Basic Research in Computer Science

BRICS NS-94-1 Peter D. Mosses (editor): 1st Workshop on ACTION SEMANTICS

Proceedings of the First International Workshop on ACTION SEMANTICS

14 April 1994, Edinburgh, Scotland

Peter D. Mosses (editor)

BRICS Notes Series **NS-94-1**

ISSN 0909-3206 **May 1994**

See back inner page for a list of recent publications in the BRICS Notes Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Proceedings of the First International Workshop on

ACTION SEMANTICS

14 April 1994 — Edinburgh, Scotland

Peter D. Mosses (editor)

Preface

Actions speak louder than words: Action Semantics is now being used in practical applications! This workshop surveyed recent achievements, demonstrated tools, and coordinated projects. It was open to all.

Brief abstracts of the presentations were handed out at the workshop. Extended abstracts / full papers were collected afterwards and are now published here.

There were 19 participants,[1] all assumed to be familiar with the basic ideas of Action Semantics.[2] A list of the registered participants is given at the end. Most of them also attended some or all of the CAAP / ESOP / CC conferences, of which the workshop was a satellite meeting; but five participants travelled specially to Edinburgh to participate in the workshop.

As can be seen from the workshop programme and from the following papers, a lot of interesting work was presented and discussed during the one day. Special thanks to the invited speakers, Dave Schmidt and Bo Stig Hansen for their stimulating contributions, and to all the authors for keeping closely to a tight schedule not only when giving their talks, but also when preparing their papers for this Proceedings.

The final discussion session revealed plans for exciting new work, and possibilities for further collaboration. A second workshop on action semantics will be held within a year or two; no definite venue has yet been fixed, although one proposal is to hold it as a satellite meeting of TAPSOFT'95 in Aarhus (22–26 May 1995). In the meantime, the action semantics mailing list[3] can be used for reporting new results, further coordination of projects, and for discussing features of action semantics and related frameworks.

The workshop was organised by Peter D. Mosses (BRICS, Dept. of Computer Science, Univ. of Aarhus, Denmark) and David A. Watt (Computing Science Dept., Univ. of Glasgow, Scotland). The workshop organisers thank the organisers of CAAP / ESOP / CC and the support staff at the Department of Computer Science, University of Edinburgh, for the provision of facilities and assistance. They also gratefully acknowledge funding and sponsorship from:

BRICS (Basic Research in Computer Science,
Centre of the Danish National Research Foundation)

COMPASS (ESPRIT Basic Research Working Group 6112)

[1] H. Moura (Brazil) was unable to attend; his paper was presented by D. A. Watt.

[2] A bibliography of published work on Action Semantics is available by anonymous FTP from ftp.daimi.aau.dk in the file pub / action / bibliography / action.bib.

[3] Subscription: send a request marked `AS Mailing List' with your name and e-mail address to pdmosses@daimi.aau.dk.

THURSDAY 14 APRIL 1994 *page*

END OF WORKSHOP

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-93-02962.

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

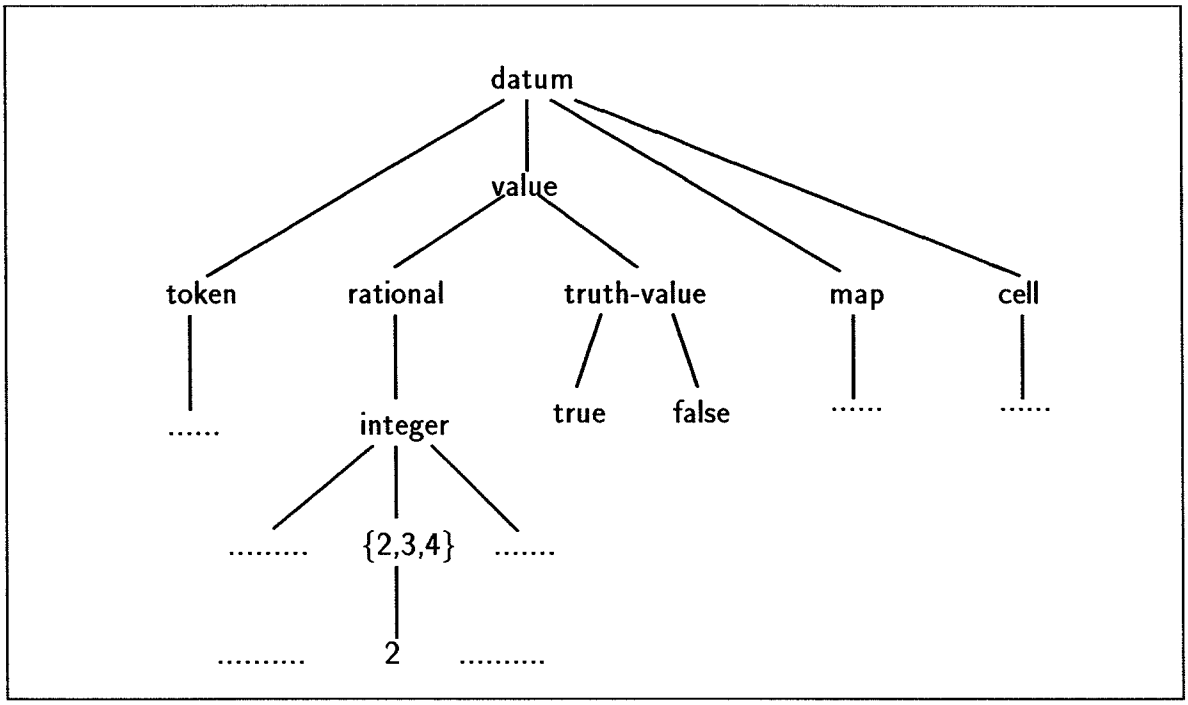


Figure 1: Sorts in Functional Facet

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} $\leq$ integer $\leq$ rational $\leq$ value $\leq$ 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 $(t = v_2) \in \rho_2$, there is a $(t = v_1) \in \rho_1$ such that $v_1 \leq v_2$. For example, {a=2,b=true} $\leq$ {a=integer,b=true} $\leq$ {a=value,b=truth-value} $\leq$ {a=value} $\leq$ {}.

