The Facets of Action Semantics: Some Principles and Applications (Extended Abstract)

Kyung-Goo Doh
The University of Aizu*

David A. Schmidt
Kansas State University†

Abstract

A distinguishing characteristic of action semantics is its facet system, which defines the variety of information flows in a language definition. The facet system can be analyzed to validate the well-formedness of a language definition, to infer the typings of its inputs and outputs, and to calculate the operational semantics of programs.

We present a single framework for doing all of the above. The framework exploits the internal subsorting structure of the facets so that sort checking, static analysis, and operational semantics are related, sound instances of the same underlying analysis. The framework also suggests that action semantics's extensibility can be understood as a kind of "weakening rule" in a "logic" of actions.

In this paper, the framework is used to perform type inference on specific programs, to justify meaning-preserving code transformations, and to "stage" an action semantics definition of a programming language into a static semantics stage and a dynamic semantics stage.

1 Introduction

Perhaps the most distinctive aspect of action semantics is its structure of facets. The facets provide a "road map" to the nature of a programming language, and in this paper we show how the internal structure of the facets also indicate the kinds of analyses that can be undertaken upon the language. In particular, the subsorting hierarchy of a facet specifies a hierarchy of properties of the facet.

Actions can be viewed as operations upon values from facets. We encode the actions' operations as sequents in a logic. In addition to providing a simple presentation, the logic lets us encode the extensibility feature of action semantics as a

*Fukushima 965-80, Japan, kg-doh@u-aizu.ac.jp
†Manhattan, Kansas 66506, U.S.A., schmidt@cis.ksu.edu . Supported by NSF Grant CCR-93-?.

weakening rule in the logic. The sequent-based format lets us state simple descriptions of operational semantics of actions, property extraction, and action equivalence. In particular, much of the technical requirements of abstract interpretation come "for free" in the representation. Finally, a staging analysis on action-semantics-coded language definitions can be undertaken.

The theme arising from this work is that the facet structure indicates the primary features of a language and guides the user and implementor to important properties and equivalences.

The structure of this paper goes as follows: Section 2 describes the facets and their orderings. Section 3 defines the inference system for actions and gives examples. Section 4 defines action equivalence in a given context and in context families. Section 5 explains the relationship between abstract interpretation and our framework. Section 6 adapts the framework to analyze semantics definitions for staging. The last section concludes the paper.

2 Facets of Actions

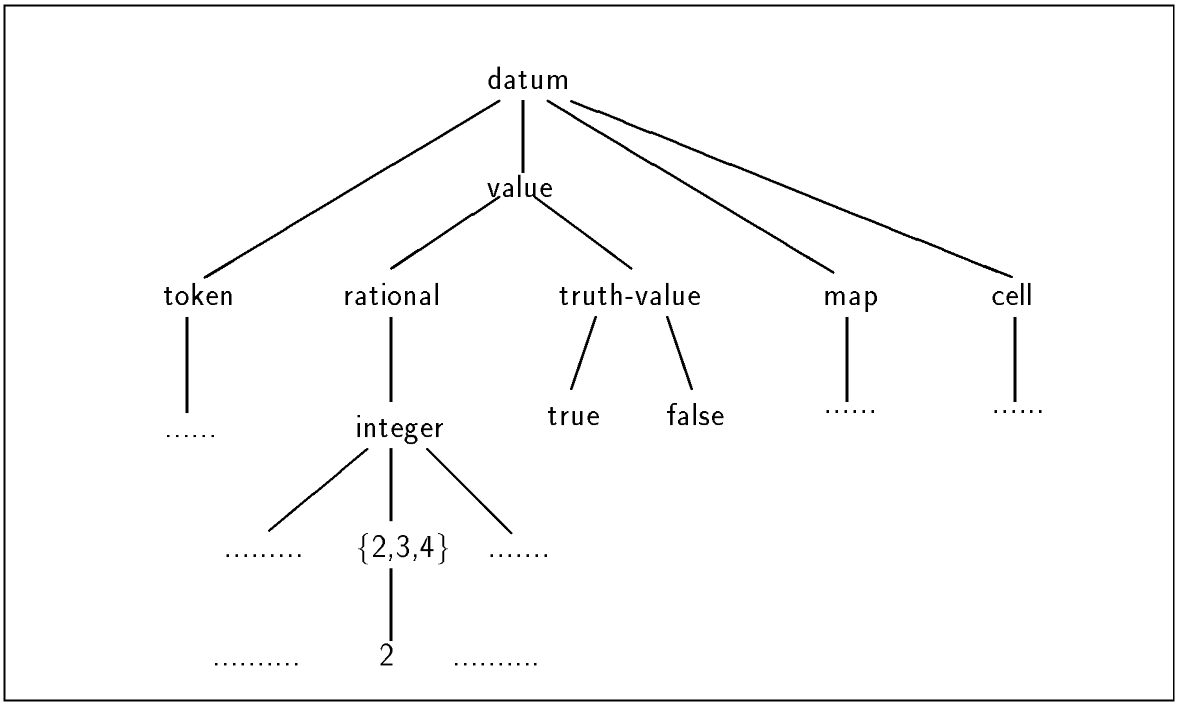


Figure 1: Sorts in Functional Facet

Data in action notation are organized into facets [8, 10]. The *functional facet* contains temporary values ("transient information") that are organized into the sorts (types) value, rational, integer, {2,3,4}, 2, truth-value, true, false, cell, token, etc. Notice that an "element", like 2, is also a sort, 2 [7]. (read as {2} if you wish.) The sorts are ordered based on subsort(subset) inclusion. Figure 1 shows a possible ordering of sorts in the functional facet. We use the notation $\leq$ for subsort ordering. For example, $2 \leq \{2,3,4\} \leq \mathsf{integer} \leq \mathsf{rational} \leq \mathsf{value} \leq \mathsf{datum}$.

