HOT-Compilation: CPS and Closure Conversion

*TA: Akiva Leffert
aleffert@andrew.cmu.edu

Out: Wednesday, November 29th
In: Friday, December 8th (Before midnight)

1 Introduction

The programming language calculi we have seen so far are relatively rich inductive systems; it is not immediately obvious how to render the execution of such a program on a computer with a very simple and direct instruction set. Transformation to continuation-passing style, or the CPS transform, is an important step from high-level languages towards the low-level simplicity of a concrete machine architecture.

The CPS transform has three main effects on our program:

1. it linearizes the computation, making explicit the sequence in which operations are to be performed,
2. it names all intermediate results, simplifying the inductive structure of the program, and
3. it reifies control flow, explaining things like exceptions and call/cc in terms of more primitive constructs.

Conversion to continuation-passing style works by systematically transforming the program to include an explicit object, a continuation, which represents the "rest of the program". Function calls never return, but rather they invoke their continuation with the appropriate value to continue computation.

Closure conversion continues this simplification by making the environment of a function in explicit. Following closure conversion, function bodies can only refer to variables bound within their scope. As such, the job of a closure converter is to ensure that all necessary environment variables are passed explictly to each function. Typically, the compilation step immediately following closure conversion is hoisting. This stage moves each function to a list of mutually recursive functions at the top of the program, separating code from other values. For hoisting to be possible all functions must be closed, as, after hoisting, functions longer have an enclosing environment.

2 Overview

In this assignment you will be implementing cps conversion and closure conversion. More specifically, you are required to implement two signatures, `CPS_CONVERT` found in the file `convert/cpsconvert-sig.sml` and `CLOSURE_CONVERT` found in the file `convert/closureconvert-sig.sml`. These should be placed in `convert/cpsconvert.sml` and `convert/closureconvert.sml` respectively. We will be making your life easier by providing a starter implementation of each of these files with some of the cases already implemented. This code will tell you which cases you still need to fill in. You can also use the compiler's exhaustive match checker to aid you. The `CPS_CONVERT` signature is also provided in Figure 1 and the `CLOSURE_CONVERT` signature in Figure 2.

You should implement CPS conversion as outlined in Section 4. Since the translation is heavily influenced by the types of the terms involved, your code should be type-derivation-directed, and produce a type for

*The CPS Conversion portion of this assignment was originally prepared by William Lovas (Fall 2005)

```
signature CPS_CONVERT =
  sig
    type context = IL3Static.Ctx.context

    exception Convert of string
    exception Error of string

    val convert_kind : IL3.kind -> IL4.kind (* just the "identity" *)
    val convert_con : IL3.con -> IL4.con
    val convert_exp : context -> IL3.exp
                   -> IL3.con * {k: IL4.expvar, k_exn: IL4.expvar} * IL4.exp
  end
```

Figure 1: CPS Conversion Signature

```
signature CLOSURE_CONVERT =
  sig
    type context = IL4Static.Ctx.context

    exception Convert of string
    exception Error of string

    val convert_kind : IL4.kind -> IL5.kind (* just the "identity" *)
    val convert_con : IL4.con -> IL5.con
    val convert_exp : context -> IL4.exp -> IL5.exp

  end
```

Figure 2: Closure Conversion Signature

each expression in addition to its translation. Section 5 defines the closure conversion algorithm you should use.

Signal any conversion errors by raising `Convert` with an appropriate error message; you may use the `Error` exception for internal consistency errors such as violated invariants or the occurrence of "impossible" conditions.

Syntax definitions for IL3, IL4 and IL5, along with assorted useful functions such as substitution, can be found in the `il3/`, `il4/` and `il5/` directories respectively. Additionally, example IL3 and IL4 programs can be found in the appropriate directories.

Each converter has its own test harness interface. These can be found in the `CPSTop` and `ClosureTop` structures. A few simple examples to get you started are shown in sections 6 and 7

Submit your code via AFS by copying your `cpsconvert.sml` to the `cps/` directory in your submission directory and your `closureconvert.sml` to the `closure/` directory. **Note:** Your submission will be graded automatically; if you submit any files that fail to compile under SML/NJ 110.59 using this assignment's base distribution, we won't be able to grade it.

3 Syntax

The syntax of IL3 and IL4 are similar to what we've seen in class and in previous assignments. You'll find IL3's syntax in subsection 3.1. IL4's syntax is shown in subsection 3.2.

Note the following subtleties or differences from previous presentations:

- Expression-level products and sums are n-ary rather than binary.
- IL3 **case** and **iftagof** expressions both still take functions, as in IL1, but in IL4 these have been converted to expressions with explicit variable binders. Your translation will have to account for this. (**handle** expressions already have an explicit binder in IL3 — exception handling is sufficiently confusing as it is.)
- In both IL3 and IL4, **fix** expressions explicitly project one function out of their mutually recursive bundle, rather than being a product of functions that you project from using $\pi_i\ (\cdot)$. This is so that the derived form for λ terms can be a value in IL4.
- The IL5 syntax is identical to that of IL4 except for the addition of existential terms and types. As such, IL5 will not be shown completely but only as an extension to IL4.

3.1 IL3 Syntax

3.1.1 IL3 Kinds

κ	::=	$\mathbf{T}$	kind of types
	|	$\mathbf{1}$	unit kind
	|	$\kappa_1 \rightarrow \kappa_2$	function kinds
	|	$\kappa_1 \times \kappa_2$	pair kinds

3.1.2 IL3 Constructors

c	::=	α	constructor variables
	|	$\lambda\alpha{:}\kappa.c$	constructor functions
	|	$c_1 c_2$	constructor application
	|	$\star$	unit constructor
	|	$\langle c_1, c_2 \rangle$	constructor pairs
	|	$\pi_1 c$	constructor pair first projection
	|	$\pi_2 c$	constructor pair second projection
	|	$\mu\alpha{:}\kappa.c$	recursive constructors
	|	int | char | string	built-in base types
	|	$c_1 \rightarrow c_2$	functions
	|	ref c	references
	|	tagged	extensible tagged unions
	|	tag c	tags
	|	$\times[c_1, \ldots, c_n]$	unlabelled products
	|	$+[c_1, \ldots, c_n]$	unlabelled sums
	|	$\forall\alpha{:}\kappa.c$	polymorphic types

