Full Control-Flow Sensitivity for Definitional Interpreters

Technical Report

Kimball Germane

Brigham Young University

The Monad A monad $m : \mathit{Type} \to \mathit{Type}$ comprises a type operator and two associated operations.

$$\mathit{return} : \forall A.A \to m(A) \qquad \mathit{bind} : \forall A.\forall B.m(A) \to (A \to m(B)) \to m(B)$$

where *return* injects a (relatively) pure computation into a monadic (effectful) computation and *bind* sequences two effectful computations with functional dependence. In general, we use the standard semicolon notation $\mathbf{do}\, x \leftarrow c; f(x)$ in place of $\mathit{bind}(c)(f)$ and allow newlines in place of semicolons in multi-line definitions headed by **do**. The *return* and *bind* operations obey the following laws:

$$\begin{aligned} \mathit{bind}(\mathit{cmp})(\mathit{return}) &\equiv \mathit{cmp} && \text{[right unit]} \\ \mathit{bind}(\mathit{return}(x))(f) &\equiv f(x) && \text{[left unit]} \\ \mathit{bind}(\mathit{bind}(\mathit{cmp})(f))(g) &\equiv \mathit{bind}(\mathit{cmp})(\lambda x.\mathit{bind}(f(x))(g)) && \text{[associativity]} \end{aligned}$$

Reader Effect A monad $m : \mathit{Type} \to \mathit{Type}$ implements a reader effect for a type r if it includes operators

$$\mathit{ask} : m(r) \qquad \mathit{inEnv} : \forall A.r \times m(A) \to m(A)$$

where *ask* obtains the environment value and *inEnv* installs a given environment value for a given computation. The *ask* and *inEnv* operations obey the following law:

$$\mathit{inEnv}(x, \mathit{ask}) \equiv \mathit{return}(x)$$

For example, the computation $\mathbf{do}\, x \leftarrow \mathit{ask}; \mathit{inEnv}(x+1, \mathit{cmp})$ runs the computation *cmp* in an environment value one greater than that of the overall computation.

State Effect A monad $m : \mathit{Type} \to \mathit{Type}$ implements a state effect for a type s if it includes operators

$$\mathit{get} : m(s) \qquad \mathit{put} : s \to m(1)$$

which satisfy the state laws [1]

$$
\begin{aligned}
\mathbf{do}\, put(x); get &\equiv \mathbf{do}\, put(x); return(x) && \text{(put–get)}\\
\mathbf{do}\, x \leftarrow get; put(x) &\equiv return() && \text{(get–put)}\\
\mathbf{do}\, put(x); put(y) &\equiv put(y) && \text{(put–put)}\\
\mathbf{do}\, x \leftarrow get; y \leftarrow get; f(x,y) &\equiv \mathbf{do}\, x \leftarrow get; f(x,x). && \text{(get–get)}
\end{aligned}
$$

The put–get law ensures that *get* sees the effect of *put*, the get–put law that *get* produces the entire state, the put–put law that the most-recent *put* takes effect, and the get–get law that *get* does not modify the state.

Nondeterminism Effect A monad $m : Type \to Type$ implements a nondeterminism effect if it includes operators

$$
\cdot\langle + \rangle\cdot : \forall A.m(A) \times m(A) \to m(A) \qquad\qquad mzero : \forall A.m(A)
$$

which satisfy the laws

$$
\begin{aligned}
mzero\langle + \rangle cmp \equiv cmp\langle + \rangle mzero &\equiv cmp\\
\mathbf{do}\, x \leftarrow mzero; f(x) &\equiv mzero\\
\mathbf{do}\, x \leftarrow cmp_0\langle + \rangle cmp_1; f(x) &\equiv [\mathbf{do}\, x \leftarrow cmp_0; f(x)]\langle + \rangle[\mathbf{do}\, x \leftarrow cmp_0; f(x)]
\end{aligned}
$$

In general, $m(A)$ must be a monoid, with $\langle + \rangle$ its associative binary operator and *mzero* its neutral element; in this work, we require $m(A)$ to be a join-semilattice, with $\langle + \rangle$ its join operator and *mzero* its bottom element.

Writer Effect A monad $m : Type \to Type$ implements a writer effect for a type w if it includes an operator

$$
log : w \to m(1).
$$

In general, w is required only to be a monoid; in this paper, we only ever use it at a join-semilattice. For example, the computation $\mathbf{do}\, log(x); log(y); return(10)$ produces the value 10 and logs the value $x \sqcup y$ for $x, y : w$ and $\sqcup$ some join operator over w.

