TRANSLATION SEMANTICS

Coeffects let us easily construct context-aware domain-specific languages. The general coeffect language framework provides a parameterized type system (flat or structural) and a translation that turns a program written in a context-aware DSL into a program in a simple (non-context-aware) functional programming langauge with additional comonadically-inspired coeffect-specific primitives.

In this section, we discuss the safety that is guaranteed by the coeffect language framework. The general coeffect language framework itself does not automatically guarantee safety of a concrete context-aware DSL. This depends on the additional coeffect-specific primitives.

However, the framework makes it easier to prove such safety – we only need to define the meaning of the comonadically-inspired coeffect-specific data type and its associated operations and then show that those do not "go wrong". In doing so, we can use the coeffect annotations provided by the type system. In this section, we:

- Define the common subset of the target functional programming language. This includes the syntax of the language, reduction rules and typing rules, but it does *not* include coeffect-specific definitions. Well-typed programs written using the common subset of the target language do not get stuck (via progress and preservation), but they may reduce to `error`, e.g. when accessing the head of an empty list.
- We then extend the language with coeffect-specific comonadically-inspired data types and primtiives for dataflow and for implicit parameters. We show that the extension preserves the progress and preservation properties.
- Next, we consider only programs in the target language that were produced by a translation from the coeffect domain-specific language and we show that such programs not only do not get stuck, but they also do not reduce to the `error` value. In other words, "well-typed coeffect programs do not get hungry" requiring more context than guaranteed by the coeffect type system.
- We show how the approach extends to structural coeffect systems and we argue that our proof can be generalized – rather than reconsidering progress and preservation of the whole language, we can rely just on the correctness of the coeffect-specific comonadically-inspired primitives and abstraction mechanism provided by languages such as ML and Haskell.

1 TARGET LANGAUGE

The target langauge for the translation is a simply typed lambda calculus with strings, numbers, tuples and lists. For each concrete coeffect domain-specific language, we then add additional primitives – a comonadically-inspired data type and its associated primitive operations. In this section, we define the common parts of the language.

LANGUAGE SYNTAX

$$
\begin{array}{rcl}
v & = & \mathsf{error} \mid s \mid n \mid \lambda x.e \mid (v_1, \ldots, v_n) \mid v_1 :: v_2 \mid [] \\
\\
e & = & x \mid \mathsf{error} \mid s \mid n \mid \pi_i\ e \mid (e_1, \ldots, e_n) \\
& \mid & e_1\ e_2 \mid \lambda x.e \mid \mathsf{head}\ e \mid \mathsf{tail}\ e \mid e_1 :: e_2 \mid [] \\
& \mid & \texttt{if}\ e_1 = e_2\ \texttt{then}\ e_3\ \texttt{else}\ e_4 \\
\\
\tau & = & \mathsf{str} \mid \mathsf{num} \mid [\tau] \mid \tau_1 \times \ldots \times \tau_n \mid \tau_1 \to \tau_2 \\
\\
C & = & (v_1, \ldots, v_{i-1}, \_, e_{i+1}, \ldots e_n) \\
& \mid & v\ \_ \mid \_\ e \mid \pi_i\ \_ \mid \mathsf{head}\ \_ \mid \mathsf{tail}\ \_ \mid \_ :: e \mid v :: \_ \\
& \mid & \texttt{if}\ \_ = e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{if}\ v = \_\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{array}
$$

REDUCTION RULES

$$
\begin{array}{lll}
(fn) & (\lambda x.e)\ v \to e[x \leftarrow v] & \\
(prj) & \pi_i(v_1, \ldots, v_n) \to v_i & \\
(hd) & \mathsf{head}\ (v_1 :: v_2) \to v_1 & \\
(tl) & \mathsf{tail}\ (v_1 :: v_2) \to v_2 & \\
(hde) & \mathsf{head}\ [] \to \mathsf{error} & \\
(tle) & \mathsf{tail}\ [] \to \mathsf{error} & \\
(err) & C[\mathsf{error}] \to \mathsf{error} & \\
(ifnt) & \texttt{if}\ n_1 = n_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to e_1 & (\text{when } n_1 = n_2) \\
(ifnt) & \texttt{if}\ n_1 = n_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to e_2 & (\text{when } n_1 \neq n_2) \\
(ifst) & \texttt{if}\ s_1 = s_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to e_1 & (\text{when } s_1 = s_2) \\
(ifst) & \texttt{if}\ s_1 = s_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \to e_2 & (\text{when } s_1 \neq s_2) \\
(ctx) & C[e] \to C[e'] & (\text{when } e \to e')
\end{array}
$$

Figure 1: Common syntax and reduction rules of the target langauge

The syntax of the target programming langauge is shown in Figure 1. The values include primitive strings s and numbers n, tuples and list values. The value error represents the result of a failed computation.

The expressions include variables x, values lambda abstraction and application, conditionals (limited to equality check) and operations on tuples and lists. We do not need pattern matching on lists and recursion (although a realistic programming language would include both). In what follows, we also use the following syntactic sugar for lists and let binding:

$$
\begin{array}{rcl}
\langle e_1, \ldots, e_n \rangle & = & e_1 :: \ldots :: e_n :: [] \\
\texttt{let}\ x = e_1\ \texttt{in}\ e_2 & = & (\lambda x.e_2)e_1
\end{array}
$$

Finally, $C[e]$ defines the context in which sub-expressions are evaluated. Together with the evaluation rules shown in Figure 1, this captures the standard call-by-name semantics of the common parts of the target language.

