UHC CoreRun
Operational Semantics
(draft)
(CoreRun version 1.1.9.2)

Atze Dijkstra

October 27, 2015

Abstract

This document describes the structure of CoreRun, its operational semantics, the API to use it, and its surface syntax. CoreRun is part of UHC and its hackage counterpart package `uhc-light`.

Contents

1 Introduction

CoreRun is an intermediate language used within UHC to directly run *Core* representations of compiled programs. It has also its own surface syntax allowing external codegenerators to target the representation and run it. For now the intent of offering these facilities is to enable experimentation and teaching.

This document aims to provide a brief explanation of the use of CoreRun in Section 3, and a more formal part describing the (big-step) operational semantics in Section 4.

Disclaimer. This document is a draft, as such it is still rough at the edges and proper citations to tools are not yet included.

2 Design

The following general choices were made for the design of CoreRun

- Similar enough to λ-calculus (to enable translation back to *Core*), yet low-level enough to relatively easily make a not-too-horribly-slow implementation for.
- Variable referencing is done by indices and offsets, not by names.
- Code is scoped, values are bound globally as part of module bindings, and are bound by arguments of a λ-term and **let**-terms. Since these can be nested, referring is done by taking into account nesting levels.
- A λ-term is the unit of scoping, its execution has its own frame with a stack and a static link pointing to the frame of the lexically enclosing scope. Nested **let**-terms use the stack of the frame to store their bindings on.
- Laziness is explicit, encoded by parameterless λ-terms, explicitly forced to evaluate.
- Choice is based on **case**-terms, once a branch is taken it does not return but tail calls.
- Tail recursion is explicit, the implementation should replace the current frame with a new one.

3 Usage by example

Figure 3 shows the concrete surface syntax of CoreRun (middle column). Figure 2 shows an example CoreRun fragment which can directly be run using `uhcr` from the `uhc-light` package cite. The code implements the corresponding (almost) Haskell fragment shown in Figure 1. Bindings are declared without their reference, although for clarity references are included in comment.

```
foreign import prim "primAddInt" (+) :: Int → Int → Int
data L = N | C Int L
l = 3 `C` (4 `C` N)
sum  N      = 0
sum (C x l) = x + sum l
main = sum l                -- IO stuff is ignored
```

Figure 1: Example 1: Haskell

We walk through the example step by step:

- **Start of execution.** On line (1) we find the module header specifying its module number, the size of the frame required to store bindings and initialize these, and the expression which starts execution of the module.
- **Laziness and forcing evaluation.** Execution starts with forcing to evaluate **m**.6, a thunk (delayed/lazy computation) computing *sum l*. The thunk for *sum l* is a zero-parameter λ-term such as $\backslash 0, 2 \rightarrow$ ... on line (17). The λ-term specifies its number of formal parameters and the total size of the stack required (including actual arguments for the formal parameters). When forced to evaluate it applies **m**.5 (*sum*) to **m**.4 (*l*).
- **Tail recursion.** The application is not expected to return hence it is wrapped inside **tail**.
- **Reference to globals.** Both **m**.5 and **m**.4 refer to globals of module 0 (the one of the example) at offsets 5 and 4 respectively.

```
module 0 → eval (m.6);                                                    -- (1)
 {-m.0 -} \2,6 → let 2 →  {-d.0.2 -} eval (d.0.0); in                     -- (2)
                 let 3 →  {-d.0.3 -} eval (d.0.1); in                     -- (3)
                 ffi "primAddInt" (d.0.2, d.0.3);                         -- (4)
 {-m.1 -} alloc 1 ();                                                     -- (5)
 {-m.2 -} \2,4 → alloc 0 (d.0.0, d.0.1);                                  -- (6)
 {-m.3 -} \0,3 → tail (app m.2 (4, m.1));                                 -- (7)
 {-m.4 -} \0,3 → tail (app m.2 (3, m.3));                                 -- (8)
 {-m.5 -} \1,9 → let 1 →  {-d.0.1 -} d.0.0; in                            -- (9)
                 let 2 →  {-d.0.2 -} eval (d.0.1); in                     -- (10)
                 case d.0.2.tag of                                        -- (11)
                   {-0 -}  → let 3 →  {-d.0.3 -} d.0.2.1; in              -- (12)
                             let 4 →  {-d.0.4 -} d.0.2.0; in              -- (13)
                             let 5 →  {-d.0.5 -}
                                        \0,2 → tail (app m.5 (d.1.3)); in -- (14)
                             tail (app m.0 (d.0.4, d.0.5));               -- (15)
                   {-1 -}  → tail (0);;                                   -- (16)
 {-m.6 -} \0,2 → tail (app m.5 (m.4));                                    -- (17)
```

