DEC 1.0 specification

Paolo Torrini, David Nowak

September 28, 2017

DEC 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 DEC 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, DEC 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 DEC state.

Relying on Coq modules, the definition of DEC 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*. With respect to Pip, the concrete module *BaseMod.v* provides an instantiation of these parameters consistent with the current output of the Digger tool (`https://github.com/2xs/digger`).

DEC types include primitive value types, which are specified by an admissibility type class over Gallina types, and function types based on primitive types. DEC programs do not bear type annotation, save for formal parameters in functions.

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

DEC values are defined by lifting Gallina terms and further hiding their Gallina types, using dependent product. We call quasi-values the head normal forms based on data, which simply include values and data variables. Analogously, we call quasi-functions the head normal forms based on functions, including functions and function variables.

Internal function definition includes local function variables, formal parameters with their types, function bodies for zero and non-zero fuel, an identifier as recursive call name (which can also be used as function name), and a natural value that represents fuel (i.e. the bound). DEC functions are structurally closed terms, in the sense that each occurrence of an identifier in their bodies is bound either as a formal parameter or as a local variable. In internal function calls, concrete parameters are given as a lifted list of expressions.

The lifting constructors in the definition of expressions are Val, which lifts values, and Return which lifts quasi-values and is additionally tagged for external, non-semantic reasons. 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 normal forms. ifThenElse is the conditional branching constructor. The constructors for internal and external function calls are apply and modify, respectively. Moreover, Var and FVar are used to lift identifiers to the corresponding head-normal forms, and similarly QV and QF to lift normal forms. FunZ and FunS are the recursive function constructors. The constructor PS lifts expression lists to parameters.

In the intended use of DEC, function definitions are known statically from the function environment. That is, the bindMS constructor and the QF lifting of functions to quasi-functions are only meant to be used in the processing of programs. Moreover, the first argument of ifThenElse is meant to be a lifted quasi-value.

Notice that functions, quasi-functions, expressions, and parameters are defined by mutual induction. The typing relation for each syntactic category is also defined inductively, modulo the well-typedness of the environments each object depends on. Usually, the definition of a typing relation is parametric in the associated typing contexts. Granted the intended use of DEC, we find it convenient to define a more expressive, algorithmic typing relation, which depends also on the function environment. In this way, the definition of the typing relation for functions can be defined inductively on the fuel. This makes it possible to obtain a stronger inductive principle for the mutually inductive datatype. Under the restrictions we mentioned, the algorithmic typing relation we have implemented is equivalent to the more usual, declarative one.

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 and a state of the mutable store. In general, the transition relation depends on value environments and function environments. The order of evaluation is fixed to left-to-right.

1 Syntax

1.1 State type

$$\mathsf{W} \ : \ \mathsf{Type} \tag{1}$$

1.2 Identifier type

$$\mathsf{Id} \ : \ \mathsf{Type} \tag{2}$$

1.3 External function record

$$\begin{array}{l} xf \ : \ \ (\ \mathsf{XFun}\ (InpT \ : \ \mathsf{Type})\ (OutT \ : \ \mathsf{Type}) \ : \ \mathsf{Type} \ := \\ \qquad \{\mathsf{b\_mod} \ : \ \mathsf{W} \to InpT \to (\mathsf{W} \ * \ OutT)\}\ ) \end{array} \tag{3}$$

1.4 Data value types

$$t \ : \ \mathsf{VTyp} \ := \ \mathsf{vtyp}\ \mathsf{Type} \tag{4}$$

1.5 Parameter types

$$ts \ : \ \mathsf{list}\ \mathsf{VTyp} \tag{5}$$

$$pt \ : \ (\ \mathsf{PTyp} \ := \ \mathsf{PT}\ (\mathsf{list}\ \mathsf{VTyp})\ ) \tag{6}$$

1.6 Identifiers

$$x \ : \ \mathsf{Id} \tag{7}$$

1.7 Data typing contexts

$$\Gamma \ : \ (\ \mathsf{ValTC} \ := \ \mathsf{list}\ (\mathsf{Id} \ * \ \mathsf{VTyp})\ ) \ \ <: \ \ \mathsf{Id} \to \mathsf{option}\ \mathsf{VTyp} \tag{8}$$

1.8 Function types

$$(\tau \ : \ \mathsf{FTyp}) \ := \ \mathsf{ValTC} \Rightarrow \mathsf{VTyp} \tag{9}$$

1.9 Function typing contexts

$$\Phi \ : \ (\ \mathsf{FunTC} \ := \ \mathsf{list}\ (\mathsf{Id} \ * \ \mathsf{FTyp})\ ) \ \ <: \ \ \mathsf{Id} \to \mathsf{option}\ \mathsf{FTyp} \tag{10}$$

1.10 Data values

$$(v \ : \ \mathsf{Value}) \ := \ \mathsf{cst} \ (T : \mathsf{Type}) \ T \tag{11}$$

1.11 Quasi-values

$$(qv \ : \ \mathsf{QValue}) \ := \ \mathsf{Var\ Id} \ | \ \mathsf{QV\ Value} \tag{12}$$

1.12 Data value environments

$$\rho \ : \ (\ \mathsf{ValEnv} \quad := \quad \mathsf{list}\ (\mathsf{Id} \ * \ \mathsf{Value})\ ) \quad <: \quad \mathsf{Id} \to \mathsf{option\ Value} \tag{13}$$