TYPING RULES

$$(var)\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad (err)\ \frac{}{\Gamma \vdash \mathsf{error} : \tau}$$

$$(str)\ \frac{}{\Gamma \vdash s : \mathsf{str}} \qquad (num)\ \frac{}{\Gamma \vdash n : \mathsf{num}}$$

$$(nil)\ \frac{}{\Gamma \vdash [] : [\tau]} \qquad (head)\ \frac{\Gamma \vdash e : [\tau]}{\Gamma \vdash \mathsf{head}\ e : \tau}$$

$$(tail)\ \frac{\Gamma \vdash e : [\tau]}{\Gamma \vdash \mathsf{tail}\ e : [\tau]} \qquad (abs)\ \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$(app)\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(cons)\ \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash e_1 :: e_2 : [\tau]}$$

$$(proj)\ \frac{\Gamma \vdash e : \tau_1 \times \ldots \tau_i \times \ldots \times \tau_n}{\Gamma \vdash \pi_i\ e : \tau_i}$$

$$(tup)\ \frac{\forall i \in \{1 \ldots n\}.\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash (e_1, \ldots, e_n) : \tau_1 \times \ldots \times \tau_n}$$

$$(ifn)\ \frac{\Gamma \vdash e_1 : \mathsf{num} \quad \Gamma \vdash e_2 : \mathsf{num} \quad \Gamma \vdash e_3 : \tau \quad \Gamma \vdash e_4 : \tau}{\Gamma \vdash \mathtt{if}\ e_1 = e_2\ \mathtt{then}\ e_3\ \mathtt{else}\ e_4 : \tau}$$

$$(ifs)\ \frac{\Gamma \vdash e_1 : \mathsf{str} \quad \Gamma \vdash e_2 : \mathsf{str} \quad \Gamma \vdash e_3 : \tau \quad \Gamma \vdash e_4 : \tau}{\Gamma \vdash \mathtt{if}\ e_1 = e_2\ \mathtt{then}\ e_3\ \mathtt{else}\ e_4 : \tau}$$

Figure 2: Typing rules for the common syntax of the target language

The (*hde*) and (*tle*) reduction rules describe the situation when we attempt to access the head or the tail of an empty list. In that case, the expression reduces to error value. An error value appearing in any context then propagates to the top-level as defined by (*err*).

The typing rules for the common expressions of the target langauge are shown in Figure 2. The rules are standard. As usual, the error term is allowed to have any type (*err*).

2 PROPERTIES

The subset of the langauge described so far models a simple ML-like functional programming langauge (or, Haskell-like language, if we choose call-by-name evaluation). We discuss the properties of the language after introducing coeffect-specific primitives. As will be seen, "well-typed programs do not get stuck" in our language, although they may produce the error value. For example, head 42 is a stuck expression ruled out by the common rules of the type system, however, head [] is a well typed expression that produces error.

3 DATAFLOW

Domain specific coeffect annotations are non-negative integers, type is a tuple and new kinds of values are lists:

$$v_c \quad = \quad \langle v_1, \ldots, v_n \rangle$$

With the following typing:

$$(\textit{val})\ \frac{\forall v_i.\ \vdash v_i : \tau}{\vdash \langle v_1, \ldots, v_n \rangle : C^n \tau}$$

And domain specific operations are defined as:

$$\mathsf{counit}_0 \langle a_0 \rangle \to a_0$$

$$\mathsf{cobind}_{m,n}\ f \langle a_0, \ldots a_{m+n} \rangle \to \langle f\langle a_0, \ldots, a_m \rangle, \ldots, f\langle a_n, \ldots, a_{m+n} \rangle \rangle$$

$$\mathsf{merge}_{m,n}(\langle a_0, \ldots, a_m \rangle, \langle b_0, \ldots, b_n \rangle) \to \langle (a_0, b_0), \ldots, (a_{\mathit{min}(m,n)}, b_{\mathit{min}(m,n)}) \rangle$$

$$\mathsf{split}_{m,n} \langle (a_0, b_0), \ldots, (a_{\mathit{max}(m,n)}, b_{\mathit{max}(m,n)}) \rangle \to (\langle a_0, \ldots, a_m \rangle, \langle b_0, \ldots, b_n \rangle)$$

$$\mathsf{lift}_{n',n} \langle a_0, \ldots, a_{n'} \rangle \to \langle a_0, \ldots, a_n \rangle \qquad (\text{when } n \leqslant n')$$

We could define those in terms of head/tail functions and then prove that they work (and translated program does not contain head/tail), but perhaps that's unnecessary overkill. Or perhaps it would be useful in order to show how we avoid errors... But doing it directly looks just simpler (we also do not need a "list" type).

Theorem 1 (Type preservation). *If* $\Gamma \vdash e : \tau$ *and* $e \to e'$ *then* $\Gamma \vdash e' : \tau$

Proof. Rule induction over $\to$
Simple for normal parts of the language
We have to check this for dataflow reductions □

Theorem 2 (Progress). *If* $\Gamma \vdash e : \tau$ *then either* e *is a value or there exits* e' *such that* $e \to e'$

Proof. By rule induction over $\vdash$.
Interesting case is $e_1 e_2$ where $e_1 = v$ is one of the primitives. Then we have to go through them and make sure they can progress. □