Figure 2: Example 1: CoreRun

- **(Lazy) list construction.** The list l is constructed as a thunk itself which, when forced to evaluate, applies the wrapper **m**.2 around the constructor C allocating a node in memory with tag 0 and two fields (the head and tail of a list). The N constructor **m**.1 uses tag 1 and has no fields.
- **λ-term execution.** The function **m**.5 (*sum*) starts with binding subexpressions locally, the first one follows the argument to **m**.5 directly on stack offset 1; it is just an alias to the argument **d**.0.0.
- **Reference to locals.** The argument and all other locals live in the same frame hence the lexical(static) level difference of 0 in all references of the form **d**.0.x, except for **d**.1.3 which refers from within a nested λ-term for the thunk created in line (14). References also allow tag and field access of heap allocated nodes, for example, in line (12) the second field **d**.0.2.1 of **d**.0.2 with offset 1 is extracted and locally bound to **d**.0.3.
- **case-term execution.** Evaluation in **m**.5 (*sum*) proceeds by evaluating the scrutinee **d**.0.1 and binding it to **d**.0.2 of which the **case**-term uses the tag **d**.0.2.**tag** to decide which branch to take.
- **ffi-term.** The branch corresponding to constructor C (tag 0) tail calls the wrapper **m**.0 around a foreign function call (ffi call) to *primAddInt* implemented by the runtime system. Inside the wrapper all arguments are forced to evaluate before passed to the ffi call, in particular the thunk for the recursive call to *sum* created on line (14).

The above example covers most of the features of CoreRun. Its execution yields 7.

4 Operational semantics

The operational semantics employs additional structures on top of the concrete syntax (see Figure 3); the operational semantics also is limited to one denotational form for constants, namely for *Int* values, and the operational semantics avoids dealing with maximum stack sizes of frames. Square brackets [and] are used to denote sequences with notation and usage taken from Haskell's

lists, in particular : is used to prepend elements and is used for pattern matching in the rules for operational semantics.

State The specification of the operational semantics (Figure 4 and onward) uses a state tuple $(\mathcal{H}, \mathcal{S})$ holding heap and frame stack. Each frame f in $\mathcal{S}$ belongs to a single λ-term. The frame also includes all bindings of nested **let**-terms. A frame holds an expression stack s and a static link sl to the frame of a lexically enclosing λ-term. The expression stack s holds λ-term argument values v followed by the values of local bindings followed by the stack used for evaluation. When evaluating for a binding the result is just left on the stack, so the part of frame used for local bindings expands during execution of **let**-term bindings thereby shifting the part of the frame used for evaluation of terms. To avoid having to deal with deallocation of local bindings the expression stack can only grow, hence no return from **let** and **case**-terms is allowed, where a return would be a further use of a result from **let** or **case**-term directly in the same λ-term. Results can only be passed on to (tail) applications etc..

Frames are also used to hold module bindings. A module is initialized in the same way let bindings are initialized, by just evaluating the initialization terms. The main module frame is also used to evaluate the term used to start execution.

The heap $\mathcal{H}$ maps pointers p to values v. Heap specific behavior like garbage collection is ignored.

Rules for expression terms e in Figure 6 use judgements taking input state $(\mathcal{H}_i, \mathcal{S}_i)$ which can be updated yielding output state $(\mathcal{H}_o, \mathcal{S}_o)$. The operational semantics is specified in 'big-step' style which does not specify an explicit ordering of control. The flow of state $(\mathcal{H}, \mathcal{S})$ however still enforces such an ordering.

Context The rules also use context information $\mathcal{C}_\mathcal{E}$ holds various pieces of information, mostly globally determined:

- Module frames used when referring to globals, each indexed by a module frame nr m.
- Import table, for each module in the import declaration *idcl* its corresponding module frame nr m, each indexed by the relative position of corresponding *idcl*.
- The context also can be queried for "being in tail recursive position", but the details for the boolean required to encode this are left largely unspecified, relying on the suggested meaning of predicates *isInTailCxt* and others in the judgements.

The rules for the program and module do not specify (or only partially) how the above is computed.

Simple expressions: constants and references The specification for the simplest expression terms *se* do not require an update to the heap $\mathcal{H}$. Figure 4 shows the rules for such simple terms. For example rule INT takes the evaluation stack s from the current frame $\mathcal{F}\ s\ sl\ m$ and pushes the i value on top.