1.13 Functions

$$\begin{array}{rcl} (f \ : \ \mathsf{Fun}) & := & \mathsf{FC}\ (\mathit{local\_function\_definitions} \ : \ \mathsf{FunEnv}) \\ & & \quad (\mathit{formal\_parameters} \ : \ \mathsf{ValTC}) \\ & & \quad (\mathit{function\_body\_zero\_case} \ : \ \mathsf{Exp}) \\ & & \quad (\mathit{function\_body\_succ\_case} \ : \ \mathsf{Exp}) \\ & & \quad (\mathit{recursive\_call\_name} \ : \ \mathsf{Id}) \\ & & \quad (\mathit{recursion\_bound} \ : \ \mathsf{nat} \end{array} \tag{14}$$

1.14 Quasi-functions

$$(qf \ : \ \mathsf{QFun}) \ := \ \mathsf{FVar\ Id} \ | \ \mathsf{QF\ Fun} \tag{15}$$

1.15 Function environments

$$\phi \ : \ (\ \mathsf{FunEnv} \ := \ (\mathsf{Id} \ * \ \mathsf{Fun})\ ) \quad <: \quad \mathsf{Id} \to \mathsf{option\ Fun} \tag{16}$$

1.16 Return tags

$$(tg \ : \ \mathsf{Tag}) \ := \ \mathsf{LL} \ | \ \mathsf{RR} \tag{17}$$

1.17 Expressions

$$\begin{array}{rcl} (e \ : \ \mathsf{Exp}) & := & \mathsf{val\ Value} \\ & | & \mathsf{return\ Tag\ QValue} \\ & | & \mathsf{bindN\ Exp\ Exp} \\ & | & \mathsf{bindS\ Id\ Exp\ Exp} \\ & | & \mathsf{bindMS\ FunEnv\ ValEnv\ Exp} \\ & | & \mathsf{ifThenElse\ Exp\ Exp\ Exp} \\ & | & \mathsf{apply\ QFun\ Prms} \\ & | & \mathsf{modify\ (XFun\ Type\ Type)\ QValue} \end{array} \tag{18}$$

1.18 Parameters

$$vs \ : \ \mathsf{list\ Value} \tag{19}$$

$$es \ : \ \mathsf{list\ Exp} \tag{20}$$

$$(ps \ : \ \mathsf{Prms}) \ := \ \mathsf{PS\ (list\ Exp)} \tag{21}$$

1.19 Programs

$$(\pi \ : \ \mathsf{Prog}) \ := \ \mathsf{prog\ Exp} \mid \mathsf{define\ Id\ Fun\ Prog} \tag{22}$$

1.20 Configurations

$$(C \ : \ \mathsf{SynCat}) \ := \ \mathsf{QValue} \mid \mathsf{QFun} \mid \mathsf{Prms} \mid \mathsf{Exp} \mid \mathsf{Prog} \tag{23}$$

$$\mathsf{Config}\ C \ := \ \langle \quad \mathsf{state : W} \quad \triangleright \quad \mathsf{program\_term} : C \quad \rangle \tag{24}$$

1.21 Static sequents

$$\begin{array}{rcl} \mathsf{Typing\_sequent} & := & \mathsf{ValTC} \ \vdash \ \mathsf{QValue} \ :: \ \mathsf{VTyp} \\ & \mid & \vdash \ \mathsf{Value} \ :: \ \mathsf{VTyp} \\ & \mid & \vdash \ \mathsf{Fun} \ :: \ \mathsf{FTyp} \\ & \mid & \vdash \ \mathsf{option\ Value} \ :: \ \mathsf{option\ VTyp} \\ & \mid & \vdash \ \mathsf{option\ Fun} \ :: \ \mathsf{option\ FTyp} \\ & \mid & \vdash \ \mathsf{ValEnv} \ :: \ \mathsf{ValTC} \\ & \mid & \vdash \ \mathsf{FunEnv} \ :: \ \mathsf{FunTC} \end{array} \tag{25}$$

$$\begin{array}{rcl} \mathsf{AlgTyping\_sequent} & := & \mathsf{FunTC; FunEnv} \ \vdash \ \mathsf{Prog} \ :: \ \mathsf{VTyp} \\ & \mid & \mathsf{FunTC; FunEnv; ValTC} \ \vdash \ \mathsf{Exp} \ :: \ \mathsf{VTyp} \\ & \mid & \mathsf{FunTC; FunEnv; ValTC} \ \vdash \ \mathsf{Prms} \ :: \ \mathsf{PTyp} \\ & \mid & \mathsf{FunTC; FunEnv} \ \vdash \ \mathsf{QFun} \ :: \ \mathsf{FTyp} \end{array} \tag{26}$$

$$\begin{array}{rcl} \mathsf{DecTyping\_sequent} & := & \mathsf{FunTC} \ \vdash \ \mathsf{Prog} \ :: \ \mathsf{VTyp} \\ & \mid & \mathsf{FunTC; ValTC} \ \vdash \ \mathsf{Exp} \ :: \ \mathsf{VTyp} \\ & \mid & \mathsf{FunTC; ValTC} \ \vdash \ \mathsf{Prms} \ :: \ \mathsf{PTyp} \\ & \mid & \mathsf{FunTC} \ \vdash \ \mathsf{QFun} \ :: \ \mathsf{FTyp} \end{array} \tag{27}$$

