

Typing Dual System FC and Sequent Core

July 16, 2015

1 Syntax

$x, y, z, f, g, h, K \in Var$	
$q, r \in KontVar$	
$a, b, T \in TypeVar$	
$c \in Command$	$::= \mathbf{let\ binds\ in} \langle v \ k \rangle$
$v \in Term$	$::= x \mid \lambda x:\tau. v \mid \mu q:\tau. c \mid lit \mid \tau \mid \gamma \mid \vec{\tau}_i^i \cdot \vec{v}_j^j$
$k \in Kont$	$::= q \mid v \cdot k \mid \gamma \triangleleft k \mid \mathbf{case\ as\ } x:\tau \mathbf{\ of\ } alts \mid \lambda \vec{a}_i:\vec{\kappa}_i^i \cdot \vec{x}_j:\vec{\tau}_j^j. c$
$binds \in Bindings$	$::= \vec{bind}_i^i$
$bind \in Binding$	$::= x:\tau = v \mid \mathbf{rec} \vec{x}_i:\vec{\tau}_i = \vec{v}_i^i \mid q:\sigma = k \mid \mathbf{rec} \vec{q}_i:\vec{\sigma}_i = \vec{k}_i^i$
$alts \in Alternatives$	$::= \vec{alt}_i^i$
$alt \in Alternative$	$::= _ \rightarrow c \mid K \vec{b}_i:\vec{\kappa}_i^i \vec{x}_j:\vec{\tau}_j^j \rightarrow c \mid lit \rightarrow c$
$\tau \in Type$	$::= \dots$
$\sigma \in SequentType$	$::= \tau \mid \exists \vec{a}_i:\vec{\kappa}_i^i. (\vec{\tau}_j^j)$
$\kappa \in Kind$	$::= \dots$
$\gamma \in Coercion$	$::= \dots$
$decls \in Declarations$	$::= \vec{decl}_i^i$
$decl \in Declaration$	$::= \dots$
$pgm \in Program$	$::= decls; c$

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 $\gamma_1 \triangleleft \gamma_2 \triangleleft q$ is the same as $\gamma_1 \triangleleft (\gamma_2 \triangleleft q)$. Both function calls and coercions share the same precedence and may be intermixed, so that $1 \cdot \gamma_2 \triangleleft 3 \cdot q$ is the same as $1 \cdot (\gamma_2 \triangleleft (3 \cdot q))$.
- If a command does not have any associated bindings with it, so that *binds* is empty in **let** k **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 **case as** $x:\tau$ **of** *alts* when it turns out that x is never referenced in *alts* or *alts*, instead writing **case** *alts*. If instead $x:\tau$ is only referenced in the default alternative $_ \rightarrow c$ 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, **case as** $x:\tau$ **of** $_ \rightarrow c$, which we write as the shortened **case** $x:\tau \rightarrow c$.

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:

$$\begin{array}{lcl}
\Gamma \in \textit{Environment} & ::= \varepsilon \mid \Gamma, x \mid \Gamma, a \\
\Delta \in \textit{KoEnvironment} & ::= \varepsilon \mid \Delta, q \\
\\
\text{Term scoping: } \Gamma \vdash v \text{ ok} \\
\\
\frac{x \in \Gamma}{\Gamma \vdash x \text{ ok}} \quad \frac{}{\Gamma \vdash \textit{lit} \text{ ok}} \quad \frac{\Gamma; q \vdash c \text{ ok}}{\Gamma \vdash \mu q. c \text{ ok}} \\
\\
\frac{\Gamma, x \vdash v \text{ ok}}{\Gamma \vdash \lambda x. v \text{ ok}} \quad \frac{\Gamma, a \vdash v \text{ ok}}{\Gamma \vdash \lambda a. v \text{ ok}} \quad \frac{\overline{\Gamma \vdash \tau_i \text{ ok}}^i \quad \overline{\Gamma \vdash v_j \text{ ok}}^j}{\Gamma \vdash \vec{\tau}_i^i \cdot \vec{v}_j^j \text{ ok}} \\
\\
\text{Continuation scoping: } \Gamma; \Delta \vdash k \text{ ok} \\
\\
\frac{q \in \Delta}{\Gamma; \Delta \vdash q \text{ ok}} \quad \frac{\Gamma \vdash v \text{ ok} \quad \Gamma; \Delta \vdash k \text{ ok}}{\Gamma; \Delta \vdash v \cdot k \text{ ok}} \quad \frac{\Gamma \vdash \tau \text{ ok} \quad \Gamma; \Delta \vdash k \text{ ok}}{\Gamma; \Delta \vdash \tau \cdot k \text{ ok}} \\
\\
\frac{\Gamma \vdash \gamma \text{ ok} \quad \Gamma; \Delta \vdash k \text{ ok}}{\Gamma; \Delta \vdash \gamma \triangleleft k \text{ ok}} \quad \frac{\Gamma, x; \Delta \vdash \textit{alts} \text{ ok}}{\Gamma; \Delta \vdash \mathbf{case as } x \mathbf{ of alts} \text{ ok}} \quad \frac{\Gamma, \vec{a}_i^i, \vec{x}_j^j; \Delta \vdash c \text{ ok}}{\Gamma; \Delta \vdash \lambda \vec{a}_i^i \cdot \vec{x}_j^j. c \text{ ok}} \\
\\
\text{Command scoping: } \Gamma; \Delta \vdash c \text{ ok} \\
\\
\frac{\Gamma; \Delta \vdash \textit{binds} : \Gamma'; \Delta' \quad \Gamma, \Gamma' \vdash v \text{ ok} \quad \Gamma, \Gamma'; \Delta, \Delta' \vdash k \text{ ok}}{\Gamma; \Delta \vdash \mathbf{let binds in } \langle v \| k \rangle \text{ ok}} \\
\\
\text{Further rules for } \Gamma \vdash \tau \text{ ok and } \Gamma \vdash \kappa \text{ ok, } \Gamma \vdash \gamma \text{ ok}
\end{array}$$

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: $\Gamma; \Delta \vdash \text{bind} : \Gamma'; \Delta'$ and $\Gamma; \Delta \vdash \text{alt} \text{ ok}$

$$\begin{array}{c}
\frac{}{\Gamma; \Delta \vdash \varepsilon : \varepsilon; \varepsilon} \quad \frac{\Gamma; \Delta \vdash \text{binds} : \Gamma'; \Delta' \quad \Gamma'; \Delta' \vdash \text{bind} : \Gamma''; \Delta''}{\Gamma; \Delta \vdash \text{binds}; \text{bind} : \Gamma''; \Delta''} \\
\\
\frac{\Gamma \vdash v \text{ ok}}{\Gamma; \Delta \vdash x = v : \Gamma, x; \Delta} \quad \frac{\overrightarrow{\Gamma, \vec{x}_j^j \vdash v_i \text{ ok}}^i}{\Gamma; \Delta \vdash \text{rec } \vec{x}_i = \vec{v}_i^i : \Gamma, \vec{x}_i^i; \Delta} \\
\\
\frac{\Gamma; \Delta \vdash k \text{ ok}}{\Gamma; \Delta \vdash q = k : \Gamma; \Delta, q} \quad \frac{\overrightarrow{\Gamma; \Delta, \vec{q}_j^j \vdash k_i \text{ ok}}^i}{\Gamma; \Delta \vdash \text{rec } q_i = \vec{k}_i^i : \Gamma; \Delta, \vec{q}_i^i} \\
\\
\frac{\overrightarrow{\Gamma; \Delta \vdash \text{alt}_i \text{ ok}}^i}{\Gamma; \Delta \vdash \overrightarrow{\text{alt}_i}^i \text{ ok}} \quad \frac{\Gamma; \Delta \vdash c \text{ ok}}{\Gamma; \Delta \vdash _ \rightarrow c \text{ ok}} \quad \frac{\Gamma, \vec{b}_i^i, \vec{x}_i^i; \Delta \vdash c \text{ ok}}{\Gamma; \Delta \vdash K \vec{b}_i^i \vec{x}_i^i \rightarrow c \text{ ok}} \quad \frac{\Gamma; \Delta \vdash c \text{ ok}}{\Gamma; \Delta \vdash \text{lit} \rightarrow c \text{ ok}} \\
\\
\text{Program scoping: } \Gamma; \Delta \vdash \text{pgm} \text{ ok} \\
\\
\frac{\Gamma \vdash \text{decls} : \Gamma' \quad \Gamma'; \Delta \vdash c \text{ ok}}{\Gamma; \Delta \vdash \text{decls}; c \text{ ok}}
\end{array}$$

