Typing Dual System FC and Sequent Core

July 17, 2015

1 Syntax

$$
\begin{array}{rll}
x, y, z, f, g, h, K \in Var & & \\
q, r \in KontVar & & \\
a, b, T \in TypeVar & & \\
c \in Command & ::= & \mathbf{let}\, binds\, \mathbf{in}\, \langle v \| k \rangle \\
v \in Term & ::= & x \mid \lambda x{:}\tau.v \mid \mu q{:}\tau.c \mid lit \mid \tau \mid \gamma \mid \vec{\tau_i}^{\,i} \cdot \vec{v_j}^{\,j} \\
k \in Kont & ::= & q \mid v \cdot k \mid \gamma \triangleleft k \mid \mathbf{case}\, \mathbf{as}\, x{:}\tau\, \mathbf{of}\, alts \mid \lambda \overrightarrow{a_i{:}\kappa_i}^{\,i} \cdot \overrightarrow{x_j{:}\tau_j}^{\,j}.c \\
binds \in Bindings & ::= & \overrightarrow{bind_i}^{\,i} \\
bind \in Binding & ::= & x{:}\tau = v \mid \mathbf{rec}\, \overrightarrow{x_i{:}\tau_i = v_i}^{\,i} \mid q{:}\sigma = k \mid \mathbf{rec}\, \overrightarrow{q_i{:}\sigma_i = k_i}^{\,i} \\
alts \in Alternatives & ::= & \overrightarrow{alt_i}^{\,i} \\
alt \in Alternative & ::= & \_ \to c \mid K\ \overrightarrow{b_i{:}\kappa_i}^{\,i}\ \overrightarrow{x_j{:}\tau_j}^{\,j} \to c \mid lit \to c \\
\tau \in Type & ::= & \ldots \\
\sigma \in SequentType & ::= & \tau \mid \exists \overrightarrow{a_i : \kappa_i}^{\,i}.(\vec{\tau_j}^{\,j}) \\
\kappa \in Kind & ::= & \ldots \\
\gamma \in Coercion & ::= & \ldots \\
decls \in Declarations & ::= & \overrightarrow{decl_i}^{\,i} \\
decl \in Declaration & ::= & \ldots \\
pgm \in Program & ::= & decls; c
\end{array}
$$

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 $1 \cdot 2 \cdot 3 \cdot q$ is the same as $1 \cdot (2 \cdot (3 \cdot q))$. Similarly, coercion continuations associate to the right as well, so that $\gamma_1 \lhd \gamma_2 \lhd q$ is the same as $\gamma_1 \lhd (\gamma_2 \lhd q)$. Both function calls and coercions share the same precedence and may be intermixed, so that $1 \cdot \gamma_2 \lhd 3 \cdot q$ is the same as $1 \cdot (\gamma_2 \lhd (3 \cdot q))$.
- If a command does not have any associated bindings with it, so that *binds* is empty in $\mathbf{let}\, k\, \mathbf{in}\, \langle binds \| v \rangle$, then we will omit the **let** form altogether and just write $\langle v \| k \rangle$.
- We will not always write the binding variable $x{:}\tau$ in the case continuation $\mathbf{case}\, \mathbf{as}\, x{:}\tau\, \mathbf{of}\, alts$ when it turns out that x is never referenced in $alts$ or $alts$, instead writing $\mathbf{case}\, alts$. If instead $x{:}\tau$ is only referenced in the default alternative $\_ \to c$ in $alts$, 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 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 variables.

This restriction makes sure that λ-abstractions cannot close over continuation variables available from its context, so that bound continuations do not escape through a returned λ-abstraction. Additionally, in all computations $\mu r.c$, the underlying command c has precisely one unique exit point, namely r, which names the result of the computation.

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:

$$\Gamma \in \mathit{Environment} \quad ::= \varepsilon \mid \Gamma, x \mid \Gamma, a$$
$$\Delta \in \mathit{KoEnvironment} \quad ::= \varepsilon \mid \Delta, q$$

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

$$\frac{x \in \Gamma}{\Gamma \vdash x\,\mathrm{ok}} \qquad \frac{}{\Gamma \vdash \mathit{lit}\,\mathrm{ok}} \qquad \frac{\Gamma; q \vdash c\,\mathrm{ok}}{\Gamma \vdash \mu q.c\,\mathrm{ok}}$$

$$\frac{\Gamma, x \vdash v\,\mathrm{ok}}{\Gamma \vdash \lambda x.v\,\mathrm{ok}} \qquad \frac{\Gamma, a \vdash v\,\mathrm{ok}}{\Gamma \vdash \lambda a.v\,\mathrm{ok}} \qquad \frac{\overrightarrow{\Gamma \vdash \tau_i\,\mathrm{ok}}^{i} \quad \overrightarrow{\Gamma \vdash v_j\,\mathrm{ok}}^{j}}{\Gamma \vdash \vec{\tau_i}^{i} \cdot \vec{v_j}^{j}\,\mathrm{ok}}$$