Similarly rule REF accesses the value of a reference r (see Figure 5) and also pushes it onto the evaluation stack. A global reference (rule GLOB in Figure 5) to a value in a same module accesses its module frame, a global reference (rule IMPORT) to a value from another (imported) module first has to access its table of imported modules provided by the context $\mathcal{C}_\mathcal{E}$ to find the module nr m of the module im being imported before it can access the frame with the index of the global variable. The CoreRun surface syntax also provides a variant $\mathbf{e}.nm.g$ which refers to a module by name; preprocessing of CoreRun modules translate these in the indexed variant described by rule IMPORT. A local reference (rules LOC and LOC0) follows the static link of the current frame and accesses the value specified by the offset. Rules TAG and FLD respectively access tag and field of a node allocated for a constructor.

$prog$	$= \mathcal{P}\ m\ [m]$		-- main module + imported
m	$= [e] \to e$	$\simeq$ **module** $nm\ sz \to e$; $body$	-- module, global bindings + main
$body$	$=$	$\simeq [idcl;]\ [edcl;]\ [dtyp;]\ [e;]$	-- imports, exports, datatypes, values
$idcl$	$=$	$\simeq$ **import** "nm"	-- imported module
$edcl$	$=$	$\simeq$ **export** "nm" $= o$	-- exported name + offset
$dtyp$	$=$	$\simeq$ **data** $str = dcon\ [, dcon]$	-- data type
$dcon$	$=$	$\simeq str \to t$	-- data constructor to tag
se	$= i$	$\simeq 5$	-- base term: int denotation
		`'x'`	-- base term: char denotation
		`"xyz"`	-- base term: string denotation
	$\mid r$		-- base terms: reference
e	$= se \mid l_e$		-- base terms
	$\mid e\ [se]$	$\simeq$ **app** $(e)\ (se, [se])$	-- application
	$\mid$ `"foreign"` $[se]$	$\simeq$ **ffi** `"foreign"` $(se, [se])$	-- ffi call
	$\mid t\ [se]$	$\simeq$ **alloc** $t\ (se, [se])$	-- node allocation
	$\mid$ **eval** e	$\simeq$ **eval** (e)	-- force evaluation
	$\mid te$		-- tail occurring term
te	$=$ **tail** e	$\simeq$ **tail** (e)	-- in tail recursive position
	$\mid$ **let** $[e]$ **in** te	$\simeq$ **let** $[e;]$ **in** te	-- local bindings
	$\mid$ **case** se **of** $[a]$	$\simeq$ **case** se **of** $[\to te;]$	-- case
l_e	$= \lambda na \to e$	$\simeq \backslash na, sz \to e$	-- lambda term taking $na \geqslant 0$ args
a	$= t \to te$	$\simeq\ \to te;$	-- case alternative
r	$= d.o$	$\simeq$ **d**$.d.o$	-- reference to local
	$\mid im.g$	$\simeq$ **i**$.im.g$	-- global of imported module
		$\simeq$ **e**$.nm.g$	-- global of named module
	$\mid o$	$\simeq$ **m**$.g$	-- reference to global of module
	$\mid r.o$	$\simeq r.o$	-- reference inside node
	$\mid r.t$	$\simeq r.$**tag**	-- tag of node
$\mathcal{S}$	$= [f]$		-- frame stack with embedded evaluation stack
$\mathcal{C_E}$	$= \mathcal{E}\ [im]\ [f]$		-- context: modules/globals, see text
f	$= \mathcal{F}\ s\ sl\ m$		-- frame: stack + static link + module nr
n	$= t\ [v]$		-- heap node
s	$= [v]$		-- evaluation stack per frame
v	$= i \mid p \mid f \mid l_v \mid n$		-- base values
	$\mid v\ [v]$		-- undersaturated application
l_v	$= {}_v\backslash sz, sl \to e$		-- lambda value taking $\geqslant 0$ args, with context
sl	$= p$		-- static link pointing to scope enclosing frame
p	$= i$		-- pointer
o	$= i$		-- offset (0-based)
t	$= i$		-- node tag, used by case
sz	$= i$		-- size (of stack, usually)
na	$= i$		-- nr of arguments
g	$= i$		-- global of module
im	$= i$		-- imported module reference
m	$= i$		-- module number (globally)
d	$= i$		-- frame reference relative to current
str	$=$ `"..."`		-- string
nm	$=$ *identifier*		-- name
$\mathcal{H}$	$= [p \mapsto v]$		-- heap

