An action semantics based on two combinators

Kyung-Goo Doh[⋆] and David A. Schmidt[⋆⋆]

Hanyang University, Ansan, SOUTH KOREA
Kansas State University, Manhattan, Kansas, USA

Abstract. We propose a naive version of action semantics that begins with a selection of "transient" and "persistent" facets, each characterized as a partial monoid. Yielders are defined as operations on the monoids' values, and actions extract values from the facets, give them to yielders, and place the results into facet output. Actions are composed with a primary combinator, andthen, which can be specialized for multiple facet flows, and the choice combinator, or. Using big-step-style deduction rules, we give the semantics of yielders and actions, and we introduce a weakening rule and a strengthening rule, which let us compose actions with different facet domain-codomains. We also introduce *Mosses abstraction*, a lambda-abstraction variant that improves the readability of action-semantics definitions. Finally, we exploit the subsort (subtype) structure within Mosses's unified algebras to use the deduction rules as both a typing definition as well as a semantics definition. Partial evaluation techniques are applied to type check and compile programs.

1 Introduction

Peter Mosses developed action semantics [9–13, 15, 16] as an antidote to the complexity of denotational semantics definitions, which use lambda-abstraction and application to model all possible language features and definitional styles. This leads to problems like the complete rewrite of a definition when moving from "direct style" to "continuation style" [14, 20].

The key innovation within action semantics is the *facet* — an "active" semantic domain, a kind of value stream, that connects one action to another. A typical imperative language has a facet that represents the flow of temporaries, a facet that represents binding (environment/symbol-table) flow, and a facet that portrays the threading of primary store. Facets are analogous to Strachey's characteristic domains for denotational semantics [21].

Actions operate on facets. An action is a "computational step," a kind of state-transition function. Actions consume facet values and produce facet values. (In the case of the store, an action updates the store and passes it to the

[⋆] doh@hanyang.ac.kr. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / Korea Science and Engineering Foundation(KOSEF), R11-2008-007-01003-0

[⋆⋆] das@ksu.edu. Supported by NSF ITR-0326577.

next action). A language's action set defines the language's computational capabilities. When one action is composed with another, the facet flows must be directed to make one action affect another; combinators are used to direct the flows. *Yielders* are the "noun phrases" within actions, evaluating to the data values that are computed upon by the actions.

The end result is an expressive, high-level, data-flow-like semantics. It is simplistic to describe action semantics as a typed combinatory logic, but similarities do exist.

It is enlightening to review the evolution of action semantics from Mosses's first papers on abstract semantic algebras [9–11] to unified algebras [12] to "linguistic-style" action semantics [13, 15, 16, 22]. In Mosses's early papers, one sees algebraic laws for infix combinators (; , ! [], etc.) that define flow of individual facets, as well as precisely stated combinations that direct multiple facet flows. The combinators evolved into a technical English of conjunctions (and, then, hence, tentatively, furthermore, before, etc.) that express a variety of flows of facet combinations [2, 13, 16].

Although the end result is a powerful, general-purpose, language-semantics toolkit, there is value in having a "lightweight" version of action semantics for tutorial purposes. In this paper, we propose a naive version of action semantics that begins with a selection of "transient" and "persistent" facets, each characterized as a partial monoid. Yielders are defined as operations on the monoids' values, and actions extract values from the facets, give them to yielders, assemble results, and place the results into facet output. Actions are composed with a primary combinator, andthen, which can be specialized for multiple facet flows. There is the choice combinator, or.

Using big-step-style deduction rules, we give the semantics of yielders and actions, and we introduce a weakening rule and a strengthening rule, which let us compose actions with different facet domain-codomains. We also reintroduce Mossses-style lambda-abstraction, which we call *Mosses abstraction*, as a readability aid [9]. Finally, we exploit the subsort (subtype) structure within Mosses's unified algebras [12] to use the deduction rules as both a typing definition as well as a semantics definition. Partial evaluation techniques [1] are applied to type check and compile programs.

2 Facets

A facet is a collection of data values, a Strachey-like characteristic domain [21], that are handled similarly, especially with respect to computational lifetime.

Formally defined, a *facet* is a (partial) monoid, that is, a set of values, S, a (partial) associative operation, $\circ : S \times S \rightarrow S$, and an identity element from S. Composition $\circ$ defines how to combine or "glue" two values.

A facet is classified as *transient* or *persistent*, based on the lifetime (*extent*) of its values. A transient facet's values are short-lived and are produced, copied, and consumed during a program's computation; a persistent facet's are longlived, fixed, data structures that are referenced and updated (and not produced, copied, and consumed).

Here are the facets we employ in this paper. First, we define by induction these sets of basic values:

$$\begin{array}{ll}
\mathsf{Identifier} = \text{identifiers} & \mathsf{F} = \mathsf{List}(\mathsf{Transient}) \\
\mathsf{Action} = \text{text of actions} & \mathsf{Transient} = \mathsf{F} \cup \mathsf{D} \cup \mathsf{Expressible} \cup \mathsf{Closure} \\
\mathsf{Cell} = \text{storage locations} & \mathsf{Closure} = \mathsf{Set}(\mathsf{Transient}) \times \mathsf{Identifier} \times \mathsf{Action} \\
\mathsf{Int} = \text{integers} & \mathsf{D} = \mathsf{Set}(\mathsf{Identifier} \times \mathsf{Denotable}) \\
\mathsf{Expressible} = \mathsf{Cell} \cup \mathsf{Int} & \mathsf{Denotable} = \mathsf{Transient} \\
 & \mathsf{I} = \mathsf{Cell} \to \mathsf{Storable} \\
 & \mathsf{Storable} = \mathsf{Int}
\end{array}$$

The facets in this paper use the basic-value sets:

1. *functional facet:* The monoid of most-transient values is written $\mathcal{F} = (\mathsf{F}, :, \langle\,\rangle)$. The value set is sequences (lists) of transients, e.g., $\langle 2, \mathit{cell99}, \{(\mathtt{x}, \mathit{cell99})\}\rangle$ is a three-transient sequence. Composition, :, is sequence append, and identity is the empty sequence, $\langle\,\rangle$. Functional-facet values have a brief extent, and the facet is a transient facet.
2. *declarative facet:* Sets of bindings are modelled by the monoid, $\mathcal{D} = (\mathsf{D}, +, \{\,\})$; The value set consists of finite sets of pairs, $\rho = \{(I_0, n_0), (I_1, n_1), \cdots (I_m, n_m), \cdots\}$, where each such set defines a function (that is, each I_j is distinct). Composition is binding override: For values ρ_1 and ρ_2, we define $\rho_1 + \rho_2 = \rho_2 \cup \{(I_j = n_j) \in \rho_1 \mid I_j \notin \mathit{domain}(\rho_2)\}$ — ρ_2's bindings take precedence over ρ_1's. Identity is the empty set of bindings. Bindings are readily generated and copied — the facet is a transient facet.
3. *imperative facet:* Stores belong to the monoid, $\mathcal{I} = (\mathsf{I}, *, [\,])$. The value set consists of finite functions, $\sigma = [\ell_0 \mapsto n_0,\ \ell_1 \mapsto n_1,\ \cdots,\ \ell_k \mapsto n_k]$, each $\ell_i \in \mathsf{Cell}$ and $n_i \in \mathsf{Storable}$. Composition, $\sigma_1 * \sigma_2$, is function union (provided that the functions' domains are disjoint) — a partial operation. The identity is the empty map. Since the imperative facet denotes persistent store, the facet is persistent.

A computation step (action) may require values from more than one facet, so we define a compound facet as a monoid of finite sets of facet elements, at most one element per facet: For *distinct* facets, $\mathcal{F} = (F, \circ_F, id_F)$ and $\mathcal{G} = (G, \circ_G, id_G)$, define the compound facet as

$$\begin{array}{l}
\mathcal{FG} = (\{\{f, g\} \mid f \in F, g \in G\},\ \circ_{FG},\ \{id_F, id_G\}) \\
\qquad\qquad \text{where } \{f_1, g_1\} \circ_{FG} \{f_2, g_2\} = \{f_1 \circ_F f_2,\ g_1 \circ_G g_2\}
\end{array}$$

That is, a compound-facet value is a set of singleton-facet values, and composition is applied on the individual values in the respective sets based on facet affiliation. The construction allows compound facets like $\mathcal{DI}$ but not $\mathcal{FFDID}$; the latter case must be folded into $\mathcal{FDI}$ by using the respective composition operators for $\mathcal{F}$ and $\mathcal{D}$.

The "empty" compound facet is the *basic* ("control" [13]) facet, and it is the one-element monoid, $\mathcal{B}$. We use Γ, Δ, Σ to stand for compound facet values.

We can embed a (compound) facet value into a larger compound facet by adding identity values: For element $f = \{f_0, f_1, ..., f_m\} \in \mathcal{F}_0\mathcal{F}_1 \cdots \mathcal{F}_m$, we embed f into $\mathcal{F}_0\mathcal{F}_1 \cdots \mathcal{F}_m\mathcal{G}_0\mathcal{G}_1 \cdots \mathcal{G}_n$ as $f \cup \{id_{G_0}, id_{G_1}, ..., id_{G_n}\}$.

Embedding is a technical device that lets us compose any two facet values together: For f and g, we define $f \circ g$ by unioning their facet domains, embedding each of f and g into the unioned-facet domain, and composing the embedded elements in the unioned monoid.

We define a "strict union" of two (compound) facet values as this partial function: for $\{f_1, ..., f_m, g_1, ..., g_n\} \in \mathcal{F}_1 \cdots \mathcal{F}_m\mathcal{G}_1 \cdots \mathcal{G}_n$ and $\{f_1, ..., f_m, h_1, ..., h_p\} \in \mathcal{F}_1 \cdots \mathcal{F}_m\mathcal{H}_1 \cdots \mathcal{H}_p$,

$$\{f_1, ..., f_m, g_1, ..., g_n\} \cup \{f_1, ..., f_m, h_1, ..., h_p\} = \{f_1, ..., f_m, g_1, ..., g_n, h_1, ..., h_p\}$$

That is, two facet values are unioned only if they agree on the values of their shared facets. In a similar manner, we define "facet restriction" and "facet subtraction":

$$\{f_1, ..., f_m, g_1, ..., g_n\} \downarrow_{\mathcal{F}_1 \cdots \mathcal{F}_m\mathcal{H}_1 \cdots \mathcal{H}_p} = \{f_1, ..., f_m\}$$
$$\{f_1, ..., f_m, g_1, ..., g_n\} \downarrow_{\sim\mathcal{F}_1 \cdots \mathcal{F}_m\mathcal{H}_1 \cdots \mathcal{H}_p} = \{g_1, ..., g_n\}$$

3 Yielders

An action semantics requires operations on values carried within a facet and operations on the facets themselves. The former are called *yielders* and the latter are called *actions*.[1] Yielders are embedded within actions in a semantics definition; for this reason, they compute on transients.

Yielders are interesting because their arguments can often be computed at earlier binding times (compile-time, link-time) than run-time. Type checking, constant folding, and partial evaluation can be profitably applied to yielders, as we investigate in Section 11.

In our naive version of action semantics, we define yielders via big-step operational-semantics rules. Figure 1 presents a sample collection. Within a rule, read the configuration, $\Gamma \vdash \mathsf{y} : \Delta$, as stating, "yielder y consumes inputs Γ to produce outputs Δ."