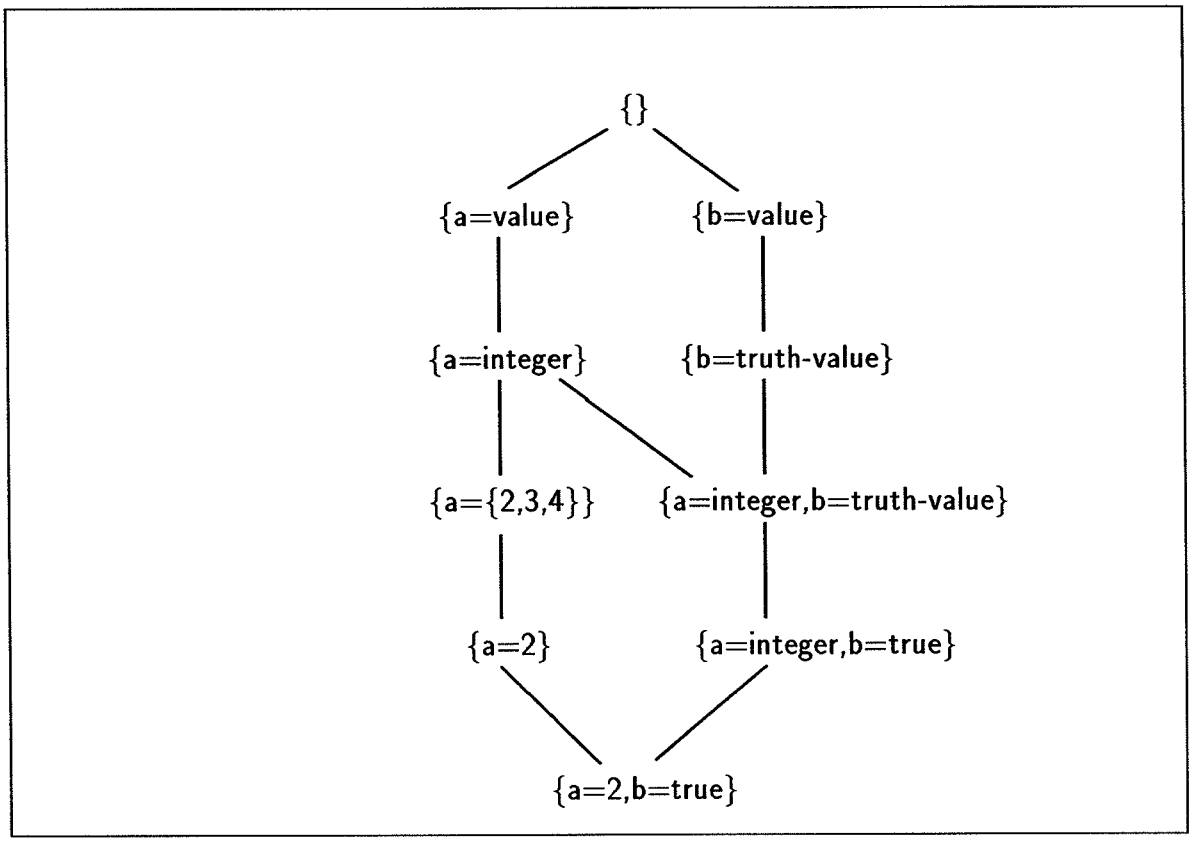


Figure 2: Some Sorts in Declarative Facet

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

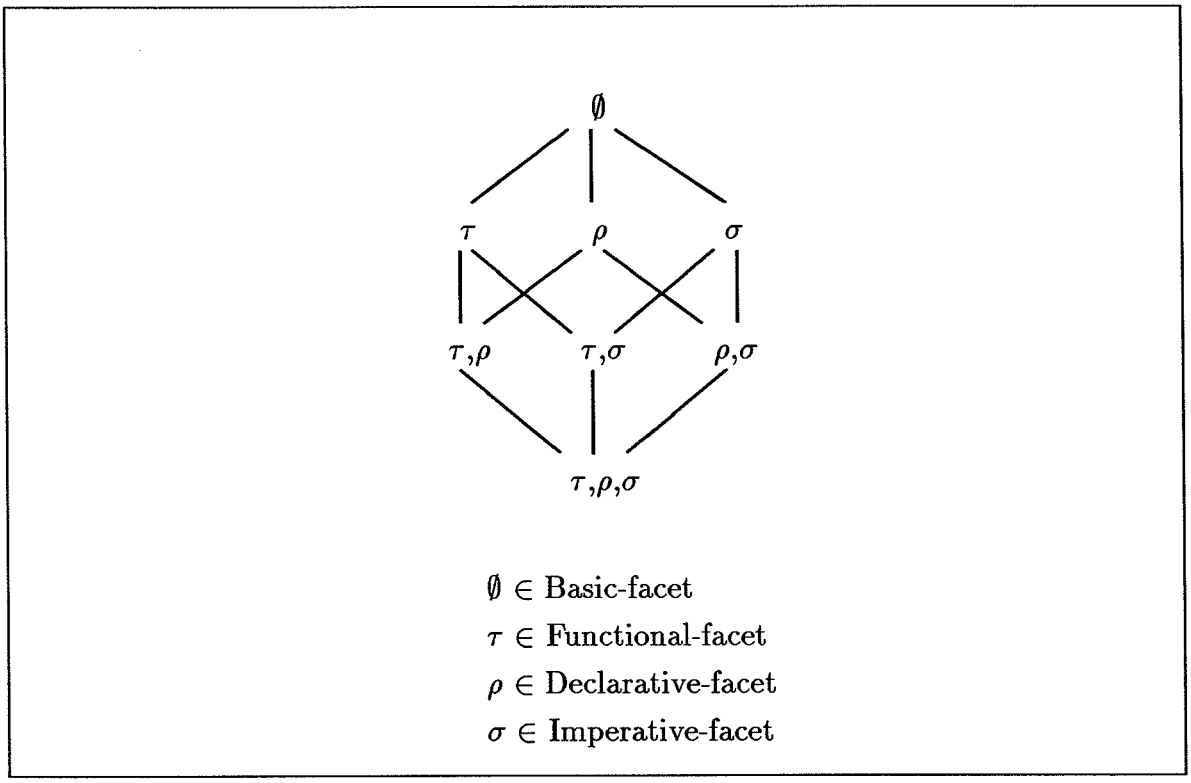


Figure 3: Facet Hierarchy

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

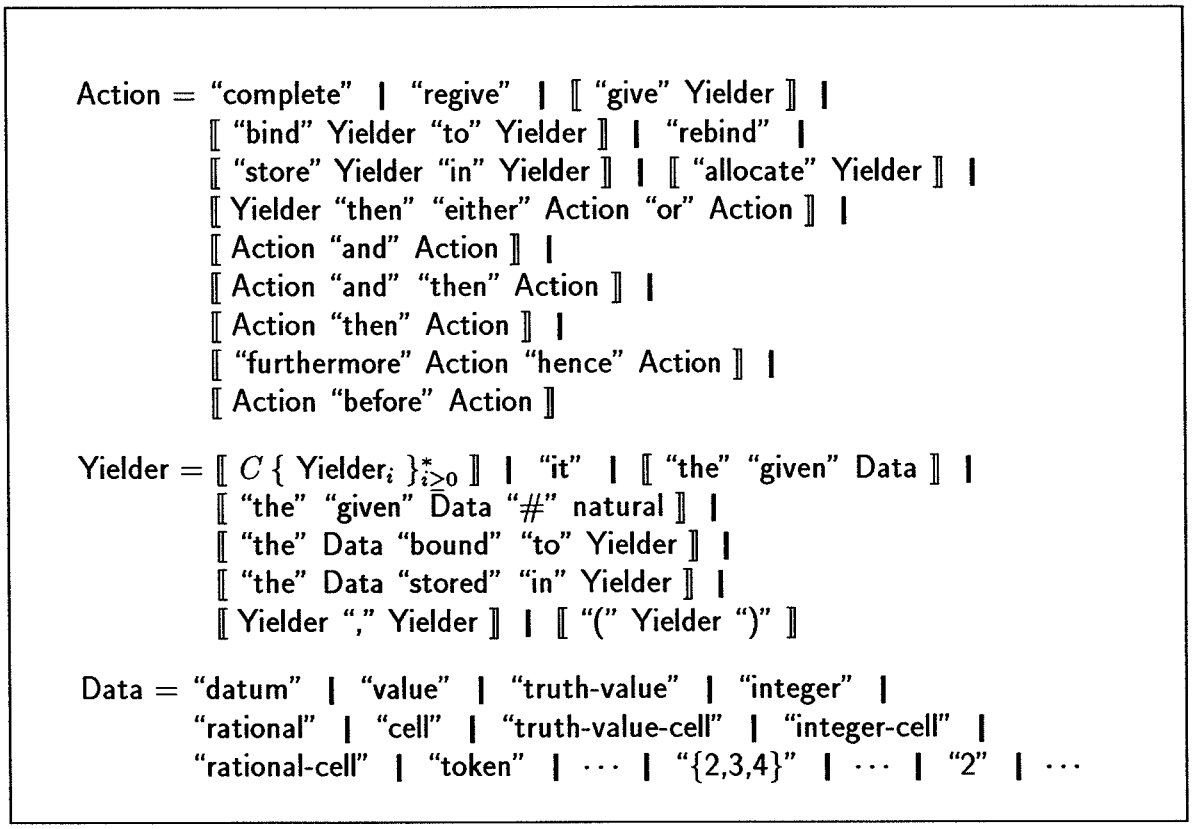


Figure 4: Syntax of Action Notation

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:

()	⊢ give 2	∈ datum
()	⊢ give 2	∈ integer
()	⊢ give 2	∈ 2
{x=3}	⊢ give 2	∈ 2
2,{x=3}	⊢ give sum(it,the integer bound to x)	∈ 5
2,{x=3}	⊢ give sum(it,the integer bound to x)	∈ integer
2,{x=integer}	⊢ give sum(it,the integer bound to x)	∈ 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

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

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

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

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

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

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

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

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

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

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

Figure 5: Rules for Yielders

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

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

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

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

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

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

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

$$\text{(A10)}\ \frac{\Gamma \vdash A_1 \in \tau_1 \quad \Gamma \vdash A_2 \in \tau_2}{\Gamma \vdash [\![\, A_1 : \mathsf{Action\ and}\ A_2 : \mathsf{Action}\ ]\!] \in (\tau_1, \tau_2)}$$

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

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

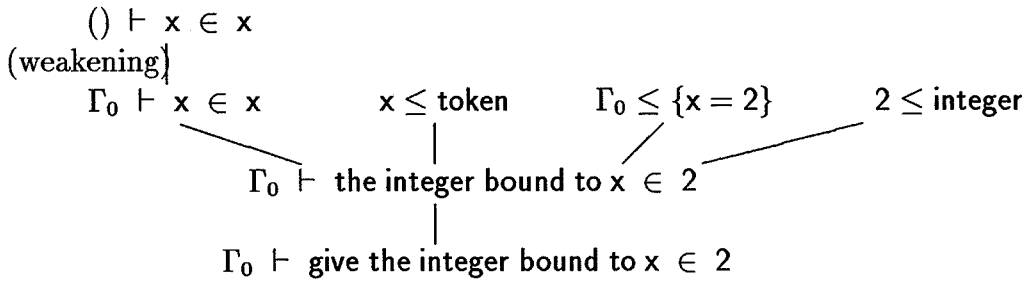
$$\text{(A13)}\ \frac{\Gamma, \rho \vdash A_1 \in \rho_1, \sigma_1 \quad \Gamma, \mathsf{overlay}(\rho_1, \rho), \sigma_1 \vdash A_2 \in \gamma}{\Gamma, \rho \vdash [\![\, \mathsf{furthermore}\ A_1 : \mathsf{Action\ hence}\ A_2 : \mathsf{Action}\ ]\!] \in \gamma}$$

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

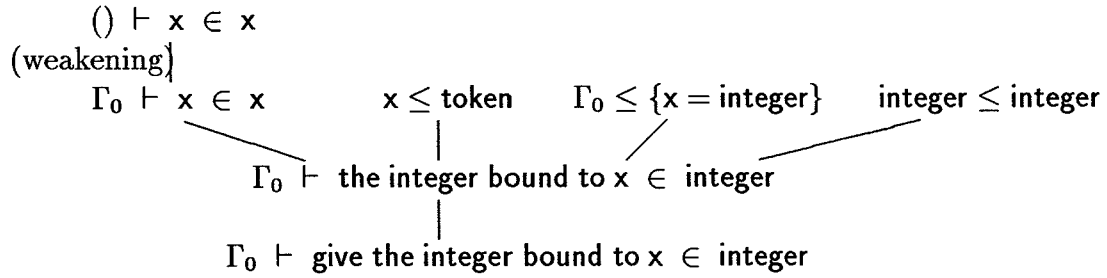
$$\text{(A15)}\ \frac{\Gamma_1 \leq \Gamma_2 \quad \Gamma_2 \vdash A \in \gamma_2 \quad \gamma_2 \leq \gamma_1}{\Gamma_1 \vdash A : \mathsf{Action} \in \gamma_1} \qquad (\textit{weakening rule})$$

