Dynamic Semantics of VDM-SL

Peter Gorm Larsen

IFAD

March 1995

Basic Semantic Domains

Models are mappings from identifiers to values:

$$ENV_{PURE} = \mathbb{E}(VAL \cup DOM \cup OPVAL \cup POLYVAL)$$

$$VAL = \bigcup \{ |A| \mid ((|A|, \sqsubseteq_A), \|A\|) \in DOM \}$$

$$DOM = \{ ((|A|, \sqsubseteq_A), \|A\|) \mid (|A|, \sqsubseteq_A) \in CPO, \|A\| \subseteq |A| \setminus \{ \bot \} \}$$

$$OPVAL = \mathbb{F}(Input \times State \times State \times Output \times Mode\ )$$

$$POLYVAL = \mathbb{L}_1(DOM) \to VAL$$

$$Input = DOM$$

$$Output = DOM$$

$$State = DOM$$

$$Mode = \{ \underline{\text{cont}}, \underline{\text{exit}}, \underline{\text{ret}} \}$$

The Style of the Definition

The direct style:

$$SemSpec : Document \to \mathbb{P}(ENV_{PURE})$$

$$SemSpec(spec) \triangleq EvalSpec(spec)$$

The relational style:

$$SemSpec : Document \to \mathbb{P}(ENV_{PURE})$$

$$SemSpec(doc) \triangleq$$
$$\{\, env \mid env \in ENV_{PURE} \cdot$$
$$IsAModelOf(env,\ doc) \,\}$$

where *IsAModelOf* corresponds to *sat* used by Hoare.

Environment Expansion

$IsAModelOf : \mathbf{Pred}(ENV_{PURE} \times Document)$

$IsAModelOf(env,\ defs) \triangleq$
  let $MkTag(\text{`}Definitions\text{'},\ t,\ epf,\ ef,\ ipf,\ if,\ v,\ eo,\ io,\ s) = defs$ **in**
    $DisjointIds(t,\ epf,\ ef,\ ipf,\ if,\ v,\ eo,\ io,\ s,\ env) \wedge$
    let $exp\_env = ExpandImplDefs(t,\ s,\ ipf,\ if,\ io)(env)$ **in**
      now verify each component (definition) of the specification
      ...

The expanded environment has the type:

$$ENV = ENV_{PURE} \cup \mathbb{E}(MOD\_FUN \cup SEL\_FUN \cup CONS\_FUN \cup STATE\_FUN \cup TAG\_ENV \cup LOC\_ENV)$$

Composite Types

Modifier functions:

$$MOD\_FUN = \mathbb{E}(VAL) \rightarrow RECORD\_VAL \rightarrow VAL$$

Selector functions:

$$SEL\_FUN = RECORD\_VAL \rightarrow VAL$$

Constructor functions:

$$CONS\_FUN = \mathbb{L}(VAL) \rightarrow RECORD\_VAL_{\perp}$$

The other extensions

Read state checking functions:

$$STATE\_FUN = \mathbb{IF}(ExtVarInf) \rightarrow (VAL \cup \{\underline{\text{nil}}\} \times VAL \cup \{\underline{\text{nil}}\}) \rightarrow \mathbb{IB}$$

The 'tag' environment:

$$TAG\_ENV = \mathbb{IE}(CompositeType)$$

Type checking of locally declared variables in block statements:

$$LOC\_ENV = \mathbb{IE}(DOM)$$

Verification Predicates in General

$VerifyAnXx : \mathbf{Pred}(\mathbb{E}(AnXx) \times ENV)$

$VerifyAnXx(Xx\_m,\ env) \triangleq$
 if $\underline{\text{dom}}(Xx\_m) \subseteq \underline{\text{dom}}(env) \wedge$
  $\forall id \in \underline{\text{dom}}(Xx\_m) \cdot env(id) \in$ right domain
 then let $sub\_models = EvalAnXx(Xx\_m)$
  $(\underline{\text{subtract}}(env,\ \underline{\text{dom}}(Xx\_m)))$ **in**
  $\exists env' \in sub\_models \cdot$
  let $env'' =$ env overwritten by env' **in**
  $\forall id \in \underline{\text{dom}}(Xx\_m) \cdot env(id) = env''(id)$
 else F