1 Recovering a Concrete Semantics

To instantiate the interpreter, we instantiate each of the three parameters. Remarkably, in doing so, we can instantiate either a concrete or an abstract semantics. In this section, we provide instantiations which recover a concrete semantics; in the next, we do the same for an abstract semantics.

We will define two concrete semantics, a standard semantics in which the meaning of a program is its result and a collecting semantics in which the meaning of a program is an association between encountered configurations and their results. Both semantics share a the value domain Val_c and the timestamp set $Time_c$, defined as follows.

$$Val_c := \mathcal{P}(\mathbb{Z} \cup Clo_c) \qquad Time_c := \mathbf{Call}^* \qquad Clo_c := Clo^{Time_c}$$

$$Env_c := Env^{Time_c} \qquad Store_c := Store^{Time_c, Val_c}$$

Figure 1 presents the injection and projection functions, primitive operation interpretation function, and timestamp increment function. As Val_c is a powerset

$$\begin{array}{ll}
I_{int} : \mathbb{Z} \to Val_c & E_{if0} : Val_c \to \mathcal{P}(Bool) \\
I_{int}(i) := \{i\} & E_{if0}(v) := \{\mathsf{true} : 0 \in v\} \cup \{\mathsf{false} : i \in v, i \neq 0\} \\
I_{clo} : Clo_c \to Val_c & E_{clo} : Val_c \to \mathcal{P}(Clo_c) \\
I_{clo}(c) := \{c\} & E_{clo}(v) := \{c : c \in v\} \\
tick : \mathbf{Call} \times Time_c \to Time_c & \delta : \mathbf{IOp} \to Val_c \times Val_c \to Val_c \\
tick(e, \tau) := e\tau & \delta[\![+]\!](v_0, v_1) := \{i_0 + i_1 : i_0 \in v_0; i_1 \in v_1\} \\
 & \delta[\![-]\!](v_0, v_1) := \{i_0 - i_1 : i_0 \in v_0; i_1 \in v_1\}
\end{array}$$

Fig. 1. The concrete value domain and timestamp set

domain, the injection and projection functions are set-oriented. However, in the concrete domain, these sets are only ever singleton or empty, and therefore the powerset domain is used merely to encode failure. The initial time $\tau_0 = \epsilon$.

1.1 Standard Semantics

We first instantiate a standard semantics in which the meaning of a program is its final value and store (encoded using nondeterminism). First, we define configurations and results as follows.

$$Config_c := \mathbf{Exp} \times Env_c \times Store_c \times Time_c \qquad Result_c := \mathcal{P}(Val_c \times Store_c)$$

We then define a type operator $Conc$ which admits a monad instance will reader, nondeterminism, and state effects as follows.

$$Conc := \forall A. Env_c \to Store_c \to Time_c \to \mathcal{P}(A \times Store_c)$$

Figure 2 contains the monad instance for $Conc$ to support the standard semantics. $Conc$ provides access to the environment and timestamp via reader effects, access to the store via a state effect, and encodes failure via a nondeterminism effect (carried through from the value domain). The reader and state effects are defined in terms of the base monad where possible to insulate their definitions from other effects, such as nondeterminism. The base monad threads the store

