Typing Dual System FC and Sequent Core

August 4, 2015

1 Syntax

$$
\begin{array}{rll}
x, y, z, f, g, h, K \in Var & & \\
q, r \in KontVar & & \\
j \in JumpVar & & \\
a, b, T \in TypeVar & & \\
c \in Command & ::= & \mathbf{let}\ bind\ \mathbf{in}\ c \mid \langle v \| k \rangle \mid \langle \mathbf{jump}\ \overrightarrow{v_n}^n \| j \rangle \\
v \in Term & ::= & x \mid \lambda x{:}\tau.v \mid \mu q{:}\tau.c \mid lit \mid \tau \mid \gamma \\
k \in Kont & ::= & \mathbf{ret}\ q \mid v \cdot k \mid \gamma \triangleleft k \mid \mathbf{case}\ \mathbf{as}\ x{:}\tau\ \mathbf{of}\ \overrightarrow{alt_n}^n \\
bind \in Binding & ::= & bp \mid \mathbf{rec}\ \left\{ \overrightarrow{bp_n}^n \right\} \\
bp \in BindPair & ::= & \mathbf{val}\ x{:}\tau = v \mid \mathbf{cont}\ j{:}\tau = \lambda \overrightarrow{x_n{:}\tau_n}^n.c \\
alt \in Alternative & ::= & \_ \to c \mid K\ \overrightarrow{x_n{:}\tau_n}^n \to c \mid lit \to c \\
\tau \in Type & ::= & \ldots \\
\kappa \in Kind & ::= & \ldots \\
\gamma \in Coercion & ::= & \ldots \\
decl \in Declaration & ::= & \ldots \\
pgm \in Program & ::= & decl; pgm \mid c
\end{array}
$$

Figure 1: Syntax of Dual System FC

The syntax for Dual System FC are shown in Figure 1. Types, kinds, coercions, and declarations are unchanged by the sequent calculus representation, so they are elided here. Note that data constructors, written K, are treated as a special sort of variable in the syntax, and additionally type constructors, written T, are treated as a special sort of type variable. We use some conventional shorthands to make programs easier to read:

- If the type annotations on variables, *ie* the τ in $x{:}\tau$, are not important to a particular example, we will often omit them.

- The function call constructor $\_ \cdot \_$ associates to the right, so $1 \cdot 2 \cdot 3 \cdot \mathbf{ret}\, q$ is the same as $1 \cdot (2 \cdot (3 \cdot \mathbf{ret}\, q))$. Similarly, coercion continuations associate to the right as well, so that $\gamma_1 \lhd \gamma_2 \lhd \mathbf{ret}\, q$ is the same as $\gamma_1 \lhd (\gamma_2 \lhd \mathbf{ret}\, q)$. Both function calls and coercions share the same precedence and may be intermixed, so that $1 \cdot \gamma_2 \lhd 3 \cdot \mathbf{ret}\, q$ is the same as $1 \cdot (\gamma_2 \lhd (3 \cdot \mathbf{ret}\, q))$.

- We will not always write the binding variable $x{:}\tau$ in the case continuation $\mathbf{case}\, \mathbf{as}\, x{:}\tau\, \mathbf{of}\, \overrightarrow{alt_n}^n$ when it turns out that x is never referenced in $alts$ or $alts$, instead writing $\mathbf{case}\, \overrightarrow{alt_n}^n$. If instead $x{:}\tau$ is only referenced in the default alternative $\_ \to c$ in $\overrightarrow{alt_n}^n$, we will prefer to write $x{:}\tau$ in place of the wildcard $\_$ pattern. This often arises in a case continuation with *only* a default alternative, $\mathbf{case}\, \mathbf{as}\, x{:}\tau\, \mathbf{of}\, \_ \to c$, which we write as the shortened $\mathbf{case}\, x{:}\tau \to c$.

2 Scope and exit analysis

The scoping rules for variables are shown in Figures 2 and 3, where the rules for scoping inside types, kinds, coercions, and declarations are elided. Continuation and jump variables are treated differently from the other sorts of variables, being placed in a separate environment Δ, in order to prevent non-functional uses of control flow.

Besides the normal rules for checking variable scope, these rules effectively also perform an *exit analysis* on a program (bindings, terms, commands, *etc*). The one major restriction that we enforce is that terms must always have a *unique* exit point and cannot jump outside their scope. The intuition is:

> Terms cannot contain any references to free continuation or jump variables.

This restriction makes sure that λ-abstractions cannot close over continuation variables available from its context, so that bound continuations and jump variables do not escape through a returned λ-abstraction. Thus, all the jump variables used within a λ-abstraction must be local to that λ-abstraction. Additionally, in all computations $\mu r.c$, the underlying command c has precisely one unique exit point, r, which names the returned result of the computation. Any jump variables referenced inside of c must be local to the term itself, and not refer to its enclosing scope.

If the command c inside the well-scoped term $\mu r.c$ stops execution with some value V sent to some continuation variable q, then we know that:

- q must be equal to r, due to the fact that r is the only allowable free continuation variable inside of c, and

- r does not appear free inside the resulting value V, again due to the scoping rules for continuation variables inside of a command.

$$\begin{array}{ll} \Gamma \in \mathit{Environment} & ::= \varepsilon \mid \Gamma, x \mid \Gamma, a \\ \Delta \in \mathit{KoEnvironment} & ::= q \mid \Delta, j \end{array}$$

Term scoping: $\Gamma \vdash_{tm} v\,\mathrm{ok}$

$$\frac{x \in \Gamma}{\Gamma \vdash_{tm} x\,\mathrm{ok}} \qquad \frac{}{\Gamma \vdash_{tm} \mathit{lit}\,\mathrm{ok}} \qquad \frac{\Gamma; q \vdash_{cmd} c\,\mathrm{ok}}{\Gamma \vdash_{tm} \mu q.c\,\mathrm{ok}} \qquad \frac{\Gamma \vdash_{cn} \gamma\,\mathrm{ok}}{\Gamma \vdash_{tm} \gamma\,\mathrm{ok}} \qquad \frac{\Gamma \vdash_{ty} \tau\,\mathrm{ok}}{\Gamma \vdash_{tm} \tau\,\mathrm{ok}}$$

$$\frac{\Gamma, x \vdash_{tm} v\,\mathrm{ok}}{\Gamma \vdash_{tm} \lambda x.v\,\mathrm{ok}} \qquad \frac{\Gamma, a \vdash_{tm} v\,\mathrm{ok}}{\Gamma \vdash_{tm} \lambda a.v\,\mathrm{ok}}$$

Continuation scoping: $\Gamma; \Delta \vdash_{ko} k\,\mathrm{ok}$

$$\frac{q \in \Delta}{\Gamma; \Delta \vdash_{ko} \mathbf{ret}\, q\,\mathrm{ok}} \qquad \frac{\Gamma \vdash_{tm} v\,\mathrm{ok} \quad \Gamma; \Delta \vdash_{ko} k\,\mathrm{ok}}{\Gamma; \Delta \vdash_{ko} v \cdot k\,\mathrm{ok}} \qquad \frac{\Gamma \vdash_{ty} \tau\,\mathrm{ok} \quad \Gamma; \Delta \vdash_{ko} k\,\mathrm{ok}}{\Gamma; \Delta \vdash_{ko} \tau \cdot k\,\mathrm{ok}}$$

$$\frac{\Gamma \vdash_{cn} \gamma\,\mathrm{ok} \quad \Gamma; \Delta \vdash_{ko} k\,\mathrm{ok}}{\Gamma; \Delta \vdash_{ko} \gamma \triangleleft k\,\mathrm{ok}} \qquad \frac{\overrightarrow{\Gamma, x; \Delta \vdash_{alt} \mathit{alt}_n\,\mathrm{ok}}^{n}}{\Gamma; \Delta \vdash_{ko} \mathbf{case\ as}\, x\, \mathbf{of}\, \overrightarrow{\mathit{alt}_n}^{n}\,\mathrm{ok}}$$

Command scoping: $\Gamma; \Delta \vdash_{cmd} c\,\mathrm{ok}$

$$\frac{\Gamma; \Delta \vdash_{bind} \mathit{bind} : (\Gamma'; \Delta') \quad \Gamma, \Gamma'; \Delta, \Delta' \vdash_{cmd} c\,\mathrm{ok}}{\Gamma; \Delta \vdash_{cmd} \mathbf{let}\, \mathit{bind}\, \mathbf{in}\, c\,\mathrm{ok}}$$

$$\frac{\Gamma \vdash_{tm} v\,\mathrm{ok} \quad \Gamma; \Delta \vdash_{ko} k\,\mathrm{ok}}{\Gamma; \Delta \vdash_{cmd} \langle v \| k \rangle\,\mathrm{ok}} \qquad \frac{j \in \Delta \quad \overrightarrow{\Gamma \vdash_{tm} v_n\,\mathrm{ok}}^{n}}{\Gamma; \Delta \vdash_{cmd} \langle \mathbf{jump}\, \overrightarrow{v_n}^{n} \| j \rangle\,\mathrm{ok}}$$