Evaluation Functions in general

For constructs with looseness:

$EvalAnXx : \mathbb{E}(XxDef) \rightarrow \ldots$

$EvalAnXx(xx\_m)(env) \triangleq$

$\{\, \lambda env \cdot$ add the objects defined by xx_m to env

$\mid$ over the looseness $\}$

For constructs without looseness:

$EvalAnXx : \mathbb{E}(XxDef) \rightarrow \ldots$

$EvalAnXx(xx\_m)(env) \triangleq$

$\lambda env \cdot$ add the objects defined by xx_m to env

Semantic Evaluation Domains

The semantics of definitions is expressed in terms of "definers":

$$Def = ENV \rightarrow (ENV \cup \{\underline{\text{err}}\})$$

If definitions can be loose, "loose definers" are used:

$$LDef = \mathbb{P}(Def)$$

For constructs used in definitions different kind of "evaluators" are used. If looseness can be present "loose evaluators" are used:

$$LXEval = \mathbb{P}(XEval)$$

Verifying Type Definitions

$$VerifyTypes : \mathbf{Pred}(\mathbb{E}(TypeDef) \times ENV)$$

$$VerifyTypes(tpdef\_m,\ env) \triangleq$$
$$\mathbf{if}\ \underline{\mathrm{dom}}(tpdef\_m) \subseteq \underline{\mathrm{dom}}(env) \wedge$$
$$\quad \forall id \in \underline{\mathrm{dom}}(tpdef\_m) \cdot env(id) \in DOM$$
$$\mathbf{then\ let}\ env' = EvalTypeDef(tpdef\_m)$$
$$(\underline{\mathrm{subtract}}(env,\ \underline{\mathrm{dom}}(tpdef\_m)))\ \mathbf{in}$$
$$\mathbf{if}\ env' = \underline{\mathrm{err}}$$
$$\mathbf{then}\ \mathsf{F}$$
$$\mathbf{else}\ \forall id \in \underline{\mathrm{dom}}(tpdef\_m) \cdot env(id) = env'(id)$$
$$\mathbf{else}\ \mathsf{F}$$

Evaluating Type Definitions

Without recursion:

$$EvalTypeDef : \mathbb{E}(TypeDef) \to Def$$

$$EvalTypeDef(td\_m)(env) \triangleq$$
$$\textbf{let } id\_s = \underline{\text{dom}}(td\_m) \textbf{ in}$$
$$\textbf{let } tev\_m = \{\, id \mapsto \text{its domain} \mid id \in id\_s \,\} \textbf{ in}$$
$$\textbf{let } itf = \{\, id \mapsto \text{its invariant} \mid id \in id\_s \,\} \textbf{ in}$$
$$\textbf{let } d\_tf = \{\, id \mapsto \textbf{let } d = tev\_m(id)(env) \textbf{ in}$$
$$\textbf{let } inv = itf(id)(env) \textbf{ in}$$
$$\mathcal{I}(\{\, e \in \|d\| \mid inv(e) = True() \,\},\ d)$$
$$\mid id \in id\_s \,\} \textbf{ in}$$
$$\underline{\text{overwrite}}(env, dt\_f)$$

Evaluating Type Definitions

With recursion:

$$EvalTypeDef : \mathbb{IE}(TypeDef) \rightarrow Def$$

$$EvalTypeDef(td\_m)(env) \triangleq$$
$$\textbf{let } id\_s = \underline{\text{dom}}(td\_m) \textbf{ in}$$
$$\textbf{let } tev\_m = \{ id \mapsto EvalType(SelShape(td\_m(id))) \mid id \in id\_s \} \textbf{ in}$$
$$\textbf{let } itf = Iota(EvalExplDef\ (ColInvFns\ (td\_m\ ))) \textbf{ in}$$
$$\textbf{let } d\_tf = Approx(t\_dm, tev\_m, itf, env) \textbf{ in}$$
$$\underline{\text{overwrite}}(env, \mathsf{Y}(dt\_f\ ))$$