Figure 6: Rules for Actions

input facet, Γ_0 = {x=2,y=true}, the following proof tree can be drawn to calculate the operational semantics of the action:



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, Γ_1 = {x = integer,y=truth-value}, we can calculate:



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., () ⊢ give 1 or give 2 ∈ 1 and () ⊢ give 1 or give 2 ∈ 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} ⊢ **give the integer bound to x** ≡ **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 () ⊢ **bind x to 2 hence give 2** ≡ **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** ⊢ **give it** ≡ **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 ⊢ **give it** ≡ **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** ⊢ **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:

Theorem 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*: $\mathcal{P}_b\mathcal{C} \to \mathcal{A}$ and *conc*: $\mathcal{A} \to \mathcal{P}_b\mathcal{C}$, 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: $\mathcal{C} \to \mathcal{A}$, $abs_0(\perp) = \perp$, $abs_0(n) = nat$, $abs_0(b) = bool$, 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 $2 \leq$ **integer**, it must be that $\tau_1 \leq \tau_2$ in $2 \vdash A_0 \in \tau_1$ and **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.

evaluate ⟦ **eq0** *E*:**Expression** ⟧ = **evaluate** *E* **then give equal-to-zero(the given integer)**

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

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

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, $\Gamma_0 \vdash$ **evaluate** E ● **integer**. The complete proof is:

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

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

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

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

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

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

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

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 $\Gamma_0 \vdash$ evaluate $E \in \tau_0$ holds. Then we draw the proof tree as follows:

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

In the process of completing the derivation, the constraint $\tau_0 \leq$ integer is acquired. This makes the induction hypothesis $\Gamma_0 \vdash$ evaluate $E \in \tau_0$ and $\tau_0 \leq$ 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 preserve monotonicity of the inference rules.

References

[1] L. Cardelli and P. 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] K.-G. Doh and D. 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] K.-G. Doh and D. A. Schmidt. Action semantics-directed prototyping. *Computer Languages*, 19(4):213–233, 1993.

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

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

[7] 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.

[8] P. 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, Edinburgh,* Lecture Notes in Computer Science 786. Springer-Verlag, 1994.

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

Design and Semantics of Action Notation

S. B. Lassen,
Aarhus University, Denmark
email: thales@daimi.aau.dk

Abstract

Action notation (AN) is the specification language of *action semantics* [Mos92]. This paper discusses AN's design, semantics, pragmatic properties, and expressive power.

Often we have the choice between actions and yielders as description domain. Actions are more extensible and also, in a certain sense, have better semantics. A new semantics for yielders is proposed that overcomes their deficiencies in these respects.

A new formulation of AN's operational semantics is sketched that uses evaluation contexts. They make certain changes and extensions of AN feasible. A first application is a new semantics for critical regions. Then AN is extended with the notion of continuations which is used to describe control operators like `call/cc` and `goto`. The idea of *subcontinuations* turns out to embody the concepts we will need.

1 Introduction

This paper discusses a range of issues concerning the design and semantics of AN. The overall theme is the requirements we put on AN as a specification language in action semantic descriptions (ASDs). AN must have an intuitive semantics and an adequate expressive power and must yield extensible ASDs.

A specification language should not just have unlimited expressive power, it should also guarantee, or at least suggest, descriptions with a realistic computational content. The applications of action semantics to automatic compiler generation [MW94, Ørb94] support that the primitives of AN are indeed possible to implement. This is an additional requirement to AN that is important for the following discussions.

Section 2 discusses and proposes a revision of the concept of yielders in AN. This is done by a reduction of AN to a simple, unparameterized kernel. In section 3, this kernel notation is given an operational semantics using evaluation

contexts. This opens up for a revision of the semantics of critical regions, in section 4, and an extension with continuations, in section 5.

2 Yielders

AN incorporates data operations via actions of the form:

give *data-operation* (|given datum#i_1 $\cdots$ given datum#i_n|) .

In [Mos92] the argument sort of give_ was extended to arbitrary *yielders* and other primitive actions also got parameterized by yielder arguments. Yielders are compounds of data operations and accesses to state information. A yielder is evaluated atomically as part of a transition of a primitive action.

In ASDs this extended AN alleviates some of the explicit control flow and data flow that only serve uninteresting bookkeeping purposes. The parameterized AN with yielders renders more fluent ASDs.

This section first presents an example that highlights the difference between actions and yielders in semantic descriptions. Then some changes are proposed to improve upon certain semantic and pragmatic properties of yielders.

2.1 Example

Suppose we have this simple imperative language fragment with assignment and a composition construct whose components are evaluated in arbitrary, interleaved order.

- Stmt = ⟦ Stmt "||" Stmt ⟧ | ⟦ Ident ":=" Expr ⟧ .
- Expr = Ident | ⟦ Expr "+" Expr ⟧ .
- exec _ :: Stmt → action[storing]

(1) exec ⟦ S_1:Stmt "||" S_2:Stmt ⟧ = exec S_1 and exec S_2 .

⋮

How many possible outcomes does the following program have when executed in a state where x is 0 and y is 1 ?

$$\{x = 0 \wedge y = 1\} \quad \mathsf{x := y + y \parallel y := x} \quad \begin{array}{l}\{\ x = 2 \ \wedge \ y = 2 \\ \vee \ x = 0 \ \wedge \ y = 0 \ \}\end{array}$$

At least two different outcomes must be possible, corresponding to left-to-right and right-to-left execution of the two statements. But we would expect more fine-grained interleavings to be possible too.

Consider these two ASDs of assignment and expressions, one that maps expressions to actions:

(2) exec ⟦ I:Ident ":=" E:Expr ⟧ = eval E then store it in the cell bound to I .

- eval _ :: Expr → action[giving integer]

(3) eval I:Ident = give the integer stored in the cell bound to I .

(4) eval ⟦ E_1:Expr "+" E_2:Expr ⟧ = | eval E_1 and eval E_2
then give sum of them .

---

and one that maps expressions to yielders:

---

(2) exec ⟦ I:Ident ":=" E:Expr ⟧ = store eval E in the cell bound to I .

- eval _ :: Expr → yielder[yielding integer]

(3) eval I:Ident = the integer stored in the cell bound to I .

(4) eval ⟦ E_1:Expr "+" E_2:Expr ⟧ = sum of (eval E_1, eval E_2) .

It is fair to say that this choice, between actions and yielders as description domain for expressions, is essentially the only choice we have to make in this description.

These two ASDs are not equivalent. The second description is very coarse-grained. An assignment is executed as one atomic transition: In one step, the entire compound expression is evaluated and stored. The example program only has the first two possible outcomes.

The first description is more fine-grained and has more possible interleavings and hence two extra possible outcomes:

$$\{x = 0 \wedge y = 1\} \quad \mathsf{x := y + y \parallel y := x} \quad \begin{array}{llll} \{ & x = 2 & \wedge & y = 2 \\ \vee & x = 0 & \wedge & y = 0 \\ \boxed{\begin{array}{ll} \vee & x = 1 \wedge y = 0 \\ \vee & x = 2 \wedge y = 0 \end{array}} & & & \} \end{array}$$

The second ASD may not be very realistic—an experienced AS user would never write it like that—yet a novice might be lured into this use of yielders by the analogy of the rôles of actions/yielders in AN and of statements/expressions in imperative languages. Of course one should know the semantics of AN before embarking on any ASD and it shouldn't be a surprise that these two ASDs are not equivalent. Still, the exact semantic difference may not be obvious (where exactly to put indivisibly in the first ASD to make the two equivalent?).

The semantic difference reflects that the second ASD packs an arbitrary amount of computations into one indivisible computation step. The atomicity of yielder evaluation is a very strong computational concept. It is a concept that

is otherwise used sparsely in AN (must be made explicit with **indivisibly**), it is not very natural in the computational model of AN (cf. the analysis in section 4), and it is rarely what we want and need in semantic descriptions. In general, the indivisible evaluation of yielders doesn't seem to be exploited in ASDs. It seems reasonable to demand that indivisible execution of several computation primitives must be explicitly enclosed in **indivisibly**.

For the language fragment above, the first, fine-grained description seems to be closest to our computational intuitions and to the operation of computers: It makes one transition per primitive state access operation and one transition per primitive state update operation. So if we change the semantics of yielders to be more fine-grained, indivisible evaluation will always require explicit use of **indivisibly**, and the two ASDs above become semantically equivalent.

2.2 Fine-grained semantics

To give a new semantics for yielders, define them as abbreviations for primitive actions that access state information. And define actions that are parameterized with yielders as abbreviations for unparameterized actions that read their inputs from the given transients.

store Y_1:yielder in Y_2:yielder = (give Y_1 and give Y_2) then update .

give the bindable bound to t:token = find$_t$.

give the storable stored in Y:yielder = give Y then contents .

give *data-operation* (| $Y_1 \cdots Y_n$ |) =
| give Y_1 and $\cdots$ and give Y_n
then
give *data-operation* (|given datum#1 $\cdots$ given datum#n|) .

A compound yielder expands to a compound action that evaluates subterms interleaved. This *changes the semantics of yielders.* The evaluation of a yielder is now split into several primitive actions that make one transition per primitive state access operation.