Further rules for $\Gamma \vdash_{ty} \tau\,\mathrm{ok}$, $\Gamma \vdash_{ki} \kappa\,\mathrm{ok}$, *and* $\Gamma \vdash_{cn} \gamma\,\mathrm{ok}$

Figure 2: Scope and exit analysis for terms, continuations, and commands

Binding scoping: $\Gamma; \Delta \vdash_{bind} bind : (\Gamma'; \Delta')$ and $\Gamma; \Delta \vdash_{bp} bp : (\Gamma'; \Delta')$

$$\frac{\Gamma; \Delta \vdash_{bp} bp : (\Gamma'; \Delta')}{\Gamma; \Delta \vdash_{bind} bp : (\Gamma'; \Delta')} \qquad \frac{\overrightarrow{\Gamma, \Gamma'; \Delta, \Delta' \vdash_{bp} bp_n : (\Gamma'_n; \Delta'_n)}^{n} \quad \Gamma' = \overrightarrow{\Gamma'_n}^{n} \quad \Delta' = \overrightarrow{\Delta'_n}^{n}}{\Gamma; \Delta \vdash_{bind} \mathbf{rec} \left\{ \overrightarrow{bp_n}^{n} \right\} : (\Gamma'; \Delta')}$$

$$\frac{\Gamma \vdash_{tm} v \,\mathrm{ok}}{\Gamma; \Delta \vdash_{bp} \mathbf{val}\, x = v : (x; \varepsilon)} \qquad \frac{\Gamma, \overrightarrow{x_n}^{n}; \Delta \vdash_{cmd} c \,\mathrm{ok}}{\Gamma; \Delta \vdash_{bp} \mathbf{cont}\, j = \lambda \overrightarrow{x_n}^{n}.c : (\varepsilon; j)}$$

Alternative scoping: $\Gamma; \Delta \vdash_{alt} alt \,\mathrm{ok}$

$$\frac{\Gamma; \Delta \vdash_{cmd} c \,\mathrm{ok}}{\Gamma; \Delta \vdash_{alt} \_ \to c \,\mathrm{ok}} \qquad \frac{\Gamma; \Delta \vdash_{cmd} c \,\mathrm{ok}}{\Gamma; \Delta \vdash_{alt} lit \to c \,\mathrm{ok}} \qquad \frac{\Gamma, \overrightarrow{x_i}^{i}; \Delta \vdash_{cmd} c \,\mathrm{ok}}{\Gamma; \Delta \vdash_{alt} K\ \overrightarrow{x_i}^{i} \to c \,\mathrm{ok}}$$

Program scoping: $\Gamma; \Delta \vdash_{pgm} pgm \,\mathrm{ok}$

$$\frac{\Gamma \vdash_{decl} decl : \Gamma' \quad \Gamma, \Gamma'; \Delta \vdash_{pgm} c \,\mathrm{ok}}{\Gamma; \Delta \vdash_{pgm} decl; pgm \,\mathrm{ok}} \qquad \frac{\Gamma; \Delta \vdash_{cmd} c \,\mathrm{ok}}{\Gamma; \Delta \vdash_{pgm} c \,\mathrm{ok}}$$

Further rules for $\Gamma \vdash_{decl} decl : \Gamma'$.

Figure 3: Scope and exit analysis for bindings, alternatives, and programs

In the simple case, this means execution of the term $\mu r.c$ yields $\mu r.\langle V \| \mathbf{ret}\, r \rangle$, which η-reduces to just the value V by the previously mentioned reasoning. Thus, evaluating a term always results in a unique value.

Notice that these scoping rules, while not very complex, still manage to tell us something about the expressive capabilities of the language. For example, we syntactically permit value and continuation bindings within the same recursive block, but can they mutually call one another? It turns out that these scoping rules disallow any sort of interesting mutual recursion between terms and continuations because terms are *prevented* from referencing continuations within their surrounding (or same) binding environment.

For example, in a simple case where we have the recursive bindings:

$$\mathbf{rec}\,\{\mathbf{val}\, f = \lambda x.v; \mathbf{cont}\, j = \lambda y.c\}$$

then by the scoping rules, q may call f through c, but f cannot jump back to j in v because $\lambda x.tm$ cannot contain the free reference to j. Therefore, since there is no true mutual recursion between both f and j, we can break the recursive bindings into two separate blocks with the correct scope:

$$\mathbf{rec}\,\{\mathbf{val}\, f = \lambda x.v\}\,; \mathbf{rec}\,\{\mathbf{cont}\, j = \lambda y.c\}$$

While we do not syntactically enforce this restriction, it would not case any loss of expressiveness. Indeed, we could further the partitions into

1. first, the list of value bindings, and

2. second, the list of continuation bindings,

since continuations can refer to previously bound terms but not vice versa. However, we do not make this distinction here.

3 Type checking

The typing rules for Dual System FC are given in Figures 4, 5, and 6. The type of a term classifies the results that it might produce, and the type of a continuation classifies the results that it expects to consume. Commands do not have a type; they are just ok to run. The eventual result of a command is returned through the unique return continuation variable available in its environment. Likewise, a program is a consistent block of code that is capable of running, meaning that a program is a command that runs with respect to some top-level declarations that introduce type information (data types, type synonyms, and axioms). The normal way to type-check a top-level program is $\Gamma_0; exit : \tau \vdash_{pgm} pgm\,\mathrm{ok}$, where Γ_0 specifies any primitive types and values provided by the run-time environment, and $exit : \tau$ is the single, top-level exit path out of the program that expects a τ result.

Compared with System FC, more of the typing rules enjoy the *sub-formula property*, meaning that the types appearing in a premise above the line of a rule appear somewhere below the line. This is a natural consequence of the sequent calculus as a logic, and was one of the primary motivations for its original development. The expected rules violating the sub-formula property are the various *cut* rules for commands that cancel out arbitrary types, where the **let**-form of command allows us to perform multiple cuts simultaneously. The other interesting violators of note are:

- The rule for a polymorphic call-stack, $\tau_1 \cdot k : \forall a{:}\kappa.\tau_2$, which substitutes the specified type τ_1 in for the variable a in τ_2 to get the type for the continuation. This rule does not have the sub-formula property since $\tau_2[\tau_1/a]$ is a new type generated by the substitution.

  Since universal quantification is dual to existential quantification, the polymorphic call-stack is dual to the existential pair $(\tau_1, v) : \exists a{:}\kappa.\tau_2$, and shares the same properties. In particular, $\tau_1 \cdot k$ does not have a *unique* type. For example, given the continuation variable r of type $Bool$, then the polymorphic call-stack $Int \cdot 1 \cdot 2 \cdot \mathbf{ret}\, q$ can be given the types $\forall a{:}\star.a \to a \to Bool$, $\forall a{:}\star.Int \to a \to Bool$, and so on. So following the bottom-up preference of a sequent calculus presentation, it is easy to type check a polymorphic call-stack if we already know the type of function it expects, but in general it is hard to guess its type.

- The rules for pattern matching on data types almost suffers from the same issue as for polymorphic call-stacks, due to substitution of the choice for

$$\begin{array}{ll} \Gamma \in \mathit{Environment} & ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, a : \kappa \\ \Delta \in \mathit{KoEnvironment} & ::= q : \tau \mid \Delta, j : \tau \end{array}$$

Term typing: $\Gamma \vdash_{tm} v : \tau$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash_{tm} x : \tau} \qquad \frac{\tau = \mathit{literalType}(\mathit{lit})}{\Gamma \vdash_{tm} \mathit{lit} : \tau} \qquad \frac{\Gamma \vdash_{cn} \gamma : \tau_1 \sim \tau_2}{\Gamma \vdash_{tm} \gamma : \tau_1 \sim \tau_2} \qquad \frac{\Gamma \vdash_{ty} \tau : \kappa}{\Gamma \vdash_{tm} \tau : \kappa}$$

$$\frac{\Gamma; q : \tau \vdash_{cmd} c\,\mathrm{ok}}{\Gamma \vdash_{tm} \mu q{:}\tau.c : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash_{tm} v : \tau_2}{\Gamma \vdash_{tm} \lambda x{:}\tau_1.v : \tau_1 \to \tau_2} \qquad \frac{\Gamma, a : \kappa \vdash_{tm} v : \tau}{\Gamma \vdash_{tm} \lambda a{:}\kappa.v : \forall a{:}\kappa.\tau}$$

