon\ \Sigma_1)$ [E-Set!]

where $\Sigma_1 = \Sigma[ref{:=}val]$

$(E[@ref_{obj}\ [@ref_{str}\ \ := val_1]]\ \varepsilon\ \Sigma) \longrightarrow (E[val_1]\ \varepsilon\ \Sigma_2)$ [E-SetFieldAdd]

where $\langle any_{cls1}, mval, \{string{:}ref, \ldots\}\rangle = \Sigma(ref_{obj})$,
$(\Sigma_1\ ref_{new}) = \text{alloc}(\Sigma, val_1)$,
$\langle any_{cls2}, string_1, any_{dict}\rangle = \Sigma(ref_{str})$,
$\Sigma_2 = \Sigma_1[ref_{obj}{:=}\langle any_{cls1}, mval, \{string_1{:}ref_{new}, string{:}ref, \ldots\}\rangle]$,
$(\text{not } (\text{member } string_1\ \ (string\ \ldots)))$

$(E[@ref_{obj}\ [@ref_{str}\ \ := val_1]]\ \varepsilon\ \Sigma)$ [E-SetFieldUpdate]

$\longrightarrow (E[val_1]\ \varepsilon\ \Sigma[ref_1{:=}val_1])$

where $\langle any_{cls1}, mval, \{string_2{:}ref_2, \ldots, string_1{:}ref_1, string_3{:}ref_3, \ldots\}\rangle = \Sigma(ref_{obj})$,
$\langle any_{cls2}, string_1, any_{dict}\rangle = \Sigma(ref_{str})$

$(E[(\text{delete } ref)]\ \varepsilon\ ((ref_1\ v_1)\ \ldots\ (ref\ v)\ (ref_n\ v_n)\ \ldots))$ [E-DeleteLocal]

$\longrightarrow (E[v]\ \varepsilon\ ((ref_1\ v_1)\ \ldots\ (ref\ \text{☠})\ (ref_n\ v_n)\ \ldots))$

$(E[(\text{delete } x)]\ ((x_1\ ref_1)\ \ldots\ (x\ ref)\ (x_n\ ref_n)\ \ldots)\ \Sigma)$ [E-DeleteGlobal]

$\longrightarrow (E[\Sigma(ref)]\ ((x_1\ ref_1)\ \ldots\ (x_n\ ref_n)\ \ldots)\ \Sigma)$

Figure 6: Various operations on mutable variables and values

$$
\begin{array}{l}
\texttt{(}E\texttt{[@}ref_{obj}\ \texttt{[@}ref_{str}\ \texttt{]]}\ \varepsilon\ \Sigma\texttt{)} \qquad \text{[E-GetField-Class/Id]} \\
\longrightarrow \texttt{(}E\texttt{[}val_{result}\texttt{]}\ \varepsilon\ \Sigma_{result}\texttt{)} \\
\quad \texttt{where}\ \langle any_{cls}, string, any_{dict} \rangle = \Sigma(ref_{str}), \\
\qquad \langle x_{cls}, mval, \{string_1 : ref_2, \ldots\} \rangle = \Sigma(ref_{obj}), \\
\qquad (\Sigma_{result}\ val_{result}) = \text{class-lookup}[\![@ref_{obj}\ ,\ \Sigma(\text{env-lookup}[\![\varepsilon,\ x_{cls}]\!]),\ string,\ \Sigma]\!], \\
\qquad \texttt{(not (member}\ string\ \ \texttt{(}string_1\ \ldots\texttt{)))} \\
\\
\qquad\quad \text{env-lookup}[\![\texttt{((}x_1\ ref_1\texttt{)}\ \ldots\ \texttt{(}x\ ref\texttt{)}\ \texttt{(}x_n\ ref_n\texttt{)}\ \ldots\texttt{),}\ x]\!] = ref
\end{array}
$$

Figure 7: Accessing fields on a class defined by an identifier

- We use `(set-field` e e e`)` to update and add fields to objects' dictionaries. We leave the fields of objects' dictionaries as references and not values to allow ourselves the ability to share references between object fields and variables. We maintain a strict separation in our current semantics, with the exception of modules, and we expect that we'll continue to need it in the future to handle patterns of `exec`.

Finally, we show the E-Delete operators, which allow a Python program to revert the value in the store at a particular location back to ☠, or to remove a global binding.