The *Approx* Function

$$Approx : \mathbb{E}(TypeDef) \rightarrow Def$$

$$Approx(td\_m, tev\_m, itf, env)(appr) \triangleq$$
$$\{\ id \mapsto \textbf{let}\ env' = \underline{\text{overwrite}}(env, appr)\ \textbf{in}$$
$$\textbf{let}\ d = tev\_m\ (id\ )(env')\ \textbf{in}$$
$$\textbf{let}\ inv = itv\ (env')(SelInvId\ (td\_m(id)))\ \textbf{in}$$
$$\mathcal{I}(\{\ e \in \|d\| \mid inv(e) = True()\ \},\ d)$$
$$\mid id \in \underline{\text{dom}}\ td\_m\ \}$$

Evaluation of Basic Types

$EvalBasicType : BasicType \rightarrow TEval$

$EvalBasicType(MkTag(\text{`}BasicType\text{'}, tp))(env) \triangleq$

cases tp :

NATONE	$\rightarrow$	nat1-dom,
NAT	$\rightarrow$	nat-dom,
INTEGER	$\rightarrow$	int-dom,
RAT	$\rightarrow$	rat-dom,
REAL	$\rightarrow$	real-dom,
BOOLEAN	$\rightarrow$	Bool-dom,
CHAR	$\rightarrow$	char-dom,
TOKEN	$\rightarrow$	token-dom,
UNIT	$\rightarrow$	nil-dom

Evaluation of other Types

$EvalXxType : XxType \to TEval$

$EvalXxType(MkTag(\text{'}XxType\text{'}, t))(env) \triangleq$
  let $d = EvalType(t)(env)$ **in**
    'build a domain from'(d)

This style can be seen by:

$EvalSetType : SetType \to TEval$

$EvalSetType(MkTag(\text{'}SetType\text{'}, t))(env) \triangleq$
  let $d = EvalType(t)(env)$ **in**
    if $d = \underline{\text{err}} \vee d \notin FLATDOM$
    then $\underline{\text{err}}$
    else $\mathcal{S}(d)$

Verifying Explicit Operations

$$VerifyExplOps : \mathbf{Pred}(\mathbb{E}(ExplOpDef) \times ENV)$$

$$VerifyExplOps(op\_m,\ env) \triangleq$$
$$\quad \mathbf{let}\ id\_s = \underline{\mathrm{dom}}(op\_m)\ \mathbf{in}$$
$$\quad\quad \mathbf{if}\ id\_s \subseteq \underline{\mathrm{dom}}(env) \wedge \forall id \in id\_s \cdot env(id) \in OPVAL$$
$$\quad\quad \mathbf{then}\ \forall id \in id\_s \cdot$$
$$\quad\quad\quad \mathbf{let}\ rel = env(id),$$
$$\quad\quad\quad\quad st\_d = GetStateDom(env)\ \mathbf{in}$$
$$\quad\quad\quad st\_d \in DOM \wedge$$
$$\quad\quad\quad rel \subseteq \{ (\|fd_1\| \times \|st\_d\| \times \|st\_d\| \times (\|fd_2\| \cup \{\underline{\mathrm{nil}}\})$$
$$\quad\quad\quad\quad \times MODE)$$
$$\quad\quad\quad\quad \mid fd_1, fd_2 \in DOM\} \wedge$$
$$\quad\quad\quad rel = EvalExplOp(op\_m(id))(env)$$
$$\quad\quad \mathbf{else}\ \mathsf{F}$$