3.1.3 IL3 Expressions

e	::=	x	variables
	|	$\overline{n}$	integer literals
	|	*'char'*	char literals
	|	*"string"*	string literals
	|	$unop\ e$	primitive unary operations
	|	$e_1\ binop\ e_2$	primitive binary operations
	|	$\mathbf{fix}_i\ fb_1, \ldots, fb_n\ \mathbf{end}$	anonymous recursive function from bundle
	|	$e_1 e_2$	application
	|	$\mathbf{handle}\ e_1\ \mathbf{with}\ x.e_2$	exception handling
	|	$\mathbf{raise}^c\ e$	exception raising
	|	$\mathbf{ref}\ e$	reference cell allocation
	|	$\mathbf{get}\ e$	reference cell derefererence
	|	$\mathbf{set}(e_1, e_2)$	reference cell update
	|	$\mathbf{roll}^c\ e$	coercion into recursive type
	|	$\mathbf{unroll}\ e$	coercion out of recursive type
	|	$(e_1, \ldots, e_n)$	unlabelled products
	|	$\pi_i\ e$	product projection
	|	$\mathbf{inj}_i^c e$	sum injection
	|	$\mathbf{case}^c\ e\ \mathbf{of}\ e_1, \ldots, e_n$	sum analysis
	|	$\mathbf{tag}(e_1, e_2)$	injection into type tagged
	|	$\mathbf{newtag}[c]$	extension of type tagged
	|	$\mathbf{iftagof}\ e_1\ \mathbf{is}\ e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4$	tag analysis
	|	$\Lambda\alpha{:}\kappa.e$	polymorphic abstraction
	|	$e[c]$	polymorphic application
	|	$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	expression let-binding

3.1.4 IL3 Function Bindings

fb	::=	$f\ (x{:}c) : c'.e$	function bindings

3.1.5 IL3 Derived Forms

κs	::=	$\cdot \mid \kappa, \kappa s$	kind lists
cs	::=	$\cdot \mid c, cs$	constructor lists

$\times[\cdot]$	$\stackrel{\text{def}}{=}$	$\mathbf{1}$	0-ary product kind
$\times[\kappa, \kappa s]$	$\stackrel{\text{def}}{=}$	$\kappa \times (\times[\kappa])$	non-empty n-ary product of kinds
$\langle\cdot\rangle$	$\stackrel{\text{def}}{=}$	$\star$	0-ary product of constructors
$\langle c, cs\rangle$	$\stackrel{\text{def}}{=}$	$\langle c, \langle cs\rangle\rangle$	non-empty n-ary product of constructors
$\pi^1 c$	$\stackrel{\text{def}}{=}$	$\pi_1 c$	first projection from n-ary product
$\pi^i c$	$\stackrel{\text{def}}{=}$	$\pi^{i-1}(\pi_2 c) \quad (i > 1)$	ith projection from n-ary product
$\lambda(x{:}c) : c'.e$	$\stackrel{\text{def}}{=}$	$\mathbf{fix}_1\ \_\ (x{:}c) : c'.e\ \mathbf{end}$	non-recursive anonymous function
unit	$\stackrel{\text{def}}{=}$	$\times[\cdot]$	unit type
bool	$\stackrel{\text{def}}{=}$	$+[\mathsf{unit}, \mathsf{unit}]$	boolean type

3.2 IL4 Syntax

3.2.1 IL4 Kinds

κ	::=	$\mathbf{T}$	kind of types
	|	$\mathbf{1}$	unit kind
	|	$\kappa_1 \rightarrow \kappa_2$	function kinds
	|	$\kappa_1 \times \kappa_2$	pair kinds

3.2.2 IL4 Constructors

c	::=	α	constructor variables
	|	$\lambda\alpha{:}\kappa.c$	constructor functions
	|	$c_1 c_2$	constructor applications
	|	$\star$	unit constructor
	|	$\langle c_2, c_2 \rangle$	constructor pairs
	|	$\pi_1 c$	constructor projection left
	|	$\pi_2 c$	constructor projection right
	|	$\mu\alpha{:}\kappa.c$	recursive constructors
	|	int | char | string	built-in base types
	|	$\neg c$	continuations
	|	ref c	references
	|	tagged	extensible tagged unions
	|	tag c	tags
	|	$\times[c_1, \ldots, c_n]$	products
	|	$+[c_1, \ldots, c_n]$	sums
	|	$\forall\alpha{:}\kappa.c$	polymorphic types

3.2.3 IL4 Values

v	::=	x	variables
	|	$\overline{n}$ | *'char'* | *"string"*	integer, character, and string literals
	|	$\mathbf{fix}_i\ fb_1, \ldots, fb_n$ **end**	recursive functions from bundle
	|	$\mathbf{roll}^c\ v$	coercion into recursive type
	|	$(v_1, \ldots, v_n)$	products
	|	$\mathbf{inj}_i^c\ v$	sum injection
	|	$\mathbf{tag}(v_1, v_2)$	injection into type tagged
	|	$\Lambda\alpha{:}\kappa.v$	polymorphic abstraction
	|	$v[c]$	polymorphic application

3.2.4 IL4 Function Bindings

fb	::=	$f\ (x{:}c).e$	function binding

3.2.5 IL4 Expressions

e	::=	**halt** v	initial continuation
	|	$v_1 v_2$	application
	|	**case** v **of** $x_1.e_1, \ldots, x_n.e_n$	sum analysis
	|	**iftagof** v_1 **is** v_2 **then** $x.e$ **else** e	tag analysis
	|	**let** b **in** e	let binding

3.2.6 IL4 Value Bindings

b	::=	$x = v$	value binding
	$\mid$	$x = unop\ v$	primitive unary operations
	$\mid$	$x = v_1\, binop\ v_2$	primitive binary operations
	$\mid$	$x = \mathbf{ref}\, v$	reference cell allocation
	$\mid$	$x = \mathbf{get}\ v$	reference cell dereference
	$\mid$	$x = \mathbf{set}(v_1, v_2)$	reference cell update
	$\mid$	$x = \mathbf{unroll}\ v$	coercion out of recursive type
	$\mid$	$x = \pi_i\ v$	product projection
	$\mid$	$x = \mathbf{newtag}[c]$	extension of type tagged

3.2.7 IL4 Derived Forms

κs	::=	$\cdot \mid \kappa, \kappa s$	kind lists
cs	::=	$\cdot \mid c, cs$	constructor lists

$\times[\cdot]$	$\stackrel{\text{def}}{=}$	$\mathbf{1}$	0-ary product kind
$\times[\kappa, \kappa s]$	$\stackrel{\text{def}}{=}$	$\kappa \times (\times[\kappa])$	non-empty n-ary product of kinds
$\langle\cdot\rangle$	$\stackrel{\text{def}}{=}$	$\star$	0-ary product of constructors
$\langle c, cs\rangle$	$\stackrel{\text{def}}{=}$	$\langle c, \langle cs\rangle\rangle$	non-empty n-ary product of constructors
$\pi^1 c$	$\stackrel{\text{def}}{=}$	$\pi_1 c$	first projection from n-ary product
$\pi^i c$	$\stackrel{\text{def}}{=}$	$\pi^{i-1}(\pi_2 c) \quad (i > 1)$	ith projection from n-ary product
$\lambda x{:}c.e$	$\stackrel{\text{def}}{=}$	$\mathbf{fix}_{1\ \_}\,(x{:}c).e\ \mathbf{end}$	non-recursive anonymous function
unit	$\stackrel{\text{def}}{=}$	$\times[\cdot]$	unit type
bool	$\stackrel{\text{def}}{=}$	$+[\mathsf{unit}, \mathsf{unit}]$	boolean type

3.3 IL5 Syntax

c	::=	$\ldots$	constructors of IL4
	$\mid$	$\exists\alpha{:}\kappa.c$	existential types
v	::=	$\ldots$	values of IL4
	$\mid$	$\mathbf{pack}[c_1, v]$ **as** c_2	pack an existential
e	::=	$\ldots$	expressions of IL4
	$\mid$	**unpack** $\alpha, x = v$ **in** e	unpack an existential

4 CPS Conversion

CPS conversion on constructors is defined by a syntax-directed mapping $\overline{c}$, shown in Figure 3. At most constructors $\overline{\cdot}$ is simply a congruence that pushes the translation down to the next level. The two exceptions are

1. type $\tau_1 \to \tau_2$. Functions become continuations that expect three arguments: a continuation to pass the result to, an exception handler, and the actual function argument.