1.22 Dynamic sequents

$$\begin{array}{rcl} \mathsf{Step\_sequent} & := & \mathsf{FunEnv} \vdash \mathsf{Config\ Prog} \longrightarrow \mathsf{Config\ Prog} \\ & & \mathsf{FunEnv}; \mathsf{ValEnv} \vdash \mathsf{Config\ Exp} \longrightarrow \mathsf{Config\ Exp} \\ & & \mathsf{FunEnv}; \mathsf{ValEnv} \vdash \mathsf{Config\ Prms} \longrightarrow \mathsf{Config\ Prms} \\ & & \mathsf{ValEnv} \vdash \mathsf{Config\ QValue} \longrightarrow \mathsf{Config\ QValue} \\ & & \mathsf{FunEnv} \vdash \mathsf{Config\ QFun} \longrightarrow \mathsf{Config\ QFun} \end{array} \tag{28}$$

1.23 Relating value lists and lifted value lists

$$\vdash \mathsf{values} \ : \ \mathsf{list\ Exp} \to \mathsf{list\ Value} \to \mathsf{Prop} \tag{29}$$

$$\frac{\vdash es = \mathsf{map\ val}\ vs}{\vdash \mathsf{values}\ es\ vs} \tag{30}$$

1.24 Typing context conversion to type lists

$$\vdash \mathsf{tc2pt} \ : \ \mathsf{ValTC} \to \mathsf{PTyp} \tag{31}$$

1.25 Value environment creation

$$\vdash \mathsf{mkValEnv} \ : \ \mathsf{ValTC} \to \mathsf{list\ Value} \ \to \mathsf{ValEnv} \tag{32}$$

2 Static semantics

2.1 Data values

$$\frac{\vdash T \ : \ \mathsf{Type} \qquad \vdash n \ : \ T}{\vdash \mathsf{cst}\ T\ n \ :: \ \mathsf{vtyp}\ T} \tag{33}$$

2.2 Option-lifted normal forms

$$\frac{}{\vdash \mathsf{None} \ :: \ \mathsf{None}} \tag{34}$$

$$\frac{\vdash v \ :: \ t}{\vdash \mathsf{Some}\ v \ :: \ \mathsf{Some}\ t} \tag{35}$$

$$\frac{\vdash f \ :: \ \tau}{\vdash \mathsf{Some}\ f \ :: \ \mathsf{Some}\ \tau} \tag{36}$$

2.3 Environments

$$\frac{\bigwedge(x : \mathsf{Id})\ (\ \vdash \rho\ x\ ::\ \Gamma\ x\ )}{\vdash \rho\ ::\ \Gamma} \tag{37}$$

$$\frac{\bigwedge(x : \mathsf{Id})\ (\ \vdash \phi\ x\ ::\ \Phi\ x\ )}{\vdash \phi\ ::\ \Phi} \tag{38}$$

2.4 Quasi-values

$$\frac{\vdash \Gamma\ x\ =\ \mathsf{Some}\ t}{\Gamma \vdash \mathsf{Var}\ x\ ::\ t} \tag{39}$$

$$\frac{\vdash v\ ::\ t}{\Gamma \vdash \mathsf{QV}\ v\ ::\ t} \tag{40}$$

3 Algorithmic static semantics

3.1 Expression lists

$$\frac{}{\Phi; \phi; \Gamma \vdash \mathsf{nil}\ ::\ \mathsf{nil}} \tag{41}$$

$$\frac{\Phi; \phi; \Gamma \vdash e\ ::\ t \qquad \Phi; \phi; \Gamma \vdash es\ ::\ ts}{\Phi; \phi; \Gamma \vdash \mathsf{cons}\ e\ es\ ::\ \mathsf{cons}\ t\ ts} \tag{42}$$

3.2 Parameters

$$\frac{\Phi; \phi; \Gamma \vdash es\ ::\ ts}{\Phi; \phi; \Gamma \vdash \mathsf{PS}\ es\ ::\ \mathsf{PT}\ ts} \tag{43}$$

3.3 Functions

$$\frac{\begin{array}{c}\vdash \phi\ ::\ \Phi \\ \Phi; \phi; \Gamma \vdash e_0\ ::\ t\end{array}}{\vdash \mathsf{FC}\ \phi\ \Gamma\ e_0\ e_1\ x\ 0\ ::\ \Gamma \Rightarrow t} \tag{44}$$

$$\frac{\begin{array}{l}\vdash \phi\ ::\ \Phi \qquad \vdash n\ :\ \mathsf{nat} \\ \vdash \mathsf{FC}\ \phi\ \Gamma\ e_0\ e_1\ x\ n\ ::\ \Gamma \Rightarrow t \\ (x,\ \Gamma \Rightarrow t)/\Phi; (x,\ \mathsf{FC}\ \phi\ \Gamma\ e_0\ e_1\ x\ n)/\phi; \Gamma \vdash e_1\ ::\ t\end{array}}{\vdash \mathsf{FC}\ \phi\ \Gamma\ e_0\ e_1\ x\ (\mathsf{S}\ n)\ ::\ \Gamma \Rightarrow t} \tag{45}$$

3.4 Quasi-functions

$$\frac{\vdash \phi \ :: \ \Phi \qquad \vdash \Phi \ x \ = \ \tau}{\Phi; \phi \vdash \mathsf{FVar} \ x \ :: \ \tau} \tag{46}$$

$$\frac{\vdash f \ :: \ \tau}{\Phi; \phi \vdash \mathsf{QF} \ f \ :: \ \tau} \tag{47}$$