The *declarative facet* contains (identifier,functional-facet-sort) bindings ("scoped information"), which can also be considered as records [1, 5]. Figure 2 shows a sample declarative facet. For example, {x=2, y=true}, is a record where x binds to 2 and y to true. Similarly, {x=integer, y=truth-value}, is a record including at least two fields, x and y, where x binds to integer and y to truth-value. This record can also be read as the sort of those records that binds x to some integer and y to some truth value. The records are ordered so that $\rho_1 \leq \rho_2$ iff for every $(\mathsf{t} = v_2) \in \rho_2$, there is a $(\mathsf{t} = v_1) \in \rho_1$ such that $v_1 \leq v_2$. For example, $\{\mathsf{a{=}2,b{=}true}\} \leq \{\mathsf{a{=}integer,b{=}true}\} \leq \{\mathsf{a{=}value,b{=}truth\text{-}value}\} \leq \{\mathsf{a{=}value}\} \leq \{\}$.

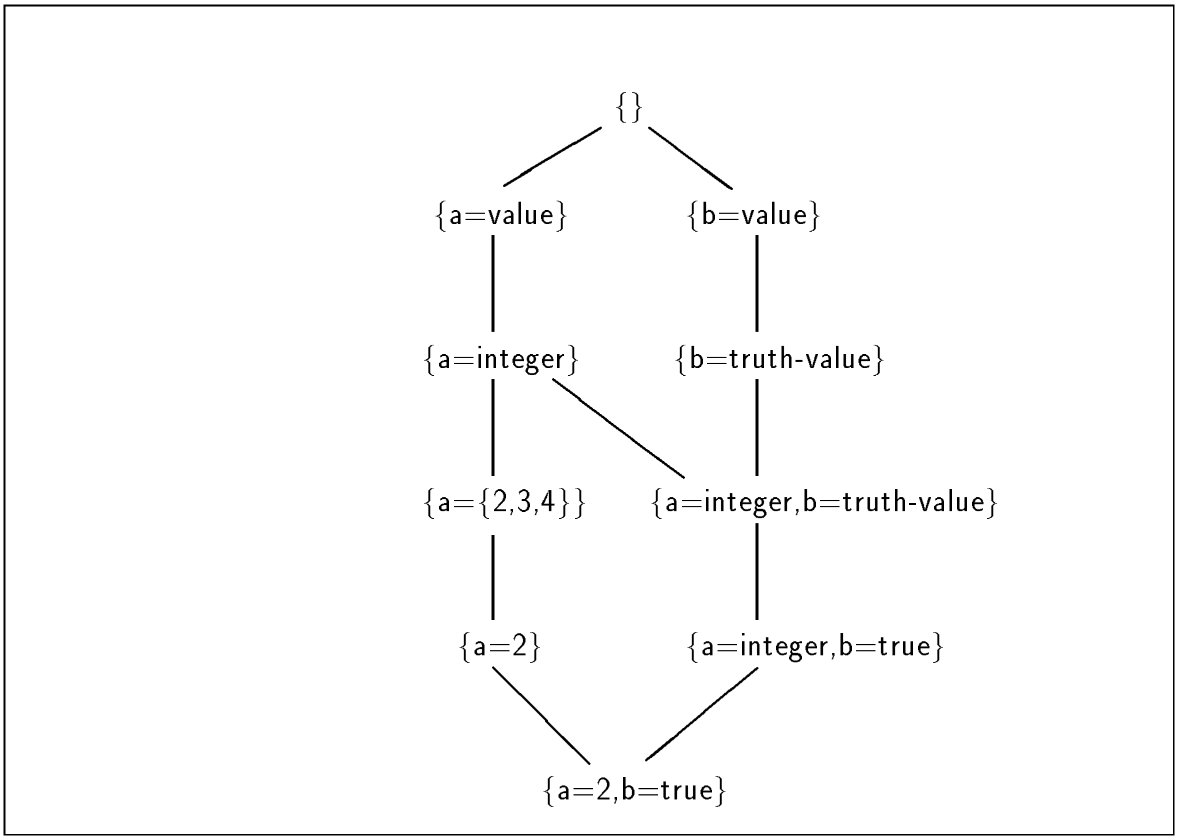
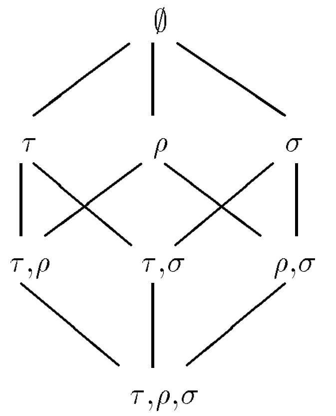


Figure 2: Some Sorts in Declarative Facet

The imperative facet contains storage structures. The storage structure is represented as a map from cell to value or uninitialized where $\mathsf{uninitialized} \leq \mathsf{datum}$. Cells are also ordered: $\mathsf{truth\text{-}value\text{-}cell} \leq \mathsf{cell}$, $\mathsf{integer\text{-}cell} \leq \mathsf{rational\text{-}cell} \leq \mathsf{cell}$, and so on.

Actions often regive values from combinations of facets, so it is helpful to give a structure for these combinations. See Figure 3. Let Γ be an element of the above lattice; an example is 2,{x=2}, which is a τ, ρ element. We call such an element a *context*. We give ordering to contexts as follows:



$\emptyset \in$ Basic-facet
$\tau \in$ Functional-facet
$\rho \in$ Declarative-facet
$\sigma \in$ Imperative-facet

Figure 3: Facet Hierarchy

Definition 2.1 (Context ordering) *Let Γ_1 and Γ_2 be contexts. $\Gamma_1 \preceq \Gamma_2$ if (facets of Γ_2) $\subseteq$ (facets of Γ_1) and for every $\gamma_2 \in \Gamma_2$, $\gamma_1 \leq \gamma_2$ where γ_1 is the corresponding facet element of Γ_1.*

For example, 2,{x=true} $\preceq$ 2,{x=truth-value} $\preceq$ integer $\preceq$ (). (The element () is the only element in the basic facet.)