For example, (add (find x) it) is a yielder that adds the value bound to x in the declarative facet to the incoming transient in the functional facet. One possible derivation, for the functional, declarative facets, $\langle n_1 \rangle, \{(\mathtt{x}, n_0), (\mathtt{z}, n_2)\}$, goes as follows:

$$\frac{\{(\mathtt{x}, n_0), (\mathtt{z}, n_2)\} \vdash \mathsf{find}\ \mathtt{x} : n_0 \qquad \langle n_1 \rangle \vdash \mathsf{it} : n_1}{\{\langle n_1 \rangle, \{(\mathtt{x}, n_0), (\mathtt{z}, n_2)\}\} \vdash \mathsf{add}\ (\mathsf{find}\ \mathtt{x})\ \mathsf{it} : add(n_0, n_1)}$$

[1] In the "linguistic" version of action semantics [13, 16, 22], yielders are "noun phrases" and actions are "verb phrases." But this distinction is not clearcut, e.g., "give (the denotable bound to x)" versus "find x," so we deemphasize this approach.

Functional-facet yielders:

primitive constant: $\dfrac{}{\vdash \mathsf{k} : k}$

n-ary operation (e.g., addition): $\dfrac{\mathit{\Gamma} \vdash \mathsf{y}_1 : \tau_1 \quad \mathit{\Delta} \vdash \mathsf{y}_2 : \tau_2}{\mathit{\Gamma} \cup \mathit{\Delta} \vdash \mathsf{add}\,\mathsf{y}_1\,\mathsf{y}_2 : add(\tau_1, \tau_2)}$

indexing: $\dfrac{1 \leq i \leq m}{\langle \tau_1, \cdots, \tau_m \rangle \vdash \#\mathsf{i} : \tau_i}$ Note: it abbreviates $\#\mathsf{1}$

sort filtering: $\dfrac{\mathit{\Gamma} \vdash \mathsf{y} : \mathit{\Delta} \quad \mathit{\Delta} \leq T}{\mathit{\Gamma} \vdash \mathsf{is}T\,\mathsf{y} : \mathit{\Delta}}$ where $\leq$ is defined in Section 10

Declarative-facet yielders: binding lookup, binding creation, and copy:

$$\frac{(\mathsf{I}, \tau) \in \rho}{\rho \vdash \mathsf{find}\,\mathsf{I} : \tau} \qquad \frac{\mathit{\Gamma} \vdash \mathsf{y} : \tau}{\mathit{\Gamma} \vdash \mathsf{bind}\,\mathsf{I}\,\mathsf{y} : \{(\mathsf{I}, \tau)\}} \qquad \frac{}{\rho \vdash \mathsf{currentbindings} : \rho}$$

Fig. 1. Yielders

$\cup$ combines the input requirements of the component yielders in the consequent sequent.

Note there is a derivation for $\mathsf{is}T\,\mathsf{y}$ exactly when y yields a value that belongs to sort (type) T.

4 Actions

Actions compute on facets in well-defined steps. In particular, actions enumerate the steps taken upon persistent-facet values like stores, databases, and i/o buffers.

There are "structural" actions that hand facet values to yielders and place the yielders' results in facets; there are actions that operate on persistent-facet values; and there are actions that define and apply closures containing action-text. Figure 2 presents a sample action set, whose behaviors are defined with big-step deduction rules. Read $\mathit{\Gamma} \vdash \mathsf{a} \Rightarrow \mathit{\Delta}$ as asserting that action a receives facets $\mathit{\Gamma}$ and produces facets $\mathit{\Delta}$.

Of the structural actions, $\mathsf{give}_{\mathcal{G}}\,\mathsf{y}$ hands yielder y its inputs and places its outputs into the facet stream named by $\mathcal{G}$. $\mathsf{complete}$ is an empty computation step.

Actions lookup, update, and $\mathsf{allocate}$ define computation steps on the persistent facet, $\mathcal{I}$; a store value must be provided as input. The rule for $\mathsf{allocate}$ shows that the action produces a functional-facet value (c) as well as an updated store.

The last two rules in the Figure portray closure construction and application. Assuming that yielder y evaluates as $\mathit{\Gamma} \vdash \mathsf{y} : \mathit{\Delta}$, then $\mathit{\Gamma} \vdash \mathsf{recabstract}_{\mathcal{G}}\ I\ \mathsf{y}\ \mathsf{a} : [\mathit{\Delta}\downarrow_{\mathcal{G}}, I, \mathsf{a}]_{\mathcal{G}}$ yields a closure that holds the $\mathcal{G}$-facet portion of $\mathit{\Delta}$, the closure's name, I, and the unevaluated action, a. Later, $\mathsf{exec}\,\mathsf{y}_1\,\mathsf{y}_2$ evaluates y_1 to the closure, evaluates y_2 to an argument, τ, and evaluates the closure body, a, to $\langle \tau \rangle \circ (\mathit{\Delta} \cup (\mathit{\Gamma}\downarrow_{\sim\mathcal{G}})) \circ \{(I, [\mathit{\Delta}, I, \mathsf{a}]_{\mathcal{G}})\} \vdash \mathsf{a} \Rightarrow \mathit{\Sigma}$, that is, input τ is composed with

Structural actions:

$$\frac{\varGamma \vdash \mathsf{y} : \varDelta \qquad \varDelta \in \mathcal{G}}{\varGamma \vdash \mathsf{give}_{\mathcal{G}}\ \mathsf{y} \Rightarrow \varDelta} \quad \text{where } \mathcal{G} \text{ names the facet that receives value } \varDelta$$

$$\frac{}{\vdash \mathsf{complete} \Rightarrow \mathit{completing}} \quad \text{where } \mathit{completing} \text{ is the sole element in the basic-facet monoid}$$

Imperative-facet actions:

$$\frac{\varGamma \vdash \mathsf{y} : c \qquad c \leq \mathsf{Cell}}{\varGamma \cup \sigma \vdash \mathsf{lookup}\ \mathsf{y} \Rightarrow \langle \sigma(c) \rangle} \qquad \frac{\varGamma_1 \vdash \mathsf{y}_1 : c \qquad c \leq \mathsf{Cell} \qquad \varGamma_2 \vdash \mathsf{y}_2 : \tau \qquad \tau \leq \mathsf{Storable}}{\varGamma_1 \cup \varGamma_2 \cup \sigma \vdash \mathsf{update}\ \mathsf{y}_1\ \mathsf{y}_2 \Rightarrow \sigma[c \mapsto \tau]}$$

$$\frac{c \notin \mathit{domain}(\sigma)}{\sigma \vdash \mathsf{allocate} \Rightarrow \langle c \rangle,\ \sigma * [c \mapsto ?]} \quad \text{Note: the output belongs to compound facet, } \mathcal{FI}$$

Closure yielder and action:

$$\frac{\varGamma \vdash \mathsf{y} : \varDelta}{\varGamma \vdash \mathsf{recabstract}_{\mathcal{G}}\ I\ \mathsf{y}\ \mathsf{a}\ :\ [\varDelta{\downarrow}_{\mathcal{G}}, I, \mathsf{a}]_{\mathcal{G}}} \quad \text{where } \mathcal{G} \text{ names only transient facets}$$

$$\frac{\begin{array}{l} \varGamma_1 \vdash \mathsf{y}_1 : [\varDelta, I, \mathsf{a}]_{\mathcal{G}} \\ \varGamma_2 \vdash \mathsf{y}_2 : \tau \qquad\qquad \langle \tau \rangle \circ (\varDelta \cup (\varGamma{\downarrow}_{\sim\mathcal{G}})) \circ \{(I, [\varDelta, I, \mathsf{a}]_{\mathcal{G}})\} \vdash \mathsf{a} \Rightarrow \varSigma \\ \varGamma = \varGamma_1 \cup \varGamma_2 \end{array}}{\varGamma \vdash \mathsf{exec}\ \mathsf{y}_1\ \mathsf{y}_2 \Rightarrow \varSigma}$$

Note: y_2 is optional.

Fig. 2. actions

weaken-L: $\dfrac{\varGamma \vdash \mathsf{a} \Rightarrow \varDelta}{\varSigma \cup \varGamma \vdash \mathsf{a} \Rightarrow \varDelta}$

strengthen-R: $\dfrac{\varGamma \vdash \mathsf{a} \Rightarrow \varDelta \qquad \varGamma \cup \sigma = \varGamma}{\varGamma \vdash \mathsf{a} \Rightarrow \varDelta \cup \sigma}$ where $\sigma \in \mathcal{I}$

Fig. 3. weakening and strengthening rules

the data, $\varDelta$, saved in the closure along with the facets within $\varGamma$ that are allowed as the inputs to a. Name I rebinds to the same closure for recursive calls.

For example, $\mathsf{abstract}_{\mathcal{D}}\ \mathsf{f}\ \mathsf{currentbindings}\ (\mathsf{give}_{\mathcal{F}}(\mathsf{add}\ (\mathsf{find}\ \mathsf{x})\ \mathsf{it}))$ defines a statically scoped closure, f, that adds `x`'s value in the scope of definition to the argument supplied at the point of application.

Actions are polymorphic depending on the inputs and outputs of their embedded yielders, and we need rules to assemble compound actions. They are listed in Figure 3. The *weaken-L* rule states that an action can consume more facets than what are needed to conduct the action, and no harm occurs. The *strengthen-R* rule states that an action whose input includes a persistent value, σ, passes forwards that value unaltered (provided that the action did not itself alter the value — recall that $\varDelta \cup \sigma$ is "strict union," defined iff $\varDelta$ holds no $\mathcal{I}$-value distinct from σ).

Here is an example. We can derive that

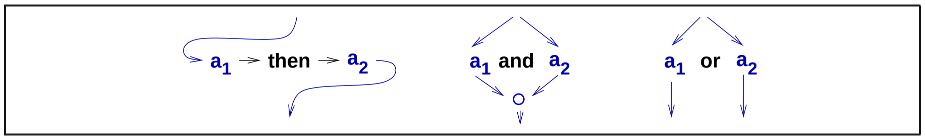


Fig. 4. Facet flows

$$\{(\mathsf{x},2)\} \vdash \mathsf{give_F}(\mathsf{find}\,\mathsf{x}) \Rightarrow 2$$

Using the *weaken-L* rule, we deduce

$$\langle\,\rangle, \{(\mathsf{x},2)\}, \sigma_0 \vdash \mathsf{give_F}(\mathsf{find}\,\mathsf{x}) \Rightarrow 2$$

which shows that the empty sequence of transients and the persistent store do not alter the outcome. The *strengthen-R* rule lets us deduce that the store propagates unaltered:

$$\langle\,\rangle, \{(\mathsf{x},2)\}, \sigma_0 \vdash \mathsf{give_F}(\mathsf{find}\,\mathsf{x}) \Rightarrow 2, \sigma_0$$

5 Combinators

When two actions are composed, there are three possible patterns of a facet's flow: sequential, parallel, and conditional; see Figure 4. Parallel flow finishes by composing the outputs from the two component actions by monoid composition. Conditional flow allows only one action to produce the output.