Continuation typing: $\Gamma; \Delta \vdash_{ko} k : \tau$

$$\frac{q : \tau \in \Delta}{\Gamma; \Delta \vdash_{ko} \mathbf{ret}\, q : \tau} \qquad \frac{\Gamma \vdash_{tm} v : \tau_1 \quad \Gamma; \Delta \vdash_{ko} k : \tau_2}{\Gamma; \Delta \vdash_{ko} v \cdot k : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_{ty} \tau_1 : \kappa \quad \Gamma; \Delta \vdash_{ko} k : \tau_2[\tau_1/a]}{\Gamma; \Delta \vdash_{ko} \tau_1 \cdot k : \forall a{:}\kappa.\tau_2}$$

$$\frac{\Gamma \vdash_{cn} \gamma : \tau_1 \sim \tau_2 \quad \Gamma; \Delta \vdash_{ko} k : \tau_2}{\Gamma; \Delta \vdash_{ko} \gamma \triangleleft k : \tau_1} \qquad \frac{\overrightarrow{\Gamma, x : \tau; \Delta \vdash_{alt} \mathit{alt}_n : \tau}^{n}}{\Gamma; \Delta \vdash_{ko} \mathbf{case}\,\mathbf{as}\, x{:}\tau\, \mathbf{of}\, \overrightarrow{\mathit{alt}_n}^{n} : \tau}$$

Command typing: $\Gamma; \Delta \vdash_{cmd} c\,\mathrm{ok}$

$$\frac{\Gamma; \Delta \vdash_{bind} \mathit{bind} : (\Gamma'; \Delta') \quad \Gamma, \Gamma'; \Delta, \Delta' \vdash_{cmd} c\,\mathrm{ok}}{\Gamma; \Delta \vdash_{cmd} \mathbf{let}\, \mathit{bind}\, \mathbf{in}\, c\,\mathrm{ok}}$$

$$\frac{\Gamma \vdash_{tm} v : \tau \quad \Gamma; \Delta \vdash_{ko} k : \tau}{\Gamma; \Delta \vdash_{cmd} \langle v \| k \rangle\,\mathrm{ok}} \qquad \frac{j : \tau \in \Delta \quad \Gamma \vdash_{jtm} \mathbf{jump}\, \overrightarrow{v_n}^{n} : \tau}{\Gamma; \Delta \vdash_{cmd} \langle \mathbf{jump}\, \overrightarrow{v_n}^{n} \| j \rangle\,\mathrm{ok}}$$

Further rules for $\Gamma \vdash_{ty} \tau : \kappa$, $\Gamma \vdash_{ki} \kappa : \delta$, *and* $\Gamma \vdash_{cn} \gamma : \tau_1 \sim \tau_2$.

Figure 4: Type checking rules for terms, continuations, and commands

Binding typing: $\Gamma; \Delta \vdash_{bind} bind : (\Gamma'; \Delta')$ and $\Gamma; \Delta \vdash_{bp} bp : (\Gamma'; \Delta')$

$$\frac{\Gamma; \Delta \vdash_{bp} bp : (\Gamma'; \Delta')}{\Gamma; \Delta \vdash_{bind} bp : (\Gamma'; \Delta')} \qquad \frac{\overrightarrow{\Gamma, \Gamma'; \Delta, \Delta' \vdash_{bp} bp_n : (\Gamma'_n; \Delta'_n)}^{n} \quad \Gamma' = \overrightarrow{\Gamma'_n}^{n} \quad \Delta' = \overrightarrow{\Delta'_n}^{n}}{\Gamma; \Delta \vdash_{bind} \mathbf{rec} \left\{ \overrightarrow{bp_n}^{n} \right\} : (\Gamma'; \Delta')}$$

$$\frac{\Gamma \vdash_{tm} v : \tau}{\Gamma; \Delta \vdash_{bp} \mathbf{val}\, x{:}\tau = v : (x : \tau; \varepsilon)} \qquad \frac{\Gamma; \Delta \vdash_{jko} \lambda \overrightarrow{x_n{:}\tau_n}^{n}.c : \tau}{\Gamma; \Delta \vdash_{bp} \mathbf{cont}\, j = \lambda \overrightarrow{x_n{:}\tau_n}^{n}.c : (\varepsilon; j : \tau)}$$

Alternative typing: $\Gamma; \Delta \vdash_{alt} alt : \tau$

$$\frac{\Gamma; \Delta \vdash_{cmd} c\,\mathrm{ok}}{\Gamma; \Delta \vdash_{alt} \_ \to c : \tau} \qquad \frac{\Gamma \vdash_{tm} lit : \tau \quad \Gamma; \Delta \vdash_{cmd} c\,\mathrm{ok}}{\Gamma; \Delta \vdash_{alt} lit \to c : \tau}$$

$$\frac{\begin{array}{l} K : \overrightarrow{\forall a_m{:}\kappa_m.}^{m} \overrightarrow{\forall b_{m'}{:}\kappa'_{m'}.}^{m'} \overrightarrow{\tau_n \to}^{n} T\ \overrightarrow{a_m}^{m} \in \Gamma \\ \Gamma, \overrightarrow{b_{m'} : \theta(\kappa'_{m'})}^{m'}, \overrightarrow{x_n : \theta(\tau_n)}^{n}; \Delta \vdash_{cmd} c\,\mathrm{ok} \qquad \theta = \overrightarrow{[\tau'_m/a_m]}^{m} \end{array}}{\Gamma; \Delta \vdash_{alt} K\ \overrightarrow{b_{m'}{:}\theta(\kappa'_{m'})}^{m'}\ \overrightarrow{x_n{:}\theta(\tau_n)}^{n} \to c : T\ \overrightarrow{\tau'_m}^{m}}$$

Program typing: $\Gamma; \Delta \vdash_{pgm} pgm\,\mathrm{ok}$

$$\frac{\Gamma \vdash_{decl} decl : \Gamma' \quad \Gamma, \Gamma'; \Delta \vdash_{pgm} pgm\,\mathrm{ok}}{\Gamma; \Delta \vdash_{pgm} decl; pgm\,\mathrm{ok}} \qquad \frac{\Gamma; \Delta \vdash_{cmd} c\,\mathrm{ok}}{\Gamma; \Delta \vdash_{pgm} c\,\mathrm{ok}}$$

Further rules for $\Gamma \vdash_{decl} decl : \Gamma'$.

Figure 5: Type checking rules for bindings, alternatives, and programs

Term argument typing: $\Gamma \vdash_{jtm} \mathbf{jump}\, \overrightarrow{v_n}^n : \tau$

$$\frac{}{\Gamma \vdash_{jtm} \mathbf{jump} : Unit\#} \qquad \frac{\Gamma \vdash_{tm} v' : \tau' \quad \Gamma \vdash_{jtm} \mathbf{jump}\, \overrightarrow{v_n}^n : \tau}{\Gamma \vdash_{jtm} \mathbf{jump}\, v', \overrightarrow{v_n}^n : \tau' \times_\# \tau}$$

$$\frac{\Gamma \vdash_{ty} \tau' : \kappa' \quad \Gamma \vdash_{jtm} \mathbf{jump}\, \overrightarrow{v_n}^n : \tau[\tau'/a]}{\Gamma \vdash_{jtm} \mathbf{jump}\, \tau', \overrightarrow{v_n}^n : \exists_\# a{:}\kappa.\tau}$$

Parameterized continuation typing: $\Gamma; \Delta \vdash_{jko} \lambda\overrightarrow{x_n{:}\tau_n}^n.c : \tau$

$$\frac{\Gamma; \Delta \vdash_{cmd} c\,\mathrm{ok}}{\Gamma; \Delta \vdash_{jko} \lambda.c : Unit\#} \qquad \frac{\Gamma, y : \tau'; \Delta \vdash_{jko} \lambda\overrightarrow{x_n{:}\tau_n}^n.c : \tau}{\Gamma; \Delta \vdash_{jko} \lambda y{:}\tau'\ \overrightarrow{x_n{:}\tau_n}^n.c : \tau' \times_\# \tau}$$

$$\frac{\Gamma, a : \kappa; \Delta \vdash_{jko} \lambda\overrightarrow{x_n{:}\tau_n}^n.c : \tau}{\Gamma; \Delta \vdash_{jko} \lambda a{:}\kappa\ \overrightarrow{x_n{:}\tau_n}^n.c : \exists_\# a{:}\kappa.\tau}$$

Figure 6: Type checking rules for parameterized continuations and their arguments

polymorphic type variables. However, the type annotations on variables bound by pattern matching already specify the specialized types, so the issue is avoided.

Also note that the problem with existential pairs (or in general, existential data structures) mentioned in the previous point is avoided on the term side. This is because we do not represent data structures directly, as a fully-applied constructor like (τ, v), but rather we represent them indirectly as a constructor evaluated with a polymorphic call-stack, $\langle (,) \| \tau \cdot v \cdot \mathbf{ret}\, q \rangle$. That means that all the troubles with checking user-defined existential data structures are contained solely in polymorphic call stacks.