The elaborate AN with yielders and parameterized actions is reduced to a simple kernel of unparameterized actions: **update**, **find**, **contents** etc. **give** becomes the only parameterized action, its parameters have the form of a data-operation working on given transients. This simple kernel notation is essentially just the old, unparameterized version of AN before the introduction of yielders [Mos83].

When we use the translation rules, the two ASDs from before are translated into the same simple, unparameterized ASD:

---

(2) exec [[I:Ident ":=" E:Expr]] = | eval E and find$_I$
then update .

(3) eval I:Ident = find$_I$ then contents .

(4) eval ⟦ E_1:Expr "+" E_2:Expr ⟧ = | eval E_1 and eval E_2
then give sum of (given datum#1,
given datum#2) .

So we see that the new semantics makes the two natural ASDs equivalent. And the second, coarse-grained description gets a more intuitive (?), fine-grained interpretation.

2.3 Extensibility

There is a second problem concerning the choice we often have between yielders and actions. If we choose yielders, we commit ourselves to a much more restricted description domain. Yielders only describe finite computations without side-effects. For instance if we add a diverging expression to the language fragment from before:

- Expr = Ident | ⟦ Expr "+" Expr ⟧ | "Ω" .
- exec _ :: Stmt → action[storing | diverging] .

---

then the first ASD, that uses actions, is easily extended:

- eval _ :: Expr → action[giving integer | diverging] .

(5) eval "Ω" = diverge .

The second ASD has to be completely rewritten from yielders into actions to accommodate the extension with divergence

How can this be remedied? Recall that the elaborate notion of yielders is reduced to just the yielders of the form: a data-operation working on the given transients. Even if we hold on to the simple kernel that the whole of AN is reduced to, we could have an even richer notion of yielders to begin with, and still reduce it to this simple kernel.

How can we enrich yielders to also include effects in the most simple way? First observe that give maps yielders into actions that gives transients but have no other effects:

give _ :: yielder[yielding data] → action[giving data] .

These two sorts are nearly isomorphic. Every deterministic action A that gives transients and does nothing else can be approximated by **give** Y_A for some yielder Y_A.[1] We can extend yielders to make them completely isomorphic to actions giving transients, by means of a new yielder:

the data given by _ :: **action[giving data]** → **yielder[yielding data]** .

which is the inverse of **give**_ , and a corresponding rule that reduces this new yielder to the simple kernel notation:

give the data given by a:**action** = a .

The aim is to make yielders as extensible as actions, such that, even if we have chosen to describe something with yielders, we can still extend it with divergence and side-effects. What we need to do is to extend the sort of yielders even more, such that they become isomorphic to the sort of actions giving transients, or diverging, and possibly with side-effects:

affecting = **diverging** | **committing** | **storing** | **communicating** | ... .
give _ :: **yielder[yielding data** [| **affecting**]]
→ **action[giving data** [| **affecting**]] .
the data given by _ :: **action[giving data** [| **affecting**]]
→ **yielder[yielding data** [| **affecting**]] .

This can be accomplished simply by extending the domain of **the data given by** _ . This larger sort of yielders gives the headroom we need in semantic descriptions. Now we can easily extend the second ASD, that uses yielders, to diverge:

- **eval** _ :: **Expr** → **yielder[yielding integer** [| **diverging**]] .

(5) [**eval "Ω" = the nothing given by diverge .**]

And both ASDs again translate into the same kernel notation description:

---

(5) [**eval "Ω" = diverge .**]

---

[1] An action A:**action[giving data]** without any "internal" non-determinism (i.e. A's outcome is determined by its income) can be written as A' = **give** Y_A. (Note that A cannot enact abstractions.) The two actions A, A' will be equivalent except that A may impose a more deterministic sequence of its state accesses than A'. Thus, with an appropriate definition of an implementation relation, $\sqsubseteq$, meaning "less deterministic than", we have $A' \sqsubseteq A$. Because A, A' are actions without side-effects, enclosing them in **indivisibly**_ makes the time sequence of state accesses indifferent, hence we have **indivisibly** A' = **indivisibly** A.

2.4 Yielders vs. Actions

A comparison with standard denotational semantics will clarify the pragmatic issues involved in the choice between yielders and actions as description domain in ASDs.

Consider the following terms of some ordinary programming language:

"1" "¬x" "Ω"

They will be instances of some syntactic categories that are mapped to certain semantic domains by the semantic functions for any denotational or action semantic description. Here are some examples of semantic domains and denotations for the example language terms, first in standard denotational semantics and secondly in action semantics:

semantic domain	1	¬x	Ω
Z	1		
$V = B + Z + L + \ldots$	$\iota_2 1$		
$V_\perp$	$\lfloor \iota_2 1 \rfloor$		$\perp_V$
$Env \to V_\perp$	$\lambda\rho.\lfloor \iota_2 1 \rfloor$	$\lambda\rho.\lfloor \iota_1(\text{not}(\rho x)) \rfloor$	$\perp_{Env \to V_\perp}$
$Sto \to Env \to V_\perp$	$\lambda\sigma\lambda\rho.\lfloor \iota_2 1 \rfloor$	$\lambda\sigma\lambda\rho.\lfloor \iota_1(\text{not}(\rho x)) \rfloor$	$\perp_{\ldots}$
$(V \to Ans) \to Env \to Ans$	$\lambda\kappa\lambda\rho.\kappa(\iota_2 1)$	$\lambda\kappa\lambda\rho.\kappa(\iota_1(\text{not}(\rho x)))$	$\perp_{\ldots}$
integer	1		
data	1		
yielder	1	not the truth-value bound to "x"	the nothing given by diverge
action	give 1	give not the truth-value bound to "x"	diverge

In denotational semantics, the denotations are very sensitive to the underlying model. (This becomes much more conspicuous when side-effects of other kinds than divergence are included.) In action semantics, this is not the case. There is essentially just the choice between yielders (that include data) and actions. Note that because integer≤data≤yielder, the change of description domain from integer into yielder doesn't affect denotations (they are all implicitly injected into the enlarged domain). But the sorts yielder and action are disjoint and all denotations are sensitive to this choice between domains.

The pragmatic gain from our extension of the sort yielder above is the ability to write yielder-denotations such as the nothing given by diverge. One might argue

that this enhanced expressiveness of yielders is an attempt to patch up the consequences of a wrong choice of semantic domain in the first place. If we look at the problem in this way, then a more thorough solution is to make **yielder** a direct subsort of **action**. Then we would have **data**≤**yielder**≤**action** and whatever we describe, the denotations should not change when we extend (or restrict) semantic domains.

In practice, this would mean that:

- **give** _ :: **yielder** → **action[giving data]** .

  give Y:**yielder** = Y .

which implies e.g.:

$$(Y_1, Y_2) = \textbf{give}\ (Y_1, Y_2) = \textbf{give}\ Y_1\ \textbf{and give}\ Y_2 = Y_1\ \textbf{and}\ Y_2\ .$$

and the expansion of parameterized into unparameterized actions looks like this:

- **store** _ **in** _ :: **yielder, yielder** → **action[completing | storing]** .
- **update** : **action[completing | storing]** .

  store Y_1:**yielder in** Y_2:**yielder** = (Y_1 **and** Y_2) **then update** .

In section 2.3 we showed that yielders must be extended to ensure extensibility of ASDs that use yielders. Instead of having (almost?) **yielder** = **action[giving data]**, we must extend to **yielder** = **action[giving data | affecting]**.

We might even give up the concept of yielders altogether. The arguments of parameterized actions could be just arbitrary actions. Then the above expansions, with **actions** instead of **yielders**, specifies that **give**_ is just the identity unary action combinator, and tupling (_ , _) is the same as the binary action combinator **and**. In general, each primitive parameterized action is performed by interleaving the argument actions followed by performing some primitive unparameterized action on the given data.

This scheme has a minimum of overlapping concepts and it is a thorough solution that maximizes AN's contribution to the extensibility AS. But the semantics of AN may become less transparent at some points. A disadvantage of allowing actions as arguments of "primitive" actions like **store_in_** is that control flow is no longer determined purely by the standard combinators but will be hidden in parameterized actions and data operations too (arguments are implicitly interleaved).

All the changes and extensions to yielders discussed above are highly tentative. The reduction of AN to an unparameterized kernel expounded in sections 2.2–2.3 ended up in the notation of an earlier version of AN [Mos83]. It has been used as description language in its own right [DS93] which indicates that it would be a sensible kernel notation. The rest of this paper will use this kernel notation.

3 Operational Semantics

This section sketches an operational semantics of the simple, unparameterized kernel notation. Another formalism will be used than the structural operational semantics in [Mos92, App.C]. This new formulation may make it easier to do operational reasoning about actions and action equivalences. In this paper, the main benefit will be that it will enable us to extend AN with continuations. The key concepts are:

An *evaluation context* [FF87] is an *action term with a hole at a legal point of execution in the term.* An evaluation context is either just a hole, a hole in the LHS of any binary action combinator, or a hole in the RHS of any interleaving action combinator (like **and** or **or**):

$$\textsf{evaluation-context } E ::= [\,] \mid E \textit{ binary } A \mid A \textit{ interleaving } E \ .$$

We will only describe the functional and declarative facets.

An *intermediate configuration* decomposes into an evaluation context with a redex filled into its hole.

$$\textsf{intermediate configuration} ::= E[R] \ .$$

This decomposition is not necessarily unique. If there is an interleaving combinator in the configuration, then we have the choice between choosing a redex on the LHS or the RHS of the interleaving combinator.