Figure 3: Structures

$$\boxed{\mathcal{C}_\mathcal{E};(\mathcal{H},\mathcal{S}_i)\vdash^{se} se \rightsquigarrow \mathcal{S}_o}$$

$$\frac{}{\mathcal{C}_\mathcal{E};(\mathcal{H},(\mathcal{F}\ s\ sl\ m){:}\mathcal{S})\vdash^{se} i \rightsquigarrow (\mathcal{F}\ (i{:}s)\ sl\ m){:}\mathcal{S}}\ (\textsc{int})$$

$$\frac{\mathcal{C}_\mathcal{E};\mathcal{H};\mathcal{F}\ s\ sl\ m\vdash^{r} r \rightsquigarrow v}{\mathcal{C}_\mathcal{E};(\mathcal{H},(\mathcal{F}\ s\ sl\ m){:}\mathcal{S})\vdash^{se} r \rightsquigarrow (\mathcal{F}\ (v{:}s)\ sl\ m){:}\mathcal{S}}\ (\textsc{ref})$$

Figure 4: Simple expression

$$\boxed{\mathcal{C}_\mathcal{E};\mathcal{H};f\vdash^{r} r \rightsquigarrow v}$$

$$\frac{}{\mathcal{E}\ \_\ ([...,(\mathcal{F}\ [...,v_o,...,v_0]\ \_\ \_)_m,...]);\mathcal{H};\mathcal{F}\ \_\ \_\ m\vdash^{r} o \rightsquigarrow v_o}\ (\textsc{glob})$$

$$\frac{}{\mathcal{E}\ (...,m_{im},...)\ ([...,(\mathcal{F}\ [...,v_g,...,v_0]\ \_\ \_)_{m_{im}},...]);\mathcal{H};f\vdash^{r} im.g \rightsquigarrow v_g}\ (\textsc{import})$$

$$\frac{}{\mathcal{C}_\mathcal{E};\mathcal{H};\mathcal{F}\ [...,v_o,...,v_0]\ \_\ \_\vdash^{r} 0.o \rightsquigarrow v_o}\ (\textsc{loc0}) \qquad \frac{f=\mathcal{H}\ [sl] \quad \mathcal{C}_\mathcal{E};\mathcal{H};f\vdash^{r} d-1,o \rightsquigarrow v}{\mathcal{C}_\mathcal{E};\mathcal{H};\mathcal{F}\ \_\ sl\ \_\vdash^{r} d.o \rightsquigarrow v}\ (\textsc{loc})$$

$$\frac{\mathcal{C}_\mathcal{E};\mathcal{H};f\vdash^{r} r \rightsquigarrow t\ [...]}{\mathcal{C}_\mathcal{E};\mathcal{H};f\vdash^{r} r.\mathbf{tag} \rightsquigarrow t}\ (\textsc{tag}) \qquad \frac{\mathcal{C}_\mathcal{E};\mathcal{H};f\vdash^{r} r \rightsquigarrow t\ [...,v_o,...]}{\mathcal{C}_\mathcal{E};\mathcal{H};f\vdash^{r} r.o \rightsquigarrow v_o}\ (\textsc{fld})$$

Figure 5: Reference

Expressions: application, abstraction, and evaluation Application (rule APP in Figure 6) first pushed all arguments in reverse order onto the stack followed by the function. Specialized rules for application (Figure 7) inspect the function and dispatch on its structure and the number of available arguments. Normal non tail-recursive invocation of a saturated application (rule SAT) takes the appropriate number of arguments from the evaluation stack of the top frame and pushes a new frame. After evaluation of the application the result value is taken from this frame only to be pushed on the previous stack as the result of the application/invocation. A tail-recursive invocation (rule SATTAIL) deviates from this by omitting the push of a new frame; it just replaces the top frame. The other cases in Figure 7 deal with under- and oversaturated applications. An undersaturated application constructs a temporary value holding the function and the arguments provided sofar. In rule APP these remembered arguments are accumulated with new arguments and application is tried again.

The rules ALLOC and FFI are similar to rule APP in their gathering of arguments, respectively for allocation on the heap and call to a function provided by the runtime environment. Also similar is the rule LET which gathers its local bindings on the stack and then proceeds with the body of the **let**-term.

λ-terms fullfill both the role of functions ($\geqslant 1$ arguments and thunks ($\equiv 0$ arguments). Both are values to be pushed on the stack, but in addition a thunk requires to be accessed via a pointer in order to allow updates (see Figure 8). In both cases the λ-value is set up with a pointer to the current frame as its static link. Evaluation of a thunk (see Figure 8) consists of an application (to 0 arguments) combined with a $\mathcal{H}$ update.

