Typing Dual System FC and Sequent Core

July 10, 2015

1 Syntax

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

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 $\gamma_1 \triangleleft \gamma_2 \triangleleft q$ 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 $\mathbf{let}\, k\, \mathbf{in}\, \langle \mathit{binds} \| v \rangle$, 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 $\mathbf{case\,as}\, x{:}\tau\, \mathbf{of}\, \mathit{alts}$ when it turns out that x is never referenced in *alts* or *alts*, instead writing $\mathbf{case}\, \mathit{alts}$. 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, $\mathbf{case\,as}\, x{:}\tau\, \mathbf{of}\, \_ \to c$, 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:

- 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.

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

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

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

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

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

$$\frac{\Gamma \vdash \gamma \,\mathrm{ok} \quad \Gamma; \Delta \vdash k \,\mathrm{ok}}{\Gamma; \Delta \vdash \gamma \triangleleft k \,\mathrm{ok}} \qquad \frac{\Gamma, x; \Delta \vdash \mathit{alts} \,\mathrm{ok}}{\Gamma; \Delta \vdash \mathbf{case\ as}\, x \,\mathbf{of}\, \mathit{alts} \,\mathrm{ok}}$$

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

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 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:

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

Binding and alternative scoping: $\Gamma; \Delta \vdash \mathit{bind} : \Gamma'; \Delta'$ and $\Gamma; \Delta \vdash \mathit{alt}\, \mathrm{ok}$

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

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

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

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

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

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

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

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

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

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

$$
\frac{\Gamma, x : \tau_1 \vdash v : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.v : \tau_1 \to \tau_2}
\qquad
\frac{\Gamma, a : \kappa \vdash v : \tau}{\Gamma \vdash \lambda a{:}\kappa.v : \forall a{:}\kappa.\tau}
$$

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

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

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

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

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

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

type information (data types, type synonyms, and axioms). The normal way to type-check a top-level program is $\Gamma_0; \mathit{exit} : \tau \vdash \mathit{pgm}\,\mathrm{ok}$, where Γ_0 specifies any primitive types and values provided by the run-time environment, and $\mathit{exit} : \tau$ 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 $\mathbf{let}\, k\, \mathbf{in}\, \langle \mathit{binds} \| v \rangle$, 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

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

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

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

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

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

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 $\mathit{Int} \cdot 1 \cdot 2 \cdot q$ can be given the types $\forall a{:}\star.a \to a \to \mathit{Bool}$, $\forall a{:}\star.\mathit{Int} \to a \to \mathit{Bool}$, 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, $\langle (,) \| \tau \cdot v \cdot q \rangle$. That means that all the troubles with checking existential structures is contained solely with polymorphic call stacks.

- The rule for coercion continuations, $\gamma \lhd k : \tau_1$, 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.