A *redex* is something that can make a primitive transition directly, i.e. a primitive action fed with transients and bindings which we write as follows:

$$\textsf{redex } R ::= (d\textsf{:data},\ b\textsf{:bindings}) \ \triangleright \ P\textsf{:primitive-action} \ .$$

And the *outcome* of an action is *to* **give** *transients*, *to* **produce** *bindings*, or *to* **fail**.

$$\textsf{outcome} ::= \textsf{give } d \mid \textsf{produce } b \mid \textsf{fail} \ .$$

Here are some examples of *transitions*:

$$E[(d,b) \ \triangleright \ \textsf{find}_t] \rightarrow E[\textsf{give } (b \textsf{ at } t)] \ .$$
$$E[(d,b) \ \triangleright \ \textsf{bind}_t] \rightarrow E[\textsf{produce (map } t \textsf{ to } d)] \ .$$
$$E[(\textsf{abstraction of } a,b) \ \triangleright \ \textsf{enact}] \rightarrow E[((),\textsf{empty-map}) \ \triangleright \ a] \ .$$

A configuration that is decomposed into an evaluation context E filled with a redex makes a transition into the same evaluation context E filled with an appropriate outcome. In the last transition, the unparameterized **enact** expects an abstraction as transients, which it invokes with empty transients and bindings.

A lot of technical details are omitted here, e.g. how to transport the transients and bindings to the redex, and how to consume the outcome into the evaluation context such that it can decompose to do the next transition. But the core of the

approach is that the execution of compound terms is determined by the algebraic specification of evaluation contexts (there are no structural rules as found in structural operational semantics).

This formulation of the operational semantics has an important property: In every configuration, the evaluation context of a decomposition is a concrete entity that represents the context, or *"the rest of the program"*, or the *continuation.* It can also be seen as a *program pointer.* The following sections will exploit this property in semantic formulations that would be difficult in a structural operational semantics.

4 Critical Regions

As a first application of the new formulation of AN's operational semantics, this section considers a new semantics for **indivisibly**.

The current structural operational semantics of **indivisibly** is

$$a\text{:}\mathsf{action} \rightarrow^* t\text{:}\mathsf{terminated} \quad \Rightarrow \quad \mathsf{indivisibly}\ a \rightarrow t\ .$$

The body of indivisibly is executed as one big step. This is a very clear and straightforward semantics that prevents such a "critical region" from being interleaved with something else.

There are some quirks, though: What if a critical region diverges? This is prohibited in [Mos92] but that means that it is undecidable whether an action is legal.[2] Also, the programming concepts involved in **indivisibly** are powerful and not easily implementable. Communication with other agents is shut down or delayed during the execution of a critical region. Some uses of **indivisibly** do not use, and are even in conflict with, these properties regarding communication and divergence. For instance uses of **indivisibly** in semantic reasoning to specify non-interference [Mos92, B.4.1]. Preferably, the semantics of **indivisibly** should only model non-interference and be closer to realistic implementations.

What if we instead added the following transitions to our new formulation of the operational semantics?

$$(entry) \qquad E[(d,b) \rhd \mathsf{indivisibly}\ a] \rightarrow E[\mathsf{indivisibly}\ (d,b) \rhd a]\ .$$

$$(exit) \qquad E[\mathsf{indivisibly}\ t] \rightarrow E[t]\ .$$

Then we have to make sure that between *entry* and *exit* of a critical region nothing else is interleaved.

[2]This is undecidable for other reasons too (other examples of illegal actions are actions that violate sort restrictions of various kinds, e.g. by trying to bind something that is not of sort **bindable** or by trying to give something that is a proper sort and not an individual), yet this undecidability is an undesirable property that we ought to minimize.

Evaluation contexts provide us with a notion of "program pointer", and this we can use to keep track of a currently operating critical region. Split the sort of intermediate configurations into those that are inside a critical region, and those that are not:

$$\textsf{intermediate configuration} ::= \textsf{critical} \mid \textsf{uncritical} .$$

Then **uncritical** are those configurations with a redex not wrapped in **indivisibly** by the enclosing evaluation context:

$$\textsf{uncritical } u ::= U[(d,b) \rhd a] .$$
$$\textsf{uncritical-context } U ::= [\,] \mid U \textit{ sequential } a \mid U \textit{ interleaving } u \mid u \textit{ interleaving } U .$$

And in a **critical** the redex is inside **indivisibly**:

$$\textsf{critical } c ::= E[\textsf{indivisibly } E'[(d,b) \rhd a]] .$$
$$\textsf{evaluation-context } E ::= [\,] \mid \textsf{indivisibly } E \mid E \textit{ sequential } a \mid E \textit{ interleaving } u \mid u \textit{ interleaving } E .$$

The algebraic specification of the sort **evaluation-context** tells the full story:

- Critical regions can be nested.
- If there is an active critical region (the configuration is **critical**), the redex must be chosen therein (the hole in the evaluation context must be on the **critical** side of any interleaving combinator because the un-chosen side has to be **uncritical** by the definition of **evaluation-context**).
- The two sides of an interleaving combinator cannot both be **critical** because initially they must both be **uncritical** and when one side gets **critical**, the other side is excluded until the **critical** becomes **uncritical** again.

This improves the semantics for critical regions on some points: It deviates less from the rest of the operational semantics of AN, it only models that a critical region cannot be interleaved with anything else, and it gives a natural interpretation of divergence inside critical regions.

5 Continuations

Continuations are a powerful programming technique in functional programming. They may be hard to understand but they do have a precise formal semantics. Yet, it is impossible to give a straightforward ASD of continuations. To remedy this deficiency of AS, this section extends AN with continuations. Later on further justification for this extension will be sought by using AN's continuations to describe control constructs in imperative languages too.

5.1 `callcc` and `throw`

Lets focus on SML/NJ's `callcc` and `throw`. They manipulate a program's continuations as first-class values.

Evaluation contexts provide the machinery to give an operational semantics to continuations: When we decompose an intermediate configuration into an evaluation context E and a redex, then the redex represents the program's *current operation*, and the evaluation context E represents the program's *current continuation, "the rest of the program"*.

$$E[\texttt{callcc}\ f] \ \rightarrow\ E[f\ E] \qquad\qquad E[\texttt{throw}\ E'\ v] \ \rightarrow\ E'[v]$$

`callcc` copies the current continuation and applies its argument to it. `throw` throws away the current continuation E, and reinstates the continuation E' with outcome v.

`callcc` and `throw` have straightforward formal semantics, both operational and denotational. Therefore we would expect to be able to describe continuations in AS too, but we cannot in any reasonable way. Continuations are a "notion of computation" missing in AN (as admitted in [Mos92, p.211]). To describe `callcc` and `throw` in AS, we need to extend AN with similar control operators.

Let **copycc in_** be a unary action combinator and let **throw_with_** be a primitive action with a continuation- and a value-parameter. **throw_with_** can be expanded into unparameterized notation as before where **throw** is an unparameterized action that expects a continuation and a value-parameter on the given transients:

throw Y_1 **with** Y_2 = (**give** Y_1 **and give** Y_2) **then throw** .

Continuations fit in smoothly with the operational semantics for AN that was sketched in section 3:

E[(d,b) ▷ **copycc in** a] → E[((E,d),b) ▷ a] .
E[((E',d),b) ▷ **throw**] → E'[**give** d] .

copycc copies the current continuation and pushes it in front of the current transients. **throw** expects a continuation as first component of the transients and reinstates it with the rest of the current transients.

Now we can easily make an ASD of SML/NJ including `callcc` and `throw`. (There is nothing to it because the troublesome control operators are just translated into the corresponding actions.)

- **value** = **abstraction** | **continuation** | □ .

- **eval** _ :: Expr → **action** .

(1) **eval** ⟦ "fn" I:Ident "=>" E:Expr ⟧ =
    give abstraction of
        furthermore bind token of I **to given value#1**
        hence eval E .

(2) eval ⟦ E_1:Expr E_2:Expr ⟧ = | eval E_1 and then eval E_2
then enact application of given abstraction#1
to given value#2 .

(3) eval ⟦ "callcc" E:Expr ⟧ = eval E then
copycc in enact application of given abstraction#2
to given continuation#1 .

(4) eval ⟦ "throw" E_1:Expr E_2:Expr ⟧ =
| eval E_1 and then eval E_2
then throw given continuation#1 with given value#2 .

Note that SML is a deterministic language with an explicit left-to-right evaluation order. What is the impact of this on the semantics of continuations? To see this, consider the following expressions that one would expect to be equivalent:

```
(callcc (fn k => throw f)) e  ?≃  f e
```

They are with SML's left-to-right evaluation of function and argument. In Scheme where the evaluation order is unspecified, either left-to-right or right-to-left, the equivalence also holds.

But if we write "**and**" instead of "**and then**" in the SML/NJ ASD, such that any interleaving evaluation is possible, then the equivalence ceases to hold. An example:

```
(callcc (fn k => throw (fn x => x))) (print "hello")
        ?≃  (fn x => x) (print "hello")
            ≃  print "hello"
```

`callcc` may copy the current continuation (or context) just before `print "hello"`. But before the continuation is reinstated by `throw`, `"hello"` may be printed. Then `throw` will rewind the RHS of the context and `"hello"` is printed again.

What we see here is that continuations are a global notion, that becomes uncontrollable if paired with non-sequentiality. The above SML/NJ ASD doesn't have interleavings and there is no problem. But are copycc and throw sensible operations in AN as such if it is possible to program something as counterintuitive as actions that rewind their contexts?

5.2 Pascal's goto

To substantiate this problem, consider the following application of AN's copycc and throw to describe `goto` in a Pascal-like language.