3 Global Scope

While local variables are handled directly via substitution, we handle global scope with an explicit environment ε that follows the computation. We do this for two main reasons. First, because global scope in Python is truly dynamic in ways that local scope is not (`exec` can modify global scope), and we want to be open to those possibilities in the future. Second, and more implementation-specific, we use global scope to bootstrap some mutual dependencies in the object system, and allow ourselves a touch of dynamism in the semantics.

For example, when computing booleans, λ_π needs to yield numbers from the δ function that are real booleans (e.g., have the built-in `%bool` object as their class). However, we need booleans to set up if-tests while we are bootstrapping the creation of the boolean class itself! To handle this, we allow global identifiers to appear in the class position of objects. If we look for the class of an object, and the class position is an identifier, we look it up in the global environment. We only use identifiers with special %-prefixed names that aren't visible to Python in this way. It may be possible to remove this touch of dynamic scope from our semantics, but we haven't yet found the desugaring strategy that lets us do so. Figure 7 shows the reduction rule for field lookup in this case.

4 `True`, `False`, and `None`

The keywords `True`, `False`, and `None` are all singleton references in Python. In λ_π, we do not have a form for `True`, instead desugaring it to a variable reference bound in the environment. The same goes for `None` and `False`. We bind each to an allocation of an object:

```
let  True = (alloc ⟨%bool,1,{}⟩)  in
let  False = (alloc ⟨%bool,0,{}⟩)  in
let  None = (alloc ⟨%none, meta-none ,{}⟩)  in
 e_prog
```

and these bindings happen before anything else. This pattern ensures that all references to these identifiers in the desugared program are truly to the same objects. Note also that the boolean values are represented simply as number-like values, but with the built-in `%bool` class, so they can be added and subtracted like numbers, but perform method lookup on the `%bool` class. This reflects Python's semantics:

```
isinstance(True, int) # ==> True
```

5 Variable-arity Functions

We implement Python's variable-arity functions directly in our core semantics, with the reduction rules shown in figure 8. We show first the two arity-mismatch cases in the semantics, where either no vararg is supplied and the argument count is wrong, or where a vararg is supplied but the count is too low. If the count is higher than the number of parameters and a vararg is present, a new tuple is allocated with the extra arguments, and passed as the vararg. Finally, the form e `(`e `...)*`e allows a variable-length collection of arguments to be *passed* to the function; this mimics `apply` in languages like Racket or JavaScript.

6 Modules

We model modules with the two rules in figure 9. E-ConstructModule starts the evaluation of the module, which is represented by a `meta-code` structure. A `meta-code` contains a list of global variables to bind for the module, a name for the module, and an expression that holds the module's body. To start evaluating the module, a new location is allocated for each global variable used in the module, initialized to ☠ in the store, and a new environment is created mapping each of these new identifiers to the new locations.

Evaluation that proceeds inside the module, *replacing* the global environment ϵ with the newly-created environment. The old environment is stored with a new `in-module` form that is left in the current context. This step also sets up an expression that will create a new module object, whose fields hold the global variable references of the running module.

$(E[@ref_{fun}\ \ (val\ \ldots)]\ \varepsilon\ \Sigma)$ [E-AppArity]

$\longrightarrow (E[(\text{err}\ \langle \%str, \text{“arity-mismatch”}, \{\}\rangle)]\ \varepsilon\ \Sigma)$

where $\langle any_c, \lambda(x\ \ldots)\ (\text{no-var}).e\ \ , any_{dict}\rangle = \Sigma(ref_{fun})$,
$(\text{not}\ (\text{equal?}\ (\text{length}\ (val\ \ldots))\ \ (\text{length}\ (x\ \ldots))))$

$(E[@ref_{fun}\ \ (val\ \ldots)]\ \varepsilon\ \Sigma)$ [E-AppVarArgsArity]

$\longrightarrow (E[(\text{err}\ \langle \%str, \text{“arity-mismatch-vargs”}, \{\}\rangle)]\ \varepsilon\ \Sigma)$

where $\langle any_c, \lambda(x\ \ldots)\ (y).e\ \ , any_{dict}\rangle = \Sigma(ref_{fun})$,
$(<\ (\text{length}\ (val\ \ldots))\ \ (\text{length}\ (x\ \ldots)))$

$(E[@ref_{fun}\ \ (val\ \ldots)]\ \varepsilon\ \Sigma)$ [E-AppVarArgs1]

