Dual System FC and Sequent Core[1]

22 June, 2015

1 Introduction

System FC has proved to be a powerful system for expressing programs and their transformations in GHC's optimization passes. However, some operations are most naturally expressed in a language that makes control flow, and the related notion of an evaluation context, explicit. Different encodings such as the continuation-passing style exist to reify contexts; an elegant alternative is to restructure the representation from a lambda-calculus to a *sequent calculus*. Dual System FC is such a calculus, and Sequent Core is a proposed intermediate representation refining Dual System FC for Haskell's particular needs.

2 Grammar

2.1 Metavariables

We will use the following metavariables:

x	Term variable names
k	Continuation variable names
α, β	Type-level variable names
N	Type-level constructor names
K	Term-level data constructor names
M	Axiom rule names
i, j, k, a, b, c, n, m	Indices to be used in lists

2.2 Literals

Literals do not play a major role, so we leave them abstract:

lit	::=	Literals

We also leave abstract the function *basicTypes/Literal.lhs*:`literalType` and the judgment *coreSyn/CoreLint.lhs*:`lintTyLit` (written $\Gamma \vdash_{\mathsf{tylit}} \mathsf{lit} : \kappa$).

2.3 Variables

Sequent Core follows GHC's use of the same datatype to represent term-level variables and type-level variables:

z	::=		Term, continuation, or type name
	|	α	Type-level name
	|	x	Term name
	|	k	Continuation name
n, m	::=		Variable names, *basicTypes/Var.lhs*:`Var`
	|	z^τ	Name, labeled with type/kind

[1]This document was originally prepared by Luke Maurer (`maurerl@cs.uoregon.edu`) based on the System FC formalism by Richard Eisenberg (`eir@cis.upenn.edu`), but it should be maintained by anyone who edits the functions or data structures mentioned in this file. Please feel free to contact Richard for more information.

Continuation	Context
$\$\, v; e$	$e[\Box\ v]$
$\triangleright\, \gamma; e$	$e[\Box \triangleright \gamma]$
$\{tick\}; e$	$e[\Box_{\{tick\}}]$
$\mathbf{case}\ \mathbf{as}\ n\ \mathbf{of}\ \overline{alt_i}^{\,i}$	$\mathbf{case}\ \Box\ \mathbf{as}\ n\ \mathbf{of}\ \overline{alt_i}^{\,i}$

Table 1: Continuations in Dual System FC and equivalent contexts in System FC.

2.4 Expressions

Being a sequent calculus, Dual System FC divides terms into the syntactic categories of *terms*, *continuations*, and *commands*. A command represents an act of computation in which a value interacts with a continuation; in particular, a term *produces* data that a continuation *consumes*.

Terms largely correspond to WHNF terms in Core:

$v,\ w$	::=		Terms, *Syntax.hs*:`Term`
	|	n	Variable
	|	lit	Literal
	|	$\lambda n.v$	Abstraction
	|	$\mathbf{compute}\ n.c$	Computation
	|	τ	Type
	|	γ	Coercion
	|	e	Continuation

Similarly to Core's `Expr` type, types and coercions should only appear in terms that are either function arguments (see the continuation syntax) or right-hand sides of **let**s (see the command syntax). Additionally, continuations as terms should only appear as right-hand sides. None of these are first-class values.

There is one major novelty in `Term`: The **compute** form wraps a general computation as a term. It binds a continuation variable for the context in which the computation is to occur. (In the literature, this is usually written as a μ-abstraction.)

Continuations represent strict Haskell contexts, only "inside-out":

e	::=		Continuations, *Syntax.hs*:`Cont`
	|	$\mathbf{ret}\ n$	Continuation variable
	|	$\$\, v; e$	Application
	|	$\triangleright\, \gamma; e$	Cast
	|	$\{tick\}; e$	Internal note
	|	$\mathbf{case}\ \mathbf{as}\ n\ \mathbf{of}\ \overline{alt_i}^{\,i}$	Pattern match
alt	::=		Case alternative, *Syntax.hs*:`Alt`
	|	$\mathbb{K}\, \overline{n_i}^{\,i} \rightarrow c$	Constructor applied to fresh names

Each constructor (besides **ret**) in `Cont` corresponds to a particular fragment of Core (Table 1).