3 Inference Rules for Actions

An action needs input from facets to perform: for example, when a functional-facet element, 2, and a declarative-facet element, {x=3}, are provided as inputs, the action give sum(it,the integer bound to x) outputs 5. But what if integer and {x=integer} are inputed to this action? The output should be integer. This might arise in type inference. We formalize the semantics of actions by presenting inference rules that can

$$
\begin{aligned}
\textsf{Action} = {} & \textsf{complete} \;\mathbf{|}\; \textsf{regive} \;\mathbf{|}\; [\![\textsf{give Yielder}]\!] \;\mathbf{|} \\
& [\![\textsf{bind Yielder to Yielder}]\!] \;\mathbf{|}\; \textsf{rebind} \;\mathbf{|} \\
& [\![\textsf{store Yielder in Yielder}]\!] \;\mathbf{|}\; [\![\textsf{allocate Yielder}]\!] \;\mathbf{|} \\
& [\![\textsf{Yielder then either Action or Action}]\!] \;\mathbf{|} \\
& [\![\textsf{Action and Action}]\!] \;\mathbf{|} \\
& [\![\textsf{Action and then Action}]\!] \;\mathbf{|} \\
& [\![\textsf{Action then Action}]\!] \;\mathbf{|} \\
& [\![\textsf{furthermore Action hence Action}]\!] \;\mathbf{|} \\
& [\![\textsf{Action before Action}]\!]
\end{aligned}
$$

$$
\begin{aligned}
\textsf{Yielder} = {} & [\![C \{ \textsf{Yielder}_i \}^*_{i \geq 0}]\!] \;\mathbf{|}\; \textsf{it} \;\mathbf{|}\; [\![\textsf{the given Data}]\!] \;\mathbf{|} \\
& [\![\textsf{the given Data} \;\#\; \textsf{natural}]\!] \;\mathbf{|} \\
& [\![\textsf{the Data bound to Yielder}]\!] \;\mathbf{|} \\
& [\![\textsf{the Data stored in Yielder}]\!] \;\mathbf{|} \\
& [\![\textsf{Yielder , Yielder}]\!] \;\mathbf{|}\; [\![\textsf{(Yielder)}]\!]
\end{aligned}
$$

$$
\begin{aligned}
\textsf{Data} = {} & \textsf{datum} \;\mathbf{|}\; \textsf{value} \;\mathbf{|}\; \textsf{truth-value} \;\mathbf{|}\; \textsf{integer} \;\mathbf{|} \\
& \textsf{rational} \;\mathbf{|}\; \textsf{cell} \;\mathbf{|}\; \textsf{truth-value-cell} \;\mathbf{|}\; \textsf{integer-cell} \;\mathbf{|} \\
& \textsf{rational-cell} \;\mathbf{|}\; \textsf{token} \;\mathbf{|}\; \cdots \;\mathbf{|}\; \{2,3,4\} \;\mathbf{|}\; \cdots \;\mathbf{|}\; 2 \;\mathbf{|}\; \cdots
\end{aligned}
$$

Figure 4: Syntax of Action Notation

describe computation as well as static analysis in the same framework. The syntax of the actions we deal with is defined in Figure 4.

In the inference system, a sequent $\Gamma \vdash A \in \gamma$ reads as "within context Γ, action A has sort γ." Here are some sound examples:

()	$\vdash$ give 2	$\in$ datum
()	$\vdash$ give 2	$\in$ integer
()	$\vdash$ give 2	$\in$ 2
{x=3}	$\vdash$ give 2	$\in$ 2
2,{x=3}	$\vdash$ give sum(it,the integer bound to x)	$\in$ 5
2,{x=3}	$\vdash$ give sum(it,the integer bound to x)	$\in$ integer
2,{x=integer}	$\vdash$ give sum(it,the integer bound to x)	$\in$ datum

As in the above examples, an action can have many possible Γ,γ pairs. However, for fixed Γ_0, there is a *least* Γ_0 such that $\Gamma_0 \vdash A \in \gamma_0$ holds. We call this the "*least sorting property*". The action give 2 can have sorts, datum, integer, 2, etc., in context (), but it has the least sort, 2. Similarly, an action, give sum(it,the integer bound to x), has least sort integer in context 2,{x=integer}.

The inference rules for yielders are presented in Figure 5. Constants with no arguments, e.g., rational, integer, 3, true, etc., yield themselves. For operations with more than one argument, rule (Y2) says that if $Y_1, \ldots, Y_n$ have sorts $\tau_1, \ldots, \tau_n$, then $[\![\ C(Y_1, \ldots, Y_n) \ ]\!]$ has sort $\overline{C}(\tau_1, \ldots, \tau_n)$, where $\overline{C}$ stands for sorting operation for C. For example, $\overline{\text{sum}}$ (3,4) yields 7, $\overline{\text{sum}}$ (integer,4) yields integer, and so on. In the functional context, τ, it yields τ and $[\![$ the given D $]\!]$ yields τ if $\tau \leq D$ is true. A constraint, $\tau \leq D$, ensures that the given datum is a subsort of D. The inference rule for $[\![$ the D bound to Y $]\!]$ asserts that in context Γ, Y must yield a token t, a binding $(t = \tau)$ must be found in the context Γ, and $\tau \leq D$ must hold for $[\![$ the D bound to Y $]\!]$ to yield τ. The other rules work similarly. Notice that a rule like (Y3) requires a context of merely τ. For example, $2 \vdash$ it $\in 2$ holds. but we require $2, \{x = 2\} \vdash$ it $\in 2$ to hold as well. To obtain this, we add rule (Y10), a "weakening" rule, which safely expands a context without altering the underlining deduction. The weakening rule also appears crucial to understanding an important modularity principle of action semantics: the understanding of an action is not altered if the action is embedded in an extended context. For example, the semantics of the action, it, should be unaltered if the context, 2, is enriched to 2,{x=2} due to extensions in the language's design.