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

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:

$$\text{rec}\{f = \lambda x.v; q = \text{case } y \rightarrow c\}$$

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:

$$\text{rec}\{f = \lambda x.v\}; \text{rec}\{q = \text{case } y \rightarrow c\}$$

So this limitation results in no loss of expressiveness. Indeed, we could further partition 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}{l}
\Gamma \in \textit{Environment} \quad \quad \quad ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, a : \kappa \\
\Delta \in \textit{KoEnvironment} \quad \quad \quad ::= \varepsilon \mid \Delta, q : \sigma \\
\\
\text{Term typing: } \Gamma \vdash v : \sigma \\
\\
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{\tau = \textit{literalType}(\textit{lit})}{\Gamma \vdash \textit{lit} : \tau} \quad \frac{\Gamma; q : \tau \vdash \textit{c 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 \rightarrow \tau_2} \quad \frac{\Gamma, a : \kappa \vdash v : \tau}{\Gamma \vdash \lambda a : \kappa . v : \forall a : \kappa . \tau} \quad \frac{\overrightarrow{\Gamma \vdash \tau_i : \kappa_i} \quad \overrightarrow{\Gamma \vdash v_j : \tau'_j[\tau_i/a_i]}}{\Gamma \vdash \overrightarrow{\tau_i}^i \cdot \overrightarrow{v_j}^j : \exists \overrightarrow{a_i} : \overrightarrow{\kappa_i}^i . (\overrightarrow{\tau_j}^j)} \\
\\
\text{Continuation typing: } \Gamma; \Delta \vdash k : \sigma \\
\\
\frac{q : \sigma \in \Delta}{\Gamma; \Delta \vdash q : \sigma} \quad \frac{\Gamma \vdash v : \tau_1 \quad \Gamma; \Delta \vdash k : \tau_2}{\Gamma; \Delta \vdash v \cdot k : \tau_1 \rightarrow \tau_2} \quad \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} \quad \frac{\Gamma, x : \tau; \Delta \vdash \textit{alts} : \tau}{\Gamma; \Delta \vdash \textbf{case as } x : \tau \textbf{ of } \textit{alts} : \tau} \\
\\
\frac{\Gamma, \overrightarrow{a_i} : \overrightarrow{\kappa_i}^i, \overrightarrow{x_j} : \overrightarrow{\tau_j}^j; \Delta \vdash \textit{c ok}}{\Gamma; \Delta \vdash \lambda \overrightarrow{a_i} : \overrightarrow{\kappa_i}^i \cdot \overrightarrow{x_j} : \overrightarrow{\tau_j}^j . c : \exists \overrightarrow{a_i} : \overrightarrow{\kappa_i}^i . (\overrightarrow{\tau_j}^j)} \\
\\
\text{Command typing: } \Gamma; \Delta \vdash \textit{c ok} \\
\\
\frac{\Gamma; \Delta \vdash \textit{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 \textbf{let binds in } \langle v \| k \rangle \textit{ ok}} \\
\\
\textit{Further rules for } \Gamma \vdash \tau : \kappa, \Gamma \vdash \kappa : \delta, \text{ and } \Gamma \vdash \gamma : \tau_1 \sim \tau_2.
\end{array}$$

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 $\Gamma_0; exit : \tau \vdash \text{pgm ok}$, 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 **let** k **in** $\langle binds \parallel v \rangle$, not only do v and k agree on the same (inferred) type τ , 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 $\forall a:\star.a \rightarrow a \rightarrow Bool$, $\forall a:\star.Int \rightarrow a \rightarrow Bool$, 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 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, $\langle (\cdot) \parallel \tau \cdot v \cdot q \rangle$. 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, the type checking algorithm for Sequent Core reveals some of its unsymmetrical bias. On the one hand, reading the type off of a Sequent Core term (including first-class coercions) is straightforward, since they are a subset of Core expressions. On the other hand, checking that a continuation has a known type is also straightforward. 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.
- Read the type off of the term in the command, confirming that the term is well-typed.
- Check that the continuation in the command has the same type as the term.