3.5 Expressions

$$\frac{\vdash v \ :: \ t}{\Phi; \phi; \Gamma \vdash \mathsf{val} \ v :: \ t} \tag{48}$$

$$\frac{\Gamma \vdash qv \ :: \ t \qquad \vdash tg \ : \ \mathsf{Tag}}{\Phi; \phi; \Gamma \vdash \mathsf{return} \ tg \ qv \ :: \ t} \tag{49}$$

$$\frac{\Phi; \phi; \Gamma \vdash e_1 \ :: \ t_1 \qquad \Phi; \phi; \Gamma \vdash e_2 \ :: \ t_2}{\Phi; \phi; \Gamma \vdash \mathsf{bindN} \ e_1 \ e_2 \ :: \ t_2} \tag{50}$$

$$\frac{\Phi; \phi; \Gamma \vdash e_1 \ :: \ t_1 \qquad \Phi; \phi; (x, \ t_1)/\Gamma \vdash e_2 \ :: \ t_2}{\Phi; \phi; \Gamma \vdash \mathsf{bindS} \ x \ e_1 \ e_2 \ :: \ t_2} \tag{51}$$

$$\frac{\vdash \phi' \ :: \ \Phi' \qquad \vdash \rho \ :: \ \Gamma' \qquad \Phi'/\Phi; \phi'/\phi; \Gamma'/\Gamma \vdash e \ :: \ t}{\Phi; \phi; \Gamma \vdash \mathsf{bindMS} \ \phi' \ \rho \ e \ :: \ t} \tag{52}$$

$$\frac{\Phi; \phi; \Gamma \vdash e_1 \ :: \ \mathsf{vtyp \ bool} \qquad \Phi; \phi; \Gamma \vdash e_2 \ :: \ t \qquad \Phi; \phi; \Gamma \vdash e_3 \ :: \ t}{\Phi; \phi; \Gamma \vdash \mathsf{ifThenElse} \ e_1 \ e_2 \ e_3 \ :: \ t} \tag{53}$$

$$\frac{\Phi; \phi \vdash qf \ :: \ \Gamma' \Rightarrow t \qquad \Phi; \phi; \Gamma \vdash ps \ :: \ \mathsf{tc2pt} \ \Gamma'}{\Phi; \phi; \Gamma \vdash \ \mathsf{apply} \ qf \ ps \ :: \ t} \tag{54}$$

$$\frac{\vdash xf \ : \ \mathsf{XFun} \ T_1 \ T_2 \qquad \Gamma \vdash qv \ :: \ \mathsf{vtyp} \ T_1}{\Phi; \phi; \Gamma \vdash \mathsf{modify} \ xf \ qv \ :: \ \mathsf{vtyp} \ T_2} \tag{55}$$

3.6 Programs

$$\frac{\vdash \phi \ :: \ \Phi \qquad \Phi; \phi \vdash e \ :: \ t}{\Phi; \phi \vdash \mathsf{prog} \ e \ :: \ t} \tag{56}$$

$$\frac{\vdash \phi \ :: \ \Phi \qquad \vdash f \ :: \ \tau \qquad (x, \ \tau)/\Phi; (x, \ f)/\phi \vdash \pi \ :: \ t}{\Phi; \phi \vdash \mathsf{define} \ x \ f \ \pi \ :: \ t} \tag{57}$$

4 Dynamic semantics

4.1 Var

$$\frac{\vdash \rho\ x\ =\ \mathsf{Some}\ v}{\rho \vdash \langle\ w \triangleright \mathsf{Var}\ x\ \rangle \longrightarrow \langle\ w \triangleright \mathsf{QV}\ v\ \rangle} \tag{58}$$

4.2 FVar

$$\frac{\vdash \phi\ x\ =\ \mathsf{Some}\ f}{\phi \vdash \langle\ w \triangleright \mathsf{FVar}\ x\ \rangle \longrightarrow \langle\ w \triangleright \mathsf{QF}\ f\ \rangle} \tag{59}$$

4.3 return

$$\frac{}{\phi; \rho \vdash \langle\ w \triangleright \mathsf{return}\ tg\ v\ \rangle \longrightarrow \langle\ w \triangleright \mathsf{val}\ tg\ v\ \rangle} \tag{60}$$

$$\frac{\rho \vdash \langle\ w \triangleright q\ \rangle \longrightarrow \langle\ w' \triangleright q'\ \rangle}{\phi; \rho \vdash \langle\ w \triangleright \mathsf{return}\ tg\ q\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{return}\ tg\ q'\ \rangle} \tag{61}$$

4.4 bindN

$$\frac{}{\phi; \rho \vdash \langle\ w \triangleright \mathsf{bindN}\ (\mathsf{val}\ v)\ e\ \rangle \longrightarrow \langle\ w \triangleright e\ \rangle} \tag{62}$$

$$\frac{\phi; \rho \vdash \langle\ w \triangleright e_1\ \rangle \longrightarrow \langle\ w' \triangleright e_1'\ \rangle}{\phi; \rho \vdash \langle\ w \triangleright \mathsf{bindN}\ e_1\ e_2\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{bindN}\ e_1'\ e_2\ \rangle} \tag{63}$$