The flows are modelled by the combinators, then, and, and or, respectively; here are their semantics:

$$\frac{\Gamma \vdash \mathsf{a}_1 \Rightarrow \Delta \quad \Delta \vdash \mathsf{a}_2 \Rightarrow \Sigma}{\Gamma \vdash \mathsf{a}_1\ \mathsf{then}\ \mathsf{a}_2 \Rightarrow \Sigma} \quad \frac{\Gamma \vdash \mathsf{a}_1 \Rightarrow \Delta_1 \quad \Gamma \vdash \mathsf{a}_2 \Rightarrow \Delta_2}{\Gamma \vdash \mathsf{a}_1\ \mathsf{and}\ \mathsf{a}_2 \Rightarrow \Delta_1 \circ \Delta_2} \quad \frac{\Gamma \vdash \mathsf{a}_i \Rightarrow \Delta \quad i \in \{1,2\}}{\Gamma \vdash \mathsf{a}_1\ \mathsf{or}\ \mathsf{a}_2 \Rightarrow \Delta}$$

At this point, we have a kind of combinatory logic for the imperative facet, $\mathcal{I}$: $\mathsf{a}_1\ \mathsf{then}\ \mathsf{a}_2$ defines composition, $a_2(a_1\ \sigma)$; $\mathsf{a}_1\ \mathsf{and}\ \mathsf{a}_2$ defines an S-combinator, $S\,\mathsf{a}_1\,\mathsf{a}_2\,\sigma = \mathsf{a}_1(\sigma) \circ \mathsf{a}_2(\sigma)$; the *L-weaken* rule defines a K-combinator, $(\mathsf{a}\,x)\sigma = \mathsf{a}\,x$; and the *R-strengthen* rule defines an I-combinator, $\mathsf{a}\,\sigma = \sigma$.

When actions consume multiple facets, it is likely that different flows are required for the individual facets. The standard example is command composition, $C_1; C_2$, where the incoming set of bindings (scope/symbol table) is given in parallel to both C_1 and C_2, and the incoming store is threaded sequentially through C_1, which updates it and passes it to C_2. Such concepts are crucial to language semantics, and denotational semantics employs lambda-abstractions to encode such flows; in contrast, action semantics makes the flows primitive and explicit.

Our naive action semantics uses the combinator, $\mathsf{and}_{\mathcal{G}}\mathsf{then}$, where $\mathcal{G}$ denotes the (compound) facet that is passed in parallel (by and) and all other facets are passed sequentially (by then). Here is its definition:

$$\frac{\Gamma \vdash \mathsf{a}_1 \Rightarrow \Delta_1 \qquad (\Gamma{\downarrow_{\mathcal{G}}}) \cup (\Delta_1{\downarrow_{\sim\mathcal{G}}}) \vdash \mathsf{a}_2 \Rightarrow \Delta_2}{\Gamma \vdash \mathsf{a}_1\ \mathsf{and}_{\mathcal{G}}\mathsf{then}\ \mathsf{a}_2 \Rightarrow \Delta_1{\downarrow_{\mathcal{G}}} \circ \Delta_2}$$

Expression: $E ::= \texttt{k} \mid E_1 + E_2 \mid N$

Command: $C ::= N \texttt{:=} E \mid C_1;C_2 \mid \texttt{while}\ E\ \texttt{do}\ C \mid D\ \texttt{in}\ C \mid \texttt{call}\ N(E)$

Declaration: $D ::= \texttt{val}\ I \texttt{=} E \mid \texttt{var}\ I \texttt{=} E \mid \texttt{proc}\ I_1(I_2) \texttt{=} C \mid \texttt{module}\ I \texttt{=} D \mid D_1;D_2$

Name: $N ::= I \mid N.I$

Identifier: I

Fig. 5. Example language syntax

In particular, then abbreviates $\mathsf{and}_{\emptyset}\mathsf{then}$ and and abbreviates $\mathsf{and}_{\mathsf{AllFacets}}\mathsf{then}$. When we omit the subscript and write andthen, we mean $\mathsf{and}_{\mathcal{D}}\mathsf{then}$, that is, only the declarative facet, $\mathcal{D}$, is consumed in parallel.

With andthen and or, we can readily model most mainstream language concepts.

6 Action equations

A language's action semantics is a set of equations, defined inductively on the language's syntax. For the syntax in Figure 5, we define one valuation function for each syntax domain, one equation for each syntactic construction. Each valuation function has an arity that lists the facets that may be consumed and must be produced. For example, expressions are interpreted by the valuation function, $\mathsf{evaluate} : \mathsf{Expression} \to \mathcal{D}\mathcal{I} \to \mathcal{F}$, which indicates that an expression might require the declarative and imperative facets to produce a functional-facet value. Figure 6 shows the action equations for the language described in Figure 5. We follow Mosses-Watt-style action notation, which elides the semantic brackets from single nonterminals, e.g., evaluate E rather than $\mathsf{evaluate}[\![E]\!]$.

Wherever possible, we employ a_1 andthen a_2 to combine actions: the declarative facet flows in parallel to a_1 and a_2 and the store and any temporaries thread from a_1 to a_2. The language is an "andthen"-sequencing language. When we deviate from using andthen in the semantics, this indicates a feature deserving further study. Here are a few general points:

– The self-reference in $\mathsf{execute}[\![\texttt{while}\ E\ \texttt{do}\ C]\!]$ is understood as a lazy, infinite unfolding of the compound action, which has the usual least-fixed-point meaning in a partial ordering of partial, finite, and ω-length phrases [6].
– Although the arity of elaborate states that an action may produce both a set of bindings (in $\mathcal{D}$) and an altered store (in $\mathcal{I}$), not all equations do so (e.g., val and proc). In these latter cases, the *R-strengthening* rule applies, passing the store through, unchanged.
– The *closure* defined in $\mathsf{elaborate}[\![\texttt{proc}\, I_1(I_2)\texttt{=}C]\!]$ embeds the current bindings at the definition point. When applied, the closure's action consumes the actual parameter, it, and the store at the point of call.

Figure 7 shows a derivation of the actions taken by a procedure call.

Here are the situations where combinators other than andthen appear:

$$\mathsf{evaluate} : \mathsf{Expression} \rightarrow \mathcal{DI} \rightarrow \mathcal{F}$$

$$\mathsf{evaluate}[\![\mathtt{k}]\!] = \mathsf{give}_{\mathcal{F}}\ \mathsf{k}$$

$$\mathsf{evaluate}[\![E_1 + E_2]\!] = \begin{array}{l}(\mathsf{evaluate}\ E_1\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{evaluate}\ E_2)\\ \ \mathsf{andthen}\ \mathsf{give}_{\mathcal{F}}(\mathsf{add}\ (\mathsf{isInt}\ \#1)\ (\mathsf{isInt}\ \#2))\end{array}$$

$$\mathsf{evaluate}[\![N]\!] = \mathsf{investigate}\ N\ \mathsf{andthen}\ \begin{array}{l}\mathsf{lookup}\ (\mathsf{isCell}\ \mathsf{it})\\ \mathsf{or}\ \mathsf{give}_{\mathcal{F}}\ (\mathsf{isInt}\ \mathsf{it})\end{array}$$

$$\mathsf{execute} : \mathsf{Command} \rightarrow \mathcal{DI} \rightarrow \mathcal{I}$$

$$\mathsf{execute}[\![N\,\mathtt{:=}E]\!] = \begin{array}{l}(\mathsf{investigate}\ N\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{evaluate}\ E)\\ \ \mathsf{andthen}\ \mathsf{update}(\mathsf{isCell}\ \#1)\ \#2\end{array}$$

$$\mathsf{execute}[\![C_1; C_2]\!] = \mathsf{execute}\ C_1\ \mathsf{andthen}\ \mathsf{execute}\ C_2$$

$$\mathsf{execute}[\![\mathtt{while}\ E\ \mathtt{do}\ C]\!] = \begin{array}{l}\mathsf{evaluate}\ E\ \mathsf{andthen}\\ \ ((\mathsf{give}_{\mathcal{F}}(\mathsf{isZero}\ \mathsf{it})\ \mathsf{andthen}\ \mathsf{complete})\\ \ \mathsf{or}\\ \ (\mathsf{give}_{\mathcal{F}}(\mathsf{isNonZero}\ \mathsf{it})\ \mathsf{andthen}\ \mathsf{execute}\ C\\ \quad \mathsf{andthen}\ \mathsf{execute}\ [\![\mathtt{while}\ E\ \mathtt{do}\ C]\!]))\end{array}$$

$$\mathsf{execute}[\![D\ \mathtt{in}\ C]\!] = (\mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}\ \mathsf{andthen}\ \mathsf{elaborate}\ D)\ \mathsf{then}\ \mathsf{execute}\ C$$

$$\mathsf{execute}[\![\mathtt{call}\ N(E)]\!] = \begin{array}{l}(\mathsf{investigate}\ N\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{evaluate}\ E)\\ \ \mathsf{andthen}\ \mathsf{exec}(\mathsf{isClosure}\ \#1)\ \#2\end{array}$$

$$\mathsf{elaborate} : \mathsf{Declaration} \rightarrow \mathcal{DI} \rightarrow \mathcal{DI}$$

$$\mathsf{elaborate}[\![\mathtt{val}\ I\ \mathtt{=}\ E]\!] = \mathsf{evaluate}\ E\ \mathsf{andthen}\ \mathsf{give}_{\mathcal{D}}\ (\mathsf{bind}\ I\ \mathsf{it})$$

$$\mathsf{elaborate}[\![\mathtt{var}\ I\ \mathtt{=}\ E]\!] = \begin{array}{l}(\mathsf{evaluate}\ E\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{allocate})\\ \ \mathsf{andthen}\ (\mathsf{give}_{\mathcal{D}}\ (\mathsf{bind}\ I\ \#2)\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{update}\ \#2\ \#1)\end{array}$$

$$\mathsf{elaborate}[\![\mathtt{proc}\ I_1(I_2)\ \mathtt{=}\ C]\!] = \mathsf{give}_{\mathcal{D}}(\mathsf{bind}\ I_1\ \mathit{closure})$$

$$\text{where } \mathit{closure} = \mathsf{recabstract}_{\mathcal{D}}\ I_1\ (\mathsf{currentbindings})\ \begin{array}{l}((\mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}\\ \ \ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{give}_{\mathcal{D}}(\mathsf{bind}\ I_2\ \mathsf{it}))\\ \ \mathsf{then}\ \mathsf{execute}\ C)\end{array}$$

$$\mathsf{elaborate}[\![\mathtt{module}\ I\ \mathtt{=}\ D]\!] = \mathsf{elaborate}\ D\ \mathsf{then}\ \mathsf{give}_{\mathcal{D}}(\mathsf{bind}\ I\ \mathsf{currentbindings})$$

$$\mathsf{elaborate}[\![D_1; D_2]\!] = \begin{array}{l}(\mathsf{elaborate}\ D_1\\ \quad \mathsf{then}\ (\mathsf{give}_{\mathcal{F}}\ \mathsf{currentbindings}\ \mathsf{and}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}))\\ \mathsf{andthen}\ ((\mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{it})\\ \qquad \mathsf{then}\ \mathsf{elaborate}\ D_2)\end{array}$$

$$\mathsf{investigate} : \mathsf{Name} \rightarrow \mathcal{D} \rightarrow \mathcal{F}$$

$$\mathsf{investigate}[\![I]\!] = \mathsf{give}_{\mathcal{F}}(\mathsf{find}\ I)$$

$$\mathsf{investigate}[\![N.I]\!] = \mathsf{investigate}\ N\ \mathsf{then}\ \mathsf{give}_{\mathcal{D}}(\mathsf{isD}\ \mathsf{it})\ \mathsf{then}\ \mathsf{give}_{\mathcal{F}}(\mathsf{find}\ I)$$