Finally, **case**-terms dispatch on a tag, a **tail** annotation sets up context $\mathcal{C}_\mathcal{E}$ properly.

Modules On the toplevel a program (Figure 9) consists of a main module and a sequence of imported modules, each of which consists of a sequence of bindings (Figure 10). A module is (at

$$\boxed{\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e e \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)}$$

$$\frac{\mathcal{C}_\mathcal{E}; (\mathcal{H}, \mathcal{S}_i) \vdash^{se} se \rightsquigarrow \mathcal{S}_o}{\mathcal{C}_\mathcal{E}; (\mathcal{H}, \mathcal{S}_i) \vdash^e se \rightsquigarrow (\mathcal{H}, \mathcal{S}_o)} \text{ (SEXP)} \qquad \frac{\begin{array}{c}\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^{se} se_{sz_@} \rightsquigarrow \mathcal{S}_{sz_@} \\ \mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_{sz_@}) \vdash^{se} se_1 \rightsquigarrow \mathcal{S}_1 \\ \mathcal{C}'_\mathcal{E} = setFalseInTailCxt\ (\mathcal{C}_\mathcal{E}) \\ \mathcal{C}'_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_1) \vdash^e e_f \rightsquigarrow (\mathcal{H}_f, \mathcal{S}_f) \\ \mathcal{C}_\mathcal{E}; (\mathcal{H}_f, \mathcal{S}_f) \vdash^{ap} sz_@ \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)\end{array}}{\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e e_f\ [se_1, ..., se_{sz_@}] \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)} \text{ (APP)}$$

$$\frac{\begin{array}{c}\mathcal{C}'_\mathcal{E} = setFalseInTailCxt\ (\mathcal{C}_\mathcal{E}) \\ \mathcal{C}'_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e e_1 \rightsquigarrow (\mathcal{H}_1, \mathcal{S}_1) \\ \mathcal{C}'_\mathcal{E}; (\mathcal{H}_1, \mathcal{S}_1) \vdash^e e_n \rightsquigarrow (\mathcal{H}_n, \mathcal{S}_n) \\ \mathcal{C}_\mathcal{E}; (\mathcal{H}_n, \mathcal{S}_n) \vdash^e e \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)\end{array}}{\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e \textbf{let}\ [e_1, ..., e_n]\ \textbf{in}\ e \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)} \text{ (LET)}$$

$$\frac{\begin{array}{c}\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^{se} se \rightsquigarrow (\mathcal{F}\ (t{:}s)\ sl\ m){:}\mathcal{S} \\ \mathcal{C}_\mathcal{E}; (\mathcal{H}_i, (\mathcal{F}\ s\ sl\ m){:}\mathcal{S}) \vdash^e e_t \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)\end{array}}{\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e \textbf{case}\ se\ \textbf{of}\ [0 \to e_0, ..., t \to e_t, ...] \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)} \text{ (CASE)}$$

$$\frac{\begin{array}{c}\mathcal{H}_{new} = (p \equiv newPtr\ (\mathcal{H}_i)) \\ \mathcal{H}_o = (\mathcal{H}_{new}[p] \equiv {}_v\backslash 0, f \to e)\end{array}}{\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, (f \mapsto (\mathcal{F}\ s\ sl\ m)){:}\mathcal{S}) \vdash^e \lambda 0 \to e \rightsquigarrow (\mathcal{H}_o, (\mathcal{F}\ (p{:}s)\ sl\ m){:}\mathcal{S})} \text{ (THUNK)}$$

$$\frac{}{\mathcal{C}_\mathcal{E}; (\mathcal{H}, (f \mapsto (\mathcal{F}\ s\ sl\ m)){:}\mathcal{S}) \vdash^e \lambda sz_\lambda \to e \rightsquigarrow (\mathcal{H}, (\mathcal{F}\ (({}_v\backslash sz_\lambda, f \to e){:}s)\ sl\ m){:}\mathcal{S})} \text{ (LAM)}$$

$$\frac{\begin{array}{c}\mathcal{C}'_\mathcal{E} = setTrueInTailCxt\ (\mathcal{C}_\mathcal{E}) \\ \mathcal{C}'_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e e \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)\end{array}}{\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e \textbf{tail}\ e \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)} \text{ (TAIL)} \qquad \frac{\begin{array}{c}\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e e \rightsquigarrow (\mathcal{H}_e, \mathcal{S}_e) \\ \mathcal{C}_\mathcal{E}; (\mathcal{H}_e, \mathcal{S}_e) \vdash^{evl} (\mathcal{H}_o, \mathcal{S}_o)\end{array}}{\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e \textbf{eval}\ e \rightsquigarrow (\mathcal{H}_o, \mathcal{S}_o)} \text{ (EVAL)}$$