Figure 6 defines inference rules for actions. The rules are read like the ones in Figure 5. The rules for the combinators and, and then, etc., assume that the action is interference-free, that is, the action is atomic. Interference is considered at the end of the paper. For reasons to be made clear shortly, or is replaced by then either – or.

These inference rules define the operational semantics of actions as well as property extraction. For example, consider an action, give the integer bound to x. Given the

$$\text{(Y1)}\quad \frac{}{() \vdash C \in C}$$

$$\text{(Y2)}\quad \frac{\Gamma \vdash Y_1 \in \tau_1 \quad \cdots \quad \Gamma \vdash Y_n \in \tau_n}{\Gamma \vdash [\![\, C(Y_1 : \textsf{Yielder}, \ldots, Y_n : \textsf{Yielder}) \,]\!] \in \overline{C}(\tau_1, \ldots, \tau_n)}$$

$$\text{(Y3)}\quad \frac{}{\tau \vdash \textsf{it} \in \tau}$$

$$\text{(Y4)}\quad \frac{\tau \leq D}{\tau \vdash [\![\, \textsf{the given } D \textsf{: Data} \,]\!] \in \tau}$$

$$\text{(Y5)}\quad \frac{\tau' = \textsf{component\#}\ n\ \tau \qquad \tau' \leq D}{\tau \vdash [\![\, \textsf{the given } D \textsf{: Data \#}\ n \textsf{: natural} \,]\!] \in \tau'}$$

$$\text{(Y6)}\quad \frac{\Gamma \vdash Y \in t \qquad t \leq \textsf{token} \qquad \Gamma \leq \{t = \tau\} \qquad \tau \leq D}{\Gamma \vdash [\![\, \textsf{the } D \textsf{: Data bound to } Y \textsf{: Yielder} \,]\!] \in \tau}$$

$$\text{(Y7)}\quad \frac{\Gamma, \sigma \vdash Y \in \tau \qquad \tau \leq D\textsf{cell} \qquad \mathit{allocated?}(\tau, \sigma) \qquad \mathit{initialized?}(\tau, \sigma)}{\Gamma, \sigma \vdash [\![\, \textsf{the } D \textsf{: Data stored in } Y \textsf{: Yielder} \,]\!] \in \sigma \textsf{ at } \tau}$$

$$\text{(Y8)}\quad \frac{\Gamma \vdash Y_1 \in \tau_1 \qquad \Gamma \vdash Y_2 \in \tau_2}{\Gamma \vdash [\![\, Y_1 : \textsf{Yielder}, Y_2 : \textsf{Yielder} \,]\!] \in (\tau_1, \tau_2)}$$

$$\text{(Y9)}\quad \frac{\Gamma \vdash Y \in \tau}{\Gamma \vdash [\![\, (\ Y : \textsf{Yielder}\ ) \,]\!] \in \tau}$$

$$\text{(Y10)}\quad \frac{\Gamma_1 \leq \Gamma_2 \qquad \Gamma_2 \vdash Y \in \tau_2 \qquad \tau_2 \leq \tau_1}{\Gamma_1 \vdash Y \in \tau_1} \qquad (\mathit{weakening\ rule})$$

Figure 5: Rules for Yielders

$$\text{(A1)}\ \frac{}{\emptyset \vdash \mathsf{complete} \in ()} \qquad \text{(A2)}\ \frac{}{\tau \vdash \mathsf{regive} \in \tau}$$

$$\text{(A3)}\ \frac{\Gamma \vdash Y \in \tau}{\Gamma \vdash \llbracket\, \mathsf{give}\ Y\!:\!\mathsf{Yielder} \,\rrbracket \in \tau} \qquad \text{(A4)}\ \frac{}{\rho \vdash \mathsf{rebind} \in \rho}$$

$$\text{(A5)}\ \frac{\Gamma \vdash Y_1 \in t \quad t \leq \mathsf{token} \quad \Gamma \vdash Y_2 \in \tau \quad \tau \leq \mathsf{value}}{\Gamma \vdash \llbracket\, \mathsf{bind}\ Y_1\!:\!\mathsf{Yielder}\ \mathsf{to}\ Y_2\!:\!\mathsf{Yielder} \,\rrbracket \in \{t = \tau\}}$$

$$\text{(A6)}\ \frac{\Gamma, \sigma \vdash Y_1 \in \tau_1 \quad \tau_1 \leq \mathsf{value} \quad \Gamma, \sigma \vdash Y_2 \in \tau_2 \quad \tau_2 \leq \mathsf{cell} \quad \mathit{compatible?}(\tau_1, \tau_2) \quad \mathit{allocated?}(\tau_2, \sigma)}{\Gamma, \sigma \vdash \llbracket\, \mathsf{store}\ Y_1\!:\!\mathsf{Yielder}\ \mathsf{in}\ Y_2\!:\!\mathsf{Yielder} \,\rrbracket \in \mathsf{overlay}(\mathsf{map}\ \tau_2\ \mathsf{to}\ \tau_1, \sigma)}$$

$$\text{(A7)}\ \frac{\Gamma, \sigma \vdash Y \in \tau \quad \tau \leq \mathsf{cell}}{\Gamma, \sigma \vdash \llbracket\, \mathsf{allocate}\ Y\!:\!\mathsf{Yielder} \,\rrbracket \in \tau, \mathsf{overlay}(\mathsf{map}\ \tau\ \mathsf{to\ uninitialized}, \sigma)}$$