Fig. 6. Action equations

To make linear the big-step deductions that follow, we use this notation for subgoaling: For big-step rule,

$$\frac{\{\mathit{\Gamma}_i \vdash \mathsf{e}_i \Rightarrow \tau_i\}_{i\in I} \quad \tau = f\{\tau_i\}_{i\in I}}{\mathit{\Gamma} \vdash \mathsf{op}(\mathsf{e_i})_{\mathsf{i\in I}} \Rightarrow \tau}$$

and goal, $\mathit{\Gamma} \vdash \mathsf{op}(\mathsf{e_i})_{\mathsf{i\in I}} \Rightarrow \tau$, we depict the subgoaling and computation of the result in this form:

$$:- \quad f\{\mathit{\Gamma}_i \vdash \mathsf{e}_i \Rightarrow \tau_i\}_{i\in I} = \tau$$

Let $\rho_{\mathsf{xp}} = \{(\mathsf{x}, \ell_0), (\mathsf{p}, closure_p)\}$,
$closure_p = [\{(\mathsf{x}, \ell_0)\}, \mathsf{p}, [\![\mathtt{x} := \mathtt{y}]\!]_{\mathcal{D}}$, and
$\sigma_{\mathsf{x}} = [\ell_0 \mapsto 2]$. This goal,

$$\rho_{\mathsf{xp}}, \sigma_{\mathsf{x}} \vdash \mathsf{execute}[\![\mathtt{call}\ \mathsf{p}(3)]\!] \Rightarrow \sigma_{\mathsf{f}}$$

is solved for σ_{f} as follows:

$= \rho_{\mathsf{xp}}, \sigma_{\mathsf{x}} \vdash (\mathsf{investigate\ p\ and}_{\mathcal{FD}}\mathsf{then\ evaluate\ 3})$
$\qquad\qquad \mathsf{andthen\ exec(isClosure\#1)\ \#2} \Rightarrow \sigma_{\mathsf{f}}$
$:- \quad \rho_{\mathsf{xp}}, \sigma_{\mathsf{x}} \vdash (\mathsf{investigate\ p\ and}_{\mathcal{FD}}\mathsf{then\ evaluate\ 3}) \Rightarrow \tau_1, \sigma_{\mathsf{x}},$
$\quad \tau_1, \rho_{\mathsf{xp}}, \sigma_{\mathsf{x}} \vdash \mathsf{exec(isClosure\#1)\ \#2} \Rightarrow \sigma_2 = \sigma_{\mathsf{f}}$

The first subgoal computes as follows:

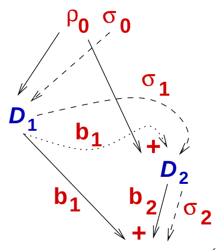
$((\rho_{\mathsf{xp}} \vdash \mathsf{investigate\ p} \Rightarrow closure_p);\ (\rho_{\mathsf{xp}} \vdash \mathsf{evaluate\ 3} \Rightarrow 3)) = \tau_1$
$= \langle closure_p, 3\rangle = \tau_1$

So, the second subgoal proceeds as follows:

$\langle closure_p, 3\rangle, \rho_{\mathsf{xp}}, \sigma_{\mathsf{x}} \vdash \mathsf{exec(isClosure\#1)\ \#2} \Rightarrow \sigma_{\mathsf{f}}$
$:- \quad \rho_{\mathsf{xp}} + \{(\mathsf{y}, 3)\}, \sigma_{\mathsf{x}} \vdash \mathsf{execute}[\![\mathsf{x} :=\ \mathsf{y}]\!] \Rightarrow \sigma_{\mathsf{f}}$
$= \rho_{\mathsf{xp}} + \{(\mathsf{y}, 3)\}, \sigma_{\mathsf{x}} \vdash (\mathsf{investigate\ x\ and}_{\mathcal{FD}}\mathsf{then\ evaluate\ y})$
$\qquad\qquad\qquad \mathsf{andthen\ update(isCell\ \#1)\ \#2} \Rightarrow \sigma_{\mathsf{f}}$
$\cdots = [\ell_0 \mapsto 3] = \sigma_{\mathsf{f}}$

Fig. 7. Derivation of the actions defined by $\mathsf{call\ p(3)}$

- The or used in $\mathsf{evaluate}[\![N]\!]$ gives opportunity to both its clauses to complete; at most one will do so. This is also true for $\mathsf{execute}[\![\mathtt{while}\ E\ \mathtt{do}\ C]\!]$.
- $\mathsf{and}_{\mathcal{FD}}\mathsf{then}$ appears in situations (e.g., $\mathsf{evaluate}[\![E_1 + E_2]\!]$) where two arguments must be evaluated independently ("in parallel") and supplied to a yielder, indicating that the transient values will be held for a longer extent than usual. (An implementation might employ a stack to hold the longer-living transients.)
- then appears where the usual scoping is ignored (e.g., $\mathsf{execute}[\![D\ \mathtt{in}\ C]\!]$). The strict sequential flow warns us that bindings are made locally and override the incoming scope. See in particular, $\mathsf{investigate}[\![N.I]\!]$, where the module (binding set) computed as N's meaning replaces the current scope in determining I's meaning.



$$\begin{aligned}\mathcal{D}[\![D_1; D_2]\!]\rho_0\sigma_0 = \ & let\ b_1, \sigma_1 = \mathcal{D}[\![D_1]\!]\rho_0\sigma_0 \\ & let\ b_2, \sigma_2 = \mathcal{D}[\![D_2]\!](\rho_0 + b_1)\sigma_1 \\ & in\ b_1 + b_2, \sigma_2\end{aligned}$$

$$\mathsf{elaborate}[\![D_1; D_2]\!] = \begin{array}{l}(\mathsf{elaborate}\ D_1 \\ \quad \mathsf{then}\ (\mathsf{give}_{\mathcal{F}}\ \mathsf{currentbindings}\ \mathsf{and}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings})) \\ \mathsf{andthen}\ ((\mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{it}) \\ \qquad\qquad \mathsf{then\ elaborate}\ D_2)\end{array}$$

Fig. 8. Facet flow of $D_1; D_2$ and its denotational- and action-semantics codings

– The binding flow for elaborate$[\![D_1; D_2]\!]$ is complex: The semantics assembles the bindings made by D_1 and D_2, where D_1 uses the entry scope, but D_2 uses the scope made from the entry scope plus D_1's bindings. This means D_1's bindings are part of the output as well as part of the input to elaborate$[\![D_2]\!]$. For this reason, they are produced both as a declarative-facet value as well as a functional-facet value; see Figure 8.
  The denotational-semantics coding of the semantics, included in the Figure, states the correct distribution of bindings, but it ignores the language's "semantic architecture" (i.e., the facets), which must accommodate D_1's binding flows.
  Mosses named this complex flow pattern elaborate D_1 before elaborate D_2 [13]. Figure 9 shows a example derivation that uses the pattern.

As an exercise, one might rewrite the semantic equations so that the default combinator is and$_{\mathcal{FD}}$then (or, for that matter, then or and), to see what form of language results. (High-level declarative languages tend to be "and-languages," and low-level imperative languages are "then-languages" where the store, symbol table, and temporary-value stack are passed sequentially.)

7 Pronoun unambiguity and Mosses abstraction

Although they are functions on facet values, the yielders it and currentbindings look like pronouns. For this reason, it is important these "pronouns" are understood unambiguously. Consider elaborate$[\![D_1; D_2]\!]$ in Figure 8; there are three occurrences of pronoun currentbindings, but the first and second occurrences refer to the bindings generated from D_1 and the third occurrence refers to the bindings incoming to $D_1; D_2$. We can repair the ambiguity with an abstraction form first proposed by Mosses [9], which we call the *Mosses abstraction*. Our version is an action of form, (p : $\mathcal{G}$) => a, where pattern p predicts the shape of the incoming value from facet $\mathcal{G}$ to action a. Not a naive binder, pattern p

Let $\rho_0 = \{\}$ and $\sigma_0 = []$. The goal

$$\rho_0, \sigma_0 \vdash \mathsf{elaborate}[\![\texttt{var x = 2; proc p(y) = (x := y)}]\!] \Rightarrow \rho_{\mathsf{xp}}, \sigma_{\mathsf{x}}$$

is solved as follows:

$$\rho_0, \sigma_0 \vdash \begin{array}{l}(\mathsf{elaborate}[\![\texttt{var x = 2}]\!] \\ \quad \mathsf{then}\ (\mathsf{give}_{\mathcal{F}}\ \mathsf{currentbindings}\ \mathsf{and}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings})) \\ \mathsf{andthen}\ ((\mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{it}) \\ \qquad\quad \mathsf{then}\ \mathsf{elaborate}[\![\texttt{proc p(y) = (x := y)}]\!])\end{array} \Rightarrow \rho_{\mathsf{xp}}, \sigma_{\mathsf{x}}$$

$$\begin{array}{l}:-\ \ (\rho_0, \sigma_0 \vdash \mathsf{elaborate}[\![\texttt{var x = 2}]\!] \\ \quad \mathsf{then}\ (\mathsf{give}_{\mathcal{F}}\ \mathsf{currentbindings}\ \mathsf{and}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}) \Rightarrow \tau_1, \rho_{\mathsf{x}}, \sigma_1) \\ +(\tau_1, \rho_0, \sigma_1 \vdash (\mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{it}) \\ \qquad\quad \mathsf{then}\ \mathsf{elaborate}[\![\texttt{proc p(y) = (x := y)}]\!] \Rightarrow \rho_{\mathsf{p}}, \sigma_{\mathsf{x}}) = \rho_{\mathsf{xp}}, \sigma_{\mathsf{x}}\end{array}$$

The first subgoal simplifies to

$$\begin{array}{l} = \{(\mathsf{x}, \ell_0)\}, [\ell_0 \mapsto 2] \vdash \mathsf{give}_{\mathcal{F}}\ \mathsf{currentbindings}\ \mathsf{and}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings} \Rightarrow \tau_1, \rho_{\mathsf{x}}, \sigma_{\mathsf{x}} \\ = \langle\{(\mathsf{x}, \ell_0)\}\rangle, \{(\mathsf{x}, \ell_0)\}, [\ell_0 \mapsto 2] = \tau_1, \rho_{\mathsf{x}}, \sigma_{\mathsf{x}}\end{array}$$

Note how the binding, $\{(\mathsf{x}, \ell_0)\}$, is copied to the functional facet as well as to the declarative facet. The overall denotation has progressed to

$$\begin{array}{l}(\{(\mathsf{x}, \ell_0)\}+ \\ \langle\{(\mathsf{x}, \ell_0)\}\rangle, \rho_0, [\ell_0 \mapsto 2] \vdash (\mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{it}) \\ \qquad\qquad \mathsf{then}\ \mathsf{elaborate}[\![\texttt{proc p(y) = (x := y)}]\!] \Rightarrow \rho_{\mathsf{p}}, \sigma_{\mathsf{f}}) = \rho_{\mathsf{xp}}, \sigma_{\mathsf{x}}\end{array}$$