$$\frac{\begin{array}{c}\mathcal{H}_{new} = (p \equiv newPtr\ (\mathcal{H}_i)) \\ \mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^{se} se_n \rightsquigarrow \mathcal{S}_n \\ \mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_n) \vdash^{se} se_1 \rightsquigarrow (\mathcal{F}\ ([v_1, ..., v_n]{:}s)\ sl\ m){:}\mathcal{S}_o \\ \mathcal{H}_o = (\mathcal{H}_{new}[p] \equiv t\ [v_1, ..., v_n])\end{array}}{\mathcal{C}_\mathcal{E}; (\mathcal{H}_i, \mathcal{S}_i) \vdash^e t\ [se_1, ..., se_n] \rightsquigarrow (\mathcal{H}_o, (\mathcal{F}\ (p{:}s)\ sl\ m){:}\mathcal{S}_o)} \text{ (ALLOC)}$$

$$\frac{\begin{array}{c}\vdash^{\textbf{ffi}} foreign\ [v_1, ..., v_n] \rightsquigarrow v \\ \mathcal{C}_\mathcal{E}; (\mathcal{H}, \mathcal{S}_i) \vdash^{se} se_n \rightsquigarrow \mathcal{S}_n \\ \mathcal{C}_\mathcal{E}; (\mathcal{H}, \mathcal{S}_n) \vdash^{se} se_1 \rightsquigarrow (\mathcal{F}\ ([v_1, ..., v_n]{:}s)\ sl\ m){:}\mathcal{S}_o\end{array}}{\mathcal{C}_\mathcal{E}; (\mathcal{H}, \mathcal{S}_i) \vdash^e \texttt{"foreign"}\ [se_1, ..., se_n] \rightsquigarrow (\mathcal{H}, (\mathcal{F}\ (v{:}s)\ sl\ m){:}\mathcal{S}_o)} \text{ (FFI)}$$

Figure 6: Expression

$$\boxed{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,\mathcal{S}_i)\vdash^{ap} sz \rightsquigarrow (\mathcal{H}_o,\mathcal{S}_o)}$$

$$\frac{\begin{array}{c}isInTailCxt\ (\mathcal{C}_{\mathcal{E}})\\ \mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ [v_{sz_\lambda},...,v_1]\ sl_l\ m):\mathcal{S}_i)\vdash^{e} e\rightsquigarrow(\mathcal{H}_o,\mathcal{S}_o)\end{array}}{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ ([(_v\backslash sz_\lambda,sl_l\rightarrow e),v_1,...,v_{sz_\lambda}]:s)\ sl\ m):\mathcal{S}_i)\vdash^{ap} sz_\lambda\rightsquigarrow(\mathcal{H}_o,\mathcal{S}_o)}\ (\textsc{sattail})$$

$$\frac{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ [v_{sz_\lambda},...,v_1]\ sl_l):(\mathcal{F}\ s\ sl\ m):\mathcal{S}_i)\vdash^{e} e\rightsquigarrow(\mathcal{H}_o,(\mathcal{F}\ (v{:}\_)\ m):(\mathcal{F}\ s\ sl\ m):\mathcal{S}_o)}{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ ([(_v\backslash sz_\lambda,sl_l\rightarrow e),v_1,...,v_{sz_\lambda}]:s)\ sl\ m):\mathcal{S}_i)\vdash^{ap} sz_\lambda\rightsquigarrow(\mathcal{H}_o,(\mathcal{F}\ (v{:}s)\ sl\ m):\mathcal{S}_o)}\ (\textsc{sat})$$

$$\frac{sz_@ < sz_\lambda}{\mathcal{C}_{\mathcal{E}};(\mathcal{H},(\mathcal{F}\ ([(_v\backslash sz_\lambda,sl_l\rightarrow e),v_1,...,v_{sz_@}]:s)\ sl\ m):\mathcal{S})\vdash^{ap} sz_@\rightsquigarrow(\mathcal{H},(\mathcal{F}\ (((_v\backslash sz_\lambda,sl_l\rightarrow e)\ [v_1,...,v_{sz_@}]):s)\ sl\ m):\mathcal{S})}\ (\textsc{usat})$$