2. type $\forall\alpha{:}\kappa.c$. Polymorphic functions' bodies are required to be values in IL4, so they must be suspended by a double negation. The translation is roughly equivalent to $\forall\alpha{:}\kappa.\overline{\star \to \overline{c}}$.

$$
\begin{array}{rcl}
\overline{\alpha} & \stackrel{\text{def}}{=} & \alpha \\
\overline{\lambda\alpha{:}\kappa.c} & \stackrel{\text{def}}{=} & \lambda\alpha{:}\kappa.\overline{c} \\
\overline{cc} & \stackrel{\text{def}}{=} & \overline{c}\,\overline{c} \\
\overline{\star} & \stackrel{\text{def}}{=} & \star \\
\overline{\langle c, c\rangle} & \stackrel{\text{def}}{=} & \langle \overline{c}, \overline{c}\rangle \\
\overline{\pi_1 c} & \stackrel{\text{def}}{=} & \pi_1 \overline{c} \\
\overline{\pi_2 c} & \stackrel{\text{def}}{=} & \pi_2 \overline{c} \\
\overline{\mu\alpha{:}\kappa.c} & \stackrel{\text{def}}{=} & \mu\alpha{:}\kappa.\overline{c} \\
\overline{\mathsf{int}} & \stackrel{\text{def}}{=} & \mathsf{int} \\
\overline{\mathsf{char}} & \stackrel{\text{def}}{=} & \mathsf{char} \\
\overline{\mathsf{string}} & \stackrel{\text{def}}{=} & \mathsf{string} \\
\overline{c_1 \to c_2} & \stackrel{\text{def}}{=} & \neg(\times[\neg\overline{c_2}, \neg\mathsf{tagged}, \overline{c_1}]) \\
\overline{\mathsf{ref}\ c} & \stackrel{\text{def}}{=} & \mathsf{ref}\ \overline{c} \\
\overline{\mathsf{tagged}} & \stackrel{\text{def}}{=} & \mathsf{tagged} \\
\overline{\mathsf{tag}\ c} & \stackrel{\text{def}}{=} & \mathsf{tag}\ \overline{c} \\
\overline{\times[c_1, \ldots, c_n]} & \stackrel{\text{def}}{=} & \times[\overline{c_1}, \ldots, \overline{c_n}] \\
\overline{+[c_1, \ldots, c_n]} & \stackrel{\text{def}}{=} & +[\overline{c_1}, \ldots, \overline{c_n}] \\
\overline{\forall\alpha{:}\kappa.c} & \stackrel{\text{def}}{=} & \forall\alpha{:}\kappa.\neg \times [\neg\overline{c}, \neg\mathsf{tagged}]
\end{array}
$$

Figure 3: CPS conversion on constructors

CPS conversion on expressions is formalized using a type-derivation-directed judgment $\Gamma \vdash e : c \rightsquigarrow k, k_{exn}, e'$. This judgment is interpreted as saying that in context Γ, IL3 expression e has type c, and CPS converts to IL4 expression e', which passes its result to a continuation k and sends any exceptions to an exception handler k_{exn}.

The following theorem states the static correctness condition for this translation.

Theorem 1. If $\Gamma \vdash e : c \rightsquigarrow k, k_{exn}, e'$, then $\overline{\Gamma}, k{:}\neg\overline{c}, k_{exn}{:}\neg\mathsf{tagged} \vdash e'$ **ok**. It is useful to keep this theorem in mind when constructing the rules defining the judgment. For reference, the static semantics of IL4 are included in Appendix A.

You will be required to implement the following cases of CPS Conversion

- Binary operators
- Functions
- Function application
- Exception handlers

- Exception raising
- Constructor abstraction
- Constructor application
- Case expressions

5 Closure Conversion

5.1 Constructor Translation

Most types are translated recursively. The exception is functions. Functions are translated so that a tuple containing all of the free variables of a function is passed to the function along with its actual argument. The type of this environment is then abstracted away into an existential so that functions with different free variables, but otherwise the same type, can be used in the same places. We specify a translation $\overline{c}$ from IL4 constructors to IL5 constructors:

$$
\begin{array}{rcl}
\overline{\alpha} & \stackrel{\text{def}}{=} & \alpha \\
\overline{\lambda\alpha{:}\kappa.c} & \stackrel{\text{def}}{=} & \lambda\alpha{:}\kappa.\overline{c} \\
\overline{cc} & \stackrel{\text{def}}{=} & \overline{c}\,\overline{c} \\
\overline{\star} & \stackrel{\text{def}}{=} & \star \\
\overline{\langle c, c\rangle} & \stackrel{\text{def}}{=} & \langle \overline{c}, \overline{c}\rangle \\
\overline{\pi_1 c} & \stackrel{\text{def}}{=} & \pi_1 \overline{c} \\
\overline{\pi_2 c} & \stackrel{\text{def}}{=} & \pi_2 \overline{c} \\
\overline{\mu\alpha{:}\kappa.c} & \stackrel{\text{def}}{=} & \mu\alpha{:}\kappa.\overline{c} \\
\overline{\mathsf{int}} & \stackrel{\text{def}}{=} & \mathsf{int} \\
\overline{\mathsf{char}} & \stackrel{\text{def}}{=} & \mathsf{char} \\
\overline{\mathsf{string}} & \stackrel{\text{def}}{=} & \mathsf{string} \\
\overline{\neg c} & \stackrel{\text{def}}{=} & \exists\alpha{:}\mathbf{T}.\times[\neg(\times[\overline{c},\alpha]),\alpha] \\
\overline{\mathsf{ref}\ c} & \stackrel{\text{def}}{=} & \mathsf{ref}\ \overline{c} \\
\overline{\mathsf{tagged}} & \stackrel{\text{def}}{=} & \mathsf{tagged} \\
\overline{\mathsf{tag}\ c} & \stackrel{\text{def}}{=} & \mathsf{tag}\ \overline{c} \\
\overline{\times[c_1,\ldots,c_n]} & \stackrel{\text{def}}{=} & \times[\overline{c_1},\ldots,\overline{c_n}] \\
\overline{+[c_1,\ldots,c_n]} & \stackrel{\text{def}}{=} & +[\overline{c_1},\ldots,\overline{c_n}] \\
\overline{\forall\alpha{:}\kappa.c} & \stackrel{\text{def}}{=} & \forall\alpha{:}\kappa.\overline{c}
\end{array}
$$