$$
\begin{aligned}
&\mathit{Conc} := \forall A.\mathit{Env}_c \to \mathit{Store}_c \to \mathit{Time}_c \to \mathcal{P}(A \times \mathit{Store}_c)\\
&\mathit{return}^{\mathit{Conc}} : \forall A.A \to \mathit{Conc}(A)\\
&\mathit{return}^{\mathit{Conc}}(x)(\rho)(\sigma)(\tau) := \{(x,\sigma)\}\\
&\mathit{bind}^{\mathit{Conc}} : \forall A.\forall B.\mathit{Conc}(A) \to (A \to \mathit{Conc}(B)) \to \mathit{Conc}(B)\\
&\mathit{bind}^{\mathit{Conc}}(\mathit{cmp})(f)(\rho)(\sigma)(\tau) := f(x_1)(\rho)(\sigma_1)(\tau) \cup \cdots \cup f(x_n)(\rho)(\sigma_n)(\tau)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{where } \{(x_1,\sigma_1),\ldots,(x_n,\sigma_n)\} := \mathit{cmp}(\rho)(\sigma)(\tau)\\
&\cdot\langle+\rangle\cdot : \forall A.\mathit{Conc}(A) \times \mathit{Conc}(A) \to \mathit{Conc}(A)\\
&\mathit{cmp}_0\langle+\rangle\mathit{cmp}_1 := \lambda\rho.\lambda\sigma.\lambda\tau.\mathit{cmp}_0(\rho)(\sigma)(\tau) \cup \mathit{cmp}_1(\rho)(\sigma)(\tau)\\
&\mathit{mzero}^{\mathit{Conc}} : \forall A.\mathit{Conc}(A)\\
&\mathit{mzero}^{\mathit{Conc}}(\rho)(\sigma)(\tau) := \emptyset\\
&\mathit{ask}_\rho^{\mathit{Conc}} : \mathit{Conc}(\mathit{Env}_c)\\
&\mathit{ask}_\rho^{\mathit{Conc}}(\rho)(\sigma)(\tau) := \mathit{return}^{\mathit{Conc}}(\rho)(\rho)(\sigma)(\tau)\\
&\mathit{inEnv}_\rho^{\mathit{Conc}} : \forall A.\mathit{Env}^{\mathit{Time}_c} \times \mathit{Conc}(A) \to \mathit{Conc}(A)\\
&\mathit{inEnv}_\rho^{\mathit{Conc}}(\rho, \mathit{cmp})(\_)(\sigma)(\tau) := \mathit{cmp}(\rho)(\sigma)(\tau)\\
&\mathit{get}_\sigma^{\mathit{Conc}} : \mathit{Conc}(\mathit{Store}_c)\\
&\mathit{get}_\sigma^{\mathit{Conc}}(\rho)(\sigma)(\tau) := \mathit{return}^{\mathit{Conc}}(\sigma)(\rho)(\sigma)(\tau)\\
&\mathit{put}_\sigma^{\mathit{Conc}} : \mathit{Store}_c \to m(1)\\
&\mathit{put}_\sigma^{\mathit{Conc}}(\sigma)(\rho)(\_)(\tau) := \mathit{return}^{\mathit{Conc}}(\langle\rangle)(\rho)(\sigma)(\tau)\\
&\mathit{ask}_\tau^{\mathit{Conc}} : \mathit{Conc}(\mathit{Time}_c)\\
&\mathit{ask}_\tau^{\mathit{Conc}}(\rho)(\sigma)(\tau) := \mathit{return}^{\mathit{Conc}}(\tau)(\rho)(\sigma)(\tau)\\
&\mathit{inEnv}_\tau^{\mathit{Conc}} : \forall A.\mathit{Time}_c \times \mathit{Conc}(A) \to \mathit{Conc}(A)\\
&\mathit{inEnv}_\tau^{\mathit{Conc}}(\tau, \mathit{cmp})(\rho)(\sigma)(\_) := \mathit{cmp}(\rho)(\sigma)(\tau)
\end{aligned}
$$

Fig. 2. The concrete monad for the standard semantics

with each value, treating it path-sensitively. The concrete semantics is deterministic, with observations on values producing at most one value, so each result set is in fact singleton or empty, though the definitions are able to handle the more general case.

Lemma 1. *The instance for Conc satisfies the monad, nondeterminism, reader, and state laws.*

The result follows exactly from the definitions of the instance.

Instantiated with $Conc$, the type of the interpreter $eval^{Conc}$ after receiving its open-recursive argument is

$$\mathbf{Exp} \to Conc(Val_c) := \mathbf{Exp} \to Env_c \to Store_c \to Time_c \to Result_c.$$

In particular, it maps an expression to an effectful computation producing a result. In anticipation of our framework, we have in contrast defined the program's meaning in terms of configurations and results. To bridge between these different notions of meaning, we define in Figure 3 a pair of functions γ^{Conc} and α^{Conc} which translate to and from the former notion to the latter, respectively. This figure also defines $interp^{Conc}$ to evaluate a program pr. using the

$$\begin{aligned}
&\gamma^{Conc} : (\mathbf{Exp} \to Conc(Val_c)) \to Config_c \to Result_c \\
&\gamma^{Conc}(eval)(e, \rho, \sigma, \tau) := eval(e)(\rho)(\sigma)(\tau) \\
&\alpha^{Conc} : (Config_c \to Result_c) \to \mathbf{Exp} \to Conc(Val_c) \\
&\alpha^{Conc}(exec)(e)(\rho)(\sigma)(\tau) := exec(e, \rho, \sigma, \tau) \\
&interp^{Conc} : \mathbf{Exp} \to Result_c \\
&interp^{Conc}(pr) := \gamma^{Conc}(fix(eval^{Conc}))(\mathcal{I}(pr))
\end{aligned}$$