- The rule for coercion continuations, $\gamma \triangleleft k : \tau_1$, in which the type of the continuation k is hidden by the coercion. Interestingly, the typing rules for casting resembles a special sort of function application, both with terms in System FC and continuations in Dual System FC.

Due to the difficulty of inferring the type of a polymorphic call-stack, the type checking algorithm for Sequent Core reveals some asymmetrical bias of type information. As designed, reading the type off of a Sequent Core term is always straightforward. However, continuations may have several different types (due to the polymorphic form of call stacks). Instead, it is straightforward to check if a continuation has a particular type. Contrarily, the type of the term arguments, $\mathbf{jump}\, \overrightarrow{v}$, is easy to check but hard to infer, whereas the type of the parameterized continuation $\lambda\overrightarrow{x : \tau}.c$ is easy to infer. This flow of type-inferring

versus type-checking is captured in the structure of a command and gives us a type checking algorithm:

- If the command is a **let**-binding, gather the types for the bound local variables and confirm that they are bound to well-typed terms and continuations.
- If the command evaluates a term in the eye of a continuation, $\langle v \| k \rangle$, then read the type off of v, thereby confirming that it is well-typed, and check that k has the same type.
- If the command is a jump, $\langle \mathbf{jump}\ \vec{v} \| j \rangle$, then read the type off of j, and check that $\mathbf{jump}\ \vec{v}$ has the same type.

Determining the explicit flow of type information (from term to continuation or continuation to term, depending on the form of the command) likely has something to do with the traditional form of bi-directional type checking from the literature. Intuitively, the problem with inferring the type of polymorphic call stacks is identical to the problem with inferring the general existential tuples in the sequent calculus. Thus, solving the problem with forall gives us a solution for exists by duality in the sequent calculus. Symmetry saves the day!

4 Semantics

Here we illustrate the semantics of Sequent Core programs in two forms:

- An equational theory of program transformations that a compiler might perform, in any order, to simplify a Sequent Core program.
- A small-step, operational semantics that suggests how an interpreter might evaluate a Sequent Core program.

For simplicity, the presented semantics Sequent Core only accounts for call-by-name evaluation and non-recursive bindings. A more thorough semantics would combine both call-by-name evaluation (for lifted types) and call-by-value evaluation (which is necessary for unlifted types), along with recursive **let**-bindings.

4.1 Equational Theory

First, we have the sequent equivalent of the usual β and η axioms for functions:

$$\begin{aligned} \langle \lambda x{:}\tau.v \| v' \cdot k \rangle &= \mathbf{let}\ \mathbf{val}\ x{:}\tau = v'\ \mathbf{in}\ \langle v \| k \rangle && \mathbf{where}\ x \notin FV(k) \\ \lambda x{:}\tau.\mu r{:}\tau'.\ \langle v \| x \cdot \mathbf{ret}\ r \rangle &= v && \mathbf{where}\ x \notin FV(v) \\ & && \quad v \neq \bot : \tau \to \tau' \end{aligned}$$

For the β axiom, we simplify a λ-abstraction that is evaluated with respect to a calling continuation by **let**-binding the argument and evaluating the body in the return return continuation. For the η axiom, we recognize that the

functional abstraction $\lambda x.\mu r.\,\langle f \| x \cdot \mathbf{ret}\, r\rangle$ immediately delegates to f, so the two are equivalent. Note that, due to the presence of unconstrained, polymorphic strictness (for example,`seq`), the η axiom can only be applied to terms of a function type that do not evaluate to bottom. In other words, the η axiom only applies to terms that are already equal to a λ-abstraction without the use of η.

Computation abstractions (that is, μ-abstractions), also follow a form of β and η axioms:

$$\begin{aligned} \langle \mu r.c \| k\rangle &= c[k/r] \\ \mu r.\,\langle v \| \mathbf{ret}\, r\rangle &= v \qquad\qquad \mathbf{where}\, r \notin FV(v) \end{aligned}$$

The β axiom for a μ-abstraction substitutes its entire continuation for the bound continuation variable. To avoid unnecessary code duplication, we may only want to apply the β axiom for μ when k is small (and we can force k to be small by introducing auxiliary bindings naming arguments and join points). The η axiom for a μ-abstraction is more well-behaved than the one for functions: it applies to *any* term that does not reference the bound continuation variable.

Case continuations have their own form of β and η axioms. The β axiom for a case continuation selects the appropriate branch based on the structure of input, in the case that it is given a literal or a constructed data structure.

$$\langle lit \| \mathbf{case\ as}\, x{:}\tau\ \mathbf{of}\ \ldots; lit \to c; \ldots\rangle = c[lit/x]$$

$$\begin{aligned} &\langle K \| \overrightarrow{v_n}^n \cdot \mathbf{case\ as}\, x{:}\tau\ \mathbf{of}\ \ldots; K\ \overrightarrow{y_n{:}\sigma_n}^n \to c; \ldots\rangle \\ &= \overrightarrow{\mathbf{let\ val}\, y_n{:}\sigma_n = v_n\ \mathbf{in}}^n\ \mathbf{let\ val}\, x{:}\tau = \mu r{:}\tau.\,\langle K \| \overrightarrow{y_n}^n \cdot \mathbf{ret}\, r\rangle\ \mathbf{in}\, c \end{aligned}$$

The default alternative for a case continuation is selected when no other alternative matches, and only applies when the continuation is given a weak-head normal form (WHNF) as input.

$$\begin{aligned} \langle v' \| \overrightarrow{v_n}^n \cdot \mathbf{case\ as}\, x{:}\tau\ \mathbf{of}\ \ldots; \_ \to c\rangle &= \mathbf{let\ val}\, x{:}\tau = v''\ \mathbf{in}\, c \\ &\quad \mathbf{where}\, v'' = \mu r{:}\tau.\,\langle v' \| \overrightarrow{v_n}^n \cdot \mathbf{ret}\, r\rangle \in WHNF \end{aligned}$$

Note that the list of arguments, $\overrightarrow{v_n}^n$, can only be non-empty when the head term v is a constructor: $\mu r{:}\tau.\,\langle K \| \overrightarrow{v_n}^n \cdot \mathbf{ret}\, r\rangle$ is a WHNF. The other possibility is that $\overrightarrow{v_n}^n$ is empty and v' is a literal or a λ-abstraction. We also have some extensional properties about case continuations. First, we may replace the name for the input with the pattern in any alternative, or vice versa:

$$\mathbf{case\ as}\, x{:}\tau\ \mathbf{of}\ \ldots; lit \to c; \ldots = \mathbf{case\ as}\, x{:}\tau\ \mathbf{of}\ \ldots; lit \to c[lit/x]; \ldots$$

$$\begin{aligned} &\mathbf{case\ as}\, x{:}\tau\ \mathbf{of}\ \ldots; K\ \overrightarrow{y_n{:}\sigma_n}^n \to c; \ldots \\ &= \mathbf{case\ as}\, x{:}\tau\ \mathbf{of}\ \ldots; K\ \overrightarrow{y_n{:}\sigma_n}^n \to c[\mu r{:}\tau.\,\langle K \| \overrightarrow{y_n}^n \cdot r\rangle / x]; \ldots \end{aligned}$$

Second, we have a form of η axiom for case continuations which eliminates a redundant case analysis wrapped around another continuation.

$$\mathbf{case\ as}\, x{:}\tau\ \mathbf{of}\ \overrightarrow{pat_n \to \langle x \| k\rangle}^n = k \quad \mathbf{where}\, \{x\} \cup BV(\overrightarrow{pat_n}^n) \notin FV(k), k : \tau$$

These extensionality laws are dual to the η laws for λ- and μ-abstractions. For instance, for pairs, we have the dual equality to functional η:

$$\mathbf{case}\,\mathbf{of}(x{:}\tau_1, y{:}\tau_2) \to \langle (x,y) \| k \rangle = k \qquad \mathbf{where}\, x, y \notin FV(k), k : (\tau_1, \tau_2)$$

Furthermore, for a case continuation with only a default alternative, we have the dual equality to μ η:

$$\mathbf{case}\,\mathbf{of}\, x \to \langle x \| k \rangle = k \qquad\qquad \mathbf{where}\, x \notin FV(k)$$

The **let** form of commands substitute their bindings into the underlying command. First, we interpret each type of binding pair as a substitution:

$$\begin{aligned} [\mathbf{val}\, x = v] &= [v/x] \\ [\mathbf{cont}\, j = \lambda \overrightarrow{x_n}^n.c] &= [\langle \mathbf{jump}\, \overrightarrow{v_n}^n \| j \rangle \,/\overrightarrow{\mathbf{let}\,\mathbf{val}\, x_n = v_n\,\mathbf{in}}^n c] \end{aligned}$$