4.5 bindS

$$\frac{}{\begin{array}{r}\phi; \rho \vdash \langle\ w \triangleright \mathsf{bindS}\ x\ (\mathsf{val}\ v)\ e\ \rangle \longrightarrow \\ \langle\ w \triangleright \mathsf{bindMS}\ \emptyset\ (x,\ v)\ e\ \rangle\end{array}} \tag{64}$$

$$\frac{\phi; \rho \vdash \langle\ w \triangleright e_1\ \rangle \longrightarrow \langle\ w' \triangleright e_1'\ \rangle}{\phi; \rho \vdash \langle\ w \triangleright \mathsf{bindS}\ x\ e_1\ e_2\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{bindS}\ x\ e_1'\ e_2\ \rangle} \tag{65}$$

4.6 bindMS

$$\frac{}{\phi; \rho \vdash \langle\ w \triangleright \mathsf{bindMS}\ \_\ \_\ (\mathsf{val}\ v)\ \rangle \longrightarrow \langle\ w \triangleright \mathsf{val}\ v\ \rangle} \tag{66}$$

$$\frac{\phi'/\phi; \rho'/\rho \vdash \langle\ w \triangleright e\ \rangle \longrightarrow \langle\ w \triangleright e'\ \rangle}{\phi; \rho \vdash \langle\ w \triangleright \mathsf{bindMS}\ \phi'\ \rho'\ e\ \rangle \longrightarrow \langle\ w \triangleright \mathsf{bindMS}\ \phi'\ \rho'\ e')\ \rangle} \tag{67}$$

4.7 ifThenElse

$$\frac{}{\phi;\rho \vdash \langle\ w \triangleright \mathsf{ifThenElse}\ (\mathsf{val}\ (\mathsf{cst}\ \mathsf{bool}\ \mathsf{true}))\ e_1\ e_2\ \rangle \longrightarrow \langle\ w \triangleright e_1\ \rangle} \tag{68}$$

$$\frac{}{\phi;\rho \vdash \langle\ w \triangleright \mathsf{ifThenElse}\ (\mathsf{val}\ (\mathsf{cst}\ \mathsf{bool}\ \mathsf{false}))\ e_1\ e_2\ \rangle \longrightarrow \langle\ w \triangleright e_2\ \rangle} \tag{69}$$

$$\frac{\phi;\rho \vdash \langle\ w \triangleright e_1\ \rangle \longrightarrow \langle\ w' \triangleright e_1'\ \rangle}{\phi;\rho \vdash \langle\ w \triangleright \mathsf{ifThenElse}\ e_1\ e_2\ e_3\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{ifThenElse}\ e_1'\ e_2\ e_3\ \rangle} \tag{70}$$

4.8 apply

$$\frac{\vdash \mathsf{values}\ es\ vs}{\begin{array}{r}\phi;\rho \vdash \langle\ w \triangleright \mathsf{apply}\ (\mathsf{fun}\ \phi\ \Gamma\ e_0\ e_1\ x\ 0)\ (\mathsf{PS}\ es)\ \rangle \longrightarrow \\ \langle\ w \triangleright \mathsf{bindMS}\ \phi\ (\mathsf{mkValEnv}\ \Gamma\ vs)\ e_0\ \rangle\end{array}} \tag{71}$$

$$\frac{\vdash \mathsf{values}\ es\ vs}{\begin{array}{l}\phi;\rho \vdash \langle\ w \triangleright \mathsf{apply}\ (\mathsf{fun}\ \phi\ \Gamma\ e_0\ e_1\ x\ (\mathsf{S}\ n))\ (\mathsf{PS}\ es)\ \rangle \longrightarrow \\ \quad \langle\ w \triangleright \mathsf{bindMS}\ ((x,\ \mathsf{FC}\ \phi\ \Gamma\ e_0\ e_1\ x\ n)/\phi)\ (\mathsf{mkValEnv}\ \Gamma\ vs)\ e_1\ \rangle\end{array}} \tag{72}$$

$$\frac{\phi;\rho \vdash \langle\ w \triangleright ps\ \rangle \longrightarrow \langle\ w' \triangleright ps'\ \rangle}{\phi;\rho \vdash \langle\ w \triangleright \mathsf{apply}\ f\ ps\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{apply}\ f\ ps'\ \rangle} \tag{73}$$

$$\frac{\phi \vdash \langle\ w \triangleright qf\ \rangle \longrightarrow \langle\ w' \triangleright qf'\ \rangle}{\phi;\rho \vdash \langle\ w \triangleright \mathsf{apply}\ qf\ ps\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{apply}\ qf'\ ps\ \rangle} \tag{74}$$

4.9 modify

$$\frac{\vdash xf\ :\ \mathsf{XFun}\ T_1\ T_2 \qquad \vdash xf.\mathsf{b\_mod}\ w\ n\ =\ (w', n')}{\phi;\rho \vdash \langle\ w \triangleright \mathsf{modify}\ xf\ (\mathsf{cst}\ T_1\ n)\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{Val}\ (\mathsf{cst}\ T_2\ n')\ \rangle} \tag{75}$$

$$\frac{\rho \vdash \langle\ w \triangleright qv\ \rangle \longrightarrow \langle\ w' \triangleright qv'\ \rangle}{\phi;\rho \vdash \langle\ w \triangleright \mathsf{modify}\ qv\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{modify}\ xf\ qv'\ \rangle} \tag{76}$$