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

$$\frac{q \in \Delta}{\Gamma; \Delta \vdash q\,\mathrm{ok}} \qquad \frac{\Gamma \vdash v\,\mathrm{ok} \quad \Gamma; \Delta \vdash k\,\mathrm{ok}}{\Gamma; \Delta \vdash v \cdot k\,\mathrm{ok}} \qquad \frac{\Gamma \vdash \tau\,\mathrm{ok} \quad \Gamma; \Delta \vdash k\,\mathrm{ok}}{\Gamma; \Delta \vdash \tau \cdot k\,\mathrm{ok}}$$

$$\frac{\Gamma \vdash \gamma\,\mathrm{ok} \quad \Gamma; \Delta \vdash k\,\mathrm{ok}}{\Gamma; \Delta \vdash \gamma \lhd k\,\mathrm{ok}} \qquad \frac{\Gamma, x; \Delta \vdash \mathit{alts}\,\mathrm{ok}}{\Gamma; \Delta \vdash \mathbf{case\ as}\ x\ \mathbf{of}\ \mathit{alts}\,\mathrm{ok}} \qquad \frac{\Gamma, \vec{a_i}^{i}, \vec{x_j}^{j}; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash \lambda \vec{a_i}^{i} \cdot \vec{x_j}^{j}.c\,\mathrm{ok}}$$

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

$$\frac{\Gamma; \Delta \vdash \mathit{binds} : \Gamma'; \Delta' \quad \Gamma, \Gamma' \vdash v\,\mathrm{ok} \quad \Gamma, \Gamma'; \Delta, \Delta' \vdash k\,\mathrm{ok}}{\Gamma; \Delta \vdash \mathbf{let}\ \mathit{binds}\ \mathbf{in}\ \langle v \| k \rangle\,\mathrm{ok}}$$

Further rules for $\Gamma \vdash \tau\,\mathrm{ok}$ *and* $\Gamma \vdash \kappa\,\mathrm{ok}$, $\Gamma \vdash \gamma\,\mathrm{ok}$

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

- 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.

In the simple case, this means execution of the term $\mu r.c$ yields $\mu r.\langle V \| r \rangle$, 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, recursive bindings can be between only terms or only continuations. But why not allow for is mutual recursion between both terms and continuations in the same binding block? 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 and alternative scoping: $\Gamma; \Delta \vdash \mathit{bind} : \Gamma'; \Delta'$ and $\Gamma; \Delta \vdash \mathit{alt}\,\mathrm{ok}$

$$\frac{}{\Gamma; \Delta \vdash \varepsilon : \varepsilon; \varepsilon} \qquad \frac{\Gamma; \Delta \vdash \mathit{binds} : \Gamma'; \Delta' \quad \Gamma'; \Delta' \vdash \mathit{bind} : \Gamma''; \Delta''}{\Gamma; \Delta \vdash \mathit{binds}; \mathit{bind} : \Gamma''; \Delta''}$$

$$\frac{\Gamma \vdash v\,\mathrm{ok}}{\Gamma; \Delta \vdash x = v : \Gamma, x; \Delta} \qquad \frac{\overrightarrow{\Gamma, \vec{x_j}^{\,j} \vdash v_i\,\mathrm{ok}}^{\,i}}{\Gamma; \Delta \vdash \mathbf{rec}\,\overrightarrow{x_i = v_i}^{\,i} : \Gamma, \vec{x_i}^{\,i}; \Delta}$$

$$\frac{\Gamma; \Delta \vdash k\,\mathrm{ok}}{\Gamma; \Delta \vdash q = k : \Gamma; \Delta, q} \qquad \frac{\overrightarrow{\Gamma; \Delta, \vec{q_j}^{\,j} \vdash k_i\,\mathrm{ok}}^{\,i}}{\Gamma; \Delta \vdash \mathbf{rec}\,\overrightarrow{q_i = k_i}^{\,i} : \Gamma; \Delta, \vec{q_i}^{\,i}}$$

$$\frac{\overrightarrow{\Gamma; \Delta \vdash \mathit{alt}_i\,\mathrm{ok}}^{\,i}}{\Gamma; \Delta \vdash \overrightarrow{\mathit{alt}_i}^{\,i}\,\mathrm{ok}} \qquad \frac{\Gamma; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash \_ \to c\,\mathrm{ok}} \qquad \frac{\Gamma, \vec{b_i}^{\,i}, \vec{x_i}^{\,i}; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash K\ \vec{b_i}^{\,i}\ \vec{x_i}^{\,i} \to c\,\mathrm{ok}} \qquad \frac{\Gamma; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash \mathit{lit} \to c\,\mathrm{ok}}$$

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

$$\frac{\Gamma \vdash \mathit{decls} : \Gamma' \quad \Gamma'; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash \mathit{decls}; c\,\mathrm{ok}}$$

Further rules for $\Gamma \vdash \mathit{decls} : \Gamma'$ *and* $\Gamma \vdash \mathit{decl} : \Gamma'$.