Finally, a `Command` describes the interaction between a value and its continuation; in addition, it may introduce bindings into the context:

c	::=		Commands, *Syntax.hs*:`Command`
	|	$\mathbf{let}\ bindings\ \mathbf{in}\ \langle v \mid e \rangle$	Command
$bindings$	::=		Series of let-bindings
	|	$\overline{binding_i}^{\,i}$	List of bindings
$binding$	::=		Let-bindings, *Syntax.hs*:`Bind`

	$n = v$	Non-recursive binding
	$\mathbf{rec}\ \overline{n_i = v_i}^{\,i}$	Recursive binding

Some of the invariants in the Core datatypes carry over into Sequent Core:

- The right-hand sides of all top-level and recursive bindings must be of lifted type.
- The right-hand side of a non-recursive binding and the argument of an application may be of unlifted type, but only if the expression is ok-for-speculation. See `#let_app_invariant#` in *coreSyn/CoreSyn.lhs*. [2]
- We allow a non-recursive binding to bind a type variable, coercion variable, or continuation variable.

Finally, as in Core, a program is just a list of bindings:

program	::=		A Sequent Core program, *Syntax.hs*:`SeqCoreProgram`
		$\overline{binding_i}^{\,i}$	List of bindings

2.5 Types

Types are unchanged from System FC, except that we introduce a type $\mathsf{Cont}_{\#}\,\tau$ for each type τ of kind $*$ or $\#$. This is the type of a continuation expecting a τ. Since continuations cannot be passed as arguments, continuation types can't parameterize polymorphic functions; thus we introduce the kinds $\mathsf{ContKind}\,*$ and $\mathsf{ContKind}\,\#$ for $\mathsf{Cont}_{\#}\,\tau$ when τ has kind $*$ or $\#$, respectively.

The type $\mathsf{Cont}_{\#}\,\tau$ is only used when a computation is bound by a **let**; as a continuation used in a command, we write that its type is simply τ.

3 Typing judgments

Here we elide many gory details, such as binding consistency, to focus on the areas that are particular to Dual System FC. Other judgments are largely unchanged from Core.

The typing judgments for terms have no surprises:

$\boxed{\Gamma \vdash_{\mathsf{vl}} v : \tau}$ Term typing, *Lint.hs*:`lintCoreTerm`

$$\frac{\begin{array}{l} x^{\tau} \in \Gamma \\ \neg\,(\exists \gamma \text{ s.t. } \tau = \gamma) \end{array}}{\Gamma \vdash_{\mathsf{vl}} x^{\tau} : \tau}\ \textsc{Vl\_Var}$$

$$\frac{\tau = \mathsf{literalType}\,\mathsf{lit}}{\Gamma \vdash_{\mathsf{vl}} \mathsf{lit} : \tau}\ \textsc{Vl\_Lit}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \sigma : \kappa_2 \\ \Gamma, x^{\tau} \vdash_{\mathsf{vl}} v : \sigma \end{array}}{\Gamma \vdash_{\mathsf{vl}} \lambda x^{\tau}.v : \tau \to \sigma}\ \textsc{Vl\_LamId}$$

[2] Allowing strict binds stretches the idea that Dual System FC makes the evaluation order explicit using commands; however, by design, a computation that's ok for speculation has negligible cost and always succeeds, and thus the "detour" it causes is minor.