$\longrightarrow (E[(\text{frame}\ \text{subst}[\![(x\ \ldots\ y_{varg}), (ref_{arg}\ \ldots\ ref_{tupleptr}), e]\!])]\ \varepsilon\ \Sigma_3)$

where $\langle any_c, \lambda(x\ \ldots)\ (y_{varg}).e\ \ , any_{dict}\rangle = \Sigma(ref_{fun})$,
$(>=\ (\text{length}\ (val\ \ldots))\ \ (\text{length}\ (x\ \ldots)))$,
$(val_{arg}\ \ldots) = (\text{take}\ (val\ \ldots)\ \ (\text{length}\ (x\ \ldots)))$,
$(val_{rest}\ \ldots) = (\text{drop}\ (val\ \ldots)\ \ (\text{length}\ (x\ \ldots)))$,
$val_{tuple} = \langle \%\text{tuple}, (val_{rest}\ \ldots), \{\}\rangle$,
$(\Sigma_1\ (ref_{arg}\ \ldots)) = \text{extend-store/list}[\![\Sigma, (val_{arg}\ \ldots)]\!]$,
$(\Sigma_2\ ref_{tuple}) = \text{alloc}(\Sigma_1, val_{tuple})$,
$(\Sigma_3\ ref_{tupleptr}) = \text{alloc}(\Sigma_2, @ref_{tuple}\ )$

$(E[@ref_{fun}\ \ (val\ \ldots)*@ref_{var}\ ]\ \varepsilon\ \Sigma)$ [E-AppVarArgs2]

$\longrightarrow (E[@ref_{fun}\ \ (val\ \ldots\ val_{extra}\ \ldots)]\ \varepsilon\ \Sigma)$

where $\langle any_c, (val_{extra}\ \ldots), any_{dict}\rangle = \Sigma(ref_{var})$

Figure 8: Variable-arity functions

$$(E[\texttt{(construct-module @}ref_{mod}\texttt{)}]\ \varepsilon_{old}\ \Sigma) \qquad \text{[E-ConstructModule]}$$

$$\longrightarrow (E[\texttt{(in-module }e_{body}\ \varepsilon_{old}\texttt{)}\ \texttt{(alloc }\langle \texttt{\$module,(no-meta),\{}string_{arg}\texttt{:}ref_{new}\texttt{,...\}}\rangle\texttt{)]}\ \varepsilon_{new}\ \Sigma_1)$$

$$\texttt{where }\langle any_{cls}\texttt{,(meta-code (}x_{arg}\texttt{ ...) }x_{name}\ e_{body}\texttt{),}any_{dict}\rangle = \Sigma(ref_{mod}),$$
$$(\Sigma_1\ \varepsilon_{new}\ (ref_{new}\ \texttt{...})) = \text{vars->fresh-env}[\![\Sigma, (x_{arg}\ \texttt{...})]\!],$$
$$(string_{arg}\ \texttt{...}) = \texttt{(map symbol->string (}x_{arg}\ \texttt{...))}$$

$$(E[\texttt{(in-module }v\ \varepsilon_{old}\texttt{)]}\ \varepsilon_{mod}\ \Sigma) \longrightarrow (E[\texttt{vnone]}\ \varepsilon_{old}\ \Sigma) \qquad \text{[E-ModuleDone]}$$

$$\text{vars->fresh-env}[\![\Sigma, ()]\!] = (\Sigma\ ()\ ())$$

$$\text{vars->fresh-env}[\![\Sigma, (x)]\!] = (\Sigma_1\ ((x\ ref))\ (ref))$$
$$\texttt{where } (\Sigma_1\ (ref)) = \texttt{(alloc }\Sigma\ \text{☠}\texttt{)}$$

$$\text{vars->fresh-env}[\![\Sigma, (x\ x_r\ \texttt{...})]\!] = (\Sigma_2\ ((x\ ref)\ (x_{rest}\ ref_{rest})\ \texttt{...})\ (ref\ ref_{rest}\ \texttt{...}))$$
$$\texttt{where } (\Sigma_1\ ref) = \texttt{alloc(}\Sigma\texttt{,}\text{☠}\texttt{),}$$
$$(\Sigma_2\ ((x_{rest}\ ref_{rest})\ \texttt{...})\ (ref_{rest}\ \texttt{...})) = \text{vars->fresh-env}[\![\Sigma_1, (x_r\ \texttt{...})]\!]$$

Figure 9: Simple modules in λ_π

When the evaluation of the module is complete (the `in-module` term sees a value), the old global environment is reinstated.

To desugar to these core forms, we desugar the files to be imported, and analyze their body to find all the global variables they use. The desugared expression and variables are combined with the filename to create the `meta-code` object. This is roughly an implementation of Python's `compile`, and it should be straightforward to extend it to implement `exec`, though for now we've focused on specifically the module case.