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

binding environment.

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

$$\mathbf{rec}\{f = \lambda x.v; q = \mathbf{case}\,y \to c\}$$

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

$$\mathbf{rec}\{f = \lambda x.v\}; \mathbf{rec}\{q = \mathbf{case}\,y \to c\}$$

So this limitation results in no loss of expressiveness. Indeed, we could further the partitions into

1. first, the list of term 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

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

Term typing: $\Gamma \vdash v : \sigma$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\tau = \mathit{literalType}(\mathit{lit})}{\Gamma \vdash \mathit{lit} : \tau} \qquad \frac{\Gamma; q : \tau \vdash c\,\mathrm{ok}}{\Gamma \vdash \mu q{:}\tau.c : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash v : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.v : \tau_1 \to \tau_2} \qquad \frac{\Gamma, a : \kappa \vdash v : \tau}{\Gamma \vdash \lambda a{:}\kappa.v : \forall a{:}\kappa.\tau} \qquad \frac{\overrightarrow{\Gamma \vdash \tau_i : \kappa_i}^{\,i} \quad \overrightarrow{\Gamma \vdash v_j : \tau'_j[\overrightarrow{\tau_i/a_i}^{\,i}]}^{\,j}}{\Gamma \vdash \vec{\tau}_i^{\,i} \cdot \vec{v}_j^{\,j} : \exists \overrightarrow{a_i : \kappa_i}^{\,i}.(\vec{\tau}'^{\,j}_j)}$$

Continuation typing: $\Gamma; \Delta \vdash k : \sigma$

$$\frac{q : \sigma \in \Delta}{\Gamma; \Delta \vdash q : \sigma} \qquad \frac{\Gamma \vdash v : \tau_1 \quad \Gamma; \Delta \vdash k : \tau_2}{\Gamma; \Delta \vdash v \cdot k : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa \quad \Gamma; \Delta \vdash k : \tau_2[\tau_1/a]}{\Gamma; \Delta \vdash \tau_1 \cdot k : \forall a{:}\kappa.\tau_2}$$

$$\frac{\Gamma \vdash \gamma : \tau_1 \sim \tau_2 \quad \Gamma; \Delta \vdash k : \tau_2}{\Gamma; \Delta \vdash \gamma \triangleleft k : \tau_1} \qquad \frac{\Gamma, x : \tau; \Delta \vdash \mathit{alts} : \tau}{\Gamma; \Delta \vdash \mathbf{case\ as}\, x{:}\tau\, \mathbf{of}\, \mathit{alts} : \tau}$$

$$\frac{\Gamma, \overrightarrow{a_i : \kappa_i}^{\,i}, \overrightarrow{x_j : \tau_j}^{\,j}; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash \lambda \overrightarrow{a_i : \kappa_i}^{\,i} \cdot \overrightarrow{x_j : \tau_j}^{\,j}.c : \exists \overrightarrow{a_i : \kappa_i}^{\,i}.(\vec{\tau}_j^{\,j})}$$

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

$$\frac{\Gamma; \Delta \vdash \mathit{binds} : \Gamma'; \Delta' \quad \Gamma, \Gamma' \vdash v : \sigma \quad \Gamma, \Gamma' \vdash \sigma : \star \quad \Gamma, \Gamma'; \Delta, \Delta' \vdash k : \sigma}{\Gamma; \Delta \vdash \mathbf{let}\, \mathit{binds}\, \mathbf{in}\, \langle v \| k \rangle\, \mathrm{ok}}$$

Further rules for $\Gamma \vdash \tau : \kappa$, $\Gamma \vdash \kappa : \delta$, *and* $\Gamma \vdash \gamma : \tau_1 \sim \tau_2$.

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

The typing rules for Dual System FC are given in Figures 4 and 5. 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 "escapes" through one of its available continuation variables. 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 $\Gamma_0; exit : \tau \vdash pgm\,\text{ok}$, 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 that cancel out arbitrary types, given by the rules for typing commands and bindings, which effectively perform multiple cuts simultaneously. This is the reason that we must check that in the command $\mathbf{let}\, binds\, \mathbf{in}\, \langle v \| k \rangle$, not only do v and k agree on the same (inferred) type σ, but that inferred type actually has to be of kind $\star$. 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 $Int \cdot 1 \cdot 2 \cdot q$ 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 the terms and continuations of existential stacks exhibit exactly the dual issues that are raised for polymorphic call stacks. This is expected, as they are designed to be a "dualish" form of fully-general call stacks.