$$\frac{\Gamma \vdash_{\mathsf{k}} \kappa\ \mathsf{ok} \quad \Gamma, \alpha^{\kappa} \vdash_{\mathsf{vl}} v : \sigma}{\Gamma \vdash_{\mathsf{vl}} \lambda \alpha^{\kappa}.v : \forall \alpha^{\kappa}.\sigma}\ \textsc{Vl\_LamTy}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa \quad \Gamma, k^{\tau} \vdash_{\mathsf{cm}} c}{\Gamma \vdash_{\mathsf{vl}} \mathbf{compute}\, k^{\tau}.c : \tau}\ \textsc{Vl\_Compute}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim^{\kappa}_{\mathsf{N}} \tau_2}{\Gamma \vdash_{\mathsf{vl}} \gamma : \tau_1 \sim^{\kappa}_{\#} \tau_2}\ \textsc{Vl\_CoercionNom}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim^{\kappa}_{\mathsf{R}} \tau_2}{\Gamma \vdash_{\mathsf{vl}} \gamma : (\sim_{\mathsf{R}\#})\, \kappa\, \tau_1\, \tau_2}\ \textsc{Vl\_CoercionRep}$$

Whereas a term has the type of the data it produces, a continuation has the type of the data it *consumes*. Thus a function application supplies a term, but it always has a function type (whose argument type, of course, must match the type of the term).

$\boxed{\Gamma \vdash_{\mathsf{ct}} e : \tau}$ Continuation typing, *Lint.hs*:`lintCoreCont`

$$\frac{k^{\tau} \in \Gamma}{\Gamma \vdash_{\mathsf{ct}} \mathbf{ret}\, k^{\tau} : \tau}\ \textsc{Ct\_Return}$$

$$\frac{\Gamma \vdash_{\mathsf{vl}} v : \tau \quad \neg\,(\exists e \text{ s.t. } v = e) \quad \Gamma \vdash_{\mathsf{ct}} e : \sigma}{\Gamma \vdash_{\mathsf{ct}} \$\, v; e : \tau \to \sigma}\ \textsc{Ct\_ValApp}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa \quad \Gamma \vdash_{\mathsf{k}} \kappa\ \mathsf{ok} \quad \Gamma \vdash_{\mathsf{ct}} e : \sigma[\alpha^{\kappa} \mapsto \tau]}{\Gamma \vdash_{\mathsf{ct}} \$\, \tau; e : \forall \alpha^{\kappa}.\sigma}\ \textsc{Ct\_TyApp}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa \quad \overline{\Gamma, z^{\tau}; \tau \vdash_{\mathsf{alt}} alt_i}^{\,i}}{\Gamma \vdash_{\mathsf{ct}} \mathbf{case}\, \mathbf{as}\, z^{\tau}\, \mathbf{of}\, \overline{alt_i}^{\,i} : \tau}\ \textsc{Ct\_Case}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim^{\kappa}_{\mathsf{R}} \tau_2 \quad \Gamma \vdash_{\mathsf{ct}} e : \tau_2}{\Gamma \vdash_{\mathsf{ct}} \triangleright \gamma; e : \tau_1}\ \textsc{Ct\_Cast}$$

$$\frac{\Gamma \vdash_{\mathsf{ct}} e : \tau}{\Gamma \vdash_{\mathsf{ct}} \{tick\}; e : \tau} \quad \textsc{Ct\_Tick}$$

A command does not in itself have a type, but each of its bindings must be well-typed, and its term's type must match its continuation's type:

$\boxed{\Gamma \vdash_{\mathsf{cm}} c}$ Command typing, *Lint.hs*:`lintCoreCommand`

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{binds}} \mathit{bindings} \\ \Gamma' = \Gamma, \mathsf{vars\_of}\ \mathit{bindings} \\ \Gamma' \vdash_{\mathsf{vl}} v : \tau \\ \Gamma' \vdash_{\mathsf{ct}} e : \tau\end{array}}{\Gamma \vdash_{\mathsf{cm}} \mathbf{let}\ \mathit{bindings}\ \mathbf{in}\ \langle v \mid e \rangle} \quad \textsc{Cm\_Cut}$$

3.1 Linearity

There is an important condition that has not yet been mentioned, namely that of *linearity*. Properly, any continuation parameter should have a linear type, and any bound continuation variable should have an affine type (since it may not be used in all branches). This is not checked explicitly by Lint. However, an equivalent condition is checked, and indeed is required, by the translation functions:

> At any point in the program besides a top-level term, there is a continuation that acts as "the current continuation." Every **ret** must invoke either:
>
> 1. the current continuation, or
> 2. a **let**-bound continuation whose current continuation is the same as that of the **ret**.