Evaluation Explicit Operations

$$EvalExplOp : ExplOpDef \to ENV \to OPVAL$$

$$EvalExplOp(MkTag(\text{`}ExplOpDef\text{'}, (h, body)))(env) \triangleq$$
$$\textbf{let } MkTag(\text{`}OpHeading\text{'}, (MkTag(\text{`}Par\text{'}, (id, tp)), restp)) = h \textbf{ in}$$
$$\textbf{let } i\_s = \|EvalType(tp)(env)\| \cup \{\bot\},$$
$$o\_s = \|EvalType(restp)(env)\| \cup \{\bot\},$$
$$st\_s = \|GetStateDom(env)\| \cup \{\bot\} \textbf{ in}$$
$$\{(i\_v, i\_st, o\_st, o\_v, m)$$
$$\mid i\_v \in i\_s, i\_st \in st\_s, o\_st \in st\_s, o\_v \in o\_s, m \in MODE \cdot$$
$$\textbf{let } (e', e'') = ExtExplOpEnv(id, i\_v, i\_st, o\_st, ext)(env) \textbf{ in}$$
$$CheckStateConstancy(e')(ext)(i\_st, o\_st) = True() \wedge$$
$$\textbf{let } lsev = EvalStmt(body) \textbf{ in}$$
$$\exists\, sev \in lsev \cdot sev(env') = (e'', m, o\_v)\}$$

Semantics of Expressions

Without looseness in expressions:

$$EvalExpr : Expr \to ENV \to VAL$$

First attempt with looseness:

$$EvalExpr : Expr \to ENV \to \mathbb{P}(VAL)$$

Final signature for *EvalExpr*

$$EvalExpr : Expr \to \mathbb{P}(ENV \to VAL)$$

Semantic Functions for Expressions

$$EvalAnExpr : AnExpr \to LEEval$$

$$EvalAnExpr(MkTag(\text{`}AnExpr\text{'}, (expr_1,\ldots,op,\ldots,expr_n))) \triangleq$$
$$\{\, \lambda env \;.\; \textbf{let}\; val_1 = ev_1(env),$$
$$\ldots,$$
$$val_n = ev_n(env)\; \textbf{in}$$
$$AnOp(val_1,\ldots,val_n,op)$$
$$\mid\; ev_1 \in EvalExpr(expr_1),\ldots,\; ev_n \in EvalExpr(expr_n)\,\}$$

Semantics of If-then-else Expressions

$$EvalIfExpr : IfExpr \to LEEval$$

$$EvalIfExpr(MkTag(\text{'}IfExpr\text{'}, (test, cons, altn))) \triangleq$$
$$\{\lambda env.\textbf{if } testev(env) \notin BOOL\_VAL$$
$$\textbf{then } \bot$$
$$\textbf{else if } testev(env) = True()$$
$$\textbf{then } consev(env)$$
$$\textbf{else } altnev(env)$$
$$\mid testev \in EvalExpr(test),$$
$$consev \in EvalExpr(cons),$$
$$altnev \in EvalExpr(altn)\}$$

Semantics of Let-be Expressions

$$EvalLetBeSTExpr : LetBeSTExpr \to LEEval$$

$$EvalLetBeSTExpr(MkTag(\text{`}LetBeSTExpr\text{'}, (bind, st, in))) \triangleq$$
$$PropE(\{\, \lambda env.\textbf{let } venv\_s = \bigcup\{\, bev(env) \mid bev \in EvalBind(bind)\,\} \textbf{ in}$$
$$\textbf{let } venv\_s' = \{\, venv \mid venv \in venv\_s \cdot$$
$$venv \neq \underline{\text{err}} \wedge \neg BotEnv(venv) \wedge$$
$$stev(\underline{\text{overwrite}}(env, venv)) =$$
$$True()\,\} \textbf{ in}$$
$$\{\, inev(\underline{\text{overwrite}}(env, venv)) \mid venv \in venv\_s'\,\}$$
$$\mid stev \in EvalExpr(st),\ inev \in EvalExpr(in)\,\})$$