- 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 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, $\langle (,) \| \tau \cdot v \cdot q \rangle$. That means that all the troubles with checking existential structures is contained solely with 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 or an existential stack, the type checking algorithm for Sequent Core reveals some conditional bias towards terms or continuations. On the one hand, reading the type off of a Sequent Core term is straightforward if it has a Core type, since those terms correspond to a subset of Core expressions. Then, checking that a continuation has a known Core type is easy. On the other hand, reading the type off a Sequent Core continuation accepting an existential stack is is straightforward for exactly the dual reason. Then, checking that a term has a known existential stack type is easy. This flow of type-reading versus type-checking is captured in the structure of a command:

- If the command has any local bindings, gather the types for the bound local variables and confirm that they are bound to well-typed terms and continuations.
- Determine whether the main term and continuation of the command share some (as of yet unknown) Core type or share some (as of yet unknown) existential stack type. This is a simple syntactic check on the structure of the term and continuation. If they both share some Core type:
  - Read the type off of the term, thereby confirming that it is well-typed.
  - Check that the continuation has the same type as the term.

  If they both share some existential stack type:
  - Read off the type of the continuation, thereby confirming that it is well-typed.
  - Check that the term has the same type as the continuation.

  Otherwise, the command is clearly ill-typed, since there is no way for the term and continuation to share some common type.

This explicit flow of type information (from term to continuation or continuation to term, depending) likely has something to do with bi-directional type checking. 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 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!

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

$$\begin{aligned}
Seq\,[\![x : \tau]\!] &= x \\
Seq\,[\![lit : \tau]\!] &= lit \\
Seq\,[\![(\lambda x{:}\tau.e) : \tau \to \tau']\!] &= \lambda x{:}\tau.Seq\,[\![e : \tau']\!] \\
Seq\,[\![(\lambda a{:}\kappa.e) : \forall a{:}\kappa.\tau]\!] &= \lambda a{:}\kappa.Seq\,[\![e : \tau]\!] \\
Seq\,[\![\gamma : \tau \sim \tau']\!] &= \gamma \\
Seq\,[\![\tau : \kappa]\!] &= \tau
\end{aligned}$$

Rather, most of the work of translating Core into Sequent Core is in handling the apparent computation. In the compositional translation, apparent computation expressions all correspond to computation μ-abstractions. Note that we now actually use the given type for a Core expression in order to record the type of the continuation variable introduced by the computation.

$$\begin{aligned}
Seq\,[\![e_1\ e_2 : \tau]\!] &= \mu r{:}\tau.\,\langle Seq\,[\![e_1 : \tau' \to \tau]\!] \| Seq\,[\![e_2 : \tau']\!] \cdot r\rangle \\
&\quad \textbf{where}\, e_1 : \tau' \to \tau \\
Seq\,[\![e_1\ \tau_2 : \tau]\!] &= \mu r{:}\tau.\,\langle Seq\,[\![e : \forall a{:}\kappa.\tau_2]\!] \| \tau_1 \cdot r\rangle \\
&\quad \textbf{where}\, e_1 : \forall a{:}\kappa.\tau_1 \\
&\qquad\quad \tau = \tau_1[\tau_2/a] \\
Seq\,[\![\textbf{let}\, bind\, \textbf{in}\, e : \tau]\!] &= \mu r{:}\tau.\,\textbf{let}\, Seq\,[\![bind]\!]\, \textbf{in}\, \langle e : \tau \| r\rangle \\
Seq\,[\![\textbf{case}\, e\, \textbf{as}\, x{:}\tau'\, \textbf{of}\, alts : \tau]\!] &= \mu r{:}\tau.\,\langle Seq\,[\![e : \tau']\!] \| \textbf{case as}\, x{:}\tau'\, \textbf{of}\, Seq\,[\![alts : \tau']\!]\, r : \tau\rangle \\
Seq\,[\![e \triangleright \gamma : \tau]\!] &= \mu r{:}\tau.\,\langle Seq\,[\![e : \tau']\!] \| \gamma \triangleleft r\rangle \\
&\quad \textbf{where}\, \gamma : \tau' \sim \tau
\end{aligned}$$

[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.