The current continuation is the one that represents the immediate context—in other words, $\mathbf{ret}\, k^\tau$ represents the context $\Box$ when k^τ is the current continuation. A bound continuation corresponds to a join point in Core (and indeed is translated as one by `Translate.hs`); the requirement above that its current continuation is the same as the **ret** that calls it means that the current continuation will always eventually be invoked ($\bot$ notwithstanding).

An alternative formulation can be given in terms of the operation of the type checker:

> The current continuation is the most recently bound continuation parameter. (By parameter we mean a variable arising as a bound identifier from a **compute** or a λ-abstraction.) Whenever recursing underneath a continuation parameter, clear all other continuations from the context.

Clearing bound continuations from the context frees us from explicitly restricting the argument to **ret**.

Subtle though it may seem, this rule is very important: Without it, Sequent Core would accidentally have added control operators to Haskell!

4 Operational semantics

4.1 Disclaimer

The Sequent Core library does not implement an operational semantics in any concrete form. Most of the rules below are implied by algorithms in, for example, the simplifier and optimizer. Yet, there is no one place in the library that states these rules, analogously to `Lint.hs`. Nevertheless, these rules are included in this document to help the reader understand Dual System FC.

4.2 Operational semantics rules

$\boxed{\Sigma \vdash_{\mathsf{op}} c \longrightarrow c'}$ Single step semantics

$$\frac{\Sigma(n) = v}{\Sigma \vdash_{\mathsf{op}} \langle n \mid e \rangle \longrightarrow \langle v \mid e \rangle} \quad \text{c\_Var}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} \langle \lambda n.v \mid \$\, v'; e \rangle \longrightarrow \langle v\,[n \mapsto v'] \mid e \rangle} \quad \text{c\_Beta}$$

$$\frac{\begin{array}{l}\gamma_0 = \mathsf{sym}\,(\mathsf{nth}_0\,\gamma) \\ \gamma_1 = \mathsf{nth}_1\,\gamma \\ \neg\,\exists \tau \text{ s.t. } v_2 = \tau \\ \neg\,\exists \gamma \text{ s.t. } v_2 = \gamma \\ \tau \to \sigma = \mathsf{coercionKind}\,\gamma_0\end{array}}{\Sigma \vdash_{\mathsf{op}} \langle v \mid \triangleright\,\gamma; \$\, v_2; e \rangle \longrightarrow \langle v \mid \$\, \mathbf{compute}\, k^\sigma.\langle v_2 \mid \triangleright\,\gamma_0; \mathbf{ret}\, k^\sigma \rangle; \triangleright\,\gamma_1; e \rangle} \quad \text{c\_Push}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} \langle v \mid \triangleright\,\gamma; \$\,\tau; e \rangle \longrightarrow \langle v \mid \$\,\tau; \triangleright\,\gamma\,\tau; e \rangle} \quad \text{c\_TPush}$$

$$\frac{\begin{array}{l}\gamma_0 = \mathsf{nth}_1\,(\mathsf{nth}_0\,\gamma) \\ \gamma_1 = \mathsf{sym}\,(\mathsf{nth}_2\,(\mathsf{nth}_0\,\gamma)) \\ \gamma_2 = \mathsf{nth}_1\,\gamma\end{array}}{\Sigma \vdash_{\mathsf{op}} \langle v \mid \triangleright\,\gamma; \$\,\gamma'; e \rangle \longrightarrow \langle v \mid \$\,(\gamma_0 \fatsemi \gamma' \fatsemi \gamma_1); \triangleright\,\gamma_2; e \rangle} \quad \text{c\_CPush}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} \mathbf{let}\; n = w \;\mathbf{in}\; c \longrightarrow c\,[n \mapsto w]} \quad \text{c\_LetNonRec}$$