Fig. 3. Semantic bridge functions for the standard semantics

operator $fix : \forall A.\forall B.((A \to B) \to (A \to B)) \to A \to B$ which yields the fixed point of an argument function of the appropriate type and the program injector $\mathcal{I}$ which injects a program pr into an initial configuration $(pr, \rho_0, \sigma_0, \tau_0)$. $interp^{Conc}$ merely wraps the fixed point of the evaluator—becoming an uninstrumented interpreter—with γ^{Conc} to unpack the incoming configuration and package the produced result.

1.2 Collecting Semantics

Whereas the standard semantics provides a summary of (only) the initial configuration's evaluation, in terms of its resulting value and store, a collecting semantics produces a summary of *each* configuration's evaluation in those same terms. These summaries are recorded in a cache $Cache_c := Config_c \mapsto Result_c$.

To produce this cache, we instrument the evaluator by composing it with a function which intercepts and records the evaluator's results before passing them on to its context. The composed function records the results into a cache specific to each evaluation path and which is conglomerated using a monadic writer effect.

We first define a type operator Coll_c whose sole responsibility is to manage the cache, which it does via a writer effect.

$$\begin{aligned}&\mathit{Coll}_c : \mathit{Type} \rightarrow \mathit{Type}\\&\mathit{Coll}_c := \forall A.A \times \mathit{Cache}_c\end{aligned}$$

In terms of Coll_c, we define the type operator Coll_e to parameterize the evaluation function.[1]

$$\begin{aligned}&\mathit{Coll}_e : \mathit{Type} \rightarrow \mathit{Type}\\&\mathit{Coll}_e := \forall A.\mathit{Env}_c \rightarrow \mathit{Store}_c \rightarrow \mathit{Time}_c \rightarrow \mathit{Coll}_c(\mathcal{P}(A \times \mathit{Store}_c))\end{aligned}$$

Figure 4 presents the monad instance for Coll_c and Coll_e to support the collecting semantics. In each monadic operation, the cache acts as a log. When

$$\begin{aligned}&\mathit{Coll}_c : \mathit{Type} \rightarrow \mathit{Type}\\
&\mathit{Coll}_c := \forall A.A \times \mathit{Cache}_c\\
&\mathit{Coll}_e : \mathit{Type} \rightarrow \mathit{Type}\\
&\mathit{Coll}_e := \forall A.\mathit{Env}_c \rightarrow \mathit{Store}_c \rightarrow \mathit{Time}_c \rightarrow \mathit{Coll}_c(\mathcal{P}(A \times \mathit{Store}_c))\\
&\mathit{return}^{\mathit{Coll}_e} : \forall A.A \rightarrow \mathit{Coll}_e(A)\\
&\mathit{return}^{\mathit{Coll}_e}(x)(\rho)(\sigma)(\tau) := (\{(x,\sigma)\}, \bot)\\
&\mathit{bind}^{\mathit{Coll}_e} : \forall A.\forall B.\mathit{Coll}_e(A) \rightarrow (A \rightarrow \mathit{Coll}_e(B)) \rightarrow \mathit{Coll}_e(B)\\
&\mathit{bind}^{\mathit{Coll}_e}(\mathit{cmp})(f)(\rho)(\sigma)(\tau) := (\emptyset, \$) \sqcup f(x_1)(\rho)(\sigma_1)(\tau) \sqcup \cdots \sqcup f(x_n)(\rho)(\sigma_n)(\tau)\\
&\qquad\qquad\qquad\qquad\text{where } (\{(x_1,\sigma_1),\ldots,(x_n,\sigma_n)\}, \$) := \mathit{cmp}(\rho)(\sigma)(\tau)\\
&\cdot\langle+\rangle\cdot : \forall A.\mathit{Coll}_e(A) \times \mathit{Coll}_e(A) \rightarrow \mathit{Coll}_e(A)\\
&\mathit{cmp}_0\langle+\rangle\mathit{cmp}_1 := \lambda\rho.\lambda\sigma.\lambda\tau.\mathit{cmp}_0(\rho)(\sigma)(\tau) \sqcup \mathit{cmp}_1(\rho)(\sigma)(\tau)\\
&\mathit{mzero}^{\mathit{Coll}_e} : \forall A.\mathit{Coll}_e(A)\\
&\mathit{mzero}^{\mathit{Coll}_e}(\rho)(\sigma)(\tau) := (\emptyset, \bot)\end{aligned}$$