Figure 4: Closure conversion on constructors

5.2 Term Translation

We need to translate each IL4 function so that it takes its environment and unpacks that into the environment variables it is expecting. Additionally, since our functions are mutually recursive, the translation needs to replace its usage of itself and the other functions in the bundle to conform to a closure passing style. The translation we will use has one environment for all of the functions in a bundle, though other methods are possible. However, you should implement the given translation. The translation is specified as a type-directed translation, though you will want to implement it as syntax-directed and do any necessary type-checking as you go. The translation is described by the following judgments:

- $\Gamma \vdash v : c \rightsquigarrow v'$
- $\Gamma \vdash e \ \mathbf{ok} \rightsquigarrow e'$
- $\Gamma \vdash b : x{:}c \rightsquigarrow b'$

You will be required to implement the following cases:

- Constructor Application
- Functions
- Function application
- Case expressions
- Binary operator bindings
- Reference set bindings

5.2.1 Value Translation

$\boxed{\Gamma \vdash v : c \rightsquigarrow v'}$

$$\frac{}{\Gamma \vdash x : c \rightsquigarrow x} \tag{1}$$

$$\frac{}{\Gamma \vdash \overline{n} : c \rightsquigarrow \overline{n}} \tag{2}$$

$$\frac{}{\Gamma \vdash \textit{`char'} : c \rightsquigarrow \textit{`char'}} \tag{3}$$

$$\frac{}{\Gamma \vdash \textit{``string''} : c \rightsquigarrow \textit{``string''}} \tag{4}$$

$$\frac{\begin{array}{l}\Gamma, f_1{:}\neg c_1, \ldots, f_n{:}\neg c_n, x_i{:}c_i \vdash e_i \ \mathbf{ok} \rightsquigarrow e'_i \quad \text{forall}_i\ 1 \le i \le n \\ x_1, \ldots, x_n, y_1, \ldots, y_n, f_1, \ldots, f_n, g_1, \ldots, g_n, env \notin \mathrm{dom}(\Gamma)\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{fix}_j\ [f_i\ (x_i{:}c_i).e_i]_{i=1}^n\ \mathbf{end} : \neg c_j \rightsquigarrow \\ \quad \mathbf{pack}[c_{env}, (\\ \quad\ \mathbf{fix}_j\ [g_i\ (y_i{:} \times [\overline{c_i}, c_{env}]). \\ \quad\quad \mathbf{let}\ x_i = \pi_1\ y_i\ \mathbf{in} \\ \quad\quad \mathbf{let}\ env = \pi_2\ y_i\ \mathbf{in} \\ \quad\quad \mathbf{let}\ z_1 = \pi_1\ env\ \mathbf{in} \\ \quad\quad \vdots \\ \quad\quad \mathbf{let}\ z_n = \pi_n\ env\ \mathbf{in} \\ \quad\quad \mathbf{let}\ f_1 = \mathbf{pack}[c_{env}, (g_1, env)]\ \mathbf{as}\ \overline{\neg c_1}\ \mathbf{in} \\ \quad\quad \vdots \\ \quad\quad \mathbf{let}\ f_n = \mathbf{pack}[c_{env}, (g_n, env)]\ \mathbf{as}\ \overline{\neg c_n}\ \mathbf{in} \\ \quad\quad\ e'_i \\ \quad]_{i=1}^n\ \mathbf{end} \\ \quad , (z_1, \ldots, z_n))]\ \mathbf{as}\ \overline{\neg c_j} \\ \text{where } c_{env} \text{ is } \times [c_1^z, \ldots, c_n^z] \\ \quad \text{forall } z_i : c_i^z \in \Gamma \mid z_i \in FV(\mathbf{fix}_j\ [f_i\ (x_i{:}c_i).e_i]_{i=1}^n\ \mathbf{end})\end{array}} \tag{5}$$

$$\frac{\Gamma \vdash c \equiv \pi^i(\mu\alpha{:}\kappa.c') : \mathbf{T} \quad \Gamma \vdash v : \pi^i([\mu\alpha{:}\kappa.c'/\alpha]c') \rightsquigarrow v'}{\Gamma \vdash \mathbf{roll}^c\ v : c \rightsquigarrow \mathbf{roll}^{\overline{c}}\ v'} \tag{6}$$

$$\frac{\Gamma \vdash v_i : c_i \rightsquigarrow v'_i \quad \text{forall}_i\ 1 \le i \le n}{\Gamma \vdash (v_1, \ldots, v_n) : \times[c_1, \ldots, c_n] \rightsquigarrow (v_1, \ldots, v'_n)} \tag{7}$$

$$\frac{\Gamma \vdash c \equiv +[c_1, \ldots, c_n] : \mathbf{T} \quad \Gamma \vdash v : c_i \rightsquigarrow v'}{\Gamma \vdash \mathbf{inj}_i^c\ v : c \rightsquigarrow \mathbf{inj}_i^{\overline{c}}\ v'} \tag{8}$$

$$\frac{\Gamma \vdash v_1 : \mathsf{tag}\ c \rightsquigarrow v'_1 \quad \Gamma \vdash v_2 : c \rightsquigarrow v'_2}{\Gamma \vdash \mathbf{tag}(v_1, v_2) : \mathsf{tagged} \rightsquigarrow \mathbf{tag}(v'_1, v'_2)} \tag{9}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash v : c \rightsquigarrow v' \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \Lambda\alpha{:}\kappa.v : \forall\alpha{:}\kappa.c \rightsquigarrow \Lambda\alpha{:}\kappa.v'} \tag{10}$$

$$\frac{\Gamma \vdash v : \forall\alpha{:}\kappa.c' \rightsquigarrow v'}{\Gamma \vdash v[c] : [c/\alpha]c' \rightsquigarrow v'[\overline{c}]} \tag{11}$$

5.2.2 Expression Translation

$\boxed{\Gamma \vdash e\ \mathbf{ok} \rightsquigarrow e'}$

$$\frac{\Gamma \vdash v\ \mathbf{ok} \rightsquigarrow v'}{\Gamma \vdash \mathbf{halt}\ v\ \mathbf{ok} \rightsquigarrow \mathbf{halt}\ v'} \tag{12}$$

$$\frac{\Gamma \vdash v_1 : \neg c \rightsquigarrow v'_1 \quad \Gamma \vdash v_2 : c \rightsquigarrow v'_2 \quad \alpha, x, env, f \notin \mathrm{dom}(\Gamma)}{\begin{array}{l}\Gamma \vdash v_1 v_2\ \mathbf{ok} \rightsquigarrow \\ \quad \mathbf{unpack}\ \alpha, x = v'_1\ \mathbf{in} \\ \quad \mathbf{let}\ f = \pi_1\ x\ \mathbf{in} \\ \quad \mathbf{let}\ env = \pi_2\ x\ \mathbf{in} \\ \quad\quad f(v'_2, env)\end{array}} \tag{13}$$