A block consists of declarations and a body:

```
label l
function f(...)  label m
                 begin ... goto m ... end
function g(...)  begin ... goto l ... end
variable x
begin goto l
   l: x := f + g
end
```

The block's labels are visible inside the blocks in the declarations. So it is possible to jump within the body of a block or to the body of an enclosing block.

This is a fragment of an ASD using continuations:

---

(1) activate ⟦ L:Label* F:Function* V:Variable*
"begin" S:Stmt* "end" ⟧ =
furthermore declare L before declare F before declare V
hence run S .

(2) declare ⟦ "label" I:Ident ⟧ = indirectly bind token of I to unknown .

(3) declare ···

(4) run ⟨ S_1:Stmt* S_2:Unlabeled-Stmt ⟩ = run S_1 and then exec S_2 .

(5) run ⟨ S_1:Stmt* I ":" S_2:Unlabeled-Stmt ⟩ =
| copycc in
| | redirect token of I to given continuation#1
| | and then run S_1
and then exec S_2 .

(6) run ⟨ ⟩ = complete .

(7) exec ⟦ "goto" I:Ident ⟧ = throw the continuation bound to token of I with () .

(8) exec ⟦ I:Ident ":=" E:Expr ⟧ =
eval E then store it in the cell bound to token of I .

(9) eval ⟦ E_1 "+" E_2 ⟧ = (eval E_1 and eval E_2) then give sum of them .

(10) eval ···

The body of a block starts with a series of copyccs that copy the continuations to be bound to the labels.

The interleaving evaluation of `f` and `g` in `f + g` clashes with the use of continuations in the ASD: The continuation (or context) that `f` copies and binds `m` to, includes the interleaved evaluation of `g`. When `f` throws this continuation and `goto m`, then the evaluation of `g` is rewound to the state when the `m`-continuation was once copied. *This is certainly not the intended semantics of* `goto`. We definitely expect a local `goto` not to have such bizarre global effects.

5.3 Subcontinuations

Recall the semantics of copycc and throw. copycc copies its global context, possibly including interleaved computations. When this global context is thrown, interleaved computations are rewound to their state at the time of copycc. This way, something in an interleaved branch may, inadvertently, be executed twice.

There exist several proposals for control delimiters to tame the global power of continuations [Fel88, DF90]. *Subcontinuations* [HDA94] is the idea most relevant for our purposes. Subcontinuations have been proposed for concurrent settings, and they address exactly the problems posed by interleaving actions.[3]

Introduce a unary action combinator spawn_as_ with some kind of identification id, and a local version of copycc called copy_in_ with a parameter referring to an enclosing spawn. copy only captures the local context inside the corresponding spawn. Call such a local context a subcontext or subcontinuation. When we throw a subcontinuation we only replace the appropriate subcontext:

$$E[(d,b) \rhd \textsf{spawn } id \textsf{ as } a] \rightarrow E[\textsf{spawn } id \textsf{ as } (d,b) \rhd a] \,.$$

$$E[S_{id}[(d,b) \rhd \textsf{copy } id \textsf{ in } a]] \rightarrow E[S_{id}[((S_{id},d),b) \rhd a] \,.$$

$$E[S_{id}[((S'_{id},d),b) \rhd \textsf{throw}]] \rightarrow E[S'_{id}[\textsf{give } d]] \,.$$

where S_{id} is a subcontext of the form spawn id as $E_{\neg id}$, and $E_{\neg id}$ is an evaluation context without occurrences of spawn id as .

The point is that now we can put a spawn around our copys and thereby enforce locality on our continuation-manipulations. When we copy and throw subcontinuations, we don't affect the context outside the corresponding spawn.

This doesn't solve all our problems. copy and throw can still do counterintuitive computations that rewind their contexts. But now we have the means to control this undesirable feature; it is only within the subcontext in question

[3] [Mor94] is a different approach that, in parallel settings, makes control operators simulate the behaviour of sequential execution. This also ensures some locality such that the goto-example would work. But in a semantic specification language like AN, a more primitive semantics with tighter control of locality and scope of continuations seems preferable.

that such rewinding takes place. Therefore these operations may still not be altogether reassuring, but they are a powerful description tool and the locality of the new operations seem to be expressive of exactly the locality we need.

In the semantics of our SML/NJ fragment, we put a **spawn** at the root of the program, and all `callccs` **copy** everything within that global **spawn**.

In the "Pascal" semantics we can now enclose every block in its own spawn and make local labels local. Now it is only the body of `f` that is replaced and affected when `f` makes a local `goto m`.

---

(1) activate ⟦ L:Label* F:Function* V:Variable*
"begin" S:Stmt* "end" ⟧ =
furthermore declare L before declare F before declare V
hence
generate a block-id then spawn it as run S .

(2) run ⟨ S_1:Stmt* I ":" S_2:Unlabeled-Stmt ⟩ =
copy it in
redirect token of I to given continuation#1
and then
give given block-id#2 then run S_1
and then exec S_2 .

5.4 Control filters

Why hasn't this powerful description tool of continuations been part of AS right from its origin?

There is a big problem with the continuation-description of `gotos`: We might want to describe clean up on exit from a block. And there is no way to combine this obligation to clean up with the throwing of continuations.

As an example, suppose we relinquish local variables on exit from a block as follows:

---

(1) activate ⟦ L:Label* F:Function* V:Variable*
"begin" S:Stmt* "end" ⟧ =
furthermore declare L before declare F before declare V
hence
generate a block-id then spawn it as run S
thereafter relinquish V .

What should A_1 **thereafter** A_2 mean? Certainly, on normal completion of A_1, A_2 should be executed. But what if the body is left by means of a `goto` to an enclosing block, i.e. A_1 throws a subcontinuation?

There is a concept of *control-filters* associated with subcontinuations that meets out purposes very nicely: The idea is to let **throw** slide out through the subcontext it replaces. During this, all control-filters that are encountered are executed on the way out. In our case, A_1 **thereafter** A_2 is a control-filter that insists that A_2 is executed, even if a subcontinuation is thrown by A_1.

Using the expressiveness of evaluation contexts, we can write the semantics of **throw** and **thereafter** like this:

$$E[T_{id}[((S_{id},d),b) \triangleright \textbf{throw}]] \rightarrow E[S_{id}[\textbf{give } d]] \ .$$

$$E[E_{\neg id}[((S_{id},d),b) \triangleright \textbf{throw}] \textbf{ thereafter } a] \rightarrow E[((S_{id},d),b) \triangleright \textbf{throw thereafter } a] \ .$$

$$\textbf{evaluation-context } E ::= \ldots \mid \textbf{throw } S \textbf{ with } d \textbf{ thereafter } E \ .$$

where T_{id} is of the form **spawn** id **as** $E_{\neg id}$, and $E_{\neg id}$ is an evaluation context without occurrences of either **spawn** id **as** or **thereafter** . (The flow of transients and bindings through **thereafter** should probably be chosen like **trap**.)

This also links the semantics of continuations and the semantics of **escape** and **trap**. Subcontinuations subsume these exception handling actions.

Subcontinuations could form a powerful control facet in AN. The syntax of the constructs presented here may not be particularly well-chosen and several semantic details have to be worked out. But the subcontinuation operations seem to come close to the control concepts we really need for the description of real programming languages.

6 Conclusion

A range of topics concerning AN has been explored in this paper.

First we proposed a new, fine-grained semantics for yielders, and a way to make ASDs that use yielders more extensible. This we did by reducing the elaborate, parameterized AN to a simple, unparameterized kernel.

Then, we formulated the operational semantics of this AN kernel in terms of evaluation contexts. The applications of this formulation were to revise the semantics of critical regions and to extend AN with continuations.

The latter made it possible to describe the control operators `callcc` and `throw`. Then the mismatch of interleavings and continuations led to the concept of subcontinuations. Coupled with control-filters, subcontinuations appear to embody the concepts we need in ASDs. This was the case in the example of `gotos`.

Acknowledgements. I would like to thank Peter Mosses and Olivier Danvy for valuable discussions and guidance in parts of this work.

References

[DF90] O. Danvy and A. Filinski. Abstracting control. In *Conference of LISP and Functional Programming*, pages 151–160. ACM, 1990.

[DS93] Kyung-Goo Doh and David Schmidt. Action semantics-directed prototyping. *Comput. Lang.*, 19(4):213–233, 1993.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *POPL*, pages 180–190. ACM, 1988.

[FF87] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III, Proc. IFIP TC2 Working Conference, Gl. Avernæs, 1986.* IFIP, North-Holland, 1987.

[HDA94] Robert Hieb, R. Kent Dybvig, and Claude W. Anderson. Subcontinuations. *LISP and Symbolic Computation*, 7(1):83–110, 1994.

[Mor94] Luc Moreau. The PCKS-machine: an abstract machine for sound evaluation of parallel functional programs with first-class continuations. In *Programming Languages and Systems – ESOP'94*, volume 788 of *Lecture Notes in Computer Science*, pages 424–438. Springer-Verlag, 1994.

[Mos83] Peter D. Mosses. Abstract semantic algebras! In Dines Bjørner, editor, *Formal Description of Programming Concepts II, Proc. IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982.* IFIP, North-Holland, 1983.