Fig. 4. The concrete monad for the collecting semantics

monadic computations are sequenced or occur nondeterministically, their logs

[1] It is possible and even convenient to define Coll_c in terms of monad transformers. We refrain from doing so to reduce the amount of machinery and to make the raw types more legible.

are joined. We omit the definitions for the reader and state effects, which are defined analogously to *Conc*.

Lemma 2. *The instance for $Coll_e$ satisfies the monad, nondeterminism, reader, and state, and writer laws.*

The result follows exactly from the definitions of the instance.

The join-semilattice instance for $Cache_c$ is defined

$$\bot_{Cache_c} := \lambda\varsigma.\emptyset \qquad\qquad \$_0 \sqcup_{Cache_c} \$_1 := \lambda\varsigma.\$_0(\varsigma) \cup \$_1(\varsigma)$$

where the \$ metavariable indicates a cache. We will omit the subscripts when context makes the reference clear.

These two monads embody the two notions of evaluation from the previous section—one in which configurations are *exec*uted to results and another in which expressions are *eval*uated to values. (The subsequent use of the names *eval* and *exec* will be consistent with this distinction.) As with *Conc*, the functions α^{Coll} and γ^{Coll} defined in Figure 5 convert between these notions. The $exec^{Coll_c}_{cache}$

$$\begin{aligned}
&\gamma^{Coll_e} : (\mathbf{Exp} \to Coll_c(Val)) \to Config \to Coll_c(Result_c)\\
&\gamma^{Coll_e}(eval)(e,\rho,\sigma,\tau) := eval(e)(\rho)(\sigma)(\tau)\\
&\alpha^{Coll_e} : (Config \to Coll_c(Result_c)) \to \mathbf{Exp} \to Coll_e(Val_c)\\
&\alpha^{Coll_e}(exec)(e)(\rho)(\sigma)(\tau) := exec(e,\rho,\sigma,\tau)\\
&exec^{Coll_c}_{cache} : (Config_c \to Coll_c(Result_c)) \to Config_c \to Coll_c(Result_c)\\
&exec^{Coll_c}_{cache}(exec)(\varsigma) := (r, \$ \sqcup [\varsigma \mapsto r]) \text{ where } (r,\$) := exec(\varsigma)\\
&exec^{Coll_c} : Config_c \to Coll_c(Result_c)\\
&exec^{Coll_c} := fix(\lambda exec.exec^{Coll_c}_{cache}(\gamma^{Coll_e}(eval^{Coll_e}(\alpha^{Coll_e}(exec)))))\\
&interp : \mathbf{Exp} \to Cache_c\\
&interp(pr) := \$ \text{ where } (r,\$) := exec^{Coll_c}(\mathcal{I}(pr))
\end{aligned}$$

Fig. 5. Semantic bridge functions, caching function, execution function, and top-level interpretation function for the collecting semantics

function executes a given configuration by way of its open-recursive argument, intercepts the result and cache, and produces the result and cache joined with a cache that associates the result to the configuration. The definition of the $exec^{Coll_c}$ function composes this caching function and the evaluation function $eval^{Coll_e}$, mediated by α^{Coll} and γ^{Coll}, which, after taking the fixed point, yields an execution function that produces a result and cache. The top-level interpretation function invokes $exec^{Coll_c}$ on the initial configuration and produces the cache, discarding the result.

Given that it recovers the concrete semantics, this interpretation function is not computable: it will diverge if handed a diverging program. To obtain a computable (and sound) account of evaluation, we now turn to recovering the abstract semantics.

2 Abstract Semantics Monad and Effects

Figure 6 presents the monad instance for Full_{eval} in terms of that for *Full* to support the abstract semantics. The result follows exactly from the definitions of the instance.

Figure 7 defines accessors to configuration components, such as environments and stores, in terms of the monadic reader and state effects.