$$\frac{\begin{array}{c}sz_@ > sz_\lambda\\ \mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ ((_v\backslash sz_\lambda,sl_l\rightarrow e):s)\ sl\ m):\mathcal{S})\vdash^{ap} sz_\lambda\rightsquigarrow(\mathcal{H}_f,\mathcal{S}_f)\\ \mathcal{C}_{\mathcal{E}};(\mathcal{H}_f,\mathcal{S}_f)\vdash^{ap}(sz_@-sz_\lambda)\rightsquigarrow(\mathcal{H}_o,\mathcal{S}_o)\end{array}}{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ ((_v\backslash sz_\lambda,sl_l\rightarrow e):s)\ sl\ m):\mathcal{S})\vdash^{ap} sz_@\rightsquigarrow(\mathcal{H}_o,\mathcal{S}_o)}\ (\textsc{osat})$$

$$\frac{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ (e_f{:}[v_1,...,v_{sz_@}]:s)\ sl\ m):\mathcal{S})\vdash^{ap}(sz_@+sz_\lambda)\rightsquigarrow(\mathcal{H}_o,\mathcal{S}_o)}{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ ((e_f\ [v_1,...,v_{sz_@}]):s)\ sl\ m):\mathcal{S})\vdash^{ap} sz_\lambda\rightsquigarrow(\mathcal{H}_o,\mathcal{S}_o)}\ (\textsc{app})$$

Figure 7: Application

$$\boxed{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,\mathcal{S}_i)\vdash^{evl}(\mathcal{H}_o,\mathcal{S}_o)}$$

$$\frac{\begin{array}{c}\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ ((_v\backslash 0,sl_l\rightarrow e):s)\ sl\ m):\mathcal{S}_i)\vdash^{ap} 0\rightsquigarrow(\mathcal{H}_{ap},(\mathcal{F}\ (v{:}s)\ sl\ m):\mathcal{S}_o)\\ \mathcal{H}_o=(\mathcal{H}_{ap}[p]\equiv v)\end{array}}{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,(\mathcal{F}\ ((p\mapsto(_v\backslash 0,sl_l\rightarrow e)):s)\ sl\ m):\mathcal{S}_i)\vdash^{evl}(\mathcal{H}_o,(\mathcal{F}\ (v{:}s)\ sl\ m):\mathcal{S}_o)}\ (\textsc{force})$$

Figure 8: Evaluation

$$\boxed{\vdash^{prog} prog\rightsquigarrow v}$$

$$\frac{\begin{array}{c}\mathcal{E}\ \_\ ([f_1,...,f_n]+\!\!+[f_0]);(\mathcal{H}_0,[f_0])\vdash^{e} e_0\rightsquigarrow(\_,[\mathcal{F}\ (v{:}\_)\ \_\ m])\\ \mathcal{E}\ \_\ [];([],[\mathcal{F}\ []\perp m])\vdash^{m}\mathbf{m}_1\rightsquigarrow\_;(\mathcal{H}_1,[f_1])\\ \mathcal{E}\ \_\ [f_1,...,f_n];(\mathcal{H}_n,[\mathcal{F}\ []\perp m])\vdash^{m}\mathbf{m}_0\rightsquigarrow e_0;(\mathcal{H}_0,[f_0])\\ \mathcal{E}\ \_\ [f_1,...];(\mathcal{H}_1,[\mathcal{F}\ []\perp m])\vdash^{m}\mathbf{m}_n\rightsquigarrow\_;(\mathcal{H}_n,[f_n])\end{array}}{\vdash^{prog}\mathcal{P}\ \mathbf{m}_0\ [\mathbf{m}_1,...,\mathbf{m}_n]\rightsquigarrow v}\ (\textsc{prog})$$

Figure 9: Program

$$\boxed{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,\mathcal{S}_i)\vdash^{m} m\rightsquigarrow e;(\mathcal{H}_o,\mathcal{S}_o)}$$

$$\frac{\begin{array}{c}\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,\mathcal{S}_i)\vdash^{e} e_0\rightsquigarrow(\mathcal{H}_0,\mathcal{S}_0)\\ \mathcal{C}_{\mathcal{E}};(\mathcal{H}_0,\mathcal{S}_0)\vdash^{e} e_n\rightsquigarrow(\mathcal{H}_o,\mathcal{S}_o)\end{array}}{\mathcal{C}_{\mathcal{E}};(\mathcal{H}_i,\mathcal{S}_i)\vdash^{m}[e_0,...,e_n]\rightarrow e\rightsquigarrow e;(\mathcal{H}_o,\mathcal{S}_o)}\ (\textsc{mod})$$

Figure 10: Module

runtime) treated as a frame holding all bindings, similar as done for **let**-expressions. The context $\mathcal{C}_\mathcal{E}$ is set up to be a sequence of frames for each of the modules, the main entry point of the program is evaluated in the frame of the main module. A subtle point not made explicit in PROG is that mutually recursive references are assumed to either occur in a "define before use order", or, when referred to before available, are hidden inside a *lambd*-expression.