$$\frac{\begin{array}{l}\Gamma \vdash v : +[c_1, \ldots, c_n] \rightsquigarrow v' \\ \Gamma, x_i{:}c_i \vdash e_i \textbf{ ok} \rightsquigarrow e_i' \qquad \text{forall}_i\ 1 \le i \le n \\ x_1, \ldots, x_n \notin \text{dom}(\Gamma)\end{array}}{\begin{array}{c}\Gamma \vdash \textbf{case } v \textbf{ of } x_1.e_1, \ldots, x_n.e_n \textbf{ ok} \rightsquigarrow \\ \textbf{case } v' \textbf{ of } x_1.e_1', \ldots, x_n.e_n'\end{array}} \tag{14}$$

$$\frac{\begin{array}{ll}\Gamma \vdash v_1 : \mathsf{tagged} \rightsquigarrow v_1' & \Gamma \vdash v_2 : \mathsf{tag}\ c \rightsquigarrow v_2' \\ \Gamma, x{:}c \vdash e_1 \textbf{ ok} \rightsquigarrow e_1' & \Gamma \vdash e_2 \textbf{ ok} \rightsquigarrow e_2' \qquad x \notin \text{dom}(\Gamma)\end{array}}{\begin{array}{c}\Gamma \vdash \textbf{iftagof } v_1 \textbf{ is } v_2 \textbf{ then } x.e_1 \textbf{ else } e_2 \textbf{ ok} \rightsquigarrow \\ \textbf{iftagof } v_1' \textbf{ is } v_2' \textbf{ then } x.e_1' \textbf{ else } e_2'\end{array}} \tag{15}$$

$$\frac{\Gamma \vdash b : x{:}c \rightsquigarrow b' \quad \Gamma, x{:}c \vdash e \textbf{ ok} \rightsquigarrow e' \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash \textbf{let } b \textbf{ in } e \textbf{ ok} \rightsquigarrow \textbf{let } b' \textbf{ in } e'} \tag{16}$$

5.2.3 Binding Translation

$\boxed{\Gamma \vdash b : x{:}c \rightsquigarrow b'}$

$$\frac{\Gamma \vdash v : c \rightsquigarrow v'}{\Gamma \vdash x = v : x{:}c \rightsquigarrow x = v'} \tag{17}$$

$$\frac{O_1(\mathit{unop}) = (c, c_r) \quad \Gamma \vdash v : c' \rightsquigarrow v' \quad \Gamma \vdash c \equiv c' : \mathbf{T}}{\Gamma \vdash x = \mathit{unop}\ v : c_r{:}x \rightsquigarrow x = \mathit{unop}\ v'} \tag{18}$$

$$\frac{O_2(\mathit{binop}) = (c_1, c_2, c_r) \quad \Gamma \vdash v_1 : c_1' \rightsquigarrow v_1' \quad \Gamma \vdash v_2 : c_2' \rightsquigarrow v_2' \quad \Gamma \vdash c_1 \equiv c_1' : \mathbf{T} \quad \Gamma \vdash c_2 \equiv c_2' : \mathbf{T}}{\Gamma \vdash x = v_1\, \mathit{binop}\ v_2 : x{:}c_r \rightsquigarrow x = v_2'\, \mathit{binop}\ v_2'} \tag{19}$$

$$\frac{\Gamma \vdash v : c \rightsquigarrow v'}{\Gamma \vdash x = \textbf{ref } v : x{:}\mathsf{ref}\ c \rightsquigarrow x = \textbf{ref } v'} \tag{20}$$

$$\frac{\Gamma \vdash v : \mathsf{ref}\ c \rightsquigarrow v'}{\Gamma \vdash x = \textbf{get } v : x{:}c \rightsquigarrow x = \textbf{get } v'} \tag{21}$$

$$\frac{\Gamma \vdash v_1 : \mathsf{ref}\ c \rightsquigarrow v_1' \quad \Gamma \vdash v_2 : c' \rightsquigarrow v_2' \quad \Gamma \vdash c \equiv c' : \mathbf{T}}{\Gamma \vdash x = \textbf{set}(v_1, v_2) : x{:}\mathsf{unit} \rightsquigarrow x = \textbf{set}(v_1', v_2')} \tag{22}$$

$$\frac{\Gamma \vdash v : \pi_i(\mu\alpha{:}\kappa.c) \rightsquigarrow v'}{\Gamma \vdash x = \textbf{unroll } v : x{:}\pi_i([\mu\alpha{:}\kappa.c/\alpha]c) \rightsquigarrow x = \textbf{unroll } v'} \tag{23}$$

$$\frac{\Gamma \vdash v : \times[c_1, \ldots, c_n] \rightsquigarrow v'}{\Gamma \vdash x = \pi_i\ v : x{:}c_i \rightsquigarrow x = \pi_i\ v'} \tag{24}$$

$$\frac{}{\Gamma \vdash x = \textbf{newtag}[c] : x{:}\mathsf{tag}\ c \rightsquigarrow x = \textbf{newtag}[\overline{c}]} \tag{25}$$

6 CPS Conversion Examples

6.1 Constructors

```
- CPSTop.convert_con IL3Examples.arrow;
  |- int -> char
 ~~> ~*[~char, ~tagged, int]
val it = () : unit
```

6.2 Values

```
- CPSTop.convert_term IL3Examples.const;
  |- "look Mom, no flow control!"
 ~~> k_33, k_exn_34.
     k_33 "look Mom, no flow control!"
     :
     string
val it = () : unit
```

```
- CPSTop.convert_term IL3Examples.function;
  |- fix[1] _f_3 (x_2 : int) : int |-> x_2 end
 ~~> k_35, k_exn_36.
     k_35
     fix[1]
         _f_3 (cx_39 : *[~int, ~tagged, int]) |->
             let k_37 = #1 cx_39
             in
             let k_exn_38 = #2 cx_39 in let x_2 = #3 cx_39 in k_37 x_2
     end
     :
     int -> int
```

6.3 Operations

```
- CPSTop.convert_term IL3Examples.add;
  |- 5 + 7
 ~~> k_40, k_exn_41.
     let k_42 =
             fix[1]
                 __50 (x1_46 : int) |->
                     let k_44 =
                             fix[1]
                                 __49 (x2_47 : int) |->
                                     let y_48 = x1_46 add x2_47 in k_40 y_48
                             end
                     in
                     let k_exn_45 = k_exn_41 in k_44 7
             end
     in
     let k_exn_43 = k_exn_41 in k_42 5
     :
     int
val it = () : unit
```

```
- CPSTop.convert_term IL3Examples.eq;
  |- 10 = 12
 ~~> k_51, k_exn_52.
     let k_53 =
             fix[1]
                 __61 (x1_57 : int) |->
                     let k_55 =
                             fix[1]
                                 __60 (x2_58 : int) |->
                                     let y_59 = x1_57 equals x2_58
                                     in
                                     k_51 y_59
                             end
                     in
                     let k_exn_56 = k_exn_52 in k_55 12
             end
     in
     let k_exn_54 = k_exn_52 in k_53 10
     :
     +[*[], *[]]
val it = () : unit

- CPSTop.convert_term IL3Examples.proj;
  |- #2 ('d', 6)
 ~~> k_72, k_exn_65.
     let k_64 =
             fix[1]
                 __75 (x_73 : *[char, int]) |->
                     let y_74 = #2 x_73 in k_72 y_74
             end
     in
     let k_66 =
             fix[1]
                 __71 (a_62 : char) |->
                     let k_68 =
                             fix[1]
                                 __70 (a_63 : int) |-> k_64 (a_62, a_63)
                             end
                     in
                     let k_exn_69 = k_exn_65 in k_68 6
             end
     in
     let k_exn_67 = k_exn_65 in k_66 'd'
     :
     int
val it = () : unit
```