3 Example Analysis

Figure 9 contains an analysis of the program in Figure 8 in which the store is treated flow-sensitively. We present a 0CFA analysis in which the address space is simply the space of program variables. In this setting, environments are not informative (as they map each variable to itself) so we omit them. Thus, the control-sensitive quantities are instantiated as follows.

$$U = 1 \qquad V = 1 \qquad W = \widehat{\mathit{Store}} \qquad Y = 1$$

Under these instantiations, *Control* is isomorphic to **Exp**. Within the analysis, expressions are identified by line numbers except for *then* and *else* clauses which do not occupy their own line and are identified by line numbers and a clause indication. This analysis subjects N to a sign abstraction and x and y to a powerset abstraction. The presentation proceeds by iteration. At iteration 1, the analysis assumes that the program is reachable and that N has any value.

References

1. Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: Chakravarty, M.M.T., Hu, Z., Danvy, O. (eds.) Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011. pp. 2–14. ACM (2011). https://doi.org/10.1145/2034773.2034777, https://doi.org/10.1145/2034773.2034777

$$\begin{aligned}
&\mathit{Full}_{eval} : \mathit{Type} \to \mathit{Type}\\
&\mathit{Full}_{eval}(a) := U \to V \to \mathit{Full}(a \times V)
\end{aligned}$$

$$\begin{aligned}
&\mathit{return}^{\mathit{Full}_{eval}} : \forall A.A \to \mathit{Full}_{eval}(A)\\
&\mathit{return}^{\mathit{Full}_{eval}}(x)(u)(v) := \mathit{return}^{\mathit{Full}}(x, v)\\
&\mathit{bind}^{\mathit{Full}_{eval}} : \forall A.\forall B.\mathit{Full}_{eval}(A) \to (A \to \mathit{Full}_{eval}(B)) \to \mathit{Full}_{eval}(B)\\
&\mathit{bind}^{\mathit{Full}_{eval}}(\mathit{cmp})(f)(u)(v_0) := \mathit{bind}^{\mathit{Full}}(\mathit{cmp}(u)(v_1))(\lambda(x, v_1).f(x)(u)(v_1))\\
&\_\langle+\rangle^{\mathit{Full}_{eval}}\_ : \forall A.\mathit{Full}_{eval}(A) \times \mathit{Full}_{eval}(A) \to \mathit{Full}_{eval}(A)\\
&\mathit{cmp}_0\langle+\rangle^{\mathit{Full}_{eval}}\mathit{cmp}_1 := \lambda u.\lambda v.\mathit{cmp}_0(u)(v) \sqcup \mathit{cmp}_1(u)(v)\\
&\mathit{mzero}^{\mathit{Full}_{eval}} : \forall A.\mathit{Full}_{eval}(u)\\
&\mathit{mzero}(u)(v) := \mathit{mzero}^{\mathit{Full}}\\
&\mathit{ask}_U^{\mathit{Full}_{eval}} : \mathit{Full}_{eval}(U)\\
&\mathit{ask}_U^{\mathit{Full}_{eval}}(u)(v) := \mathit{return}^{\mathit{Full}}(u, v)\\
&\mathit{inEnv}_U^{\mathit{Full}_{eval}} : \forall A.U \times \mathit{Full}_{eval}(A) \to \mathit{Full}_{eval}(A)\\
&\mathit{inEnv}_U^{\mathit{Full}_{eval}}(u, \mathit{cmp})(\_)(v) := \mathit{cmp}(u)(v)\\
&\mathit{get}_V^{\mathit{Full}_{eval}} : \mathit{Full}_{eval}(V)\\
&\mathit{get}_V^{\mathit{Full}_{eval}}(u)(v) := \mathit{return}^{\mathit{Full}}(v, v)\\
&\mathit{put}_V^{\mathit{Full}_{eval}} : V \to \mathit{Full}_{eval}(1)\\
&\mathit{put}_V^{\mathit{Full}_{eval}}(v)(u)(\_) := \mathit{return}^{\mathit{Full}_{eval}}(\langle\rangle)(u)(v)\\
&\mathit{get}_W^{\mathit{Full}_{eval}} : \mathit{Full}_{eval}(W)\\
&\mathit{get}_W^{\mathit{Full}_{eval}}(u)(v) := \mathit{bind}^{\mathit{Full}}(\mathit{get}_C^{\mathit{Full}})(\lambda w.\mathit{return}^{\mathit{Full}}(w, v))\\
&\mathit{put}_W^{\mathit{Full}_{eval}} : W \to \mathit{Full}_{eval}(1)\\
&\mathit{put}_W^{\mathit{Full}_{eval}}(c)(u)(v) := \mathit{put}_C^{\mathit{Full}}(c); \mathit{return}^{\mathit{Full}}(\langle\rangle, v)\\
&\mathit{get}_Y^{\mathit{Full}_{eval}} : \mathit{Full}_{eval}(Y)\\
&\mathit{get}_Y^{\mathit{Full}_{eval}}(u)(v) := \mathit{bind}^{\mathit{Full}}(\mathit{get}_Y^{\mathit{Full}})(\lambda y.\mathit{return}^{\mathit{Full}}(y, v))\\
&\mathit{put}_Y^{\mathit{Full}_{eval}} : Y \to \mathit{Full}_{eval}(1)\\
&\mathit{put}_Y^{\mathit{Full}_{eval}}(y)(u)(v) := \mathit{put}_Y^{\mathit{Full}}(y); \mathit{return}^{\mathit{Full}}(\langle\rangle, v)
\end{aligned}$$