We also need to translate the binding of a let expression and alternatives of a case expression:

$$Seq\,[\![x{:}\tau = e]\!] = (x{:}\tau = Seq\,[\![e : \tau]\!])$$
$$Seq\,[\![\mathbf{rec}\{\overrightarrow{x_i{:}\tau_i = e_i}^{\,i}\}]\!] = (\mathbf{rec}\{\overrightarrow{x_i{:}\tau_i = Seq\,[\![e_i : \tau_i]\!]}^{\,i}\})$$

$$Seq\,[\![\_ \to e]\!]_{r:\tau} = \_ \to \langle Seq\,[\![e : \tau]\!] \| r\rangle$$
$$Seq\left[\!\!\left[K\ \overrightarrow{b_i{:}\kappa_i}^{\,i}\ \overrightarrow{x_j{:}\tau_j}^{\,j} \to e\right]\!\!\right]_{r:\tau} = K\ \overrightarrow{b_i{:}\kappa_i}^{\,i}\ \overrightarrow{x_j{:}\tau_j}^{\,j} \to \langle Seq\,[\![e : \tau]\!] \| r\rangle$$
$$Seq\,[\![lit \to e]\!]_{r:\tau} = lit \to \langle Seq\,[\![e : \tau]\!] \| r\rangle$$

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:

$$Seq\,[\![decls; e]\!]_{r:\tau} = decls; \langle Seq\,[\![e : \tau]\!] \| r\rangle$$

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

$$\begin{aligned}
Seq\,[\![x : \tau]\!] &= x \\
Seq\,[\![lit : \tau]\!] &= lit \\
Seq\,[\![(\lambda x{:}\tau.e) : \tau \to \tau']\!] &= \lambda x{:}\tau.Seq\,[\![e : \tau']\!] \\
Seq\,[\![(\lambda a{:}\kappa.e) : \forall a{:}\kappa.\tau]\!] &= \lambda a{:}\kappa.Seq\,[\![e : \tau]\!] \\
Seq\,[\![\gamma : \tau \sim \tau']\!] &= \gamma \\
Seq\,[\![\tau : \kappa]\!] &= \tau \\
Seq\,[\![e : \tau]\!] &= \mu q : \tau.Seq\,[\![e : \tau]\!]\ \ q\ \varepsilon \\
&\qquad \textbf{where}\ e \text{ is an apparent computation}
\end{aligned}$$

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

[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*).

this case, we introduce *one* μ-abstraction, and then begin to collect the outermost continuation and bindings of the computation:

$$\begin{aligned}
Seq\,[\![e_1\ e_2 : \tau]\!]\ k\ binds &= Seq\,[\![e_1 : \tau' \to \tau]\!]\ (Seq\,[\![e_2 : \tau']\!] \cdot k)\ binds \\
&\textbf{where}\ e_1 : \tau' \to \tau \\
Seq\,[\![e_1\ \tau_2 : \tau]\!]\ k\ binds &= Seq\,[\![e : \forall a{:}\kappa.\tau_2]\!]\ (\tau_1 \cdot k)\ binds \\
&\textbf{where}\ e_1 : \forall a{:}\kappa.\tau_1 \\
&\tau = \tau_1[\tau_2/a] \\
Seq\,[\![\textbf{let}\ bind\ \textbf{in}\ e : \tau]\!]\ k\ binds &= Seq\,[\![e : \tau]\!]\ k\ (binds; Seq\,[\![bind]\!]) \\
&\textbf{where}\ \emptyset = BV(bind) \cap FV(k) \\
Seq\,[\![\textbf{case}\ e\ \textbf{as}\ x{:}\tau'\ \textbf{of}\ alts : \tau]\!]\ k\ binds &= Seq\,[\![e : \tau']\!]\ k'\ binds' \\
&\textbf{where}\ k' = \textbf{case}\ \textbf{as}\ x{:}\tau'\ \textbf{of}\ Seq\,[\![alts]\!]\ (q : \tau)\ \varepsilon \\
&binds' = (binds; q{:}\tau = k) \\
Seq\,[\![e \triangleright \gamma : \tau]\!]\ k\ binds &= Seq\,[\![e : \tau']\!]\ (\gamma \triangleleft k)\ binds \\
&\textbf{where}\ \gamma : \tau' \sim \tau \\
Seq\,[\![e : \tau]\!]\ k\ binds &= \textbf{let}\ binds\ \textbf{in}\ \langle Seq\,[\![e : \tau]\!] \| k \rangle \\
&\textbf{where}\ e\ \text{is an apparent value}
\end{aligned}$$

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.

$$\begin{aligned}
Seq\,[\![x{:}\tau = e]\!] &= (x{:}\tau = Seq\,[\![e : \tau]\!]) \\
Seq\,[\![\textbf{rec}\{\overrightarrow{x_i{:}\tau_i = e_i}^{\,i}\}]\!] &= (\textbf{rec}\{\overrightarrow{x_i{:}\tau_i = Seq\,[\![e_i : \tau_i]\!]}^{\,i}\})
\end{aligned}$$