[Mos92] Peter D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[MW94] H. Moura and D. Watt. Action transformations in the ACTRESS compiler generator. In *CC'94*, volume 786 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[Ørb94] Peter Ørbæk. OASIS: an optimizing action-based compiler generator. In *CC'94*, volume 786 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

The Formal Specification of ANDF

An Application of Action Semantics

Jens Ulrik Toft DDC International A/S

Bo Stig Hansen Technical University of Denmark

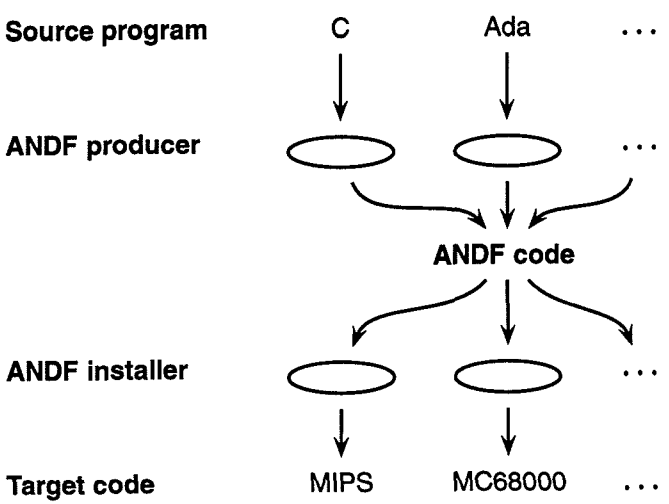
ESPRIT PROJECT 6062 omi/glue

Contents

What is ANDF?

Stands for: Architecture Neutral Distribution Format of the Open Software Foundation (OSF).

Description: General intermediate language which may be used as target when compiling usual high-level languages.



Example Program

Factorial in C

```
int fac(int arg)
{
  int res = 1;

  while (arg > 1) {
    res := res * arg;
    arg--;
  }

  return res;
}
```

Example Program

Factorial in ANDF

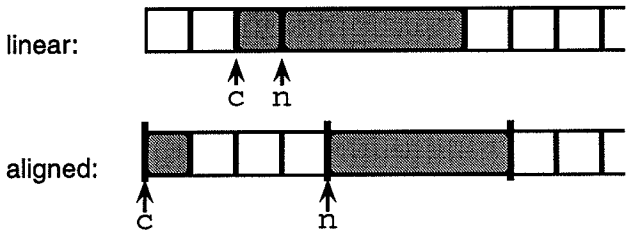
```
DEFINE int:sort = INTEGER(0..2^32-1)

def fac =
 proc(arg:int) : int

  variable res := 1:int

  labelled
      start_at l1

  l1: goto l2 if_not
              c(arg:int)>1:int;
      res := c(res:int)*c(arg:int);
      arg := c(arg:int)-1:int;
      goto l1;

  l2: return c(r:int)
```

Compound Data Representations

In C

```
struct S { unsigned char c; S* n; }
```

Storage Layout



In ANDF

```
COMPOUND(sz)
where sz =
      pad(size(INTEGER(0..255)),
          alignment(POINTER)     )
      + size(POINTER)
```

ANDF Alignment and Size Algebras

Are sizes and alignments natural numbers?
When can sizes be added?

Alignment requirements are partially ordered by implication, e.g.:

$$\texttt{alignment(POINTER)} \Rightarrow \texttt{alignment(INTEGER(0..255))}$$

Sizes of data representations are divided into classes (types) according to their alignment requirements:

$$\texttt{size}(r) : \texttt{SIZE}(\texttt{alignment}(r))$$

$$s : \texttt{SIZE}(a_1) \Rightarrow \texttt{pad}(s, a_2) : \texttt{SIZE}(a_1 \wedge a_2)$$

$$s_1 : \texttt{SIZE}(a_1) \wedge s_2 : \texttt{SIZE}(a_2) \wedge (a_1 \Rightarrow a_2) \Rightarrow s_1 + s_2 : \texttt{SIZE}(a_1)$$

Note: The size algebra is not quite right. In the ANDF specification an algebra of offsets is used instead.

ANDF Specification Challenges
and their Solution in Action Semantics

- Under-specified language notions, e.g., alignment and size.
  AN: algebraic specification
- Partial functions (intended non-termination)
  AN: operational semantics
- Under-specified order of evaluation
  AN: actions composed with "and"
- Abnormal sequencing (jumps)
  AN: escape_with, trap
- *Concurrency (future extension of ANDF)*
  AN: communicative facet

ANDF Formal Specification

General Requirements

1. Must be unambiguous, consistent and complete regarding the meaning of ANDF language constructs and features.
2. Must leave open all possibilities of making correct implementations.
3. Must be comprehensible and concise.
4. Must have a maintainable form.
5. Should support stepwise developments of implementations.
6. Should support the kinds of proofs which are relevant for the anticipated users/uses.

ANDF Formal Specification

Specification Language Requirements

1. Must support modularisation.
2. Must be supported by tools to help eliminate simple kinds of errors, e.g., grammatical errors, use of identifiers not declared, and type errors.
3. Must be supported by tools for easy production of revised specifications, e.g., automatic pretty printing and automatic formula and line numbering.
4. Must be supported by a proof editing/checking tool.

ANDF Formal Specification

The RAISE Specification Language (RSL)

- Supports algebraic as well as model-oriented, VDM like specification.
- Applicative, imperative and concurrent specification styles.
- Full featured module notion.
- Supported by a commercial toolkit:
  - syntax-directed editor
  - type checker
  - proof editor
  - LaTeX pretty printer
  - library with version control
  - code generators for executable subset

ANDF Formal Specification

Choice of Specification Language

Action Notation does not have the tool support required.

RSL has, if used straight-forwardly, some weaknesses:

- Not as comprehensible and concise.
- Difficult description of intended partiality.

Otherwise, it meets *all* requirements.

Solution: Embed (a subset of) Action Notation in RSL.

ANDF Formal Specification

Action Notation in RSL

Abstract syntax for Action Notation

Action = ConstantAct | DyadicAct | MonadicAct ...
ConstantAct == ESCAPE | COMPLETE | ...
DyadicAct = Action * InfixActOp * Action
InfixActOp == THEN | AND | OR | ...

Example

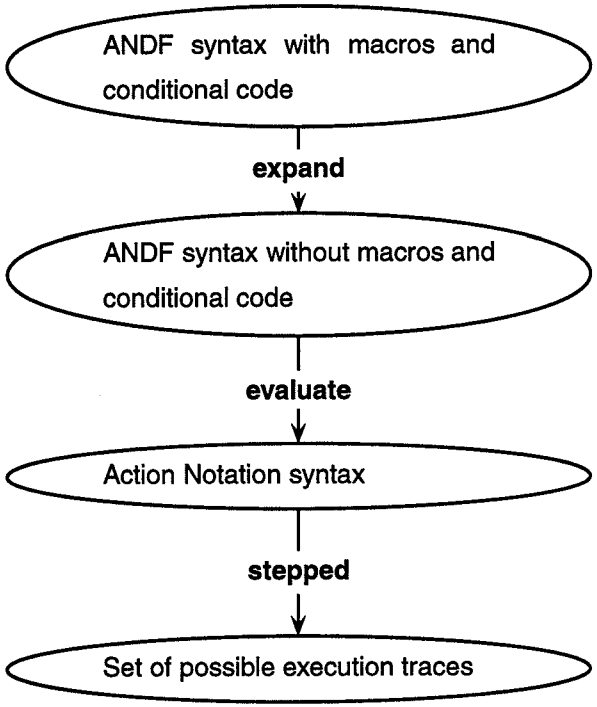
(A1, THEN, (A2, OR, A3))

Operational Semantics

Stepped: Action * State → (Action * State)-**set**

ANDF Formal Specification

Overall structure



ANDF Formal Specification

Example:

Action Semantics for "bitwise and"

```
evaluate(and(arg1,arg2)) ≡

    ((evaluate(arg1), AND, evaluate(arg2)),
THEN,
    (GIVE(bitwise_and(the_GIVEN_integer(1),
                      the_GIVEN_integer(2) )), OR,
        (check_undef_args,
    THEN,
        undef_and_args)))
```

Conclusion

Results: A complete specification of ANDF abstract syntax, static semantics and dynamic semantics (800 pages/34000 lines)

Ressources: 2 man years

Hardest challenges:

- Not overspecifying the semantics
- Interpreting the informal description correctly

Uses:

- Reference for precise semantics
- Basis for development of ANDF interpreter

References

Jens P. Nielsen and Jens Ulrik Toft. *Formal Specification of ANDF, Existing Subset.* Technical report DDC-I 202104/RPT/19, issue 2, DDC International A/S, 1994.

Jens Ulrik Toft. *Feasibility of using RSL as the Specification Language for the ANDF Formal Specification.* Technical report DDC-I 202104/RPT/12, issue 2, DDC International A/S, 1993.

Bo Stig Hansen and Jørgen Bundgaard. *The Rôle of the ANDF Formal Specification.* Technical report DDC-I 202104/RPT/5, issue 2, DDC International A/S, 1992.

Copies can be obtained by contacting Jens Ulrik Toft: jut@ddci.dk

Comparing Action Semantics and Evolving Algebra Based Specifications with Respect to Applications

Arnd Poetzsch-Heffter
Fakultät für Informatik
Technische Universität
D–80290 München
poetzsch@informatik.tu-muenchen.de

Abstract

Action semantics is compared to evolving algebra based language specifications. After a short introduction to and a general comparison of these two frameworks, we discuss different aspects of the frameworks relevant to language documentation and programming tool development.