The substitution of a value binding is the normal capture-avoiding substitution of terms. The substitution of a continuation binding Then a **let**-binding command substitutes its binding pair:

$$\mathbf{let}\, bp\, \mathbf{in}\, c = c[bp]$$

We now consider axioms for pushing casts around the program. First, we introduce syntactic sugar for casting the output of a term via the dual form of casting the input of its continuation:

$$v \triangleright \gamma = \mu r.\, \langle v \| \gamma \triangleleft \mathbf{ret}\, r \rangle$$

A sequence of casts can be combined into a single cast of the composed coercion:

$$\gamma \triangleleft (\gamma' \triangleleft k) = (\gamma; \gamma') \triangleleft k$$

Casting a coercion itself can be reduced to a modified coercion that pipes the type conversions together:

$$\begin{aligned} \langle \gamma \| \gamma' \triangleleft k \rangle = \langle \gamma_0'; \gamma; \gamma_1' \| k \rangle \qquad & \mathbf{where}\, \gamma : \tau_0 \sim \tau_1 \\ & \gamma' : (\tau_0 \sim \tau_1) \sim (\tau_0' \sim \tau_1') \\ & \gamma_0' : \tau_0' \sim \tau_0 = \mathrm{sym}(\mathrm{nth}_0\, \gamma') \\ & \gamma_1' : \tau_1 \sim \tau_1' = \mathrm{nth}_1\, \gamma' \end{aligned}$$

In general, we will always push casts down structures. For coercions between function type, this means pushing the cast down a call-stack:

$$\begin{aligned} \gamma \triangleleft (v \cdot k) &= (v \triangleright \gamma_0) \cdot (\gamma_1 \triangleleft k) \\ \mathbf{where}\, \gamma &: (\tau_0 \to \tau_1) \sim (\tau_0' \to \tau_1') \\ v &\notin Type \\ \gamma_0 &: \tau_0' \sim \tau_0 = \mathrm{sym}(\mathrm{nth}_0\, \gamma) \\ \gamma_1 &: \tau_1 \sim \tau_1' = \mathrm{nth}_1\, \gamma \end{aligned}$$

Similarly, casts down polymorphic call-stacks for coercions between universally quantified types:

$$\begin{aligned}\gamma \lhd (\tau \cdot k) &= \tau \cdot (\gamma' \lhd k) \\ \textbf{where}\, \gamma &: \forall a : \kappa.\tau_1' \sim \forall a : \kappa.\tau_2' \\ \gamma' &: \tau_1'[\tau/a] \sim \tau_2'[\tau/a] = \gamma @ \tau\end{aligned}$$

Casting a literal is a no-op, so long as the coercion represents the reflexive equality of the literal type with itself:

$$\langle lit \| \gamma \lhd k \rangle = \langle lit \| k \rangle \qquad\qquad \textbf{where}\, \gamma : \tau \sim \tau, lit : \tau$$

The most complicated push axiom is for data types. Here, we push a cast on a constructed data structure into its sub-terms:

$$\begin{aligned}\left\langle K \middle\| \overrightarrow{\tau_m}^m \cdot \overrightarrow{\sigma_{m'}}^{m'} \cdot \overrightarrow{v_n}^n \cdot \gamma \lhd k \right\rangle &= \left\langle K \middle\| \overrightarrow{\tau_m'}^m \cdot \overrightarrow{\sigma_{m'}}^{m'} \cdot \overrightarrow{(v_n \rhd \gamma_n')}^n \cdot k \right\rangle \\ \textbf{where}\, \gamma &: T\ \overrightarrow{\tau_m}^m \sim T\ \overrightarrow{\tau_m'}^m \\ K &: \tau_K = \overrightarrow{\forall a_m : \kappa_m.}^m\ \overrightarrow{\forall b_{m'} : \kappa'_{m}.}^{m'}\ \overrightarrow{\sigma_n'}^n \to T\ \overrightarrow{a_m}^m \\ \gamma_i' &: (\sigma_i'\overrightarrow{[\tau_m/a_m]}^m\overrightarrow{[\sigma_{m'}/b_{m'}]}^{m'}) \sim (\sigma_i'\overrightarrow{[\tau_m'/a_m]}^m\overrightarrow{[\sigma_{m'}/b_{m'}]}^{m'}) \\ \gamma_i' &= \arg_i(\langle \tau_K \rangle\ \overrightarrow{\mathrm{nth}_m\, \gamma}^m\ \overrightarrow{\langle \sigma_{m'} \rangle}^{m'})\end{aligned}$$

Part of the complication is the different behavior of universally and existentially quantified types in the structure. The universally quantified types, $\overrightarrow{\tau_m}^m$, are replaced the type of the coercion, representing a change in the externally visible type of the structure itself. The existentially quantified types, $\overrightarrow{\sigma_{m'}}^{m'}$, don't change at all, as they are a hidden internal part of the structure. The value sub-components of the structure, $\overrightarrow{v_n}^n$, are get cast by the new coercions $\overrightarrow{\gamma_n'}^n$. These new coercions are formed by selecting the matching argument type from the type of the constructor, and replacing the original universally quantified types with the coerced ones.

4.2 Operational Semantics

Since we do not consider recursive bindings, the reduction steps in operational semantics of Sequent Core always apply the to the top of a command. The reductions of non-recursive **let**-bindings, μ-abstractions, and λ-abstractions are:

$$\textbf{let}\, bp\, \textbf{in}\, c \mapsto c[bp]$$

$$\langle \mu r.c \| k \rangle \mapsto c[k/r]$$

$$\langle \lambda x.v \| v' \cdot k \rangle \mapsto \langle v[v'/x] \| k \rangle$$

For the purpose of the simplicity of the operational semantics, we assume fully saturated constructors, so an unapplied constructor $K : \overrightarrow{\forall a_m{:}\kappa_m}^m \overrightarrow{\tau_n \Rightarrow}^n \tau'$ is represented as $\overrightarrow{\lambda a_m.}^m \overrightarrow{\lambda x_n.}^n \mu r. \langle K\ \overrightarrow{a_m}^m\ \overrightarrow{x_n}^n \| \mathbf{ret}\ r \rangle$. The constructed form of term greatly simplifies the presentation of the operational semantics for case continuations. The rules for reducing case continuations are:

$$\langle lit \| \mathbf{case}\ \mathbf{as}\ x\ \mathbf{of}\ \ldots; lit \to c; \ldots \rangle \mapsto c[lit/x]$$
$$\langle K\ \overrightarrow{\tau_m}^m\ \overrightarrow{v_n}^n \| \mathbf{case}\ \mathbf{as}\ x\ \mathbf{of}\ \ldots; K\ \overrightarrow{y_n}^n \to c; \ldots \rangle \mapsto c[(K\ \overrightarrow{\tau_m}^m\ \overrightarrow{v_n}^n)/x]\overrightarrow{[v_n/y_n]}^n$$
$$\langle W \| \mathbf{case}\ \mathbf{as}\ x\ \mathbf{of}\ \ldots; \_ \to c \rangle \mapsto c[W/x]$$

Note that the W in the last rule stands for a WHNF, which is either a literal, a λ-abstraction, or a constructed term $K\ \overrightarrow{v_n}^n$.

Finally, we have the reductions which push casts out of the way of β reductions at the last moment. For function and type abstractions, we have:

$$\langle \lambda x.v \| \gamma \triangleleft (v' \cdot k) \rangle \mapsto \langle \lambda x.v \| (v' \triangleright \mathrm{sym}(\mathrm{nth}_0\ \gamma)) \cdot (\mathrm{nth}_1\ \gamma \triangleleft k) \rangle \quad \mathbf{where}\ v' \notin Type$$
$$\langle \lambda a.v \| \gamma \triangleleft (\tau \cdot k) \rangle \mapsto \langle \lambda a.v \| \tau \cdot (\gamma @ \tau \triangleleft k) \rangle$$

For a matching literal case, we have:

$$\langle lit \| \gamma \triangleleft \mathbf{case}\ \mathbf{as}\ x\ \mathbf{of}\ \ldots; lit \to c; \ldots \rangle \mapsto \langle lit \| \mathbf{case}\ \mathbf{as}\ x\ \mathbf{of}\ \ldots; lit \to c; \ldots \rangle$$

For a matching constructed term, we have:

$$\left\langle K\ \overrightarrow{\tau_m}^m\ \overrightarrow{\sigma_{m'}}^{m'}\ \overrightarrow{v_n}^n \middle\| \gamma \triangleleft \mathbf{case}\ \mathbf{as}\ x\ \mathbf{of}\ \ldots; K\ \overrightarrow{b_{m'}}^{m'}\ \overrightarrow{y_n}^n \to c; \ldots \right\rangle$$
$$\mapsto \left\langle K\ \overrightarrow{\tau'_m}^m\ \overrightarrow{\sigma_{m'}}^{m'}\ \overrightarrow{(v_n \triangleright \gamma'_n)}^n \middle\| \mathbf{case}\ \mathbf{as}\ x\ \mathbf{of}\ \ldots; K\ \overrightarrow{b_{m'}}^{m'}\ \overrightarrow{y_n}^n \to c; \ldots \right\rangle$$
$$\mathbf{where}\ \gamma : T\ \overrightarrow{\tau_m}^m \sim T\ \overrightarrow{\tau'_m}^m$$
$$K : \overrightarrow{\forall a_m{:}\kappa_m.}^m \overrightarrow{\forall b_{m'}{:}\kappa'_{m'}.}^{m'} \overrightarrow{\sigma'_n \to}^n T\ \overrightarrow{a_m}^m$$
$$\gamma'_i = \sigma'_i \overrightarrow{[\mathrm{nth}_m\ \gamma / a_m]}^m \overrightarrow{[\langle \sigma_{m'} \rangle / b_{m'}]}^{m'}$$