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:

$$\begin{aligned}
Seq\,[\![_ \to e]\!]\ (k : \tau)\ binds &= _ \to Seq\,[\![e : \tau]\!]\ k\ binds \\
Seq\,[\![K\ \overrightarrow{b_i{:}\kappa_i}^{\,i}\ \overrightarrow{x_j{:}\tau_j}^{\,j} \to e]\!]\ (k : \tau)\ binds &= K\ \overrightarrow{b_i{:}\kappa_i}^{\,i}\ \overrightarrow{x_j{:}\tau_j}^{\,j} \to Seq\,[\![e : \tau]\!]\ k\ binds \\
Seq\,[\![lit \to e]\!]\ (k : \tau)\ binds &= lit \to Seq\,[\![e : \tau]\!]\ k\ binds
\end{aligned}$$

$$Seq\,[\![decls; e]\!]_{r:\tau} = decls; Seq\,[\![e : \tau]\!]\ r\ \varepsilon$$

Translating Sequent Core back into Core brings up a twist: Sequent Core has types that don't exist in Core! In particular, the existential stack values don't exist directly as a Core expression. Therefore, we have to perform some selective *negation*[3] correspond to anything in Core. We represent the negation

[3]But only a single negation. Using a double negation means doing a full-blown continuation-passing style translation into Core, which is exactly what we are trying to avoid.

of a type τ as the normal encoding $\tau \to \tau'$, where the ultimate result type τ' stands in for "false." Thus, every Sequent Core type σ at least has a negated meaning in Core with respect to some result type τ', $Core^{\neg}[\![\sigma]\!]_{\tau'}$:

$$Core^{\neg}[\![\exists \overrightarrow{a_i : \kappa_i}^i.(\overrightarrow{\tau_j}^j)]\!]_{\tau'} = \overrightarrow{\forall a_i : \kappa_i.}^i \overrightarrow{\tau_j \to}^j \tau'$$
$$Core^{\neg}[\![\tau]\!]_{\tau'} = \tau \to \tau'$$

where we negate "exists" into "forall" and a stack producing a bunch of values to a function consuming a bunch of values.

By distinguishing Sequent Core types that exist in Core from ones that don't, we can get a mostly direct-style translation back to Core. For Sequent Core terms v of a Core type τ, we have the direct translation to Core, $Core[\![v : \tau]\!]$:

$$\begin{aligned}
Core[\![x : \tau]\!] &= x \\
Core[\![\mu r{:}\tau.c : \tau]\!] &= Core[\![c]\!]_{r:\tau} \\
Core[\![lit : \tau]\!] &= lit \\
Core[\![(\lambda x{:}\tau.v) : \tau \to \tau']\!] &= \lambda x{:}\tau.Core[\![v : \tau']\!] \\
Core[\![(\lambda a{:}\kappa.v) : \forall a{:}\kappa.\tau]\!] &= \lambda a{:}\kappa.Core[\![v : \tau]\!] \\
Core[\![\gamma : \tau \sim \tau']\!] &= \gamma \\
Core[\![\tau : \kappa]\!] &= \tau
\end{aligned}$$

Most clauses straightforward; the interesting one is for μ-abstractions, $\mu r{:}\tau.c$, where we translate the underlying command c and "read off" the result returned to the continuation variable r.

Likewise, Sequent Core continuations of a Core type τ have a direct translation to Core evaluation contexts, $Seq[\![k : \tau]\!]_{r:\tau'}$, which is parameterized by the return continuation variable r for the entire local computation:

$$Core[\![q : \tau]\!]_{r:\tau'} = \begin{cases} \Box & \text{if } q = r \\ q\ \Box & \text{if } q \neq r \end{cases}$$
$$\begin{aligned}
Core[\![v \cdot k : \tau_1 \to \tau_2]\!]_{r:\tau'} &= Core[\![k : \tau_2]\!]_{r:\tau'}\ [\Box\ Core[\![v : \tau_1]\!]] \\
Core[\![\tau_1 \cdot k : \forall a{:}\kappa.\tau_2]\!]_{r:\tau'} &= Core[\![k : \tau_2[\tau_1/a]]\!]_{r:\tau'}\ [\Box\ \tau_1] \\
Core[\![\gamma \triangleleft k : \tau_1]\!]_{r:\tau'} &= Core[\![k : \tau_2]\!]_{r:\tau'}\ [\Box \triangleright \gamma] \\
&\textbf{where}\ \gamma : \tau_1 \sim \tau_2 \\
Core[\![\textbf{case as}\ x{:}\tau\ \textbf{of}\ alts : \tau]\!]_{r:\tau'} &= \textbf{case}\ \Box\ \textbf{as}\ x{:}\tau\ \textbf{of}\ Core[\![alts]\!]_{r:\tau'}
\end{aligned}$$