Each monadic definition focuses on distributing the per-path component u and threading the state component v appropriately.

Fig. 6. The monad instance for Full_{eval} with reader, state, and nondeterminism effects

$$
\begin{aligned}
&\mathit{ask}_{\rho}^{\mathit{Full}_{eval}} : \mathit{Full}_{eval}(\widehat{\mathit{Env}})\\
&\mathit{ask}_{\rho}^{\mathit{Full}_{eval}} := \mathbf{do}\,(\rho, \_) \leftarrow \mathit{ask}_{U}^{\mathit{Full}_{eval}}; \mathit{return}^{\mathit{Full}_{eval}}(\rho)\\
&\mathit{inEnv}_{\rho}^{\mathit{Full}_{eval}} : \forall A.\widehat{\mathit{Env}} \times \mathit{Full}_{eval}(A) \rightarrow \mathit{Full}_{eval}(A)\\
&\mathit{inEnv}_{\rho}^{\mathit{Full}_{eval}}(\rho, \mathit{cmp}) := \mathbf{do}\,\tau \leftarrow \mathit{get}_{\tau}^{\mathit{Full}_{eval}}; \mathit{inEnv}_{U}^{\mathit{Full}_{eval}}((\rho, \tau), \mathit{cmp})\\
&\mathit{ask}_{\tau}^{\mathit{Full}_{eval}} : \mathit{Full}_{eval}(\widehat{\mathit{Time}})\\
&\mathit{ask}_{\tau}^{\mathit{Full}_{eval}} := \mathbf{do}\,(\_, \tau) \leftarrow \mathit{ask}_{U}^{\mathit{Full}_{eval}}; \mathit{return}^{\mathit{Full}_{eval}}(\tau)\\
&\mathit{inEnv}_{\tau}^{\mathit{Full}_{eval}} : \forall A.\widehat{\mathit{Time}} \times \mathit{Full}_{eval}(A) \rightarrow \mathit{Full}_{eval}(A)\\
&\mathit{inEnv}_{\tau}^{\mathit{Full}_{eval}}(\tau, \mathit{cmp}) := \mathbf{do}\,\rho \leftarrow \mathit{get}_{\rho}^{\mathit{Full}_{eval}}; \mathit{inEnv}_{U}^{\mathit{Full}_{eval}}((\rho, \tau), \mathit{cmp})\\
&\mathit{get}_{\sigma}^{\mathit{Full}_{eval}} : \mathit{Full}_{eval}(\widehat{\mathit{Store}})\\
&\mathit{get}_{\sigma}^{\mathit{Full}_{eval}} := \mathit{get}_{V}^{\mathit{Full}_{eval}}\\
&\mathit{put}_{\sigma}^{\mathit{Full}_{eval}} : \widehat{\mathit{Store}} \rightarrow \mathit{Full}_{eval}(1)\\
&\mathit{put}_{\sigma}^{\mathit{Full}_{eval}} := \mathit{put}_{V}^{\mathit{Full}_{eval}}
\end{aligned}
$$