4.10 parameters

$$\frac{\begin{array}{l}\vdash \mathsf{values}\ es_1\ \_ \\ \vdash ps\ =\ \mathsf{PS}\ (\mathsf{append}\ es_1\ (\mathsf{cons}\ e\ es_2)) \\ \vdash ps'\ =\ \mathsf{PS}\ (\mathsf{append}\ es_1\ (\mathsf{cons}\ e'\ es_2)) \\ \phi;\rho \vdash \langle\ w \triangleright e\ \rangle \longrightarrow \langle\ w' \triangleright e'\ \rangle\end{array}}{\phi;\rho \vdash \langle\ w \triangleright ps\ \rangle \longrightarrow \langle\ w' \triangleright ps'\ \rangle} \tag{77}$$

4.11 prog

$$\frac{\phi; \rho \vdash \langle\ w \triangleright e\ \rangle \longrightarrow \langle\ w' \triangleright e'\ \rangle}{\phi; \rho \vdash \langle\ w \triangleright \mathsf{prog}\ e\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{prog}\ e'\ \rangle} \tag{78}$$

4.12 define

$$\frac{}{\phi \vdash \langle\ w \triangleright \mathsf{define}\ \_\ \_\ (\mathsf{prog}\ (\mathsf{val}\ v))\ \rangle \longrightarrow \langle\ w \triangleright \mathsf{prog}\ (\mathsf{val}\ v)\ \rangle} \tag{79}$$

$$\frac{(x,\ f)/\phi \vdash \langle\ w \triangleright \pi\ \rangle \longrightarrow \langle\ w' \triangleright \pi'\ \rangle}{\phi \vdash \langle\ w \triangleright \mathsf{define}\ x\ f\ \pi\ \rangle \longrightarrow \langle\ w' \triangleright \mathsf{define}\ x\ f\ \pi'\ \rangle} \tag{80}$$

5 Declarative static semantics

5.1 Expression lists

$$\frac{}{\Phi; \Gamma \vdash \mathsf{nil}\ ::\ \mathsf{nil}} \tag{81}$$

$$\frac{\Phi; \Gamma \vdash e\ ::\ t \qquad \Phi; \Gamma \vdash es\ ::\ ts}{\Phi; \Gamma \vdash \mathsf{cons}\ e\ es\ ::\ \mathsf{cons}\ t\ ts} \tag{82}$$

5.2 Parameters

$$\frac{\Phi; \Gamma \vdash es\ ::\ ts}{\Phi; \Gamma \vdash \mathsf{PS}\ es\ ::\ \mathsf{PT}\ ts} \tag{83}$$

5.3 Functions

$$\frac{\begin{array}{l}\vdash \phi\ ::\ \Phi \qquad \vdash n\ :\ \mathsf{nat} \\ \Phi; \Gamma \vdash e_0\ ::\ t \\ (x,\ \Gamma \Rightarrow t)/\Phi; \Gamma \vdash e_1\ ::\ t\end{array}}{\vdash \mathsf{FC}\ \phi\ \Gamma\ e_0\ e_1\ x\ n\ ::\ \Gamma \Rightarrow t} \tag{84}$$

5.4 Quasi-functions

$$\frac{\vdash \Phi\ x\ =\ \mathsf{Some}\ \tau}{\Phi \vdash \mathsf{FVar}\ x\ ::\ \tau} \tag{85}$$

$$\frac{\vdash f\ ::\ \tau}{\Phi \vdash \mathsf{QF}\ f\ ::\ \tau} \tag{86}$$

5.5 Expressions

$$\frac{\vdash v \ :: \ t}{\Phi;\Gamma \vdash \mathsf{val}\ v :: \ t} \tag{87}$$

$$\frac{\Gamma \vdash qv \ :: \ t \qquad \vdash tg \ : \ \mathsf{Tag}}{\Phi;\Gamma \vdash \mathsf{return}\ tg\ qv \ :: \ t} \tag{88}$$

$$\frac{\Phi;\Gamma \vdash e_1 \ :: \ t_1 \qquad \Phi;\Gamma \vdash e_2 \ :: \ t_2}{\Phi;\Gamma \vdash \mathsf{bindN}\ e_1\ e_2 \ :: \ t_2} \tag{89}$$

$$\frac{\Phi;\Gamma \vdash e_1 \ :: \ t_1 \qquad \Phi;(x, \ t_1)/\Gamma \vdash e_2 \ :: \ t_2}{\Phi;\Gamma \vdash \mathsf{bindS}\ x\ e_1\ e_2 \ :: \ t_2} \tag{90}$$

$$\frac{\vdash \phi \ :: \ \Phi' \qquad \vdash \rho \ :: \ \Gamma' \qquad \Phi'/\Phi;\Gamma'/\Gamma \vdash e \ :: \ t}{\Phi;\Gamma \vdash \mathsf{bindMS}\ \phi\ \rho\ e \ :: \ t} \tag{91}$$

$$\frac{\Phi;\Gamma \vdash e_1 \ :: \ \mathsf{vtyp\ bool} \qquad \Phi;\Gamma \vdash e_2 \ :: \ t \qquad \Phi;\Gamma \vdash e_3 \ :: \ t}{\Phi;\Gamma \vdash \mathsf{ifThenElse}\ e_1\ e_2\ e_3 \ :: \ t} \tag{92}$$

$$\frac{\Phi \vdash qf \ :: \ \Gamma' \Rightarrow t \qquad \Phi;\Gamma \vdash ps \ :: \ \mathsf{tc2pt}\ \Gamma'}{\Phi;\Gamma \vdash \ \mathsf{apply}\ qf\ ps \ :: \ t} \tag{93}$$

$$\frac{\vdash xf \ : \ \mathsf{XFun}\ T_1\ T_2 \qquad \Gamma \vdash qv \ :: \ \mathsf{vtyp}\ T_1}{\Phi;\Gamma \vdash \mathsf{modify}\ xf\ qv \ :: \ \mathsf{vtyp}\ T_2} \tag{94}$$