The default alternative of a case continuation doesn't have anything to do with a particular type, so we can't really push the cast anywhere. Instead, we include it with the substitution in another form of the default case reduction:

$$\langle W \| \gamma \triangleleft \mathbf{case}\ \mathbf{as}\ x\ \mathbf{of}\ \ldots; \_ \to c \rangle \mapsto c[W \triangleright \gamma / x]$$

And finally, when we have two casts in a row, we merge them into a single cast of the composed coercion:

$$\langle V \| \gamma \triangleleft \gamma' \triangleleft k \rangle \mapsto \langle V \| \gamma; \gamma' \triangleleft k \rangle$$

5 Translation

An important aspect of Sequent Core is that it can be translated both to and from Core, which has some benefits:

- We are sure that Sequent Core are at least as expressive as Core, so that it can represent every Core program and type.
- We are sure that Sequent Core is not *more* expressive than Core, which is a prevailing concern since the sequent calculus (like continuation-passing style) is a natural setting for first-class control. We don't want to introduce unusual control flow that can't be represented at least somewhat directly in Core.
- The compiler can process a program represented in Core for a bit, then represented in Sequent Core, then back in Core again. This capability is what makes our use of the plugin architecture possible, so that we can add Sequent Core to GHC without modifying GHC itself!

5.1 From Core to Sequent Core

Let's start with the simplest translation of Core into Sequent Core: a compositional translation of Core expressions into Sequent Core terms, $Seq\,[\![e]\!]$. Translating apparent[1] values from Core is not hard:

$$\begin{aligned} Seq\,[\![x]\!] &= x \\ Seq\,[\![lit]\!] &= lit \\ Seq\,[\![\lambda x{:}\tau.e]\!] &= \lambda x{:}\tau.Seq\,[\![e]\!] \\ Seq\,[\![\gamma]\!] &= \gamma \\ Seq\,[\![\tau]\!] &= \tau \end{aligned}$$

Rather, most of the work of translating Core into Sequent Core is in handling the apparent computation. In the compositional translation, apparent computation

[1] We say "apparent" because the structures of a data type are not expressed directly in Core, but are written as a chain of function applications with a constructor (a special sort of variable identifier) at the head. Since $K\ x\ y\ z$ "looks like" a function application at first glance, it is apparently not a value even though in actuality it is.

expressions all correspond to computation μ-abstractions.

$$\begin{aligned}
Seq\,[\![e_1\ e_2]\!] &= \mu r{:}\tau.\,\langle Seq\,[\![e_1]\!] \| Seq\,[\![e_2]\!] \cdot r\rangle \\
&\quad \mathbf{where}\,(e_1\ e_2) : \tau \\
Seq\,[\![\mathbf{let}\,bind\,\mathbf{in}\,e]\!] &= \mu r{:}\tau.\,\mathbf{let}\,Seq\,[\![bind]\!]\,\mathbf{in}\,\langle e \| r\rangle \\
&\quad \mathbf{where}\,(\mathbf{let}\,bind\,\mathbf{in}\,e) : \tau \\
Seq\left[\!\!\left[\mathbf{case}\,e\,\mathbf{as}\,x{:}\tau\,\mathbf{of}\,\overrightarrow{alt_n}^n\right]\!\!\right] &= \mu r{:}\tau.\left\langle Seq\,[\![e]\!] \,\middle\|\, \mathbf{case}\,\mathbf{as}\,x{:}\tau\,\mathbf{of}\,\overrightarrow{Seq\,[\![alt_n]\!]_r}^n \right\rangle \\
&\quad \mathbf{where}\,(\mathbf{case}\,e\,\mathbf{as}\,x{:}\tau\,\mathbf{of}\,\overrightarrow{alt_n}^n) : \tau \\
Seq\,[\![e \triangleright \gamma]\!] &= \mu r{:}\tau.\,\langle Seq\,[\![e]\!] \| \gamma \triangleleft r\rangle \\
&\quad \mathbf{where}\,(e \triangleright \gamma) : \tau
\end{aligned}$$

Because the continuation variable r introduced by the computation abstractions is annotated with its type, we need to infer the type of all apparent computations to determine this annotation. We also need to translate the binding of a **let** expression and alternatives of a case expression:

$$\begin{aligned}
Seq\,[\![x{:}\tau = e]\!] &= \mathbf{val}\,x{:}\tau = Seq\,[\![e]\!] \\
Seq\,[\![\mathbf{rec}\,\{\overrightarrow{x_n{:}\tau_n = e_n}^n\}]\!] &= \mathbf{rec}\left\{\overrightarrow{\mathbf{val}\,x_n{:}\tau_n = Seq\,[\![e_n]\!]}^n\right\}
\end{aligned}$$

$$\begin{aligned}
Seq\,[\![_ \to e]\!]_r &= _ \to \langle Seq\,[\![e]\!] \| r\rangle \\
Seq\,[\![K\ \overrightarrow{x_n{:}\tau_n}^n \to e]\!]_r &= K\ \overrightarrow{x_n{:}\tau_n}^n \to \langle Seq\,[\![e]\!] \| r\rangle \\
Seq\,[\![lit \to e]\!]_r &= lit \to \langle Seq\,[\![e]\!] \| r\rangle
\end{aligned}$$

Note that because case alternatives in Sequent Core point to self-contained commands, we explicitly spell out the common continuation that every alternative "returns" to, which was introduced as the return continuation for the case expression itself. Finally, we can translate a whole program from Core to Sequent Core by giving some chosen continuation variable on which we expect to receive the final result of the program:

$$Seq\left[\!\!\left[\overrightarrow{decl_n}^n; e\right]\!\!\right]_r = \overrightarrow{decl_n}^n; \langle Seq\,[\![e]\!] \| r\rangle$$

While the compositional translation is simple, it creates unnecessarily large terms due to the fact that every apparent computation gets its own μ-abstraction. We can then take the observation on the difference between apparent values and apparent computations in Core to write a better, more compacting translation into Sequent Core. This more compacting translation is closer to the implemented one, but still quite simplified.[2] For apparent values (variables, literals,

[2]The implemented translation handles the renaming necessary to avoid static variable capture and also attempts to translate functions which represent continuations as continuations (i.e., it performs some *re-contification*).

lambda abstractions, coercions, and types), the translation is much the same:

$$
\begin{aligned}
Seq\,[\![x]\!] &= x \\
Seq\,[\![lit]\!] &= lit \\
Seq\,[\![\lambda x{:}\tau.e]\!] &= \lambda x{:}\tau.Seq\,[\![e]\!] \\
Seq\,[\![\gamma]\!] &= \gamma \\
Seq\,[\![\tau]\!] &= \tau \\
Seq\,[\![e]\!] &= \mu r : \tau.Seq\,[\![e : \tau]\!]\,r \quad \textbf{where}\, e : \tau \text{ is an apparent computation}
\end{aligned}
$$

The main difference is when we find an apparent computation (an application, let expression, case expression, or cast), identified by the last clause above. In this case, we introduce *one* μ-abstraction, and then begin to collect the outermost continuation and bindings of the computation:

$$
\begin{aligned}
Seq\,[\![e_1\ e_2]\!]\,k &= Seq\,[\![e_1]\!]\,(Seq\,[\![e_2]\!] \cdot k) \\
Seq\,[\![\textbf{let}\, bind\, \textbf{in}\, e]\!]\,k &= \textbf{let}\, Seq\,[\![bind]\!]\, \textbf{in}\, Seq\,[\![e]\!]\,k \\
&\quad \textbf{where}\, BV(bind) \notin FV(k) \\
Seq\,\left[\!\!\left[\textbf{case}\, e\, \textbf{as}\, x{:}\tau\, \textbf{of}\, \overrightarrow{alt_n}^n\right]\!\!\right]\,k &= Seq\,[\![e]\!]\,(\textbf{case}\, \textbf{as}\, x{:}\tau\, \textbf{of}\, \overrightarrow{Core\,[\![alt_n]\!]\,k}^n) \\
&\quad \textbf{where}\, k \text{ is small} \\
Seq\,\left[\!\!\left[\textbf{case}\, e\, \textbf{as}\, x{:}\tau\, \textbf{of}\, \overrightarrow{alt_n}^n\right]\!\!\right]\,k &= \left\langle \mu r{:}\tau'.Seq\,[\![e]\!]\,(\textbf{case}\, \textbf{as}\, x{:}\tau\, \textbf{of}\, \overrightarrow{Core\,[\![alt_n]\!]\,r}^n) \middle\| k \right\rangle \\
&\quad \textbf{where}\, k : \tau' \text{ is large} \\
Seq\,[\![e \triangleright \gamma]\!]\,k &= Seq\,[\![e]\!]\,(\gamma \triangleleft k) \\
Seq\,[\![e]\!]\,k &= \langle Seq\,[\![e]\!] \| k \rangle \qquad \textbf{where}\, e \text{ is an apparent value}
\end{aligned}
$$

Some care must still be taken when translating **case** expressions, since we want to avoid unnecessary duplication of large continuations inside the branches of alternatives. For now, we just check if the continuation to a **case** is small enough to duplicate, and if not, bind it with a computation abstraction. When we reach an apparent value again, in the last clause above, we write down the entire continuation and list of bindings found during translation.