$$\text{(A8)}\ \frac{\Gamma \vdash Y \in \mathsf{true} \quad \Gamma \vdash A_1 \in \gamma}{\Gamma \vdash \llbracket\, Y\!:\!\mathsf{Yielder}\ \mathsf{then\ either}\ A_1\!:\!\mathsf{Action}\ \mathsf{or}\ A_2\!:\!\mathsf{Action} \,\rrbracket \in \gamma}$$

$$\text{(A9)}\ \frac{\Gamma \vdash Y \in \mathsf{false} \quad \Gamma \vdash A_2 \in \gamma}{\Gamma \vdash \llbracket\, Y\!:\!\mathsf{Yielder}\ \mathsf{then\ either}\ A_1\!:\!\mathsf{Action}\ \mathsf{or}\ A_2\!:\!\mathsf{Action} \,\rrbracket \in \gamma}$$

$$\text{(A10)}\ \frac{\Gamma \vdash A_1 \in \tau_1, \rho_1 \quad \Gamma \vdash A_2 \in \tau_2, \rho_2}{\Gamma \vdash \llbracket\, A_1\!:\!\mathsf{Action}\ \mathsf{and}\ A_2\!:\!\mathsf{Action} \,\rrbracket \in (\tau_1, \tau_2), \mathsf{disjoint\ union}(\rho_1, \rho_2)}$$

$$\text{(A11)}\ \frac{\Gamma \vdash A_1 \in \gamma_1, \sigma_1 \quad \Gamma, \sigma_1 \vdash A_2 \in \gamma_2, \sigma_2}{\Gamma \vdash \llbracket\, A_1\!:\!\mathsf{Action}\ \mathsf{and\ then}\ A_2\!:\!\mathsf{Action} \,\rrbracket \in \gamma_2, \sigma_2}$$

$$\text{(A12)}\ \frac{\Gamma \vdash A_1 \in \tau \quad \Gamma, \tau \vdash A_2 \in \gamma}{\Gamma \vdash \llbracket\, A_1\!:\!\mathsf{Action}\ \mathsf{then}\ A_2\!:\!\mathsf{Action} \,\rrbracket \in \gamma}$$

$$\text{(A13)}\ \frac{\Gamma, \rho \vdash A_1 \in \rho_1, \sigma_1 \quad \Gamma, \mathsf{overlay}(\rho_1, \rho), \sigma_1 \vdash A_2 \in \gamma}{\Gamma, \rho \vdash \llbracket\, \mathsf{furthermore}\ A_1\!:\!\mathsf{Action}\ \mathsf{hence}\ A_2\!:\!\mathsf{Action} \,\rrbracket \in \gamma}$$

$$\text{(A14)}\ \frac{\Gamma, \rho \vdash A_1 \in \rho_1, \sigma_1 \quad \Gamma, \mathsf{overlay}(\rho_1, \rho), \sigma_1 \vdash A_2 \in \rho_2, \sigma_2}{\Gamma, \rho \vdash \llbracket\, A_1\!:\!\mathsf{Action}\ \mathsf{before}\ A_2\!:\!\mathsf{Action} \,\rrbracket \in \mathsf{overlay}(\rho_2, \rho_1), \sigma_2}$$

$$\text{(A15)}\ \frac{\Gamma_1 \leq \Gamma_2 \quad \Gamma_2 \vdash A \in \gamma_2 \quad \gamma_2 \leq \gamma_1}{\Gamma_1 \vdash A\!:\!\mathsf{Action} \in \gamma_1} \qquad (\mathit{weakening\ rule})$$

Figure 6: Rules for Actions

input facet, $\Gamma_0 = \{\mathsf{x{=}2,y{=}true}\}$, the following proof tree can be drawn to calculate the operational semantics of the action:

$$
\dfrac{\dfrac{() \vdash \mathsf{x} \in \mathsf{x}}{\Gamma_0 \vdash \mathsf{x} \in \mathsf{x}}\,\text{(weakening)} \qquad \mathsf{x} \leq \mathsf{token} \qquad \Gamma_0 \leq \{\mathsf{x} = 2\} \qquad 2 \leq \mathsf{integer}}{\dfrac{\Gamma_0 \vdash \mathsf{the\ integer\ bound\ to\ x} \in 2}{\Gamma_0 \vdash \mathsf{give\ the\ integer\ bound\ to\ x} \in 2}}
$$

This tree indeed shows that, given context Γ_0, the action produces 2. The same tree can be drawn for property extraction (e.g., type checking). Given input facet, $\Gamma_1 = \{\mathsf{x} = \mathsf{integer,y{=}truth\text{-}value}\}$, we can calculate:

$$
\dfrac{\dfrac{() \vdash \mathsf{x} \in \mathsf{x}}{\Gamma_0 \vdash \mathsf{x} \in \mathsf{x}}\,\text{(weakening)} \qquad \mathsf{x} \leq \mathsf{token} \qquad \Gamma_0 \leq \{\mathsf{x} = \mathsf{integer}\} \qquad \mathsf{integer} \leq \mathsf{integer}}{\dfrac{\Gamma_0 \vdash \mathsf{the\ integer\ bound\ to\ x} \in \mathsf{integer}}{\Gamma_0 \vdash \mathsf{give\ the\ integer\ bound\ to\ x} \in \mathsf{integer}}}
$$

When we write $\Gamma \vdash A \in \gamma$, we assert existence of derivation with $\Gamma \vdash A \in \gamma$ at the root. In practice, we can draw the derivation tree like a Prolog goal tree and apply the weakening rule only when absolutely necessary (i.e., as close to leaves of the tree as possible). Alternatively, we can view the derivation tree as a parse tree decorated with inherited (Γ) and synthesized (γ) attributes at the nodes.