$$\begin{aligned}
Core[\![_ \to c]\!]_{r:\tau'} &= _ \to Core[\![c]\!]_{r:\tau'} \\
Core\left[\!\left[K\ \overrightarrow{b_i : \kappa_i}^i\ \overrightarrow{x_j : \tau_j}^j \to c\right]\!\right]_{r:\tau'} &= K\ \overrightarrow{b_i : \kappa_i}^i\ \overrightarrow{x_j : \tau_j}^j \to Core[\![c]\!]_{r:\tau'} \\
Core[\![lit \to c]\!]_{r:\tau'} &= lit \to Core[\![c]\!]_{r:\tau'}
\end{aligned}$$

To translate the terms and continuations of non-Core types, we need to employ one level of negation which reverses their roles. The negated translation

of terms, $Core^{\neg}[\![v : \sigma]\!]$, and continuations $Core^{\neg}[\![k : \sigma]\!]_{r:\tau'}$, turns terms into Core evaluation contexts and continuations into Core expressions (specifically functions):

$$Core^{\neg}\left[\!\!\left[\overrightarrow{\tau_i}^i \cdot \overrightarrow{v_j}^j : \exists\overrightarrow{a_i{:}\kappa_i}^i.(\overrightarrow{\tau_j}^j)\right]\!\!\right] = \Box\ \overrightarrow{\tau_i}^i\ \overrightarrow{Core\,[\![v_j]\!]}^j$$

$$Core^{\neg}[\![v : \tau]\!] = \Box\ Core\,[\![v : \tau]\!]$$

$$Core^{\neg}\left[\!\!\left[\lambda\overrightarrow{a_i{:}\kappa_i}^i \cdot \overrightarrow{x_j{:}\tau_j}^j.c : \exists\overrightarrow{a_i{:}\kappa_i}^i.(\overrightarrow{\tau_j}^j)\right]\!\!\right]_{r:\tau'} = \overrightarrow{\lambda a_i{:}\kappa_i.}^i\,\overrightarrow{\lambda x_j{:}\tau_j.}^j Core\,[\![c]\!]_{r:\tau'}$$

$$Core^{\neg}[\![k : \tau]\!]_{r:\tau'} = \lambda x{:}\tau.Core\,[\![k : \tau]\!]_{r:\tau'}\,[x]$$

All of the interesting decisions on when to negate and when to be direct comes in the heart of a command. When translating a command into a Core expression, we must look at the type of the interacting term and continuation. If they have a Core type, then we can perform the direct translation whereby the term becomes an expression and a continuation its evaluation context. If they don't have a Core type, then we perform the negated translation whereby the term becomes an evaluation context for the continuation as a function.

$$Core\,[\![\mathbf{let}\,binds\,\mathbf{in}\,\langle v \| k\rangle]\!]_{r:\tau'} = Core\,[\![binds]\!]_{r:\tau'}\,[e]$$
$$\mathbf{where}\,e = \begin{cases} Core\,[\![k : \tau]\!]_{r:\tau'}\,[Core\,[\![v : \tau]\!]] & \text{if } \exists\tau.v : \tau \wedge k : \tau \\ Core^{\neg}[\![v : \sigma]\!]\,[Core^{\neg}[\![k : \sigma]\!]_{r:\tau'}] & \text{if } \exists\sigma.v : \sigma \wedge k : \sigma \end{cases}$$

With bindings, we don't have much of a choice: all bindings (for both terms and continuations) must become Core expression bindings. Thus, we always translate Sequent Core term bindings directly (which is possible since all term variables must have a Core type), and we always translate Sequent Core continuation bindings negatively (which converts them into functions).

$$Core\,[\![\varepsilon]\!]_{r:\tau'} = \Box$$
$$Core\,[\![binds; bind]\!]_{r:\tau'} = Core\,[\![binds]\!]_{r:\tau'}\,[Core\,[\![bind]\!]_{r:\tau'}]$$

$$Core\,[\![x{:}\tau = v]\!]_{r:\tau'} = \mathbf{let}\,x{:}\tau = Core\,[\![v : \tau]\!]\,\mathbf{in}\,\Box$$
$$Core\,[\![\mathbf{rec}\{\overrightarrow{x_i{:}\tau_i = v_i}^i\}]\!]_{r:\tau'} = \mathbf{let}\,\mathbf{rec}\{\overrightarrow{x_i{:}\tau_i = Core\,[\![v_i : \tau_i]\!]}^i\}\,\mathbf{in}\,\Box$$
$$Core\,[\![q{:}\sigma = k]\!]_{r:\tau'} = \mathbf{let}\,q{:}Core^{\neg}[\![\sigma]\!]_{\tau'} = Core^{\neg}[\![k]\!]_{r:\tau'}\,\mathbf{in}\,\Box$$
$$Core\left[\!\!\left[\mathbf{rec}\{\overrightarrow{q_i{:}\sigma_i = k_i}^i\}\right]\!\!\right]_{r:\tau'} = \mathbf{let}\,\mathbf{rec}\{\overrightarrow{q_i{:}Core^{\neg}[\![\sigma_i]\!]_{\tau'} = Core^{\neg}[\![k]\!]_{r:\tau'}}^i\}\,\mathbf{in}\,\Box$$