The more compacting translation of bindings, alternatives, and whole programs are effectively the same as before, except that for a binding we begin expecting the bound expression to be value-like, and for an alternative and whole program we begin expecting the resulting expression to be computation-like:

$$
\begin{aligned}
Seq\,[\![x{:}\tau = e]\!] &= \textbf{val}\, x{:}\tau = Seq\,[\![e]\!] \\
Seq\,[\![\textbf{rec}\, \{\overrightarrow{x_n{:}\tau_n = e_n}^n\}]\!] &= \textbf{rec}\, \left\{ \overrightarrow{\textbf{val}\, x_n{:}\tau_n = Seq\,[\![e_n]\!]}^n \right\}
\end{aligned}
$$

$$
\begin{aligned}
Seq\,[\![\_ \to e]\!]\,k &= \_ \to Seq\,[\![e]\!]\,k \\
Seq\,[\![K\ \overrightarrow{x_n{:}\tau_n}^n \to e]\!]\,k &= K\ \overrightarrow{x_n{:}\tau_n}^n \to Seq\,[\![e]\!]\,k \\
Seq\,[\![lit \to e]\!]\,k &= lit \to Seq\,[\![e]\!]\,k
\end{aligned}
$$

$$Seq\left[\!\left[\overrightarrow{decl_n}^n; e\right]\!\right] r = \overrightarrow{decl_n}^n; Seq\,[\![e]\!]\,r$$

5.2 From Sequent Core to Core

Translating Sequent Core terms back into Core expressions is also fairly straightforward, given by $Core\,[\![v]\!]$:

$$\begin{aligned} Core\,[\![x]\!] &= x \\ Core\,[\![\mu r{:}\tau.c]\!] &= Core\,[\![c]\!]_r \\ Core\,[\![\lambda x{:}\tau.v]\!] &= \lambda x{:}\tau.Core\,[\![v]\!] \\ Core\,[\![lit]\!] &= lit \\ Core\,[\![\gamma]\!] &= \gamma \\ Core\,[\![\tau]\!] &= \tau \end{aligned}$$

All but one Sequent Core term, namely the computation abstraction, corresponds directly to a Core value. To translate a computational term to a Core expression, we need to translate the underlying command and read off the result returned to the abstracted continuation variable, as given by $Core\,[\![c]\!]_r$:

$$\begin{aligned} Core\,[\![\mathbf{let}\,bind\,\mathbf{in}\,c]\!]_r &= \mathbf{let}\,Core\,[\![bind]\!]_r\,\mathbf{in}\,Core\,[\![c]\!]_r \\ Core\,[\![\langle v \| k\rangle]\!]_r &= Core\,[\![k]\!]_r\,[Core\,[\![v]\!]] \\ Core\,[\![\langle \mathbf{jump}\,\overrightarrow{v_n}^n \| j\rangle]\!]_r &= j\ \overrightarrow{Core\,[\![v_n]\!]}^n \end{aligned}$$

A **let**-binding translates to a **let**-binding, a command $\langle v \| k\rangle$ translates to evaluating expression corresponding to the term v inside the evaluation context corresponding to the continuation k, and a jump $\langle \mathbf{jump}\,\vec{v} \| j\rangle$ translates to a tail function call (more on this later). The evaluation contexts corresponding to continuations which return their result on r are given by $Core\,[\![k]\!]_r$:

$$\begin{aligned} Core\,[\![\mathbf{ret}\,r]\!]_r &= \Box \\ Core\,[\![v \cdot k]\!]_r &= Core\,[\![k]\!]_r\,[\Box\ Core\,[\![v]\!]] \\ Core\,[\![\gamma \triangleleft k]\!]_r &= Core\,[\![k]\!]_r\,[\Box \triangleright \gamma] \\ Core\left[\!\left[\mathbf{case}\,\mathbf{as}\,x{:}\tau\,\mathbf{of}\,\overrightarrow{alt_n}^n\right]\!\right]_r &= \mathbf{case}\,\Box\,\mathbf{as}\,x{:}\tau\,\mathbf{of}\,\overrightarrow{Core\,[\![alt_n]\!]_r}^n \end{aligned}$$

Besides translating terms, commands, and continuations, we also need to convert Sequent Core bindings and alternatives back into Core bindings and alternatives. The alternatives of case analysis have a straightforward, one-to-one correspondence between Core and Sequent Core, given by $Core\,[\![alt]\!]_r$, where Sequent Core alternatives lead to commands that output their result on a return continuation variable r:

$$\begin{aligned} Core\,[\![\_ \to c]\!]_r &= \_ \to Core\,[\![c]\!]_r \\ Core\,[\![K\ \overrightarrow{x{:}\tau}^n \to c]\!]_r &= K\ \overrightarrow{x{:}\tau}^n \to Core\,[\![c]\!]_r \\ Core\,[\![lit \to c]\!]_r &= lit \to Core\,[\![c]\!]_r \end{aligned}$$

Translating the bindings back to Core is more interesting, since Sequent Core has two different types of bindings (value and continuation bindings) whereas Core only has the one. Thus, we collapse both bound terms and continuations into bound Core expressions, given by $Core\,[\![bind]\!]_r$ and $Core\,[\![bp]\!]_r$.

$$\begin{aligned} Core\,[\![bp]\!]_r &= Core\,[\![bp]\!]_r \\ Core\left[\!\left[\mathbf{rec}\left\{\overrightarrow{bp_n}^n\right\}\right]\!\right]_r &= \mathbf{rec}\left\{\overrightarrow{Core\,[\![bp_n]\!]_r}^n\right\} \end{aligned}$$

$$\begin{aligned} Core\,[\![\mathbf{val}\,x{:}\tau = v]\!]_r &= (x{:}\tau = Core\,[\![v]\!]) \\ Core\,[\![\mathbf{cont}\,j{:}\tau = \lambda\overrightarrow{x_n{:}\tau_n}^n.c]\!]_r &= (j{:}Core^{\neg}\,[\![\tau]\!]_{\tau'} = \overrightarrow{\lambda x_n{:}\tau_n.}^n Core\,[\![c]\!]_r) \\ &\quad \mathbf{where}\, r : \tau' \end{aligned}$$

We need to know the return variable r in order to convert continuation bindings, since continuations do not just return results, but instead output them on the continuation variable r. In the conversion of the continuation bound to j into an expression, we effectively negate the type of j. So instead of standing for a continuation that accepts multiple inputs as an unboxed tuple, the converted j stands for a function from the multiple inputs to the ultimate result type given by the surrounding return continuation r. The negation of a parameterized continuation is given by $Core^{\neg}\,[\![\sigma]\!]_\tau$, where τ is the ultimate return type of the continuation:

$$\begin{aligned} Core^{\neg}\,[\![Unit\#]\!]_\tau &= \tau \\ Core^{\neg}\,[\![\tau' \times_\# \sigma]\!]_\tau &= \tau' \to Core^{\neg}\,[\![\sigma]\!]_\tau \\ Core^{\neg}\,[\![\exists_\# a{:}\kappa.\sigma]\!]_\tau &= \forall a{:}\kappa.Core^{\neg}\,[\![\sigma]\!]_\tau \end{aligned}$$

We can read the translation $Core^{\neg}\,[\![\sigma]\!]_\tau$ as a logical negation of σ if we understand the result type τ to stand in for falsehood. For instance, $Unit\#$ is logically interpreted as "true," and so its negation is the "false" return type τ. The logical negation of "there exists an a such that σ is true" is "for all a, σ is false," hence the third clause. The middle clause for converting a product type into a function type is a bit more indirect form of negation. However, it follows from the familiar De Morgan's laws if we (informally) understand the function type $\tau \to \sigma$ as a form of disjunction $(\neg\tau) \vee \sigma$:

$$\neg(\tau \wedge \sigma) = (\neg\tau) \vee (\neg\sigma) = \tau \to (\neg\sigma)$$

Hence, the negated interpretation of the product jump type, $Core^{\neg}\,[\![\tau' \times_\# \sigma]\!]_\tau$, is a function, $\tau' \to Core^{\neg}\,[\![\sigma]\!]_\tau$.