This explicit flow of type information (from term to continuation) likely has something to do with bi-directional type checking. Intuitively, this situation is identical to the problem with inferring the general existential tuples. And by duality, the solution would be to flip the flow of type information the other way: read off the type of the *continuation* that uses an existential tuple, and then check the tuple against the inferred type.

4 Design discussion

4.1 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

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

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

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

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:

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

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 :

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

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:

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

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:

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

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 :

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

which avoids duplicating c_1 at all.

4.2 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 **let** $x : \tau = \square \in e$ 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 $\lambda \vec{a}:\vec{\kappa} \cdot \vec{x}:\vec{\tau}.c$ would just be written as another form of case continuation **case of** $a:\kappa \cdot x:\tau \rightarrow c$. 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.

4.3 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: $\Gamma \vdash \sigma : \kappa$

$$\frac{\overline{\Gamma, \bar{a}_i : \kappa_i^{\rightarrow i} \vdash \tau_j : \star}^{\rightarrow j}}{\Gamma \vdash \exists \bar{a}_i : \kappa_i^{\rightarrow i}. (\bar{\tau}_j^{\rightarrow j}) : \star}$$

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

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

$$\frac{\Gamma \vdash v : \tau}{\Gamma; \Delta \vdash x : \tau = v : \Gamma, x : \tau; \Delta} \quad \frac{\overline{\Gamma, \bar{x}_j : \tau_j^{\rightarrow j} \vdash v_i : \tau_i}^{\rightarrow i}}{\Gamma; \Delta \vdash \mathbf{rec} \bar{x}_i : \tau_i = \bar{v}_i^{\rightarrow i} : \Gamma, \bar{x}_i : \tau_i^{\rightarrow i}; \Delta}$$

$$\frac{\Gamma; \Delta \vdash k : \sigma}{\Gamma; \Delta \vdash q : \sigma = k : \Gamma; \Delta, q : \sigma} \quad \frac{\overline{\Gamma; \Delta, \bar{q}_j : \sigma_j^{\rightarrow j} \vdash k_i : \sigma_i}^{\rightarrow i}}{\Gamma; \Delta \vdash \mathbf{rec} \bar{q}_i : \sigma_i = \bar{k}_i^{\rightarrow i} : \Gamma; \Delta, \bar{q}_i : \sigma_i^{\rightarrow i}}$$

Alternative typing: $\Gamma; \Delta \vdash alt : \tau$

$$\frac{\overline{\Gamma; \Delta \vdash alt_i : \tau}^{\rightarrow i}}{\Gamma; \Delta \vdash \overline{alt_i}^{\rightarrow i} : \tau} \quad \frac{\Gamma; \Delta \vdash c \text{ ok}}{\Gamma; \Delta \vdash _ \rightarrow c : \tau} \quad \frac{\Gamma \vdash lit : \tau \quad \Gamma; \Delta \vdash c \text{ ok}}{\Gamma; \Delta \vdash lit \rightarrow c : \tau}$$

$$\frac{K : \forall \bar{a}_j : \kappa_j. \forall \bar{b}_{j'} : \kappa_{j'}^{\rightarrow j'}. \overline{\tau_i \rightarrow^i T \bar{a}_j^{\rightarrow j} \in \Gamma}^{\rightarrow j'} \quad \theta = [\bar{\tau}_j / \bar{a}_j^{\rightarrow j}] \quad \Gamma, \bar{b}_{j'} : \theta(\kappa_{j'}^{\rightarrow j'}), \bar{x}_i : \theta(\tau_i)^{\rightarrow i}; \Delta \vdash c \text{ ok}}{\Gamma; \Delta \vdash K \overline{b_{j'} : \theta(\kappa_{j'}^{\rightarrow j'})}^{\rightarrow i} \overline{x_i : \theta(\tau_i)^{\rightarrow i}}^{\rightarrow i} \rightarrow c : T \bar{\tau}_j^{\rightarrow j}}$$

Program typing: $\Gamma; \Delta \vdash pgm \text{ ok}$

$$\frac{\Gamma \vdash decls : \Gamma' \quad \Gamma'; \Delta \vdash c \text{ ok}}{\Gamma; \Delta \vdash decls; c \text{ ok}}$$

Further rules for $\Gamma \vdash decls : \Gamma'$ and $\Gamma \vdash decl : \Gamma'$.

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