Note the following

- Frames are allocated on the heap, accessed via a pointer. The rules mostly ignore this detail. Other structures allocated on the heap may also be accessable via a pointer but rules also ignore the required dereferencing.
- The order of the rules matters when there is overlap, e.g. for the rules THUNK and LAM in Figure 6.
- Management of tail context is suggestively specified.
- When a sequence like $[e]$ requires use of rules this is done only for the ends of the sequence.
- Stacks are assumed to be large enough; the concrete syntax specifies the maximum required stack size.
- Heap garbage collection is unspecified.

5 Primitive binding

Primitives are assumed to be made available by the CoreRun runtime. If not, this is a runtime error. The idea is to also provide primitives at a highlevel: CoreRun interpreters are expected to be implemented in a Haskell runtime context, so primitives for CoreRun can be highlevel Haskell functions dealing with arbitrary datatypes. For example, for the default runtime Haskell *openFile* is available as primitive `primOpenFile`:

$$\mathit{openFile} :: \mathit{FilePath} \to \mathit{IOMode} \to \mathit{IO}\ \mathit{Handle}$$

Haskell representations of *FilePath*, *IOMode*, and *Handle* and their CoreRun counterparts therefore need to interact, similar to the way this is done in the Haskell FFI cite. Datatypes like these are treated in one of two ways:

- Datatypes are opaque. There is a CoreRun baked in representation wrapping. Such a datatype is not intended to (and cannot) be inspected on the CoreRun level.
- Datatypes are defined on the Haskell level as well as the CoreRun level with exactly the same definition. The CoreRun representation for memory nodes is used to represent data constructor values, and the marshalling between the Haskell and CoreRun representations is done generically with the aid of (1) (CoreRun) constructor to tag mapping embedded in a CoreRun file, and (2) Haskell (generic deriving) instances for the Haskell side corresponding datatype definitions.

For *openFile*, a *Handle* is baked in, and *FilePath* (a *String*) and *IOMode* are (un)marshalled when passed or returned through a primitive.

The inclusion of meta information about datatypes quickly becomes obligatory as basic primitives use simple datatypes. The following small Haskell fragment uses `primEqInt` returning a *Bool*.

$$\mathit{foreign}\ \textbf{import}\ \mathit{prim}\ \texttt{"primEqInt"}\ (\equiv) :: \mathit{Int} \to \mathit{Int} \to \mathit{Bool}$$
$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$
$$\mathit{main} = \textbf{if}\ 5 \equiv 5\ \textbf{then}\ 1\ \textbf{else}\ 0$$

The following is the corresponding CoreRun

```
module ifthenelse8 7 → eval (m.3);
data "Bool" = "False" → 0, "True" → 1;
 {-m.0 -} \2, 6 → let 2 →  {-d.0.2 -} eval (d.0.0); in
                  let 3 →  {-d.0.3 -} eval (d.0.1); in
                  ffi "primEqInt" (d.0.2, d.0.3);
 {-m.1 -} alloc 1 ();
 {-m.2 -} alloc 0 ();
 {-m.3 -} \0, 3 → let 0 →  {-d.0.0 -} \0, 3 → tail (app (eval (m.0)) (5, 5)); in
                  let 1 →  {-d.0.1 -} eval (d.0.0); in
                  case d.0.1.tag of
                      {-0 -}  → tail (0);
                      {-1 -}  → tail (1);;
```

6 Current limitations

For now the following limitations hold:

- No exception throwing or catching.
- The heap is garbage collected but there is no policy to estimate the 'right' size of the heap.
- A limited set of primitives is available. The intent is to implement an extension mechanism where code compiled with the CoreRun runtime can be bound to primitives and used from CoreRun, thereby allowing CoreRun to act as embedded Haskell when combined with the compiler part of `uhc-light`.
- Inclusion of name/symbol information intended for debugging is not yet properly done.
- No stacktraces in case something goes wrong.

7 Running CoreRun programs

CoreRun can be run using package `uhc-light`, in particular `uhcr` can be given a `.tcrr` (Textual CoReRun) suffixed file. Option `--corerunopt=printresult` will print the evaluation result on the output.