To translate the whole Sequent Core program back into Core, we just translate the main command with respect to the continuation on which we expect to receive the program's result.

$$Core\left[\!\left[\overrightarrow{decl_n}^n; c\right]\!\right]_r = \overrightarrow{decl_n}^n; Core\,[\![c]\!]_r$$

6 Design discussion

6.1 Join points

One of the goals of Sequent Core is to provide a good representation for naming join points in a program. A *join point* is a specified point in the control flow of a program where to different possible execution paths join back up again. The purpose of giving a name to these join points is to avoid excessive duplication of code and keep code size small. Since a join point represents the future execution path of a program, we would like to model them as continuations, and so named join points become named continuations.

The main place where new join points are named are in the branches of a case expression. For example, consider the Core expression

$$e = \mathbf{case}\, e_0\, \mathbf{of}\, \{K_1\ x\ y \to e_1; K_2 \to e_2\}$$

In Sequent Core, e would be represented as the term v:

$$v = \mu r.\, \langle \mu q.c_0 \| \mathbf{case}\, \mathbf{of}\, \{K_1\ x\ y \to c_1; K_2 \to c_2\} \rangle$$

where the term $\mu q.c_0$ corresponds to the expression e_0, and the commands c_1 and c_2 correspond to the expressions e_1 and e_2 run in the context of the continuation r which expects the end result returned by e. Now, it would be semantically correct to inline the case continuation for q inside of c_0, which could open up further simplification. For example, suppose that c_0 is:

$$c_0 = \langle z \| \mathbf{case}\, \mathbf{of}\, \{True \to \langle K_1 \| 5 \cdot 10 \cdot q \rangle\, ; False \to \langle z' \| q \rangle\} \rangle$$

Inlining for q everywhere in c_0 then corresponds to the case-of-case transformation performed by GHC on Core expressions.

However, inlining for q everywhere can duplicate the commands c_1 and c_2, leading to larger code size! We only want to inline selectively in places where we are confident that inlining the case continuation would lead to further simplification, but not elsewhere. We can achieve selective inlining by assigning the continuation to its name, q:

$$v = \mu r.\, \mathbf{let}\, q = \mathbf{case}\, \mathbf{of}\, \{K_1\ x\ y \to c_1; K_2 \to c_2\}\, \mathbf{in}\, c_0$$

and then follow heuristics to inline for q only in those places that matter. In particular, we could just inline inside of the one branch in c_0 which provides q with a constructed result, leading to the command:

$$c_0 = \langle z \| \mathbf{case}\, \mathbf{of}\, \{True \to c_1[5/x, 10/y]; False \to \langle z' \| q \rangle\} \rangle$$

In effect, selective inlining has eliminated case analysis on a known case by duplicating the command c_1, but not duplicating c_2.

There is still a problem, though: what if the command c_1 is so large that this simplification is not worth the duplication? In that case, we would like to

form a *new* join point that gives a name to the command c_1, so that we may refer to it by name rather than duplicating the entire command. Since c_1 is an open command referring to local variables introduced by the case analysis, we must first abstract over these local variables. This is the role of the polyadic form of continuation, which is able to receive multiple inputs before running. In our example, we would move c_1 and c_2 into a named, polyadic continuations:

$$\begin{aligned} v = \mu r. \quad & \mathbf{let}\, p_1 = \lambda x \cdot y.c_1; \\ & \quad\; p_2 = \lambda \cdot .c_2; \\ & \quad\; q = \mathbf{case}\, \mathbf{of}\, \{K_1\ x\ y \to \langle x \cdot y \| p_1 \rangle\, ; K_2 \to \langle\, \cdot \| p_2 \rangle\} \\ & \mathbf{in}\, c_0 \end{aligned}$$

Now, q stands for a small continuation, and so it can be inlined much more aggressively. For example, we can perform a more lightweight inlining into c_0:

$$c_0 = \langle z \| \mathbf{case}\, \mathbf{of}\, \{True \to \langle 5 \cdot 10 \| p_1 \rangle\, ; False \to \langle z' \| q \rangle\}\rangle$$

which avoids duplicating c_1 at all.

6.2 Polymorphic-polyadic continuations and the existential stack

Due to the fact that some constructors to data types contain existentially quantified types, it is important that we be able to abstract over a command with free type variables as a polymorphic continuation. Thus, the more general form of the λ-continuation may introduce several type variables in addition to term variables. This means that the stacks which are provided to these continuations are effectively a form of existential tuple. Currently, we represent these polymorphic continuations and continuations expecting multiple inputs as an uncurried λ-abstraction. However, there are multiple other design choices which are possible for this purpose:

- Unboxed tuples: Morally, the stack $1 \cdot 2 \cdot 3$ for a polyadic continuation is the same as the unboxed tuple $(\#1, 2, 3\#)$. Why then do we bother to introduce a new form of term, and a new type, instead of just using unboxed tuples for this purpose? The reason is the above-mentioned existentially-quantified types, which lie outside the form of unboxed tuples supported by Core. We cannot represent $Int \cdot 1 \cdot 2 \cdot 3 : \exists a : \star.(a, a, a)$ as an unboxed tuple.

  As a minor advantage, having the special form for existential stacks lets us (actually, requires us to) translate them back into Core as a calling context rather than a value, so $\langle 1 \cdot 2 \cdot 3 \| q \rangle$ becomes $q\ 1\ 2\ 3$ instead of $q(\#1, 2, 3\#)$. This results in Core expressions for join points looking closer to what the current GHC simplifier actually produces.

- Curried vs uncurried: Currently, the continuations which accept existential stacks are written in a totally-uncurried form. By listing all the parameters that abstract over a command, we are forced to spell out *exactly*

the number of inputs that are required by the continuation before any work can be done.

An alternative design would be to give a curried form of continuations which accept existential stacks. These would have one of two forms:

- $\lambda x{:}\tau.k$: accept the first element of the stack, which is a term of type τ, as x, then pass the rest of the stack to k (i.e., pop the top element of the stack as x and then continue as k)
- $\lambda a{:}\kappa.k$: accept the first element of the stack, which is a type of kind κ, as a, and then pass the rest of the stack to k (i.e., specialize the type variable a for the continuation k)

which could be collapsed like ordinary λ-abstractions. To model a unary continuation, like the continuation which accepts the "end" of the stack, we could introduce the dual to general computation abstractions. These are continuations of the form $\mu x{:}\tau.c$ which accept an input named x before performing some arbitrary computation c, and correspond to the context $\mathbf{let}\, x : \tau = \Box\, \mathbf{in}\, e$ in Core. However, these fundamentally introduce *non-strict* continuations, which is a whole can of worms we have been avoiding thus far. In the end, it may be worthwhile to introduce these general input continuations for independent reasons, but we leave them out for now.

- Case alternative: Instead of introducing a special new form of continuations for accepting existential stacks, we could also have added a form of case alternative for matching on the stack. That way the special continuations $\lambda\overrightarrow{a{:}\kappa} \cdot \overrightarrow{x{:}\tau}.c$ would just be written as another form of case continuation $\mathbf{case}\,\mathbf{of}\, a{:}\kappa \cdot x{:}\tau \to c$. This unification might make sense following the story that existential stacks are just a sort of unboxed data type that are not available in Core, so it makes sense that we would pattern match on them.

  For now we don't fold continuations for existential stacks, but it is unclear if doing so would be a benefit (one less type of continuation to consider) or a hinderance (if they need to be treated differently in enough of the compiler that we effectively special case them anyways). It is worth keeping this alternative design in mind, to evaluate which of the two options would pan out better in practice.

6.3 Continuation-closed terms

Currently, we make the scope and type restriction that forces all terms to never reference free continuation variables. This kind of restriction is necessary to ensure that Sequent Core programs correspond directly to Core programs, and do not introduce new kinds of control flow that would require a full continuation-passin style (CPS) translation back into Core. However, it is probably sufficient to impose this scoping restriction on *values* (and in particular, λ-abstractions).

This would allow for general computation terms (μ-abstractions $\mu q.c$) to reference continuations in its scope in order to produce a (continuation-closed) value. Since general computations cannot escape their scope, due to call-by-need semantics, this should be fine and should not require a full CPS translation back into Core.

The scope restriction on general computation terms comes up during translation from Core to Sequent Core. Currently, the restriction forces the continuation representing the context of a case expression be let-bound, rather than use a μ-abstraction. This forces more continuations to be abstracted into the local let bindings, which in effect introduces more named join points back in Core world. This is probably not bad, since we want to be introducing join points anyway, but it is something to keep in mind.