A usual property is the *least sorting property*: for a give context, Γ_0, an action, A, has a least sort, Γ_0: $\Gamma_0 \vdash A \in \gamma_0$ holds, and for all γ such that Γ_0: $\Gamma_0 \vdash A \in \gamma$ holds, $\gamma_0 \leq \gamma$. The least sorting property holds for the rules in Figure 5 and 6. (But it can fail when the usual version of or is added, e.g., $() \vdash \mathsf{give\ 1\ or\ give\ 2} \in \mathsf{1}$ and $() \vdash \mathsf{give\ 1\ or\ give\ 2} \in \mathsf{2}$, but neither 1 nor 2 is least.) This is exploited in the next section.

4 Equivalence of Actions

Applications such as compiling and code improvement require the notion of equivalence of actions [9]. We write $\Gamma \vdash A_1 \equiv A_2 \in \gamma$ to state that $\Gamma \vdash A_1 \in \gamma$ if and only if $\Gamma \vdash A_2 \in \gamma$. Two actions, A_1 and A_2, are equivalent in a context, Γ, if they have the same properties. Formally stated, we write $\Gamma \vdash A_1 \equiv A_2$ to state $\forall\gamma$,

$\Gamma \vdash A_1 \equiv A_2 \in \gamma$. When $\Gamma \vdash A_1 \equiv A_2$ holds, then A_1 and A_2 are interchangeable in context Γ. The chore of checking for equivalence is greatly simplified by the least sorting property: $\Gamma \vdash A_1 \equiv A_2$ holds if and only if $\Gamma \vdash A_1 \equiv A_2 \in \gamma_0$ where γ_0 is the least sort for A_1 and A_2 with respect to Γ. For example, {x=2} $\vdash$ give the integer bound to x $\equiv$ give 2 holds, since the least sorts for both actions is 2. This allows the simplification of the action bind x to 2 hence give the integer bound to x into bind x to 2 hence give 2. Since () $\vdash$ bind x to 2 hence give 2 $\equiv$ give 2 holds, we can simplify the original action to just give 2. Such transformations are common during compile-time processing of a program.

There is another kind of equivalence, with respect to a family of contexts. To motivate this equivalence, consider the equivalence integer $\vdash$ give it $\equiv$ give sum (it,it). Although the two actions are equivalent with respect to context integer, this does *not* imply, for an integer, n, that $n \vdash$ give it $\equiv$ give sum (it,it) holds. Therefore, a different form of equivalence is needed.

Let a symbolic expression, like $\tau \leq$ integer, refer to a set of contexts, namely those contexts τ such that $\tau \leq$ integer holds. For our purposes, a *symbolic sort expression* is a sort expression containing placeholders, possibly constrained by inequations. Examples are τ, τ *if* $\tau \leq$ integer (written as $\tau \leq$ integer for short), and {x=τ,y=τ'} *if* $\tau \leq \tau'$. Similarly, a *symbolic context expression* is a context consisting of symbolic sort expressions.

We can use the inference rules in Figures 5 and 6 to construct derivations of the form $\Gamma \vdash A \in \gamma$, where Γ and γ are symbolic contexts expressions and symbolic sort expressions, respectively. For example, we can derive $\tau \leq$ integer $\vdash$ give the given integer $\in \tau$, since $\tau \leq$ integer lets us build the proof tree.

A crucial property of derivations with symbolic expressions is soundness. The inference rules are *sound* because, for all ground substitutions, $\mathcal{U}$, if $\Gamma \vdash A \in \gamma$ holds, then so does $\mathcal{U}\Gamma \vdash A \in \mathcal{U}\gamma$. (A substitution, $\mathcal{U}$, is ground, for γ if $\mathcal{U}\gamma$ contains no placeholders. $\mathcal{U}$ is ground for Γ in a similar way. Also, $\mathcal{U}$ must make all constraints true.) A second, important property is principal sorting. For symbolic context expression, Γ, γ is the *principal sort*, if for all ground substitutions, $\mathcal{U}$, $\mathcal{U}\gamma$ is the least sort for A with respect to context $\mathcal{U}\Gamma$. With the aid of soundness and an algorithm that calculates principal sorting, we have this result:

Definition 4.1 *For symbolic context expression, Γ, if $\Gamma \vdash A_1 \in \gamma$ and $\Gamma \vdash A_2 \in \gamma$ hold, then for all ground substitutions $\mathcal{U}$, $\mathcal{U}\Gamma \vdash A_1 \equiv A_2$.*

The algorithm for calculating principal sorting should be obvious, but there is the problem that principal sorting is not upheld by the rules for conditional choice. Consider the example it then either give 1 or give 2 in the context $\tau \leq$ truth-value. The rule for the conditional gives us $1 \sqcup 2 = \{1, 2\}$, which is sound but not principal. A principal scheme would be $(\tau \leq \mathit{true} \rightarrow 1) \sqcup (\tau \leq \mathit{false} \rightarrow 2)$, but adding conditional schemes is computationally prohibitive.

Fortunately, sound but nonprincipal schemes can be used in code improvement. Since {1,2} is sound for the above example, and $\tau' \leq$ {1,2} $\vdash$ give the given integer $\in \tau'$ is sound *and* principal, we have that $\tau \leq$ truth-value $\vdash$ (it then either give 1 or give 2) then give the given integer $\equiv$ (it then either give 1 or give 2) then give it. This kind of transformation is common within compilation algorithms.