The second subgoal proceeds as follows:

$$\begin{array}{l}\langle\{(\mathsf{x}, \ell_0)\}\rangle, \rho_0, [\ell_0 \mapsto 2] \vdash (\mathsf{give}_{\mathcal{D}}\ \mathsf{currentbindings}\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{it}) \\ \qquad\qquad \mathsf{then}\ \mathsf{elaborate}[\![\texttt{proc p(y) = (x := y)}]\!] \Rightarrow \rho_{\mathsf{p}}, \sigma_{\mathsf{f}} \\ :-\ \ (\{\} + \{(\mathsf{x}, \ell_0)\} = \rho_{\mathsf{x}}), (\rho_{\mathsf{x}}, [\ell_0 \mapsto 2] \vdash \mathsf{elaborate}[\![\texttt{proc p(y) = (x := y)}]\!] \Rightarrow \rho_{\mathsf{p}}, \sigma_{\mathsf{f}}) \\ = \{(\mathsf{x}, \ell_0)\}, [\ell_0 \mapsto 2] \vdash \mathsf{elaborate}[\![\texttt{proc p(y) = (x := y)}]\!] \Rightarrow \rho_{\mathsf{p}}, \sigma_{\mathsf{f}} \\ = \{(\mathsf{x}, \ell_0)\}, [\ell_0 \mapsto 2] \vdash \mathsf{give}_{\mathcal{D}}(\mathsf{bind}\ \mathsf{p}\ \mathit{closure}_p) \Rightarrow \rho_{\mathsf{p}}, \sigma_{\mathsf{f}} \\ \quad \text{where } \mathit{closure}_p = [\{(\mathsf{x}, \ell_0)\}, \mathsf{p}, [\![\texttt{x := y}]\!]]_{\mathcal{D}} \\ :-\ \ (\{(\mathsf{p}, \mathit{closure}_p)\}, [\ell_0 \mapsto 2] = \rho_{\mathsf{p}}, \sigma_{\mathsf{f}}\end{array}$$

This makes the overall denotation equal

$$\{(\mathsf{x}, \ell_0)\} + \{(\mathsf{p}, \mathit{closure}_p)\}, [\ell_0 \mapsto 2] = \{(\mathsf{x}, \ell_0), (\mathsf{p}, \mathit{closure}_p)\}, [\ell_0 \mapsto 2] = \rho_{\mathsf{xp}}, \sigma_{\mathsf{x}}$$

Fig. 9. Actions taken by `var x = 2; proc p(y) = (x := y)`

defines named yielders that can be invoked within a. For example, the nested Mosses abstraction,

$$(\langle \mathsf{v}, \mathsf{w} \rangle : \mathcal{F}) \Rightarrow (\{(\mathtt{x}, \mathsf{d})\} : \mathcal{D}) \Rightarrow \mathsf{give}_{\mathcal{F}}(\mathsf{add\ w\ d})$$

asserts that the incoming functional-facet value is a sequence of at least two values and the incoming declarative-facet value holds at least a binding to $\mathtt{x}$. The first pattern binds the name v to the yielder $\#1$ and binds the name w to the yielder $\#2$; the second pattern binds the name d to the yielder, $\mathsf{find}\ \mathtt{x}$.

Thus, $\mathsf{give}_{\mathcal{F}}(\mathsf{add\ w\ d})$ makes the same action as does $\mathsf{give}_{\mathcal{F}}(\mathsf{add}\ \#2\ (\mathsf{find}\ \mathtt{x}))$. A Mosses abstraction can be understood as a kind of "macro expansion," much like traditional lambda notation macro-expands to De Bruijn notation. But there are crucial properties of Mosses abstractions that go beyond this simple analogy. To show this, we require a more formal development.[2]

A transient facet can be described by a pattern. The pattern we use for the functional facet, $\mathcal{F}$, is $\langle v_i \rangle_{1 \leq i \leq k}$, each v_i a name, representing a sequence of at least k values. For the declarative facet, $\mathcal{D}$, we use rho to denote the entire binding set and $\{(\mathtt{x}_i, d_i)\}_{1 \leq i \leq k}$, each $\mathtt{x}_i$ an identifier and each d_i a name, to represent a binding set that has bindings for identifiers $\mathtt{x}_i$.

Figure 10 defines how these patterns bind names to yielders. Yielders and actions are now evaluated with a *yielder environment*, ψ, a mapping of form, $\mathsf{Facet} \rightarrow \mathsf{Identifier} \rightarrow \mathsf{Yielder}$. For the example Mosses abstraction seen earlier, action $\mathsf{give}_{\mathcal{F}}(\mathsf{add\ v\ d})$ is interpreted with the yielder environment, $[\mathcal{F} \mapsto [\mathsf{v} \mapsto \#1,\ \mathsf{w} \mapsto \#2],\ \mathcal{D} \mapsto [\mathsf{d} \mapsto \mathsf{find\ x}]]$. Figure 10 shows and explains the derivation rule for yielder-name lookup, which consults ψ to extract and evaluate the corresponding yielder.

A Mosses abstraction is itself an action that operates with a yielder environment that is extended by the abstraction's pattern. *There can be at most one set of named yielders per facet.* Further, when a set of named yielders is generated for a facet, the default yielder for that same facet *cannot be used* — see the derivation rules for $\#\mathsf{i}$ and $\mathsf{currentbindings}$ in Figure 10. This removes pronoun/noun ambiguity in referencing values.

Additionally, a Mosses abstraction supports referential transparency. Within a Mosses-abstraction's body, *every reference to a yielder name evaluates to the same yielder which evaluates to the same value.* This crucial semantical property, which makes a Mosses abstraction behave like a lambda-abstraction, is ensured by the disciplined structure of the action-semantics combinator, a_1 $\mathsf{andthen}$ a_2: if a_1 generates output in facet $\mathcal{F}$ that passes sequentially to a_2, then the $\mathcal{F}$-generated named yielders used by a_1 are removed from a_2's use — see Figure 10.

We can employ Mosses abstractions to clarify two definitions in Figure 6. First, the semantics of function definition now shows better how a closure uses its bindings and argument:

[2] When he proposed the construction, Mosses stated, "The definition of '$\mathtt{x} \Rightarrow \mathsf{a}_1$' has been left informal, to avoid going into some technicalities." [9]

Each facet pattern generates a *yielder environment* of arity, $\mathsf{Facet} \to \mathsf{Identifier} \to \mathsf{Yielder}$:

$$[\![\langle v_i \rangle_{1\leq i\leq k} : \mathcal{F}]\!] = [\mathcal{F} \mapsto [v_i \mapsto \#\mathsf{i}]_{1\leq i\leq k}]$$
$$[\![\mathsf{rho} : \mathcal{D}]\!] = [\mathcal{D} \mapsto [\mathsf{rho} \mapsto \mathsf{currentbindings}]]$$
$$[\![\{(\mathtt{x}_i, d_i)\}_{1\leq i\leq k} : \mathcal{D}]\!] = [\mathcal{D} \mapsto [d_i \mapsto \mathsf{find}\ \mathtt{x}_i]_{1\leq i\leq k}]$$

Let ψ be a yielder environment. A yielder sequent now has form, $\Gamma \vdash_\psi \mathsf{y} : \tau$; the ψ annotations are uniformly added to the sequents in the rules of Figure 1. The rule for evaluating a yielder name, n, defined from the pattern for facet $\mathcal{G}$, goes as follows:

$$\frac{\mathcal{G} \in \mathit{domain}(\psi) \qquad \psi(\mathcal{G})(\mathsf{n}) = \mathsf{y} \qquad \Gamma \vdash \mathsf{y} : \tau}{\Gamma \vdash_\psi (\mathsf{n} : \mathcal{G}) : \tau}$$

The default yielders for $\mathcal{F}$ and $\mathcal{D}$ are "disabled" when a yielder environment already exists for the facet:

$$\frac{\mathcal{F} \notin \mathit{domain}(\psi) \qquad 1 \leq i \leq n}{\langle \tau_1, \cdots, \tau_n \rangle \vdash_\psi \#\mathsf{i} : \tau_i} \qquad \frac{\mathcal{D} \notin \mathit{domain}(\psi)}{\rho \vdash_\psi \mathsf{currentbindings} : \rho}$$

An action sequent has form, $\Gamma \vdash_\psi \mathsf{a} \Rightarrow \Delta$. The ψ annotations are uniformly added to the sequents in the rules of Figure 2. The new rule for Mosses abstraction reads as follows:

$$\frac{\Gamma \vdash_{\psi + [\![\mathsf{p}]\!]} \mathsf{a} \Rightarrow \Delta}{\Gamma \vdash_\psi (\mathsf{p} \Rightarrow \mathsf{a}) \Rightarrow \Delta}$$

The + denotes function override.

When an $\mathcal{F}$-value flows sequentially from action a_1 to a_2, the $\psi(\mathcal{F})$-part of ψ must be removed from a_2's use. This is enforced by the revised rule for $\mathsf{andthen}$:

$$\frac{\Gamma \vdash_\psi \mathsf{a}_1 \Rightarrow \Delta_1 \qquad (\Gamma \downarrow_{\mathcal{G}}) \cup (\Delta_1 \downarrow_{\sim\mathcal{G}}) \vdash_{\psi\downarrow\mathcal{G}} \mathsf{a}_2 \Rightarrow \Delta_2}{\Gamma \vdash_\psi \mathsf{a}_1\ \mathsf{and}_{\mathcal{G}}\mathsf{then}\ \mathsf{a}_2 \Rightarrow \Delta_1 \downarrow_{\mathcal{G}} \circ \Delta_2}$$

where $\psi \downarrow \mathcal{G}$ denotes ψ restricted to argument(s) $\mathcal{G}$ only.

Finally, the yielder environment for a closure is restricted to the facets embedded within the closure:

$$\frac{\Gamma \vdash_\psi \mathsf{y} : \Delta}{\Gamma \vdash_\psi \mathsf{recabstract}_{\mathcal{G}}\ \mathsf{y}\ I\ \mathsf{a}\ :\ [\Delta \downarrow_{\mathcal{G}}, I, \mathsf{a}]_{\mathcal{G}, \psi\downarrow\mathcal{G}}}$$

$$\frac{\begin{array}{l}\Gamma_1 \vdash_\psi \mathsf{y}_1 : [\Delta, I, \mathsf{a}]_{\mathcal{G},\psi'} \\ \Gamma_2 \vdash_\psi \mathsf{y}_2 : \tau \\ \Gamma = \Gamma_1 \cup \Gamma_2\end{array} \qquad \langle \tau \rangle \circ (\Delta \cup (\Gamma \downarrow_{\sim\mathcal{G}})) \circ \{(I, [\Delta, I, \mathsf{a}]_{\mathcal{G},\psi'})\} \vdash_{\psi'} \mathsf{a} \Rightarrow \Sigma}{\Gamma \vdash_\psi \mathsf{exec}\ \mathsf{y}_1\ \mathsf{y}_2 \Rightarrow \Sigma}$$

Fig. 10. Semantics of Mosses abstractions

$$
\begin{array}{l}
\mathsf{elaborate}[\![\texttt{proc}\ I_1(I_2)\ \texttt{=}\ C]\!] = (\mathsf{rho} : \mathcal{D})\ \texttt{=>}\ \mathsf{give}_{\mathcal{D}}(\mathsf{bind}\ I_1\ \mathit{closure}) \\
\ \text{where}\ \mathit{closure} = \mathsf{recabstract}_{\mathcal{D}}\ I_1\ \mathsf{rho} \\
\qquad\qquad\qquad\qquad ((\langle\mathsf{arg}\rangle : \mathcal{F})\ \texttt{=>} \\
\qquad\qquad\qquad\qquad (\mathsf{give}_{\mathcal{D}}\ \mathsf{rho}\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{give}_{\mathcal{D}}(\mathsf{bind}\ I_2\ \mathsf{arg})) \\
\qquad\qquad\qquad\qquad\ \mathsf{then}\ \mathsf{execute}\ C)
\end{array}
$$

This macro-expands to the definition seen in Figure 6. (The proof depends on the big-step rule for $\mathsf{and}_{\mathcal{FD}}\mathsf{then}$, which shows that the $\mathcal{F}$-value flows in parallel.)