6.4 Expressions

```
- CPSTop.convert_term IL3Examples.letexp;
  |- let y_17 = 1
     in
```

```
    y_17
 ~~> k_78, k_exn_77.
    let k_76 =
            fix[1]
                __80 (y_17 : int) |-> let k_exn_79 = k_exn_77 in k_78 y_17
            end
    in
    k_76 1
    :
    int
val it = () : unit

- CPSTop.convert_term IL3Examples.app;
  |- let id_4 = fix[1] _f_6 (x_5 : string) : string |-> x_5 end
     in
     id_4 "hello!"
 ~~> k_90, k_exn_82.
     let k_81 =
             fix[1]
                 __96 (id_4 : ~*[~string, ~tagged, string]) |->
                     let k_exn_91 = k_exn_82
                     in
                     let k_86 =
                             fix[1]
                                 __95 (fn_92 : ~*[~string, ~tagged, string]) |->
                                     let k_88 =
                                             fix[1]
                                                 __94 (x_93 : string) |->
                                                     fn_92
                                                     (k_90, k_exn_91, x_93)
                                             end
                                     in
                                     let k_exn_89 = k_exn_91 in k_88 "hello!"
                             end
                     in
                     let k_exn_87 = k_exn_91 in k_86 id_4
             end
     in
     k_81
     fix[1]
         _f_6 (cx_85 : *[~string, ~tagged, string]) |->
             let k_83 = #1 cx_85
             in
             let k_exn_84 = #2 cx_85 in let x_5 = #3 cx_85 in k_83 x_5
     end
     :
     string
val it = () : unit

- CPSTop.convert_term IL3Examples.handleexp;
  |- let tag_22 = newtag [*[]]
     in
```

```
    handle tag (tag_22, ()) with exn_21 |-> exn_21
 ~~> k_110, k_exn_99.
    let k_98 =
            fix[1]
                __113 (tag_22 : tag *[]) |->
                    let k_exn_111 = k_exn_99
                    in
                    let k_104 = k_110
                    in
                    let k_exn_101 =
                            fix[1]
                                __112 (exn_109 : tagged) |-> k_110 exn_109
                            end
                    in
                    let k_100 =
                            fix[1]
                                __108 (x_105 : tag *[]) |->
                                    let k_102 =
                                            fix[1]
                                                __107 (y_106 : *[]) |->
                                                    k_104
                                                    (tag (x_105, y_106))
                                            end
                                    in
                                    let k_exn_103 = k_exn_101 in k_102 ()
                            end
                    in
                    k_100 tag_22
            end
    in
    let x_97 = newtag [*[]] in k_98 x_97
    :
    tagged
val it = () : unit
```

7 Closure Conversion Examples

7.1 Constructors

```
- ClosureTop.convert_con IL4Examples.con1;
  |- ~int
 ~~> Exists a_40:Type. *[~*[int, a_40], a_40]

val it = () : unit
```

7.2 Expressions

```
- ClosureTop.convert_term IL4Examples.eletfun;
  |- let i_4 = 1
     in
     let j_5 = 2
     in
     let f_0 =
             fix[1] f_0 (x_2 : int) |-> let x_2 = i_4 add j_5 in halt x_2 end
     in
     f_0 2
 ~~> let i_41 = 1
     in
     let j_42 = 2
     in
     let f_43 =
             pack[
                 *[int, int],
                 (fix[1]
                      f_48 (x_50 : *[int, *[int, int]]) |->
                          let x_47 = #1 x_50
                          in
                          let env_51 = #2 x_50
                          in
                          let f_46 =
                                  pack[
                                      *[int, int], (f_48, env_51)
                                  ] as
                                  (Exists a_52:Type. *[~*[int, a_52], a_52])
                          in
                          let i_41 = #1 env_51
                          in
                          let j_42 = #2 env_51
                          in
                          let x_49 = i_41 add j_42 in halt x_49
                      end,
                      (i_41, j_42))
             ] as
             (Exists a_53:Type. *[~*[int, a_53], a_53])
     in
     unpack  __55 ,  p_58
      =  f_43
```

```
    in
    let f_57 = #1 p_58 in let env_56 = #2 p_58 in f_57 (2, env_56)
val it = () : unit

- ClosureTop.convert_term IL4Examples.eletfun2;
  |- let i_4 = 1
     in
     let j_5 = 1
     in
     let f_0 =
             fix[2]
                 f_0 (x_2 : *[int, ~int]) |-> let i_4 = #2 x_2 in i_4 j_5
                 g_1 (x_2 : int) |-> f_0 (x_2, g_1)
             end
     in
     f_0 2
 ~~> let i_59 = 1
     in
     let j_60 = 1
     in
     let f_61 =
             pack[
                 *[int],
                 (fix[2]
                      f_68 (x_76 : *[*[int,
                                      Exists a_75:Type.
                                      *[~*[int, a_75], a_75]],
                                    *[int]]) |->
                          let x_65 = #1 x_76
                          in
                          let env_77 = #2 x_76
                          in
                          let f_64 =
                                  pack[
                                      *[int], (f_68, env_77)
                                  ] as
                                  (Exists a_79:Type.
                                   *[~*[*[int,
                                          Exists a_80:Type.
                                          *[~*[int, a_80], a_80]],
                                        a_79],
                                     a_79])
                          in
                          let g_66 =
                                  pack[
                                      *[int], (g_69, env_77)
                                  ] as
                                  (Exists a_78:Type. *[~*[int, a_78], a_78])
                          in
                          let j_60 = #1 env_77
                          in
```

```
                        let i_70 = #2 x_65
                        in
                        unpack  __71 ,  p_74
                         =  i_70
                        in
                        let f_73 = #1 p_74
                        in
                        let env_72 = #2 p_74 in f_73 (j_60, env_72)
                    g_69 (x_85 : *[int, *[int]]) |->
                        let x_67 = #1 x_85
                        in
                        let env_86 = #2 x_85
                        in
                        let f_64 =
                                pack[
                                    *[int], (f_68, env_86)
                                ] as
                                (Exists a_88:Type.
                                 *[~*[*[int,
                                        Exists a_89:Type.
                                        *[~*[int, a_89], a_89]],
                                      a_88],
                                   a_88])
                        in
                        let g_66 =
                                pack[
                                    *[int], (g_69, env_86)
                                ] as
                                (Exists a_87:Type. *[~*[int, a_87], a_87])
                        in
                        let j_60 = #1 env_86
                        in
                        unpack  __81 ,  p_84
                         =  f_64
                        in
                        let f_83 = #1 p_84
                        in
                        let env_82 = #2 p_84 in f_83 ((x_67, g_66), env_82)
                end,
                (j_60))
        ] as
        (Exists a_90:Type. *[~*[int, a_90], a_90])
in
unpack  __92 ,  p_95
 =  f_61
in
let f_94 = #1 p_95 in let env_93 = #2 p_95 in f_94 (2, env_93)
val it = () : unit
```