5 Comparison to Abstract Interpretation

The development of equivalence with respect to symbolic contexts has a strong relationship to cousot-cousot-style abstract interpretation [2]. In this section, we try to explain this relationship. An analysis by means of abstract interpretation is undertaken in four stages:

1. For the given domain of concrete, computational values, $\mathcal{C}$ (e.g., $\mathcal{C} = (\mathcal{N}+\mathcal{B})_\perp$), we formulate a domain of abstract values, $\mathcal{A}$ (e.g., $\mathcal{A} = \{\perp, nat, bool, \top\}$, ordered in the usual way).

2. Next, we define abstraction and concretization maps, *abs*: $\mathcal{P}_b\mathcal{C} \rightarrow \mathcal{A}$ and *conc*: $\mathcal{A} \rightarrow \mathcal{P}_b\mathcal{C}$, respectively, such that *abs* and *conc* form a Galois connection [6]. Note that $\mathcal{P}_b\mathcal{C}$ is the lower powerdomain of $\mathcal{C}$. (For example, *abs* is the lifting of abs_0: $\mathcal{C} \rightarrow \mathcal{A}$, $abs_0(\perp) = \perp$, $abs_0(n) = nat$, $abs_0(b) = bool$, and *conc* is $conc(\perp) = \{\perp\}$, $conc(nat) = \mathcal{N}_\perp$, $conc(bool) = \mathcal{B}_\perp$, and $conc(\top) = \mathcal{C}$.)

3. Then, we define the abstract interpretation, $[\![\cdot]\!]_{abs}$, and prove that it is safe with respect to the concrete ("standard") interpretation, $[\![\cdot]\!]_{conc}$. The safety criterion is: $conc([\![E]\!]_{abs}(abs\ \Gamma)) \sqsupseteq [\![E]\!]_{conc}\ \Gamma$, for program E and context Γ.

4. Finally, we prove that the desired program transformation can be performed based on the results of the abstract interpretation. So, we rewrite $[\![\cdot]\!]_{conc}$ into a "cache semantics" or "sticky semantics", which remembers for each subphrase E_0, of E, the abstract context, Γ_0, that appears at E_0. A correspondence between the sticky semantics and the concrete semantics must be proved. Finally, if we wish to replace E_0 by E_1, we must prove, for Γ_0, the context that appears at E_0, for all concrete contexts $\Gamma \in conc\ \Gamma_0$, that $[\![E_0]\!]_{conc}\Gamma = [\![E_1]\!]_{conc}\Gamma$ holds.

The idea described above are straightforward, but the technical detail is heavy. In contrast, in the framework described in the earlier sections, stages 1 through 4 come more easily:

1. The abstract domain's values and the concrete domain's values are the sorts in the facet. (e.g., 2, true, and integer, truth-value.)

2. The relationship between "concrete" and "abstract" values is given already by the subsorting relation (e.g., $2 \leq$ integer). Indeed, a symbolic sort expression like $\tau \leq$ integer defines a down-closed set – an element of a lower power-domain.

3. The safety of the "abstract" analysis follows from the monotonicity of the inference rules. (For example, since $2 \leq \mathsf{integer}$, it must be that $\tau_1 \leq \tau_2$ in $2 \vdash A_0 \in \tau_1$ and $\mathsf{integer} \vdash A_0 \in \tau_2$.)

4. The "sticky" semantics is given by a derivation $\Gamma_0 \vdash A_0 \in \gamma_0$ (Γ_0 is the abstract context of A_0), and the condition under which A_0 can be transformed to A_1 is $\Gamma_0 \vdash A_0 \equiv A_1$.

We must emphasize that the framework in this paper contains no short-cuts or "magic" – the same steps as those required for abstract-interpretation-based code transformations are required here, but the structuring of the facets makes the formulation clearer. This is the main point of the present paper – the structure of a language's facets indicates the purposes, properties and potential equivalences in the language. Note also that the structure of the facets limit us in the properties we can and *cannot* naturally analyze – the facet structure in Figure 1 is natural for type inference, but useless for, say, available expressions analysis. It is the responsibility of the language designer to define facet structures whose sorts indicate the properties of importance of the language.

Finally, the usual issue regarding analysis of while-loops and recursion remain: The analysis of the infinite action represented by a while-loop must be "folded" into a finite action and its analysis.

6 Analysis of Language Definitions

So far we have developed a method of calculating meaning and properties of action denotations of specific programs. In this section, we show that the same framework can be applied to analyze and stage language definitions. In particular, we show that analysis of a language definition extracts a set of static constraints-checking ("type-checking") rules. Those rules constitute the "static semantics" of the language and motivate the construction of a residual ("dynamic") semantics. That is, the language definition has been "staged". The example that follows makes these points clear.

Consider the following semantics equation.

$$\mathsf{evaluate}\ [\![\ \mathsf{eq0}\ E{:}\mathsf{Expression}\ ]\!] = \mathsf{evaluate}\ E\ \mathsf{then\ give\ equal\text{-}to\text{-}zero(the\ given\ integer)}$$

Let's try to build a proof tree of the right-hand side of the equation:

$$\dfrac{\Gamma_0 \vdash \mathsf{evaluate}\ E \in \boxed{?} \qquad \Gamma_0, \boxed{?} \vdash \mathsf{give\ equal\text{-}to\text{-}zero(the\ given\ integer)} \in \boxed{?}}{\Gamma_0 \vdash \mathsf{evaluate}\ E\ \mathsf{then\ give\ equal\text{-}to\text{-}zero(the\ given\ integer)} \in \boxed{?}}\ \text{(A12)}$$