Second, the multiple occurrences of currentbindings within the semantics of sequential declaration can be resolved unambiguously as

$$
\begin{array}{l}
\mathsf{elaborate}[\![D_1; D_2]\!] = (\mathsf{rho}_0 : \mathcal{D})\ \texttt{=>} \\
\quad (\mathsf{elaborate}\ D_1\ \mathsf{then}\ (\mathsf{rho}_1 : \mathcal{D})\ \texttt{=>}\ \mathsf{give}_{\mathcal{F}}\ \mathsf{rho}_1\ \mathsf{and}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{rho}_1) \\
\quad \mathsf{andthen} \\
\quad ((\langle\mathsf{rho}_1\rangle : \mathcal{F})\ \texttt{=>}\ (\mathsf{give}_{\mathcal{D}}\ \mathsf{rho}_0\ \mathsf{and}_{\mathcal{FD}}\mathsf{then}\ \mathsf{give}_{\mathcal{D}}\ \mathsf{rho}_1)\ \mathsf{then}\ \mathsf{elaborate}\ D_2)
\end{array}
$$

Compare this semantics to the ones in Figure 8 — it is as readable as the denotational one but remains true to the underlying "semantic architecture."

8 From action equations to big-step semantics

Since the yielders and actions are defined by big-step semantic rules, one can map the action equations themselves into big-step-rule format by applying partial evaluation [1, 8]. The idea is that a valuation function of arity, $\mathsf{interp} : \mathsf{PhraseForm} \to \mathcal{F}_1 \cdots \mathcal{F}_m \to \mathcal{G}_1 \cdots \mathcal{G}_n$, suggests a big-step sequent of form, $f_1 \circ \cdots \circ f_m \vdash P \Rightarrow g_1 \circ \cdots \circ g_n$. An action equation for a phrase, $\texttt{cons}\ P_1 \cdots P_p$, relies on the actions denoted by the $\mathsf{interp}\ P_i$s to compute $\mathsf{interp}[\![\texttt{cons}\ P_1 \cdots P_p]\!]$. The corresponding big-step rule uses the sequent forms for each P_i as antecedents for the consequent sequent, $f_1 \circ \cdots \circ f_m \vdash \texttt{cons}\ P_1 \cdots P_p \Rightarrow g_1 \circ \cdots \circ g_n$.

The translation from action equations to big-step rules is a mechanical process, where the big-step rules for or, andthen, and the primitive actions and yielders are elaborated to expose the argument-passing flows of temporaries, τ, binding sets, ρ, and store, σ. Figure 11 shows representative translations from Figure 6. Of the examples shown above, only the rule for $\texttt{call}N(E)$ requires mild reformatting to match its counterpart in Figure 6, the issue being the binding of actual to formal parameter.

Figures 6 and 11 have the same information content, but Figure 6 is higher-level in its presentation of value flows, whereas Figure 11 makes explicit the connections and compositions. A reader familiar with big-step semantics may prefer the latter, but the explicit detail obscures the fundamental andthen facet flows that give the language its character. And this was indeed the issue that motivated Mosses to develop action semantics in the first place.

9 From action equations to denotational semantics

Mosses did not intend to erase from memory Scott-Strachey denotational semantics [7, 14, 19], but he desired a methodology and notation that matched

$$\frac{\rho \circ \sigma \vdash \mathsf{evaluate}\ E_1 \Rightarrow \tau_1 \qquad \rho \circ \sigma \vdash \mathsf{evaluate}\ E_1 \Rightarrow \tau_2 \qquad \begin{array}{l} \tau_1 \leq \mathsf{Int} \quad \tau_2 \leq \mathsf{Int} \\ \tau_3 = add(\tau_1, \tau_2) \end{array}}{\rho, \sigma \vdash \mathsf{evaluate}[\![E_1 + E_2]\!] \Rightarrow \tau_3}$$

$$\frac{\rho \vdash \mathsf{investigate}\ N \Rightarrow \tau \qquad \tau \leq \mathsf{Cell} \qquad \sigma(\tau_1) = \tau_2}{\rho \circ \sigma \vdash \mathsf{evaluate}\ N \Rightarrow \tau_2}$$

$$\frac{\rho \circ \sigma \vdash \mathsf{investigate}\ N \Rightarrow \tau_1 \quad \tau_1 \leq \mathsf{Cell} \quad \rho \circ \sigma \vdash \mathsf{evaluate}\ E \Rightarrow \tau_2 \quad \sigma_1 = \sigma[\tau_1 \mapsto \tau_2]}{\rho \circ \sigma \vdash \mathsf{execute}[\![N \mathtt{:=} E]\!] \Rightarrow \sigma_1}$$

$$\frac{\rho \circ \sigma \vdash \mathsf{execute}\ C_1 \Rightarrow \sigma_1 \qquad \rho \circ \sigma_1 \vdash \mathsf{execute}\ C_2 \Rightarrow \sigma_2}{\rho \circ \sigma \vdash \mathsf{execute}[\![C_1; C_2]\!] \Rightarrow \sigma_2}$$

$$\frac{\rho \circ \sigma \vdash \mathsf{evaluate}\ E \Rightarrow \tau \qquad \tau \leq \mathsf{NonZero} \qquad \begin{array}{l} \rho \circ \sigma \vdash \mathsf{execute}\ C \Rightarrow \sigma_1 \\ \rho \circ \sigma_1 \vdash \mathsf{execute}[\![\mathtt{while}\ E\ \mathtt{do}\ C]\!] \Rightarrow \sigma_2 \end{array}}{\rho \circ \sigma \vdash \mathsf{execute}[\![\mathtt{while}\ E\ \mathtt{do}\ C]\!] \Rightarrow \sigma_2}$$

$$\frac{\begin{array}{l} \rho \circ \sigma \vdash \mathsf{investigate}\ N \Rightarrow \tau_1 \\ \tau_1 \leq \mathsf{Closure} \\ \rho \circ \sigma \vdash \mathsf{evaluate}\ E \Rightarrow \tau_2 \end{array} \qquad \begin{array}{l} \tau_1 = [\mathcal{D}, \rho_1, I_1, I_2, C] \\ \rho_1 + \{(I_1, \tau_1), (I_2, \tau_2)\} \circ \sigma \vdash \mathsf{execute}\ C \Rightarrow \sigma_1 \end{array}}{\rho \circ \sigma \vdash \mathsf{execute}[\![\mathtt{call}\ N(E)]\!] \Rightarrow \sigma_1}$$

$$\frac{\rho \circ \sigma \vdash \mathsf{evaluate}\ E \Rightarrow \tau \qquad c \notin domain(\sigma) \qquad \rho_1 = \{(I, c)\} \qquad \sigma_1 = \sigma[c \mapsto \tau]}{\rho \circ \sigma \vdash \mathsf{elaborate}[\![\mathtt{var}\ I\ \mathtt{=}\ E]\!] \Rightarrow \rho_1 \circ \sigma_1}$$

$$\frac{\begin{array}{l} \rho \circ \sigma \vdash \mathsf{elaborate}\ D_1 \Rightarrow \rho_1 \circ \sigma_1 \\ \rho \circ \rho_1 \circ \sigma_1 \vdash \mathsf{elaborate}\ D_2 \Rightarrow \rho_2 \circ \sigma_2 \end{array} \qquad \rho_3 = \rho_1 \circ \rho_2}{\rho \circ \sigma \vdash \mathsf{elaborate}[\![D_1; D_2]\!] \Rightarrow \rho_3 \circ \sigma_2}$$

Fig. 11. Selected big-step rules derived from action equations

more closely a programmer's and a language designer's intuitions and was less sensitive to modelling issues (e.g., direct versus continuation semantics). Action semantics lives at a higher level of abstraction than does denotational semantics, and it is a routine but enlightening exercise to interpret its facets, yielders, and actions with Scott-domains and continuous functions:

– Each facet is mapped to a Strachey-style "characteristic domain."
– Each yielder is mapped to a continuous function on the characteristic domains such that the big-step rules in Figure 1 are proved sound, where $\Gamma \vdash \mathsf{y} : \Delta$ is interpreted as $[\![y]\!](\gamma) = \delta$, such that $[\![y]\!]$ is the continous function and γ and δ are the Scott-domain values interpreted from Γ and Δ.
– Each action is mapped to a (higher-order) continuous function in a similar manner so that the rules in Figure 2 are proved sound. When the yielders compute on transients only and actions compute on persistent values, the interpretation into denotational semantics gives a "two-level" denotational semantics as developed by Nielson and Nielson [17].
– The action equations are treated as valuation functions, and a least-fixed-point interpretation is taken of self-references.

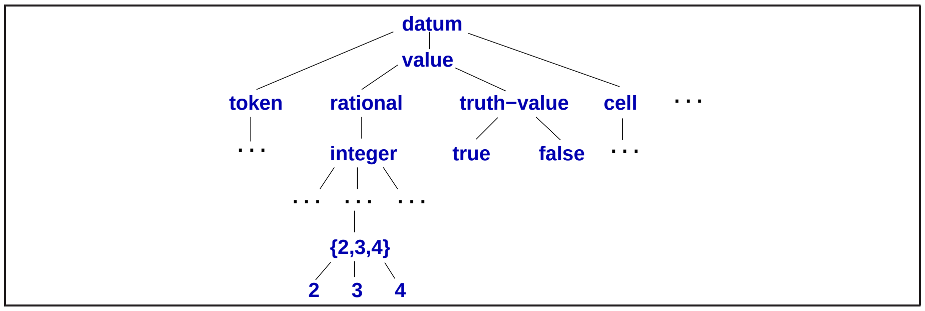


Fig. 12. Sort structure for functional facet

Once again, the information in the action-equation version and the denotational semantics version is "the same," but the former presents and emphasizes language design concepts more directly.

Related to this exercise is the relationship between direct-style and continuation-style semantics. If our example language contained jumps or fork-join constructions, the denotational-semantics domains would change as would the interpretation of the $\mathsf{andthen}$ combinator, into a tail-recursive form, *à la* $\mathsf{a}_1\ \mathsf{andthen}\ \_$, so that the continuation from a_1 could be kept or discarded as directed. The action equations themselves remain the same.

10 Subsorts within the facets

The relation, $\leq$, has been used to indicate data-type membership, e.g., $3 \leq \mathsf{Int}$. Mosses defined *unified algebra* [12] to state membership properties that go beyond the usual judgements.

Figure 12 portrays a subsort relationship, a partial ordering, for a unified algebra that lists the sorts of values that can be used with the functional facet. The diagram asserts that $\mathsf{integer} \leq \mathsf{rational} \leq \mathsf{datum}$, etc. — an implicit subset ordering applies. Even finite sets of values ($\{2, 3, 4\}$) and singletons (2) are sorts. Of course, one does not implement all the sort names, but they serve as a useful definitional tool. Each sort name is interpreted by a carrier of values that belong to the sort, e.g., $\mathsf{integer}$ is interpreted by $\{\cdots, -1, 0, 1, 2, \cdots\}$, and $\{2, 3, 4\}$ is interpreted by $\{2, 3, 4\}$. Sequences of functional-facet values are ordered pointwise.

