Notes on DEC2

Paolo Torrini
ptorrx@gmail.com

1 Introduction

DEC2 is an imperative functional language with bounded recursion and generic effects deeply embedded in Coq. It has been designed as a small intermediate language to support translation from monadic Gallina code to C. It captures the expressiveness of the monadic code used in the implementation of the Pip protokernel (`https://github.com/2xs/pipcore`). Being defined in terms of abstract datatypes, it allows for syntactic manipulation to be implemented and verified directly in Coq. In particular, this is the case for its translations to Coq representations of C, such as CompCert C (`http://compcert.inria.fr`).

The DEC2 specification includes a definition of the syntax, definitions of the static semantics, and a definition of the dynamic semantics, which is stateful and based on a small-step transition relation, in the style of SOS. Syntactic categories and semantic relations are mainly defined in terms of inductive datatypes.

Essentially, DEC2 provides the expressiveness of a fragment of C, when we restrict to C expressions and call by value. It allows for sequencing (corresponding to C comma expressions), local variables, conditional branching, internal and external function calls. As in C, functions cannot be passed as arguments by value. Passing by reference is not modelled. Internal functions are essentially primitive recursive ones, defined by a bounded *iterate*-style construct. Any Gallina function can be imported as an external one, and semantically treated as a generic effect with respect to the DEC2 state.

Relying on Coq modules, the definition of DEC2 is parametric in the type of the mutable state and in the type of identifiers. These parameters are specified in the module type *ModTyp.v*.

DEC2 types include primitive value types, which are specified by a dataype over Gallina types and their intended translation to CompCert C types, and function types based on primitive types.

DEC2 normal forms include data values and functions. DEC2 distinguishes between data variables and function variables, while relying on a single name space for identifiers. Correspondingly, DEC2 semantics distinguishes between data environments and function environments, and similarly for typing environments (or contexts). Environments are implemented as association lists, functionally interpreted as partial maps.

DEC2 values are defined by lifting Gallina terms and further hiding their Gallina types, using dependent product. Function definitions in the function

environment are intended to be mutually recursive. Recursion is on a unique fuel argument, decreasing by one (or more) at each call. Each definition consists of the formal parameters with their types, a default value for the 0-fuel case, and the function body. In internal function calls, concrete parameters are given as a lifted list of expressions.

The lifting constructor for values in the definition of expressions is Val. bindN is the sequencing constructor. The constructors for local variables are bindS, which allows for let-style binding of identifiers to expressions, and bindMS, which allows for multiple binding of identifiers to values. ifThenElse is the conditional branching constructor. The constructors for internal function calls are apply (which allows to decrease fuel arbitrarily) and call (which decreases it by one). The constructor for external function calls is modify. The constructor PS lifts expression lists to parameters.

The dynamic semantics is specified in the style of structural operational semantics (SOS) using a small-step transition relation on configurations that for each category include a program term in that category, a state of the mutable store and a fuel value. In general, the transition relation depends on value environments, function environments.

DEC2 types defined in TypSpecI1.v

DEC2 syntax defined in LangSpecI1.v

Static semantics defined in StaticSemI1.v

Dynamic semantics defined in DynamicSemI1.v

Dynamic semantics defined in DynamicSemI1.v

Interpreters defined in InterpretI1.v. This module contains the definitions of the SOS interpreter (for expressions and parameters), of the reflective interpreter, and the proof of their behavioural equality (see comments).

Translation to CompCert C defined in CTransI1.v

2 Reflective translation (outline)

$$
\begin{array}{ll}
\Theta_t\ (t:\ \mathsf{VTyp}) & :=\ T\ :\ \mathsf{Type} \\
\Theta_h\ (h:\ \mathsf{FTyp}) & :=\ (\mathsf{map}\ \Theta_t\ (\mathsf{args\_typ}\ h)) \\
 & \qquad \to (\mathsf{result\_typ}\ h)\ :\ \mathsf{Type} \\
\Theta_\tau\ (\tau:\ \mathsf{list}\ (\mathsf{Id} * \mathsf{VTyp}|\mathsf{FTyp})) & :=\ \mathsf{foldl}\ (\lambda kp.(\Theta_{t|h}\ (\pi_{val}\ p)\ *\ k))\ \mathsf{tt}\ \tau \\
 & \qquad :\ \mathsf{Tuple\ Type} \\
\Theta_v\ (v:\ \mathsf{Value}) & :=\ \mathsf{shallow\_val\ v}\ :\ \mathsf{shallow\_typ}\ v \\
\Theta_x\ (x:\ \mathsf{Id})\ \tau\ (E:\ \Theta_\tau\ \tau) & :=\ \mathsf{let}\ (n\ =\ \mathsf{position}\ x\ \tau)\ \mathsf{in} \\
 & \qquad \pi_n\ E\ :\ (\Theta_\tau\ \tau)_n
\end{array}
$$