We cannot go on because we do not know what to fill in the box, $\boxed{?}$. However, we can finish the analysis if we guess an induction hypothesis, $\Gamma_0 \vdash \mathsf{evaluate}\ E \in \mathsf{integer}$. The complete proof is:

$$\dfrac{\Gamma_0 \vdash \textsf{evaluate } E \in \textsf{integer} \qquad \dfrac{\dfrac{\dfrac{\textsf{integer} \leq \textsf{integer}}{\Gamma_0, \textsf{integer} \ \vdash \textsf{the given integer} \in \textsf{integer}}\,\text{(Y4)} \qquad \textsf{integer} \leq \textsf{value}}{\Gamma_0, \textsf{integer} \ \vdash \textsf{equal-to-zero(the given integer)} \in \textsf{truth-value}}\,\text{(Y2)}}{\Gamma_0, \textsf{integer} \ \vdash \textsf{give equal-to-zero(the given integer)} \in \textsf{truth-value}}\,\text{(A3)}}{\Gamma_0 \ \vdash \ \textsf{evaluate } E \textsf{ then give equal-to-zero(the given integer)} \in \textsf{truth-value}}\,\text{(A12)}$$

Thus we can assert that if $\Gamma_0 \vdash \textsf{evaluate } E \in \textsf{integer}$ holds, then $\Gamma_0 \vdash \textsf{evaluate}$ $[\![$ $\textsf{eq0}$ E:$\textsf{Expression}$ $]\!] \in \textsf{truth-value}$ holds. This assertion can be reformatted as

$$\frac{\Gamma_0 \vdash \textsf{evaluate } E \in \textsf{integer}}{\Gamma_0 \vdash \textsf{evaluate } [\![\textsf{eq0 } E : \textsf{Expression}]\!] \in \textsf{truth value}} \qquad (1)$$

which is a static typing rule for $[\![\textsf{eq0 } E : \textsf{Expression}]\!]$. Also given this information, we can see that

$$\Gamma_0\textsf{,integer} \vdash \textsf{give equal-to-zero(the given integer)} \equiv \textsf{give equal-to-zero(it)}$$

holds. Therefore, if E satisfies the hypothesis of the typing law, we obtain:

$$\Gamma_0 \vdash [\![\ \textsf{eq0 E:Expression} \ ]\!] \equiv \textsf{evaluate } E\textsf{:Expression then give equal-to-zero(it)} \qquad (2)$$

The semantics equation has been staged into a "static" component (1) and a "dynamic" (actually, a residual) one (2).

Now remaining is the question of how to determine the induction hypothesis. The answer is to use the symbol sort expressions from Section 4. In the above example, we say that $\Gamma_0 \vdash \textsf{evaluate } E \in \tau_0$ holds. Then we draw the proof tree as follows:

$$\dfrac{\Gamma_0 \ \vdash \ \textsf{evaluate } E \ \in \ \tau_0 \qquad \dfrac{\dfrac{\dfrac{\tau_0 \leq \textsf{integer}}{\Gamma_0, \tau_0 \ \vdash \textsf{the given integer} \in \tau_0}\,\text{(Y4)} \qquad \tau_0 \leq \textsf{value}}{\Gamma_0, \tau_0 \ \vdash \textsf{equal-to-zero(the given integer)} \in \textsf{truth-value}}\,\text{(Y2)}}{\Gamma_0, \tau_0 \ \vdash \textsf{give equal-to-zero(the given integer)} \in \textsf{truth-value}}\,\text{(A3)}}{\Gamma_0 \ \vdash \ \textsf{evaluate } E \textsf{ then give equal-to-zero(the given integer)} \in \textsf{truth-value}}\,\text{(A12)}$$

In the process of completing the derivation, the constraint $\tau_0 \leq \textsf{integer}$ is acquired. This makes the induction hypothesis $\Gamma_0 \vdash \textsf{evaluate } E \in \tau_0$ and $\tau_0 \leq \textsf{integer}$. This approach is developed in detail in [4, 3].

7 Conclusion

We have shown the application of the facet structure of action semantics to language definition, interpretation, and analysis. Although no new methods for analysis are given, existing methods become clearer and simpler when expressed within the facet framework.

The development in this paper has been for an interference-free variant of action semantics. We title this variant of action semantics *atomic action semantics*, because the analyses and equivalences assume that the actions are indivisible – atomic. The extension of the action semantics by interference should add another facet (perhaps the communicative one), and as promoted earlier in the paper, the extension should not affect the existing understanding of the atomic actions. Hence, facets and actions for interference are extensions, and a bisimulation equivalence can be defined for the extensions.

Finally, we have not treated divergence and failure as actions' outcomes. Divergence corresponds to nonderivability, but failure should be denoted by the sort, nothing. Unlike the treatment in [8], nothing appears at the top (and not the bottom) of a facet's subsorting hierarchy, due to the need to presence monotonicity of the inference rules.

References

[1] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[3] Kyung-Goo Doh and David A. Schmidt. Extraction of strong typing laws from action semantics definitions. In *ESOP'92, Proc. European Symposium on Programming, Rennes*, volume 582 of *Lecture Notes in Computer Science 349*, pages 151–166. Springer-Verlag, 1992.

[4] Kyung-Goo Doh and David A. Schmidt. Action semantics-directed prototyping. *Computer Languages*, 19(4):213–233, 1993.

[5] Susan Even and David A. Schmidt. Category-sorted algebra-based action semantics. *Theoretical Computer Science*, 77:71–95, 1990.

[6] Austin Melton, David A. Schmidt, and George Strecker. Galois connections and computer science applications. In David H. Pitt et al., editors, *Category Theory and Computer Programming*, number 240 in Lecture Notes in Computer Science, pages 299–312, Guildford, UK, September 1986.

[7] Peter 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.

[8] Peter D. Mosses. *Action Semantics.* Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.

[9] H. Moura and D. A. Watt. Action transformations in the ACTRESS compiler generator. In *CC'94, Proceedings of the 5th International Conference on Compiler Construction, Edniburgh*, Lecture Notes in Computer Science 786. Springer-Verlag, 1994.

[10] David A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.