The declarative facet's sorts can be portrayed as seen in Figure 13. There is a pointwise ordering, based on the identifiers that are named in the sorts: $\rho_1 \leq \rho_2$ iff the identifiers named in ρ_1 equal the ones named in ρ_2 and for for every (I, τ_1) in ρ_1 and (I, τ_2) in ρ_2, $\tau_1 \leq \tau_2$. A sort is interpreted by those sets of pairs that have exactly the bindings stated in the sort name, e.g., $\{(\mathsf{x}, \mathsf{integer}), (\mathsf{y}, \{2, 3, 4\})\}$ is interpreted by $\{\{(\mathsf{x}, m), (\mathsf{y}, n)\} \mid m \leq \mathsf{integer} \text{ and } n \leq \{2, 3, 4\}\}$.

The imperative facet is organized similarly to the declarative facet: its sorts are finite maps from Cell names to subsorts of datum such that $\sigma_1 \leq \sigma_2$ iff $\mathit{domain}(\sigma_1) = \mathit{domain}(\sigma_2)$ and for every $\mathsf{c} \in \mathit{domain}(\sigma_1)$, $\sigma_1(\mathsf{c}) \leq \sigma_2(\mathsf{c})$.

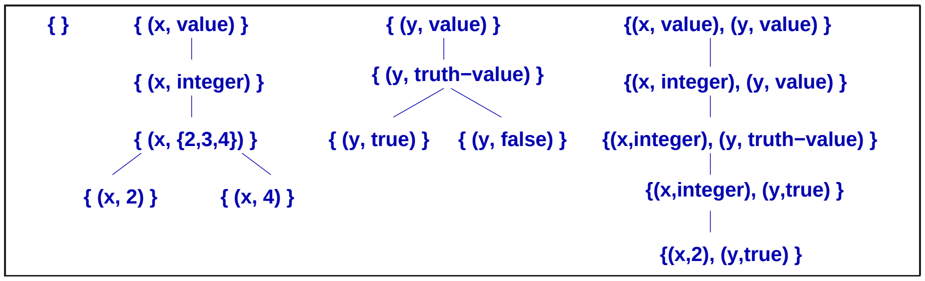


Fig. 13. Sort structure for declarative facet

Strictly speaking, subsorting does not extend to compound-facet values, but we will write $\Delta \leq \Gamma$ for compound facet values to assert that $\Delta = \{f_1, \cdots, f_m\}$, $\Gamma = \{f'_1, \cdots, f'_m\}$ and $f_i \leq f'_i$ for all $i \in 1..m$ in each respective facet, $\mathcal{F}_i$.

The sorting hierarchies suggest that yielders and actions can compute on sort names as well as individual values. For example, all these derivations are meaningful:

$\vdash \mathsf{give}\ 2 \Rightarrow 2$
$\vdash \mathsf{give}\ 2 \Rightarrow \mathsf{integer}$
$\vdash \mathsf{give}\ 2 \Rightarrow \mathsf{datum}$

$\{\langle 2\rangle, \{(\mathsf{x}, 3)\}\} \vdash \mathsf{give\ add(isInteger\ it, isInteger(find\ x))} \Rightarrow 5$
$\{\langle 2\rangle, \{(\mathsf{x}, 3)\}\} \vdash \mathsf{give\ add(isInteger\ it, isInteger(find\ x))} \Rightarrow \mathsf{integer}$
$\{\langle 2\rangle, \{(\mathsf{x}, \mathsf{integer})\}\} \vdash \mathsf{give\ add(isInteger\ it, isInteger(find\ x))} \Rightarrow \mathsf{integer}$
$\{\langle \mathsf{integer}\rangle, \{(\mathsf{x}, \mathsf{integer})\}\} \vdash \mathsf{give\ add(isInteger\ it, isInteger(find\ x))} \Rightarrow \mathsf{integer}$

The derivations justify type checking and abstract interpretation upon the semantics definition.[3]

The above assertions are connected by this weakening rule for sorting, which weakens information within individual facets:

$$\frac{\Gamma_1 \leq \Gamma_2 \quad \Gamma_2 \vdash \mathsf{a} \Rightarrow \Delta_2 \quad \Delta_2 \leq \Delta_1}{\Gamma_1 \vdash \mathsf{a} \Rightarrow \Delta_1}$$

The rule complements the two existing weakening and strengthening rules, which weaken the facet structures themselves.

11 Partial evaluation of action semantics

As in the derivation examples in the previous section, a yielder may have many possible input-output sort pairs. However, for input context Γ, there is a least output sort Δ such that $\Gamma \vdash \mathsf{y} : \Delta$ holds. This is called the *least sorting property.* For example, a yielder 2 can have output sorts datum, value, integer, {2,3,4}, 2,

[3] In the example, we assume that the operation, *add*, is extended monotonically to operate on all sorts within the functional facet. This makes all yielders and actions behave monotonically as well.

etc., in any input context, but the least sort among them is 2. Similarly, a yielder add (isInteger it) (isInteger (find x)) has the least output sort integer in the context, 2, {(x,integer)}. The sort checking rule for calculating a least sort for integer addition operation can be defined as follows:

$$\frac{\Gamma \vdash \mathsf{y}_1 : \tau_1 \quad \Delta \vdash \mathsf{y}_2 : \tau_2 \quad \tau_1 \leq \mathsf{integer} \quad \tau_2 \leq \mathsf{integer}}{\Gamma \cup \Delta \vdash \mathsf{add}\ \mathsf{y}_1\ \mathsf{y}_2 : add(\tau_1, \tau_2)}$$

Other rules for yielders in Figure 1 can be used for sort checking without modification, along with the weakening rule just introduced:

$$\frac{\Gamma_1 \leq \Gamma_2 \quad \Gamma_2 \vdash \mathsf{y} : \Delta_2 \quad \Delta_2 \leq \Delta_1}{\Gamma_1 \vdash \mathsf{y} : \Delta_1}$$

The rules for sort consistency check upon actions can also be defined in big-step deduction style. Read $\Gamma \vdash \mathsf{a} \Rightarrow \Delta$ as asserting that action a receives sorts of facets Γ and produces sorts of facets Δ. Note that the imperative facet is excluded from Γ and Δ since sort checking occurs before run-time. The sort checking rules for imperative-facet actions are defined as follows:

$$\frac{\Gamma \vdash \mathsf{y} : c \quad c \leq \mathsf{integer\text{-}cell}}{\Gamma \vdash \mathsf{lookup}\ \mathsf{y} \Rightarrow \mathsf{integer}}$$

$$\frac{\Gamma_1 \vdash \mathsf{y}_1 : c \quad c \leq \mathsf{integer\text{-}cell} \quad \Gamma_2 \vdash \mathsf{y}_2 : \tau \quad \tau \leq \mathsf{integer}}{\Gamma_1 \cup \Gamma_2 \vdash \mathsf{update}\ \mathsf{y}_1\ \mathsf{y}_2 \Rightarrow \mathsf{completing}}$$

$$\frac{}{\vdash \mathsf{allocate} \Rightarrow \mathsf{integer\text{-}cell}}$$

Sort consistency checking between closure and its arguments is done at application time. The rules are identical to those in Figure 2.

Sorts can be distinguished according to binding times — compile-time sorts and run-time sorts [3, 5]. Individual sorts, such as 1, true, etc., are known constants, and thus static sorts. All other sorts, including integer, cell, etc., are treated as dynamic sorts because their values are not known. Yielders and actions taking static sorts can be processed at compile-time, reducing the run-time computation overhead. For example, consider an action, give 2 andthen give (add it (find x)). The left subaction, give 2, is statically computable, and passes its output value to the yielder it in the right subaction. The yielder it then consumes the value, and then the whole action is semantically identical to give 2 andthen give (add 2 (find x)). Since the life of the left subaction is over, the action is safely reduced to give (add 2 (find x)). If the given context to this action is {(x,3)}, find x becomes 3, and then the yielder add 2 3 is further computed to 5. On the other hand, if the given context is {(x,integer)}, no more computation is possible and the action remains as it is. This transformation is partial evaluation of the actions.

During partial evaluation, since each yielder either gives computed sorts or reconstructs yielder code, rules for partial evaluation have to carry around both facets and reconstructed residual code. Read $\Gamma, \kappa_i \vdash \mathsf{y} : \Delta, \kappa_o$ as asserting that yielder y consumes facets Γ and residual code κ_i, and produces facets Δ and

Functional-facet yielders:

primitive constant: $\dfrac{}{\vdash \mathsf{k} : k, \emptyset}$

n-ary operation (e.g., addition):

$$\dfrac{\Gamma, \kappa \vdash \mathsf{y}_1 : \tau_1, \kappa_1 \quad \Delta, \kappa \vdash \mathsf{y}_2 : \tau_2, \kappa_2}{\Gamma \cup \Delta, \kappa \vdash \mathsf{add}\ \mathsf{y}_1\ \mathsf{y}_2 : \begin{array}{l} \text{case } \kappa_1, \kappa_2 \text{ of} \\ \quad \emptyset, \emptyset \rightarrow add(\tau_1, \tau_2), \emptyset \\ \quad \emptyset, \_ \rightarrow \tau_1 \sqcup \tau_2, [\![\mathsf{add}\ [\![\tau_1]\!]\ \kappa_2]\!] \\ \quad \_, \emptyset \rightarrow \tau_1 \sqcup \tau_2, [\![\mathsf{add}\ \kappa_1\ [\![\tau_2]\!]]\!] \\ \quad \_, \_ \rightarrow \tau_1 \sqcup \tau_2, [\![\mathsf{add}\ \kappa_1\ \kappa_2]\!] \end{array}}$$

indexing: $\dfrac{1 \leq i \leq m}{\langle \tau_1, \cdots, \tau_m \rangle, \langle \kappa_1, \cdots, \kappa_m \rangle \vdash \#\mathsf{i} : \tau_i, \kappa_i}$ Note: it abbreviates #1

sort filtering: $\dfrac{\Gamma, \kappa \vdash \mathsf{y} : \Delta, \kappa' \qquad \Delta \leq T}{\Gamma, \kappa \vdash \mathsf{is}T\ \mathsf{y} : \Delta, \text{if } \kappa' = \emptyset \text{ then } \emptyset \text{ else } [\![\mathsf{is}T\ \kappa']\!]}$

Declarative-facet yielders:

binding lookup: $\dfrac{(\mathsf{I}, \tau) \in \rho}{\rho, \kappa \vdash \mathsf{find}\ \mathsf{I} : \tau, \text{if } static(\tau) \text{ then } \emptyset \text{ else } [\![\mathsf{find}\ \mathsf{I}]\!]}$

binding creation: $\dfrac{\Gamma, \kappa \vdash \mathsf{y} : \tau, \kappa'}{\Gamma, \kappa \vdash \mathsf{bind}\ \mathsf{I}\ \mathsf{y} : \{(\mathsf{I}, \tau)\}, \text{if } \kappa' = \emptyset \text{ then } \emptyset \text{ else } [\![\mathsf{bind}\ \mathsf{I}\ \kappa']\!]}$

binding copy: $\dfrac{}{\rho, \kappa \vdash \mathsf{currentbindings} : \rho, \text{if } \kappa' = \emptyset \text{ then } \emptyset \text{ else } [\![\mathsf{currentbindings}]\!]}$

Fig. 14. Partial evaluation for yielders

residual code κ_o. If the output sort Δ is static, then code reconstruction is not necessary and thus $\kappa_o = \emptyset$, indicating no code. Otherwise, the residual code is reconstructed.