A IL4 Static Semantics

A.1 Constructor Kinding

$\boxed{\Gamma \vdash c : \kappa}$

$$\frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha : \kappa} \tag{26}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c : \kappa' \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda\alpha{:}\kappa.c : \kappa \to \kappa'} \tag{27}$$

$$\frac{\Gamma \vdash c_1 : \kappa \to \kappa' \quad \Gamma \vdash c_2 : \kappa}{\Gamma \vdash c_1 c_2 : \kappa'} \tag{28}$$

$$\frac{}{\Gamma \vdash \star : \mathbf{1}} \tag{29}$$

$$\frac{\Gamma \vdash c_1 : \kappa_1 \quad \Gamma \vdash c_1 : \kappa_1}{\Gamma \vdash \langle c_1, c_2 \rangle : \kappa_1 \times \kappa_2} \tag{30}$$

$$\frac{\Gamma \vdash c : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_1 c : \kappa_1} \tag{31}$$

$$\frac{\Gamma \vdash c : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_2 c : \kappa_2} \tag{32}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c : \kappa \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \mu\alpha{:}\kappa.c : \kappa} \tag{33}$$

$$\frac{}{\Gamma \vdash \mathsf{int} : \mathbf{T}} \tag{34}$$

$$\frac{}{\Gamma \vdash \mathsf{char} : \mathbf{T}} \tag{35}$$

$$\frac{}{\Gamma \vdash \mathsf{string} : \mathbf{T}} \tag{36}$$

$$\frac{\Gamma \vdash c : \mathbf{T}}{\Gamma \vdash \neg c : \mathbf{T}} \tag{37}$$

$$\frac{\Gamma \vdash c : \mathbf{T}}{\Gamma \vdash \mathsf{ref}\ c : \mathbf{T}} \tag{38}$$

$$\frac{}{\Gamma \vdash \mathsf{tagged} : \mathbf{T}} \tag{39}$$

$$\frac{\Gamma \vdash c : \mathbf{T}}{\Gamma \vdash \mathsf{tag}\ c : \mathbf{T}} \tag{40}$$

$$\frac{\Gamma \vdash c_i : \mathbf{T} \quad \mathrm{forall}_i\ 1 \le i \le n}{\Gamma \vdash \times[c_1, \ldots, c_n] : \mathbf{T}} \tag{41}$$

$$\frac{\Gamma \vdash c_i : \mathbf{T} \quad \mathrm{forall}_i\ 1 \le i \le n}{\Gamma \vdash +[c_1, \ldots, c_n] : \mathbf{T}} \tag{42}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c : \mathbf{T} \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \forall\alpha{:}\kappa.c : \mathbf{T}} \tag{43}$$

A.2 Constructor Equivalence

$\boxed{\Gamma \vdash c : \kappa}$

$$\frac{\Gamma \vdash c : \kappa}{\Gamma \vdash c \equiv c : \kappa} \tag{44}$$

$$\frac{\Gamma \vdash c_2 \equiv c_1 : \kappa}{\Gamma \vdash c_1 \equiv c_2 : \kappa} \tag{45}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2 : \kappa \quad \Gamma \vdash c_2 \equiv c_2 : \kappa}{\Gamma \vdash c_1 \equiv c_3 : \kappa} \tag{46}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2 : \mathbf{T}}{\Gamma \vdash \neg c_1 \equiv \neg c_2 : \mathbf{T}} \tag{47}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2 : \mathbf{T}}{\Gamma \vdash \mathsf{ref}\ c_1 \equiv \mathsf{ref}\ c_2 : \mathbf{T}} \tag{48}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2 : \mathbf{T}}{\Gamma \vdash \mathsf{tag}\ c_1 \equiv \mathsf{tag}\ c_2 : \mathbf{T}} \tag{49}$$

$$\frac{\Gamma \vdash c_i \equiv c_i' : \mathbf{T} \quad \text{forall}_i\ 1 \leq i \leq n}{\Gamma \vdash \times[c_1, \ldots, c_n] \equiv \times[c_1', \ldots, c_n'] : \mathbf{T}} \tag{50}$$

$$\frac{\Gamma \vdash c_i \equiv c_i' : \mathbf{T} \quad \text{forall}_i\ 1 \leq i \leq n}{\Gamma \vdash +[c_1, \ldots, c_n] \equiv +[c_1', \ldots, c_n'] : \mathbf{T}} \tag{51}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c_1 \equiv c_2 : \mathbf{T} \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \forall \alpha{:}\kappa.c_1 \equiv \forall \alpha{:}\kappa.c_2 : \mathbf{T}} \tag{52}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c_1 \equiv c_2 : \kappa' \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda \alpha{:}\kappa.c_1 \equiv \lambda \alpha{:}\kappa.c_2 : \kappa \to \kappa'} \tag{53}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' : \kappa \to \kappa' \quad \Gamma \vdash c_2 \equiv c_2' : \kappa}{\Gamma \vdash c_1 c_2 \equiv c_1' c_2' : \kappa'} \tag{54}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' : \kappa_1 \quad \Gamma \vdash c_2 \equiv c_2' : \kappa_2}{\Gamma \vdash \langle c_1, c_2 \rangle \equiv \langle c_1', c_2' \rangle : \kappa_1 \times \kappa_2} \tag{55}$$

$$\frac{\Gamma \vdash c \equiv c' : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_1 c \equiv \pi_1 c' : \kappa_1} \tag{56}$$

$$\frac{\Gamma \vdash c \equiv c' : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_2 c \equiv \pi_2 c' : \kappa_2} \tag{57}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c_1 \equiv c_2 : \kappa \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \mu \alpha{:}\kappa.c_1 \equiv \mu \alpha{:}\kappa.c_2 : \mathbf{T}} \tag{58}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c' : \kappa' \quad \Gamma \vdash c : \kappa \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\lambda \alpha{:}\kappa.c')c \equiv [c/\alpha]c' : \kappa'} \tag{59}$$

$$\frac{\Gamma \vdash c_1 : \kappa_1}{\Gamma \vdash \pi_1 \langle c_1, c_2 \rangle \equiv c_1 : \kappa_1} \tag{60}$$

$$\frac{\Gamma \vdash c_2 : \kappa_2}{\Gamma \vdash \pi_2 \langle c_1, c_2 \rangle \equiv c_2 : \kappa_2} \tag{61}$$