$$
\begin{array}{l}
\mathsf{ret}\ :\ T_1 \to M\ T_2 \\
\mathsf{bind}\ :\ M\ T_1 \to (T_1 \to M\ T_2) \to M\ T_2 \\
\mathsf{lft\_bind}\ :\ (T_0 \to M\ T_1) \to (T_1 * T_0 \to M\ T_2) \to T_0 \to M\ T_2 \\
\mathsf{lft\_if\_then\_else}\ :\ (T_0 \to M\ \mathsf{bool}) \to (T_0 \to M\ T_1) \to \\
\qquad\qquad (T_0 \to M\ T_1) \to T_0 \to M\ T_1 \\
\mathsf{lft\_apply}\ :\ (T_0 \to T_1 \to M\ T_2) \to (T_0 \to M\ T_1) \to T \to M\ T_2
\end{array}
$$

$$
\begin{array}{ll}
\Theta_e\ (\mathsf{Val}\ v)\ \tau & =\ \lambda\_ : \Theta_\tau\ \tau.\ \mathsf{ret}\ (\Theta_v\ v) \\
 & :\ \Theta_\tau\ \tau \to M\ (\mathsf{ext\_styp}\ v) \\
\Theta_e\ (\mathsf{Var}\ x)\ \tau & =\ \lambda E : \Theta_\tau\ \tau_\rho.\ \mathsf{ret}\ (\Theta_x\ x\ \tau\ E) \\
 & :\ \Theta_\tau\ \tau \to M\ (\Theta_t\ t_x) \\
\Theta_e\ (\mathsf{BindN}\ e_1\ e_2)\ \tau & =\ \mathsf{lft\_bind}\ (\Theta_e\ e_1\ \tau)\ (\Theta_e\ e_2\ ((\_ : t_1) :: \tau)) \\
 & :\ \Theta_\tau\ \tau \to M\ (\Theta_t\ t_2) \\
\Theta_e\ (\mathsf{BindS}\ x\ t_1\ e_1\ e_2)\ \tau & =\ \mathsf{lft\_bind}\ (\Theta_e\ e_1\ \tau)\ (\Theta_e\ e_2\ ((x : t_1) :: \tau)) \\
 & :\ \Theta_\tau\ \tau \to M\ (\Theta_t\ t_2) \\
\Theta_e\ (\mathsf{IfThenElse}\ e_1\ e_2\ e_3)\ \tau & =\ \mathsf{lft\_if\_then\_else} \\
 & (\Theta_e\ e_1\ \tau)\ (\Theta_e\ e_2\ \tau)\ (\Theta_e\ e_3\ \tau) \\
 & :\ \Theta_\tau\ \tau \to M\ (\Theta_t\ t_2) \\
\Theta_e\ (\mathsf{Call}\ x\ es\ \_)\ \tau & =\ \mathsf{lft\_apply}\ (\Theta_{es}\ es\ \tau)\ (\Theta_x\ x\ \tau) \\
 & :\ \Theta_\tau\ \tau \to M\ (\Theta_t\ (\mathsf{res\_ty}\ h_x)) \\
\Theta_e\ (\mathsf{Modify}\ t_1\ t_2\ X\ e)\ \tau & =\ X.\mathsf{mod}\ (\Theta_e\ e\ \tau) \\
 & :\ \Theta_\tau\ \tau \to M\ (\Theta_t\ t_2)
\end{array}
$$

$$
\begin{array}{ll}
\Theta_f\ \mathit{fuel}\ (\mathsf{FC}\ \tau\ v\ e)\ \tau_\phi & :=\ \Theta_e\ (\mathsf{match}\ \mathit{fuel}\ \mathsf{with}\ 0 \to \mathsf{Val}\ v \mid \mathsf{S}\ n \to e)\ \tau \\
 & :\ \Theta_h\ (\mathsf{type\_of}\ f) \\
\Theta_\rho\ (\rho:\ \mathsf{list}\ (\mathsf{Id} * \mathsf{Value})) & :=\ \mathsf{foldl}\ (\lambda kp.(\Theta_v\ (\pi_{val}\ p),\ k))\ \mathsf{tt}\ \rho\ :\ \Theta_\tau\ \tau_\rho \\
\Theta_\phi\ (\phi:\ \mathsf{list}\ (\mathsf{Id} * \mathsf{Fun})) & :=\ \mathsf{foldl}\ (\lambda kp.(\Theta_f\ (\pi_{val}\ p)\ \tau_\phi,\ k))\ \mathsf{tt}\ \rho \\
 & :\ \Theta_\tau\ \tau_\phi \\
\Theta_{es}\ es\ \tau & :=\ \mathsf{foldl}\ (\lambda kx.(\Theta_e\ x\ \tau,\ k))\ \mathsf{tt}\ es \\
 & :\ \mathsf{foldl}\ (\lambda kx.\ \mathsf{ext\_styp}\ x,\ k)\ \mathsf{unit}\ es
\end{array}
$$

$$
\Theta\ (\phi; \rho \vdash e\ :\ t) \quad :=\ (\Theta_e\ e\ \tau_{(\phi,\rho)})\ (\Theta_{\phi|\rho}(\phi,\rho))\ :\ \Theta_t\ t
$$