Checking whether or not a yielder output is static can be done by examining its output sort. The following function *static* determines if the given functional-facet sort is static.

$$\begin{array}{l} static(\langle \tau \rangle) = \text{if } \tau \text{ is an individual value then true else false} \\ static(\langle \tau_1, \ldots, \tau_n \rangle) = static(\tau_1) \wedge \cdots \wedge static(\tau_n) \\ static([\tau, I, \mathsf{a}]) = \text{false} \\ static(\{(I_1, \tau_1), \ldots, (I_n, \tau_n)\}) = static(\tau_1) \wedge \cdots \wedge static(\tau_n) \end{array}$$

A yielder output is also static when the emitted residual code is $\emptyset$.

Figure 14 presents a sample collection of rules for partial evaluation for yielders. Primitive constant yielder is always static, thus emits no residual code, $\emptyset$. If two argument yielders of addition operation are both static, they are evaluated and added to give a static output, while emitting no code. Otherwise, after two argument yielders are partially evaluated, the whole yielder code is recon-

Structural actions:

$$\frac{\Gamma, \kappa \vdash \mathsf{y} : \Delta, \kappa'}{\Gamma, \kappa \vdash \mathsf{give}_{\mathcal{G}}\ \mathsf{y} \Rightarrow \Delta, \text{if } \kappa' = \emptyset \text{ then } \emptyset \text{ else } [\![\mathsf{give}_{\mathcal{G}}\ \kappa']\!]}$$

$$\frac{}{\vdash \mathsf{complete} \Rightarrow \mathit{completing}, [\![\mathsf{complete}]\!]}$$

Imperative-facet actions:

$$\frac{\Gamma, \kappa \vdash \mathsf{y} : s\text{-}\mathsf{cell}, \kappa'}{\Gamma, \kappa \vdash \mathsf{lookup}\ \mathsf{y} \Rightarrow s, [\![\mathsf{lookup}\ \kappa']\!]}$$

$$\frac{\Gamma_1, \kappa \vdash \mathsf{y}_1 : \tau_1\text{-}\mathsf{cell}, \kappa_1 \qquad \Gamma_2, \kappa \vdash \mathsf{y}_2 : \tau_2, \kappa_2 \qquad \tau_2 \leq \tau_1}{\Gamma_1 \cup \Gamma_2, \kappa \vdash \mathsf{update}\ \mathsf{y}_1\ \mathsf{y}_2 \Rightarrow \mathit{completing}, [\![\mathsf{update}\ \kappa_1\ \kappa_2]\!]}$$

$$\frac{}{\vdash \mathsf{allocate}_s \Rightarrow s\text{-}\mathsf{cell}, [\![\mathsf{allocate}_s]\!]}$$

Fig. 15. partial evaluation for actions

structed. If the output of declarative-facet yielder is static, no code is emitted. However, the code is reconstructed otherwise.

Figure 15 presents a sample action set, whose partial-evaluation behaviors are defined with big-step deduction rules. Read $\Gamma, \kappa_i \vdash \mathsf{a} \Rightarrow \Delta, \kappa_o$ as asserting that action a receives facets Γ and a residual code κ_i, and produces facets Δ and a residual code κ_o. A structural action such as $\mathsf{give}\ \mathsf{y}$ either gives the evaluated results or reconstructs a residual code depending on its binding time. Imperative actions are all reconstructed at partial-evaluation time, but their yielder constituents are evaluated when possible. Since the termination is not guaranteed, the body of a self-referencing closure is not partially evaluated.

For action combinators, $\mathsf{and}_{\mathcal{G}}\mathsf{then}$, partial evaluation is defined differently depending on the facet flows. When $\mathcal{G} = \emptyset$, the combinator is essentially the same as then, and its partial evaluation can be defined as follows:

$$\frac{\Gamma, \kappa \vdash \mathsf{a}_1 \Rightarrow \Delta, \kappa_1 \quad \Delta, \kappa_1 \vdash \mathsf{a}_2 \Rightarrow \Sigma, \kappa_2}{\Gamma, \kappa \vdash \mathsf{a}_1\ \mathsf{then}\ \mathsf{a}_2 \Rightarrow \Sigma, \text{if } \kappa_1 = \emptyset \text{ then } \kappa_2 \text{ else } [\![\kappa_1\ \mathsf{then}\ \kappa_2]\!]}$$

In this case, when the left subaction is static, the whole action can be reduced to its right subaction. The partial evaluation of the $\mathsf{and}_{\mathcal{G}}\mathsf{then}$ combinator when $\mathcal{G} \neq \emptyset$ is defined differently as follows:

$$\frac{\Gamma, \kappa \vdash \mathsf{a}_1 \Rightarrow \Delta_1, \kappa_1 \qquad (\Gamma {\downarrow_{\mathcal{G}}}) \cup (\Delta_1 {\downarrow_{\sim\mathcal{G}}}), \kappa \vdash \mathsf{a}_2 \Rightarrow \Delta_2, \kappa_2}{\Gamma, \kappa \vdash \mathsf{a}_1\ \mathsf{and}_{\mathcal{G}}\mathsf{then}\ \mathsf{a}_2 \Rightarrow \Delta_1 {\downarrow_{\mathcal{G}}} \circ \Delta_2, \begin{array}{l} \text{case } \kappa_1, \kappa_2 \text{ of} \\ \quad \emptyset, \emptyset \rightarrow \emptyset \\ \quad \emptyset, \_ \rightarrow [\![[\![\Delta_1]\!]\ \mathsf{and}_{\mathcal{G}}\mathsf{then}\ \kappa_2]\!] \\ \quad \_, \emptyset \rightarrow [\![\kappa_1\ \mathsf{and}_{\mathcal{G}}\mathsf{then}\ [\![\Delta_2]\!]]\!] \\ \quad \_, \_ \rightarrow [\![\kappa_1\ \mathsf{and}_{\mathcal{G}}\mathsf{then}\ \kappa_2]\!] \end{array}}$$

12 Conclusion

Action semantics was an influential experiment in programming-language design and engineering. Its success rests on its relatively high level of abstraction and its sensitivity to a language's "semantic architecture," as expressed by facets. Action semantics readily maps to operational and denotational semantics definitions, giving a good entry point into language-definition methodology.

By presenting a naive formulation, based on two combinators, a weakening rule, and a strengthening rule, the present paper has attempted to expose action semantics's personality and emphasize its strengths.

In appreciation of Peter D. Mosses

Peter Mosses has made significant impact on the programming languages community, and he made significant impact on the authors of this paper as well.

David Schmidt: I met Peter in Aarhus in 1979. I was impressed by his thesis work, his Semantics Implementation System, and his ability to go to the core of an issue. (One example: After viewing a presentation on a dubiously complex programming technique, Peter contributed, "As Strachey might say, first get it working correctly, then get it working fast!")

Peter answered many of my beginner's questions about denotational semantics, and his work on binding algebras showed me that language semantics was more than Scott-domains and lambda calculus — semantics itself should have structure. This insight, plus Peter's remark to me in 1982, that semantic domains need not necessarily possess $\perp$-elements, gave me all I needed to write the key chapters of my *Denotational Semantics* text [18].

Peter's research on action semantics forms the most profound body of knowledge on programming-language principles I have encountered, and I have enjoyed studying and applying this material over the decades. I deeply appreciate Peter's insights, his perseverence, and his friendship.

Kyung-Goo Doh: It was the paper, "Abstract semantic algebras!" [10], introduced to me by David Schmidt in 1990, that impressed me and guided me into the Peter Mosses's world of programming language semantics. It did not take a long time for me to choose the subject as my Ph.D. research topic. Since then, Peter has been a good mentor to me on numerous occasions through personal communications, research collaborations, and published papers. Peter assured me that it would be possible to have a useful semantics formalism for realistic programming languages just like the syntax counterpart, BNF. Peter's outlasting works in programming-language semantics hugely influenced me and my students on understanding the principles of programming languages. I admire Peter's enduring body of research works, and I cordially thank him for his support and friendship.

Finally, this paper is a significantly expanded and revised version of research

presented at the First Workshop on Action Semantics, Edinburgh, 1994 [4]. The authors thank Peter Mosses for organizing the workshop and inviting both of us to attend.

References

1. K.-G. Doh. Action transformation by partial evaluation. In *PEPM'95*, pages 230–240. ACM Press, 1995.
2. K.-G. Doh and P.D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Computer Prog.*, 47:3–36, 2003.
3. K.-G. Doh and D. A. Schmidt. Extraction of strong typing laws from action semantics definitions. In B. Krieg-Bruckner, editor, *ESOP'92, Proceedings of the 4th European Symposium on Programming*, pages 151–166. Lecture Notes in Computer Science 582, Springer-Verlag, 1992.
4. K.-G. Doh and D. A. Schmidt. The facets of action semantics: some principles and applications. In *Workshop on Action Semantics*, pages 1–15. Univ. of Aarhus, BRICS NS-94-1, 1994.
5. K.-G. Doh and D.A. Schmidt. Action semantics-directed prototyping. *Computer Languages*, 19:213–233, 1993.
6. I. Guessarian. *Algebraic Semantics*. Springer LNCS 99, 1981.
7. C. Gunter and D.S. Scott. Semantic domains. In *Handbook of Theoretical Computer Science, Vol. B*, pages 633–674. MIT Press, 1991.
8. N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
9. P. D. Mosses. A semantic algebra for binding constructs. In *Colloq. Formalization of Programming Concepts*, pages 408–418. Springer LNCS 107, 1981.
10. P. D. Mosses. Abstract semantic algebras! In *Formal Description of Programming Concepts II, Proceedings of the IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982*, pages 45–72. IFIP, North-Holland, Amsterdam, 1983.
11. P. D. Mosses. A basic abstract semantic algebra. In *Symp. Semantics of data types*, pages 87–107. Springer LNCS 173, 1984.
12. P. D. Mosses. Unified algebras and action semantics. In *STACS'89, Proceedings of Symposium on Theoretical Aspects of Computer Science, Paderborn*. Lecture Notes in Computer Science 349, Springer-Verlag, 1989.
13. P. D. Mosses. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.
14. P.D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 575–631. MIT Press, 1990.
15. P.D. Mosses. Theory and practice of action semantics. In *Proc. Math. Found. Comp. Sci.*, pages 37–61. Springer LNCS 1113, 1996.
16. P.D. Mosses and D.A. Watt. The use of action semantics. In Martin Wirsing, editor, *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986*. IFIP, North-Holland, 1987.
17. F. Nielson and H. R. Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.
18. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

19. D.S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings of Symposium on Computers and Automata*, pages 19–46. Microwave Research Institute Symposia Series: Volume 21, Polytechnic Institute of Brooklyn, 1971.
20. J.E. Stoy. *Denotational Semantics*. MIT Press, Cambridge, MA, 1977.
21. C. Strachey. The varieties of programming language. Technical Report PRG-10, Prog. Research Group, Oxford University, 1973.
22. D.A. Watt. An action semantics of standard ML. In *Proc. Math. Found. Prog. Semantics*, pages 572–598. Springer LNCS 298, 1987.