Evaluation of Disjunction

$$EvalOrExpr : OrExpr \to LEEval$$

$$EvalOrExpr(MkTag(\text{`}OrExpr\text{'}, (left, \text{OR}, right))) \triangleq$$
$$\{\, \lambda env.\textbf{let } l\_v = l\_ev(env),$$
$$r\_v = r\_ev(env) \textbf{ in}$$
$$\textbf{if } l\_v \in BOOL\_VAL_{\perp} \wedge r\_v \in BOOL\_VAL_{\perp}$$
$$\textbf{then if } (l\_v = True()) \vee (r\_v = True())$$
$$\textbf{then } True()$$
$$\textbf{else if } (l\_v = False()) \wedge (r\_v = False())$$
$$\textbf{then } False()$$
$$\textbf{else } \perp$$
$$\textbf{else } \perp$$
$$\mid l\_ev \in EvalExpr(left),\ r\_ev \in EvalExpr(right)\,\}$$

Evaluation of Set Comprehension

$$
\begin{aligned}
&EvalSetComprehension : SetComprehension \to LEEval \\
&EvalSetComprehension(MkTag(\text{`}SetComprehension\text{'}, \\
&\qquad\qquad (elems,\ binds,\ pred\ ))) \triangleq \\
&\quad \textbf{let}\ pbev\_s = Partition(\{\ EvalBind(b) \mid b \in binds\}), \\
&\qquad plev = EvalExpr(pred)\ \textbf{in} \\
&\quad \{\lambda env\ .\ \textbf{let}\ venv\_ss = \{\ bev(env) \mid bev \in pbev\ \}\ \textbf{in} \\
&\qquad\qquad \textbf{let}\ den = \{\ ev(\underline{\text{overwrite}}(env,\underline{\text{merge}}(venvs))) \\
&\qquad\qquad\qquad\qquad \mid venvs \in Partition(venv\_ss)\ \cdot \\
&\qquad\qquad\qquad\qquad pev(\underline{\text{overwrite}}(env,venvs))\ \}\textbf{in} \\
&\qquad\qquad MkTag(\text{`}set\text{'},\ den) \\
&\quad \mid ev \in EvalExpr(elem\ ),\ pbev \in pbev\_s,\ pev \in EvalExpr(pred\ )\ \}
\end{aligned}
$$

Semantics of Lambda Expressions

$$EvalLambda : Lambda \to LEEval$$

$$\begin{aligned}
&EvalLambda(MkTag(\text{`Lambda'}, (MkTag(\text{`Par'}, (id, tp)), body))) \triangleq \\
&\quad \textbf{let } q = \lambda env \,.\, \textbf{let } d = EvalType(tp)(env) \textbf{ in} \\
&\qquad \lambda ob \in |d| \,.\, \{ \textbf{if } ob \in \|d\| \\
&\qquad\qquad \textbf{then } ev(\underline{\text{overwrite}}(env, \{ id \mapsto ob \})) \\
&\qquad\qquad \textbf{else } \bot \\
&\qquad\quad \mid ev \in EvalExpr(body) \} \textbf{ in} \\
&\quad \{ eeval \mid eeval \in EEval \cdot \\
&\qquad \forall\ env \in ENV \cdot \\
&\qquad\quad \delta_0(eeval(env)) = |EvalType(tp)(env)| \wedge \\
&\qquad\quad \forall\ ob \in |EvalType(tp)(env)| \cdot \\
&\qquad\qquad eeval(env)(ob) \in q(env)(ob)) \}
\end{aligned}$$

Evaluation of Statements

Loose statement evaluators:

$$LSEval = \mathbb{P}(SEval)$$

Statement evaluators:

$$SEval = ENV \to (ENV \times MODE \times (VAL \cup \{\underline{\text{nil}}\}))$$

Triples of:

1. modified environments
2. termination mode
3. returned value

Evaluation of Sequence Statements

$EvalSequence : Sequence \to LSEval$

$EvalSequence(MkTag(\text{`Sequence'}, stmt\_l)) \triangleq$
if $\underline{\text{len}}(stmt\_l) = 0$
then $\{\, \lambda env \,.\, (env, \underline{\text{cont}}, \underline{\text{nil}}) \,\}$
else $\{\, \lambda env$ **.let** $(env', mode, val) = sev_1(env)$ **in**
if $mode \neq \underline{\text{cont}}$
then $(env', mode, val)$
else $sev_2(env')$
$\mid sev_1 \in EvalStmt(\underline{\text{hd}}(stmt\_l)),$
$sev_2 \in EvalSequence(MkTag(\text{`Sequence'}, \underline{\text{tl}}(stmt\_l)))\,\}$