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.

$$Core\,[\![decls; c]\!]_{r:\tau} = decls; Core\,[\![c]\!]_{r:\tau}$$

5 Design discussion

5.1 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.

5.2 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 $\mathbf{let}\, x : \tau = \Box \in e$ 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.

5.3 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.

Sequent type kinding: $\Gamma \vdash \sigma : \kappa$

$$\frac{\overrightarrow{\Gamma, \overrightarrow{a_i : \kappa_i}^i \vdash \tau_j : \star}^j}{\Gamma \vdash \exists \overrightarrow{a_i : \kappa_i}^i . (\overrightarrow{\tau_j}^j) : \star}$$

Binding typing: $\Gamma; \Delta \vdash \mathit{binds} : \Gamma'; \Delta'$ and $\Gamma; \Delta \vdash \mathit{bind} : \Gamma'; \Delta'$

$$\frac{}{\Gamma; \Delta \vdash \varepsilon : \varepsilon; \varepsilon} \qquad \frac{\Gamma; \Delta \vdash \mathit{binds} : \Gamma'; \Delta' \quad \Gamma'; \Delta' \vdash \mathit{bind} : \Gamma''; \Delta''}{\Gamma; \Delta \vdash \mathit{binds}; \mathit{bind} : \Gamma''; \Delta''}$$

$$\frac{\Gamma \vdash v : \tau}{\Gamma; \Delta \vdash x{:}\tau = v : \Gamma, x : \tau; \Delta} \qquad \frac{\overrightarrow{\Gamma, \overrightarrow{x_j : \tau_j}^j \vdash v_i : \tau_i}^i}{\Gamma; \Delta \vdash \mathbf{rec}\, \overrightarrow{x_i{:}\tau_i = v_i}^i : \Gamma, \overrightarrow{x_i : \tau_i}^i; \Delta}$$

$$\frac{\Gamma; \Delta \vdash k : \sigma}{\Gamma; \Delta \vdash q{:}\sigma = k : \Gamma; \Delta, q : \sigma} \qquad \frac{\overrightarrow{\Gamma; \Delta, \overrightarrow{q_j : \sigma_i}^j \vdash k_i : \sigma_i}^i}{\Gamma; \Delta \vdash \mathbf{rec}\, \overrightarrow{q_i{:}\sigma_i = k_i}^i : \Gamma; \Delta, \overrightarrow{q_i : \sigma_i}^i}$$

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

$$\frac{\overrightarrow{\Gamma; \Delta \vdash \mathit{alt}_i : \tau}^i}{\Gamma; \Delta \vdash \overrightarrow{\mathit{alt}_i}^i : \tau} \qquad \frac{\Gamma; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash \_ \to c : \tau} \qquad \frac{\Gamma \vdash \mathit{lit} : \tau \quad \Gamma; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash \mathit{lit} \to c : \tau}$$

$$\frac{K : \overrightarrow{\forall a_j{:}\kappa_j.}^j \overrightarrow{\forall b_{j'}{:}\kappa'_{j'}.}^{j'} \overrightarrow{\tau_i \to}^i T\ \overrightarrow{a_j}^j \in \Gamma \quad \theta = [\overrightarrow{\tau_j / a_j}^j] \quad \Gamma, \overrightarrow{b_{j'} : \theta(\kappa'_{j'})}^{j'}, \overrightarrow{x_i : \theta(\tau_i)}^i; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash K\ \overrightarrow{b_{j'}{:}\theta(\kappa'_{j'})}^i\ \overrightarrow{x_i{:}\theta(\tau_i)}^i \to c : T\ \overrightarrow{\tau_j}^j}$$

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

$$\frac{\Gamma \vdash \mathit{decls} : \Gamma' \quad \Gamma'; \Delta \vdash c\,\mathrm{ok}}{\Gamma; \Delta \vdash \mathit{decls}; c\,\mathrm{ok}}$$

Further rules for $\Gamma \vdash \mathit{decls} : \Gamma'$ *and* $\Gamma \vdash \mathit{decl} : \Gamma'$.

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