$$\frac{\Sigma,\, \overline{[n_i \mapsto v_i]}^{\,i} \vdash_{\mathsf{op}} c \longrightarrow c'}{\Sigma \vdash_{\mathsf{op}} \mathbf{let\,rec}\; \overline{n_i = v_i}^{\,i} \;\mathbf{in}\; c \longrightarrow \mathbf{let\,rec}\; \overline{n_i = v_i}^{\,i} \;\mathbf{in}\; c'} \quad \text{c\_LetRec}$$

$$\frac{fv(c) \cap \overline{n_i}^{\,i} = \cdot}{\Sigma \vdash_{\mathsf{op}} \mathbf{let\,rec}\; \overline{n_i = v_i}^{\,i} \;\mathbf{in}\; c \longrightarrow c} \quad \text{c\_LetRecReturn}$$

$$\frac{\begin{array}{l} alt_b = K\ \overline{\alpha_j{}^{\kappa_j}}^{\,j}\ \overline{x_k{}^{\tau_k}}^{\,k}\ \to c \\ subst = [n \mapsto v]\,\overline{[\alpha_j{}^{\kappa_j} \mapsto \sigma_j]}^{\,j}\ \overline{[x_k{}^{\tau_k} \mapsto v_k]}^{\,k} \end{array}}{\Sigma \vdash_{\mathsf{op}} \langle K \mid \$\,\overline{\tau'_i}^{\,i}\ \overline{\sigma_j}^{\,j}\ \overline{v_k}^{\,k}; \mathbf{case\ as}\ n\ \mathbf{of}\ \overline{alt_a}^{\,a}]\rangle \longrightarrow c\ subst}\ \textsc{c\_MatchData}$$

$$\frac{alt_j = \mathsf{lit}\ \to c}{\Sigma \vdash_{\mathsf{op}} \langle \mathsf{lit} \mid \mathbf{case\ as}\ n\ \mathbf{of}\ \overline{alt_i}^{\,i}\rangle \longrightarrow c\,[n \mapsto \mathsf{lit}]}\ \textsc{c\_MatchLit}$$

$$\frac{\begin{array}{l} alt_j = \_ \to c \\ \text{no other case matches} \end{array}}{\Sigma \vdash_{\mathsf{op}} \langle v \mid \mathbf{case\ as}\ n\ \mathbf{of}\ \overline{alt_i}^{\,i}\rangle \longrightarrow c\,[n \mapsto v]}\ \textsc{c\_MatchDefault}$$

$$\frac{\begin{array}{l} T\,\overline{\tau_a}^{\,a} \sim^{\kappa}_{\#} T\,\overline{\tau'_a}^{\,a} = \mathsf{coercionKind}\,\gamma \\ \dfrac{\forall\overline{\alpha_a{}^{\kappa_a}}^{\,a}.\forall\overline{\beta_b{}^{\kappa'_b}}^{\,b}.\overline{\tau_1}^{\,c} \to T\,\overline{\alpha_a{}^{\kappa_a}}^{\,a} = \mathsf{dataConRepType}\,K}{v'_c = \mathbf{compute}\ n_c.\langle e_c \mid \triangleright(\tau_1\,\overline{[\alpha_a{}^{\kappa_a} \mapsto \mathsf{nth}_a\,\gamma]}^{\,a}\ \overline{[\beta_b{}^{\kappa'_b} \mapsto \langle\sigma_b\rangle_{\mathsf{N}}]}^{\,b}); \mathbf{ret}\ n_c\rangle}^{\,c} \end{array}}{\Sigma \vdash_{\mathsf{op}} \langle K \mid \$\,\overline{\tau_a}^{\,a}\ \overline{\sigma_b}^{\,b}\ \overline{v_c}^{\,c}; \triangleright\gamma; \mathbf{case\ as}\ n\ \mathbf{of}\ \overline{alt_i}^{\,i}]\rangle \longrightarrow \langle K \mid \$\,\overline{\tau'_a}^{\,a}\ \overline{\sigma_b}^{\,b}\ \overline{v'_c}^{\,c}; \mathbf{case\ as}\ n\ \mathbf{of}\ \overline{alt_i}^{\,i}]\rangle}\ \textsc{c\_CasePush}$$