Evaluation of loop Statements

$$EvalSeqForLoop : SeqForLoop \to LSEval$$

$$EvalSeqForLoop(MkTag(\text{'SeqForLoop'}, (cv,\ dirn,\ seq,\ body))) \triangleq$$
$$PropS(\{\ \lambda env.\textbf{if}\ l\_ev(env) \notin LIST\_VAL$$
$$\textbf{then}\ \{(env,\ \underline{\text{ret}},\ \bot)\}$$
$$\textbf{else let}\ l\_v = \textbf{if}\ dirn = \text{FORWARDS}$$
$$\textbf{then}\ StripSeqTagVal(l\_ev(env))$$
$$\textbf{else}\ Rev(StripTag(l\_ev(env)))\ \textbf{in}$$
$$\{\ sev(env) \mid sev \in DoLoop(cv,\ l\_v,\ body)\ \}$$
$$\mid l\_ev \in EvalExpr(seq)\ \})$$

Evaluation of Patterns

Loose pattern evaluators:

$$LPatEval = \mathbb{IP}(PatEval)$$

Pattern evaluators:

$$PatEval = VAL \to ENV \to (\mathbb{IE}(VAL) \cup \{\underline{\text{err}}, \underline{\text{unmatch}}\})$$

Evaluation of pattern identifiers

$$EvalPatternId : PatternId \to LPatEval$$

$$EvalPatternId(MkTag(\text{`}PatternId\text{'}, id)) \triangleq$$
$$\{\lambda val.\lambda env.\textbf{if } id = \underline{\text{nil}}$$
$$\textbf{then } \{\mapsto\}$$
$$\textbf{else } \{id \mapsto val\}\}$$

Evaluation of Tuple Patterns

$EvalTuplePattern : TuplePattern \to LPatEval$

$EvalTuplePattern(MkTag(\text{`}TuplePattern\text{'}, fields)) \triangleq$
$\{ \lambda val.\lambda env.$**if** $val \in TUPLE\_VAL$
then let $MkTag(\text{`}tuple\text{'}, l) = val$ **in**
$pev(MkTag(\text{`}seq\text{'}, l))(env)$
else if $val = \bot$
then $EvalBotPat(\underline{\text{elems}}(fields))$
else $\underline{\text{unmatch}}$
$\mid pev \in EvalPatternList(fields) \}$

Evaluation of Bindings

Loose binding evaluators:

$$LBindEval = \mathbb{P}(BindEval)$$

Binding evaluators:

$$BindEval = ENV \to \mathbb{P}(\mathbb{E}(VAL) \cup \{\underline{\text{err}}\})$$

Evaluation of set-bindings

$EvalSetBind : SetBind \rightarrow LBindEval$

$EvalSetBind(MkTag(\text{'SetBind'}, (pat, set))) \triangleq$
$\{\lambda env.\textbf{let } set\_v = ev(env) \textbf{ in}$
$\quad \textbf{if } set\_v \in SET\_VAL$
$\quad \textbf{then let } MkTag(\text{'set'}, s) = set\_v \textbf{ in}$
$\quad\quad \{pev(e)(env) \mid e \in s\} \backslash \{\underline{\text{unmatch}}\}$
$\quad \textbf{else } \{\underline{\text{err}}\}$
$\mid ev \in EvalExpr(set), pev \in EvalPattern(pat)\}$

Further Information

- Towards a Formal Semantics of The BSI/VDM Specification Language (IFIP'89)
- The Formal Semantics of ISO VDM-SL (September'95 CSI)
- The VDM Specification Language – Reading the Standard (Prentice-Hall'95)