5.6 Programs

$$\frac{\Phi \vdash e \ :: \ t}{\Phi \vdash \mathsf{prog}\ e \ :: \ t} \tag{95}$$

$$\frac{\vdash f \ :: \ \tau \qquad (x, \ \tau)/\Phi \vdash \pi \ :: \ t}{\Phi \vdash \mathsf{define}\ x\ f\ \pi \ :: \ t} \tag{96}$$

6 Reflective translation (outline)

We give an informal specification of the reflective translation to the Gallina fragment used in the executable model of Pip. We focus on the most interesting case – expression translation (Θ_e). TC stands for the typing context information, and E for environment information (moreover including information about the external monadic functions). We informally denote by $\mathsf{ext}\ \ x\ \ e\ \ TC$ as the extension of TC by associating the identifier x with the type of expression e, and by $\mathsf{ext}\ \ x\ \ e\ \ E$ as the extension of E by associating the identifier x with the value obtained from e. We leave unspecified the translation of types (Θ_t), values (Θ_v), quasi-values (Θ_{qv}) and quasi-functions (Θ_{qf}), which are not problematic, the translation of parameters (Θ_{ps}), which essentially maps expression translation on the underlying list, and the translation of functions (Θ_f), which boils down to the translation of the function bodies.

The term $\Theta_e\ e\ TC$ is meant to have type which depends on the type of e and on TC. The term $\Theta_e\ e\ TC$ is meant to have type which depends solely on the type t of e – indeed, type $\Theta_t\ t$.

$$
\begin{array}{lcl}
\Theta_e\ (\mathsf{val}\ v, \_, \_) & = & \mathsf{ret}\ (\Theta_v\ v) \\
\Theta_e\ (\mathsf{return}\ \_\ qv)\ \_\ E & = & \mathsf{ret}\ (\Theta_{qv}\ qv\ E) \\
\Theta_e\ (\mathsf{bindN}\ e_1\ e_2)\ TC\ E & = & \mathsf{bind}\ (\Theta_e\ e_1\ TC\ E)\ (\Theta_e\ e_2\ TC\ E) \\
\Theta_e\ (\mathsf{bindS}\ x\ e_1\ e_2)\ TC\ E & = & \mathsf{bind}\ (\Theta_e\ e_1\ TC\ E) \\
 & & \quad (\Theta_e\ e_2\ (\mathsf{ext}\ x\ e_1\ TC)\ (\mathsf{ext}\ x\ e_1\ E)) \\
\Theta_e\ (\mathsf{ifThenElse} & & \\
\quad (\mathsf{return}\ qv)\ e_1\ e_2)\ TC\ E & = & \mathsf{if}\ (\Theta_{qv}\ qv\ E)\ \mathsf{then}\ \ (\Theta_e\ e_1\ TC\ E) \\
 & & \qquad\qquad \mathsf{else}\ (\Theta_e\ e_2\ TC\ E) \\
\Theta_e\ (\mathsf{apply}\ qf\ ps)\ TC\ E & = & \mathsf{bind}\ (\Theta_{ps}\ ps\ TC\ E)\ (\Theta_{qf}\ qf\ TC\ E) \\
\Theta_e\ (\mathsf{modify}\ xf\ qv)\ TC\ E & = & (\Theta_{xf}\ xf\ TC\ E)\ (\Theta_{qv}\ qv\ E)
\end{array}
\tag{97}
$$

7 Translation to C (outline)

We give an informal specification of the translation to C, again focusing on expressions. Essentially, the translation returns recursively a C expression paired with a list of C variable declarations. Typing information is used pervasively, but omitted in this schematic presentation. DEC identifiers are translated to C variables. Sequencing is translated to comma expressions ($,_c$), local variables to comma expressions with assignment ($=_c$), conditionals to conditional expressions (? :), and function calls to function calls ($\_\ \_$). Function translation boils down to translating the function bodies and lifting them to statements. The top-level function environment is translated by mapping function translation on the list.