Fig. 7. Accessors for configuration components

```
1: let x :=                        |    in
2:    if0(N){                      |5:    let y :=
3:       if0(N){1} else {2}        |6:       if0(N){5} else {6}
      } else {                     |      in
4:       if0(N){3} else {4}        |7:       exit(x, y)
      }                            |
```

Fig. 8. A program to illustrate path and flow sensitivity

#	Reachable, Flow-sensitive, Results
1	$R_0 := \{1\}$, $\$^w_0 := [1 \mapsto [\mathbf{N}_{-,0,+} := N \mapsto \{-,0,+\}]]$, $\$_0 := \bot$
2	$R_1 := R_0 \cup \{2\}$, $\$^w_1 := \$^w_0[2 \mapsto [\mathbf{N}_{-,0,+}]]$, $\$_0$
3	$R_2 := R_1 \cup \{3,4\}$, $\$^w_2 := \$^w_1[3 \mapsto [\mathbf{N}_0 := N \mapsto \{0\}], 4 \mapsto [\mathbf{N}_{-,+} := N \mapsto \{-,+\}]]$, $\$_0$
4	$R_3 := R_2 \cup \{3.\mathit{then}, 4.\mathit{else}\}$, $\$^w_3 := \$^w_2[3.\mathit{then} \mapsto [\mathbf{N}_0], 4.\mathit{else} \mapsto [\mathbf{N}_{-,+}]]$, $\$_0$
5	R_3, $\$^w_3$, $\$_1 := \$_0[3.\mathit{then} \mapsto (\{1\}, [\mathbf{N}_0]), 4.\mathit{else} \mapsto (\{4\}, [\mathbf{N}_{-,+}])]$
6	R_3, $\$^w_3$, $\$_2 := \$_1[3 \mapsto (\{1\}, [\mathbf{N}_0]), 4 \mapsto (\{4\}, [\mathbf{N}_{-,+}])]$
7	R_3, $\$^w_3$, $\$_3 := \$_2[2 \mapsto (\{1,4\}, [\mathbf{N}_{-,0,+}])]$
8	$R_4 := R_3 \cup \{5\}$, $\$^w_4 := \$^w_3[5 \mapsto [\mathbf{x} := x \mapsto \{1,4\}, \mathbf{N}_{-,0,+}]]$, $\$_3$
9	$R_5 := R_4 \cup \{6\}$, $\$^w_5 := \$^w_4[6 \mapsto [\mathbf{x}, \mathbf{N}_{-,0,+}]]$, $\$_3$
10	$R_5 := R_4 \cup \{6.\mathit{then}, 6.\mathit{else}\}$, $\$^w_6 := \$^w_5[6.\mathit{then} \mapsto [\mathbf{x}, \mathbf{N}_0], 6.\mathit{else} \mapsto [\mathbf{x}, \mathbf{N}_{-,+}]]$, $\$_3$
11	R_5 $\$^w_6$, $\$_4 := \$_3[6.\mathit{then} \mapsto (\{5\}, [\mathbf{x}, \mathbf{N}_0]), 6.\mathit{else} \mapsto (\{6\}, [\mathbf{x}, \mathbf{N}_{-,+}])]]$
12	R_5 $\$^w_6$, $\$_5 := \$_4[6 \mapsto (\{5,6\}, [\mathbf{x}, \mathbf{N}_{-,0,+}])]$
13	$R_6 := R_5 \cup \{7\}$ $\$^w_7 := \$^w_6[7 \mapsto [\mathbf{x}, \mathbf{y} := y \mapsto \{5,6\}, \mathbf{N}_{-,0,+}]]$, $\$_5$
14	R_6 $\$^w_7$, $\$_6 := \$_5[7 \mapsto (\{(1,5),(1,6),(4,5),(4,6)\}, [\mathbf{x}, \mathbf{y}, \mathbf{N}_{-,0,+}])]$
15	R_6 $\$^w_7$, $\$_6 := \$_6[5 \mapsto (\{(1,5),(1,6),(4,5),(4,6)\}, [\mathbf{x}, \mathbf{y}, \mathbf{N}_{-,0,+}])]$
16	R_6 $\$^w_7$, $\$_7 := \$_6[1 \mapsto (\{(1,5),(1,6),(4,5),(4,6)\}, [\mathbf{x}, \mathbf{y}, \mathbf{N}_{-,0,+}])]$

Fig. 9. A 0CFA analysis of the example program with a flow-sensitive store