A.3 Value Typing

$\boxed{\Gamma \vdash v : c}$

$$\frac{\Gamma(x) = c}{\Gamma \vdash x : c} \tag{62}$$

$$\frac{}{\Gamma \vdash \overline{n} : \mathsf{int}} \tag{63}$$

$$\frac{}{\Gamma \vdash \textit{`char'} : \mathsf{char}} \tag{64}$$

$$\frac{}{\Gamma \vdash \textit{``string''} : \mathsf{string}} \tag{65}$$

$$\frac{\Gamma \vdash c_i : \mathbf{T} \quad \Gamma, f_1 : \neg c_1, \cdots, f_n : \neg c_n, x_i : c_i \vdash e_i \ \mathbf{ok} \quad \text{forall}_i\ 1 \leq i \leq n}{\Gamma \vdash \mathbf{fix}_j\ [f_i\ (x_j{:}c_j).e_i]_{i=1}^{n}\ \mathbf{end} : \neg c_j} \tag{66}$$

$$\frac{\Gamma \vdash c \equiv \pi^i(\mu\alpha{:}\kappa.c') : \mathbf{T} \quad \Gamma \vdash v : \pi^i([\mu\alpha{:}\kappa.c'/\alpha]c')}{\Gamma \vdash \mathbf{roll}^c\ v : c} \tag{67}$$

$$\frac{\Gamma \vdash v_i : c_i \quad \text{forall}_i\ 1 \leq i \leq n}{\Gamma \vdash (v_1, \ldots, v_n) : \times[c_1, \ldots, c_n]} \tag{68}$$

$$\frac{\Gamma \vdash c_{sum} \equiv \times[c_1, \ldots, c_n] : \mathbf{T} \quad \Gamma \vdash v : c_i}{\Gamma \vdash \mathbf{inj}_i^{c_{sum}}\ v : c_{sum}} \tag{69}$$

$$\frac{\Gamma \vdash v_1 : \mathsf{tag}\ c \quad \Gamma \vdash v_2 : c}{\Gamma \vdash \mathbf{tag}(v_1, v_2) : \mathsf{tagged}} \tag{70}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash v : c \quad \alpha \notin \text{dom}(\Gamma)}{\Gamma \vdash \Lambda\alpha{:}\kappa.v : \forall\alpha{:}\kappa.c} \tag{71}$$

$$\frac{\Gamma \vdash v : \forall\alpha{:}\kappa.c' \quad \Gamma \vdash c : \kappa}{\Gamma \vdash v[c] : [c/\alpha]c'} \tag{72}$$

$$\frac{\Gamma \vdash v : c' \quad \Gamma \vdash c' \equiv c : \mathbf{T}}{\Gamma \vdash v : c} \tag{73}$$

A.4 Expression Formation

$$\frac{\Gamma \vdash v : c}{\Gamma \vdash \mathbf{halt}\ v\ \mathbf{ok}} \tag{74}$$

$$\frac{\Gamma \vdash v_1 : \neg c \quad \Gamma \vdash v_2 : c}{\Gamma \vdash v_1 v_2\ \mathbf{ok}} \tag{75}$$

$$\frac{\Gamma \vdash v : +[c_1, \ldots, c_n] \quad \Gamma, x_i : c_i \vdash e_i\ \mathbf{ok} \quad \text{forall}_i\ 1 \leq i \leq n}{\Gamma \vdash \mathbf{case}\ v\ \mathbf{of}\ x_1.e_1, \ldots, x_n.e_n\ \mathbf{ok}} \tag{76}$$

$$\frac{\Gamma \vdash v_1 : \mathsf{tagged} \quad \Gamma \vdash v_2 : \mathsf{tag}\ c \quad \Gamma, x{:}c \vdash e_1\ \mathbf{ok} \quad \Gamma \vdash e_2\ \mathbf{ok} \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash \mathbf{iftagof}\ v_1\ \mathbf{is}\ v_2\ \mathbf{then}\ x.e_1\ \mathbf{else}\ e_2\ \mathbf{ok}} \tag{77}$$

$$\frac{\Gamma \vdash b : x{:}c \quad \Gamma, x{:}c \vdash e\ \mathbf{ok} \quad x \notin \text{dom}(\Gamma)}{\Gamma \vdash \mathbf{let}\ b\ \mathbf{in}\ e\ \mathbf{ok}} \tag{78}$$

A.5 Bindings

$\boxed{\Gamma \vdash b : x{:}c}$

$$\frac{\Gamma \vdash v : c}{\Gamma \vdash x = v : x{:}c} \tag{79}$$

$$\frac{O_1(unop) = (c, c_r)\Gamma \vdash v : c' \quad \Gamma \vdash c \equiv c' : \mathbf{T}}{\Gamma \vdash x = unop\ v : x{:}c_r} \tag{80}$$

$$\frac{O_2(binop) = (c_1, c_2, c_r) \quad \Gamma \vdash v_1 : c_1' \quad \Gamma \vdash v_2 : c_2' \quad c_1 \vdash c_1' \equiv \mathbf{T} : \quad c_2 \vdash c_2' \equiv \mathbf{T} :}{\Gamma \vdash x = v_1\, binop\ v_2 : x{:}c_r} \tag{81}$$

$$\frac{\Gamma \vdash v : c}{\Gamma \vdash x = \mathbf{ref}\ v : x{:}\mathsf{ref}\ c} \tag{82}$$

$$\frac{\Gamma \vdash v : \mathsf{ref}\ c}{\Gamma \vdash x = \mathbf{get}\ v : x{:}c} \tag{83}$$

$$\frac{\Gamma \vdash v_1 : \mathsf{ref}\ c \quad \Gamma \vdash v_2 : c}{\Gamma \vdash x = \mathbf{set}(v_1, v_2) : x{:}\mathsf{unit}} \tag{84}$$

$$\frac{\Gamma \vdash v : \pi^i(\mu\alpha{:}\kappa.c)}{\Gamma \vdash x = \mathbf{unroll}\ v : x{:}\pi^i([\mu\alpha{:}\kappa.c/\alpha]c)} \tag{85}$$

$$\frac{\Gamma \vdash v : \times[c_1, \ldots, c_n]}{\Gamma \vdash x = \pi_i\ v : x{:}c_i} \tag{86}$$

$$\frac{}{\Gamma \vdash x = \mathbf{newtag}[c] : x{:}\mathsf{tag}\ c} \tag{87}$$

B IL5 Static Semantics

As mentioned, the IL5 statics semantics are the same as the IL4 static semantics except for the addition of existentials and the closure requirement for functions. Here we give only the relevant rules.

$$\frac{\Gamma, \alpha{:}\kappa \vdash c : \mathbf{T} \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \exists\alpha{:}\kappa.c : \mathbf{T}} \tag{88}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c_1 \equiv c_2 : \kappa' \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \exists\alpha{:}\kappa.c_1 \equiv \exists\alpha{:}\kappa.c_2 : \mathbf{T}} \tag{89}$$

$$\frac{\Gamma \vdash c' \equiv \exists\alpha{:}\kappa.c'' : \mathbf{T} \quad \Gamma \vdash c : \kappa \quad \Gamma \vdash v : [c/\alpha]c''}{\Gamma \vdash \mathbf{pack}[c, v]\ \mathbf{as}\ c' : c'} \tag{90}$$

$$\frac{\Gamma \vdash v : \exists\alpha{:}\kappa.c \quad \Gamma, \alpha{:}\kappa, x{:}c \vdash e\ \mathbf{ok} \quad \alpha, x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathbf{unpack}\ \alpha, x = v\ \mathbf{in}\ e\ \mathbf{ok}} \tag{91}$$