$$
\begin{array}{lcl}
\Xi_e\ (\mathsf{val}\ v) & = & (\Xi_v\ v,\ \mathsf{nil}) \\
\Xi_e\ (\mathsf{return}\ \_\ v) & = & (\Xi_{qv}\ v,\ \mathsf{nil}) \\
\Xi_e\ (\mathsf{bindN}\ e_1\ e_2) & = & \mathsf{let}\ (ee_1, dd_1)\ :=\ (\Xi_e\ e_1) \\
 & & \quad (ee_2, dd_2)\ :=\ (\Xi_e\ e_2) \\
 & & \quad \mathsf{in}\ ((ee_1\ ,_c\ ee_2),\ \mathsf{concat}\ dd_1\ dd_2) \\
\Xi_e\ (\mathsf{bindS}\ x\ e_1\ e_2) & = & \mathsf{let}\ (ee_1, dd_1)\ :=\ (\Xi_e\ e_1) \\
 & & \quad (ee_2, dd_2)\ :=\ (\Xi_e\ e_2) \\
 & & \quad\ \ i\ :=\ \Xi_i\ x \\
 & & \quad \mathsf{in}\ ((i\ =_c\ ee_1\ ,_c\ ee_2), \\
 & & \quad\ \ \mathsf{ext}\ i\ (\Xi_t\ e_1)\ (\mathsf{concat}\ dd_1\ dd_2)) \\
\Xi_e\ (\mathsf{ifThenElse} & & \\
\quad (\mathsf{return}\ qv)\ e_1\ e_2) & = & \mathsf{let}\ (ee_1, dd_1)\ :=\ (\Xi_e\ e_1) \\
 & & \quad (ee_2, dd_2)\ :=\ (\Xi_e\ e_2) \\
 & & \quad \mathsf{in}\ ((\Xi_{qv}\ qv)\ ?\ ee_1\ :\ ee_2, \\
 & & \quad\ \ \mathsf{concat}\ dd_1\ dd_2) \\
\Xi_e\ (\mathsf{apply}\ qf\ ps) & = & \mathsf{let}\ (pps, dd)\ =\ (\Xi_e\ ps) \\
 & & \quad \mathsf{in}\ ((\Xi_{qf}\ qf)\ pps,\ dd) \\
\Xi_e\ (\mathsf{modify}\ xf\ qv) & = & ((\Xi_{xf}\ xf)\ (\Xi_{qv}\ qv),\ \mathsf{nil})
\end{array}
\tag{98}
$$

We can refine the above translation function to one which targets CompCert C (`http://compcert.inria.fr/compcert-C.html`). We can then rely on the CompCert representation of C expressions as a datatype (in *Csyntax.expr*). Here the translation returns recursively a C expression paired with its type, which is needed by CompCert, and a list of C global declarations. The latter results in a list of CompCert type (ident $*$ globdef fundef type), given the translation of DEC identifiers to C identifiers. As before, sequencing is translated to comma expressions (Ecomma), local variables to comma expressions with assignment (Eassign), conditionals to conditional expressions (Econdition), and function calls to function calls (Ecall).

$$
\begin{array}{lcl}
\Psi_e\ (\mathsf{val}\ v) & = & (\Psi_v\ v,\ \mathsf{nil}) \\
\Psi_e\ (\mathsf{return}\ \_\ v) & = & (\Psi_{qv}\ v,\ \mathsf{nil}) \\
\Psi_e\ (\mathsf{bindN}\ e_1\ e_2) & = & \mathsf{let}\ (ee_1,\ \_,\ dd_1)\ :=\ (\Psi_e\ e_1) \\
 & & \quad\ (ee_2,\ ty,\ dd_2)\ :=\ (\Psi_e\ e_2) \\
 & & \quad\ \mathsf{in}\ ((\mathsf{Ecomma}\ ee_1\ ee_2\ ty), \\
 & & \quad\quad ty,\ \mathsf{concat}\ dd_1\ dd_2) \\
\Psi_e\ (\mathsf{bindS}\ x\ e_1\ e_2) & = & \mathsf{let}\ (ee_1,\ ty_1,\ dd_1)\ :=\ (\Psi_e\ e_1) \\
 & & \quad\ (ee_2,\ ty_2,\ dd_2)\ :=\ (\Psi_e\ e_2) \\
 & & \quad\ i\ :=\ \Psi_i\ x \\
 & & \quad\ \mathsf{in}\ (\mathsf{Ecomma}\ (\mathsf{Eassign}\ i\ ee_1\ ty_1)\ ee_2\ ty_2, \\
 & & \quad\quad ty_2,\ \mathsf{cons}\ (i,\ ty_1)\ (\mathsf{concat}\ dd_1\ dd_2)) \\
\Psi_e\ (\mathsf{ifThenElse} & & \\
\quad (\mathsf{return}\ qv)\ e_1\ e_2) & = & \mathsf{let}\ (vv,\ \mathsf{Tint\ IBool}\ \_\ \_)\ :=\ (\Psi_{qv}\ qv) \\
 & & \quad\ (ee_1, ty,\ dd_1)\ :=\ (\Psi_e\ e_1) \\
 & & \quad\ (ee_2,\ ty,\ dd_2)\ :=\ (\Psi_e\ e_2) \\
 & & \quad\ \mathsf{in}\ (\mathsf{Econdition}\ vv\ ee_1\ ee_2\ ty, \\
 & & \quad\quad ty,\ \mathsf{concat}\ dd_1\ dd_2) \\
\Psi_e\ (\mathsf{apply}\ qf\ ps) & = & \mathsf{let}\ (pps,\ tts,\ dd)\ :=\ (\Psi_{ps}\ ps) \\
 & & \quad\ (ff,\ \mathsf{Tfunction}\ tys\ ty\ \_)\ :=\ (\Psi_{fs}\ qf) \\
 & & \quad\ \mathsf{in}\ (\mathsf{Ecall}\ ff\ pps\ ty,\ ty,\ dd) \\
\Psi_e\ (\mathsf{modify}\ xf\ qv) & = & \mathsf{let}\ tys\ :=\ \mathsf{Tcons}\ tt\ \mathsf{Tnil} \\
 & & \quad\ (qqv,\ tts)\ :=\ (\Psi_{qv}\ qv) \\
 & & \quad\ (ff,\ \mathsf{Tfunction}\ tys\ ty\ \_)\ :=\ (\Psi_{xf}\ xf) \\
 & & \quad\ \mathsf{in}\ (\mathsf{Ecall}\ ff\ qv\ ty,\ ty,\ \mathsf{nil})
\end{array}
\tag{99}
$$