1 Introduction

In the last twenty years, many different frameworks for the formal specification of programming languages have been developed: e.g. denotational, structural operational, action, and evolving algebra semantics. Whereas a lot of work has been spent to develop these frameworks and to apply them to more and more realistic languages, almost no effort has been made so far to compare and relate the different approaches. Comparisons should reveal for which class of languages a specification framework is most appropriate and for which language implementation tasks a framework provides a suitable formal basis. Relating frameworks should help to improve or even combine them in order to exploit the advantages of different frameworks.

In this extended abstract, we summarize a comparison between action semantics and evolving algebra semantics. Section 2 provides tiny introductions into these frameworks and compares the underlying specification principles. Section 3 discusses the frameworks with respect to language documentation and tool development.

2 General Comparison

Action semantics is an oprational language specification framework developed by P. Mosses (see [4]). An action semantics specification consists of three parts: (1.) the context-free syntax; (2.) the specification of data types and auxiliary actions (based on an elaborate set of predefined data types and actions); (3.) the semantic functions mapping each syntax tree into a (composed) action. The semantic functions are inductively defined over the syntax trees, composing the action for a tree from the actions of its subtrees. Actions are semantic entities used to express control behaviour (possibly nondeterministic, parallel) and the manipulation of sophisticated implicit computation environments consisting of name bindings, stored information, temporary results, and data communicated between distributed actions. Actions are described by applying action combinators to primitive actions. All parts of an action semantics description are completely formalized by so-called universal algebras.

Evolving algebras are an operational specification framework developed by Y. Gurevich (the following comparison is based on the introduction in [3]; for evolving algebras with several demons cf. [2]). They are used to specify the dynamic semantics of programming languages (other applications are protocol and architecture specification). Syntax and static semantics are usually described in an informal way, but confer [5] where attributed occurrence algebras are used for these purposes. An evolving algebra specification consists of a set of rules describing how configurations are related to possible successor configurations. A configuration includes all information necessary for expressing the dynamic behaviour of a program, in particular it incorporates the program itself. Configurations are formally modeled by first-order algebras. The semantics of a program is given by the set of its traces/runs starting with an initial configuration. Evolving algebras support modularization based on the rule set and the configuration structure: Different aspects of the language specification are handled by different rules allowing e.g. to seperate the value propagation in expression evaluation from control flow aspects or aspects concerning parallel execution from the rest of the specification. Beside this, evolving algebras enable very loose specifications of configurations, thereby supporting different refinement techniques.

The different specification principles and properties of action semantics and evolving algebras are summarized in the following table:

specification principle	action semantics	evolving algebra
composition principle of semantics	according to syntax tree structure	according to configuration structure
computation environment	implicit with local and global parts	explicit and global (part of configuration)
specification method	mapping syntax tree to composed action	specifying transition relation based on rich program representations
design principle	using sophisticated set of powerful action combinators (language independent)	designing a language dependent computation model (from scratch)
main semantic entities	equivalent classes of actions	sets of traces over algebras
range of formalization	syntax and semantics	focussing on dynamic semantics

3 Using Language Specifications

Language specifications are written for different purposes. In this section, we sketch a comparison of action semantics and evolving algebras w.r.t. language documentation and tool development.

3.1 Reading & Writing Language Specifications

When language specifications are used mainly for language documentation and standardization purposes, the main comparison criteria are readability, applicability to a wide language class, reusability of existing specifications, and the complexity of writing specification.

A general advantage of action semantics over evolving algebras is that it provides (a) a standardized, elegant, and sorted notation covering the whole task of language specification and (b) a well-developed module concept. For the other aspects of the comparison two factors are of major importance:

1. Can knowledge of the specification framework be assumed?
2. Is the specified language essentially a variant or mixture of existing imperative or functional languages?

If a good knowledge of action semantics is assumed, the rather large number of predefined actions with all their incorporated know-how are a great help for reading and writing specifications. Otherwise, evolving algebras have the advantage that they are easier to learn, so that one can concentrate on the design of the language specification. (The importance of this advantage in practical situations should not be underestimated.)

In case that (2.) is true, the specification knowledge built into the action facets and the clear specification methodology of action semantics can be very helpful to guide the specification process and allow for reuse and adaption of existing specification parts. Whereas reuse and adaption is possible in evolving algebras as well, the management of transient information and the use of language-specific tasks[1] are the primitive the can cause some overhead.

On the other hand, when it comes to the specification of languages with new inventive constructs (where formal specification is essential to gain clarity from the very beginning), the fixed methodology of action semantics (mapping syntax trees to actions) can create unnecessary difficulties or even unsolvable problems (continuation handling, dynamic program modification) whereas the flexibility of evolving algebras allows to design suitable specifications for languages based on extremly different paradigms (e.g. logical languages, object oriented languages making extensive use of messages as call mechanism, assembler languages, ..). The main advantage of evolving algebras in this respect is that they enable to specify the dynamic semantics over the most appropriate static structure which may be much richer than abstract syntax trees.

3.2 Developing Language-Specific Tools

Developing language-specific tools (e.g. language-based editors, browsers, interpreters, compilers, optimizer, program analyser) from language specifications is a major issue of language design and implementation. Up to now, different tools are based on different, unconnected specification techniques; e.g. many tools are based on attribute grammars, but optimization methods need flow graph representations. An integrated framework where tool development is considered as specification refinement could support use and reuse of specifications and increase the correctness of programming tools. With this goal in mind, a comparison of action semantics and evolving algebras can be summarized as follows:

- Action Semantics: The advantage of action semantics is that optimization and implementation technology can be based on actions, i.e. is language independent. Therefore action semantics is a good candidate for automatic compiler generation. On the other hand, it is difficult to express language-specific optimizations and even harder to use an action semantics specification as a basis for interactive tools, because a distinction between static and dynamic semantics is not supported by the framework.

- Evolving Algebras: The strength of evolving algebras is the stepwise development of tools starting with the language specification. The flexibility of evolving algebras allows to perform refinements in the framework itself:

[1]The basic operations of a language are usually called *tasks* in evolving algebra specifications; such a task can be considered as a language-specific action.

e.g. the possibility to explicitly distinguish between static and dynamic aspects or to integrate control flow graph based optimizations. Whereas refinement of data types can be performed in both frameworks, refinement of the basic operations (usually called tasks in evolving algebras) can only be done within evolving algebras. In addition to this, having the programs (possibly including attributes and control informations) as part of the configurations is a big advantage for interactive applications.

A very interesting aspect is to compare the suitability of the frameworks for verification tools. The advantage of action semantics in this respect is certainly that it provides completely formal specifications and an explicit notion of program equivalence whereas in evolving algebra specifications syntax, static semantics, and a program equivalence notion is often kept informal. The strength of evolving algebras lies in correctness proofs of compilation schemes (cf. e.g. [1]) and as a foundation for interactive program verifier.

4 Conclusions

We compared action semantics to evolving algebra based language specifications and discussed their application to different tasks of language design and implementation. The goal of the comparison was not only to provide some criteria that may guide people to chose between action semantics and evolving algebra for a specification task, but to encourage to close the gap between these frameworks in order to combine their respective advantages.

References

[1] E. Börger and D. Rosenzweig. The WAM-definition and compiler correctness. Technical Report TR–14/92, dipartimento di informatica, universita di Pisa, 1992.

[2] P. Glavan and D. Rosenzweig. Communicating evolving algebras. In E. B. et al., editor, *Computer Science Logic*, pages 182–215, 1992. LNCS 702.

[3] Y. Gurevich. *Evolving Algebras*, volume 43, pages 264–284. EATCS Bulletin, 1991.

[4] P. Mosses. *Action Semantics.* Cambridge University Press, 1992. "Tracts in Theoretical Computer Science".

[5] A. Poetzsch-Heffter. Developing efficient interpreters based on formal language specifications. In P. Fritzson, editor, *Compiler Construction*, 1994. LNCS 786.

A Framework for Generating Compilers from Natural Semantics Specifications

Stephen McKeever
swm@comlab.ox.ac.uk

Programming Research Group,
Oxford University

Abstract

We consider the problem of automatically deriving correct compilers from Natural Semantics specifications [Kahn 87, NN 92] of programming languages. Our method is based on the idea that a programming language is inherently a specification of a computation done in stages. Certain phrases in a language expression are intended to be evaluated at compile time whereas others are left until run time. We divide the computation described in the semantics into two parts: a compile time translator and a run time executor.

1 Introduction

Staging transformations were introduced in [JS 86] as a general approach to separating stages or phases of a computation based on the availability of data. We consider two general strategies for staging: partial evaluation and pass separation. In both cases we assume given an interpreter *interp*, a source program *prog* and its data *data* the task of separating the computations formed by *interp*(*prog*,*data*).

Partial Evaluation is the process of specialising a program with respect to part of its input in order to generate a residual program. In our case we are interested in calculating $interp_{prog}$ such that:

$$interp_{prog}\ data = interp\ (prog,data)$$

Thus, the partial evaluation step represents the compilation phase of the computation, and the application of the residual program to the original program's data represents the evaluation phase. The drawback of this approach is that the generated code is in the partial evaluator's output language, typically the language the interpreter is written in. Partial evaluation will not devise a target language suitable for the source language or invent new runtime data structures. However, partial evaluation is automatable and has an established research base [JGS 93].

Pass separation is the process of constructing from a program p a pair of programs p_1,p_2 such that [JS 86]:

$$p(x,y) = p_2(p_1(x),y)$$

The computation $p(x,y)$ can therefore be split into a first stage computing $p_1(x)$, yielding some value v, followed by a second stage computing $p_2(v,y)$. In our

compilation scenario, if we define p to be an interpreter for a programming language, x to be a program in that language and y to be some input data to program x then p_1 becomes the compiler and p_2 executes this compiled code on the data.

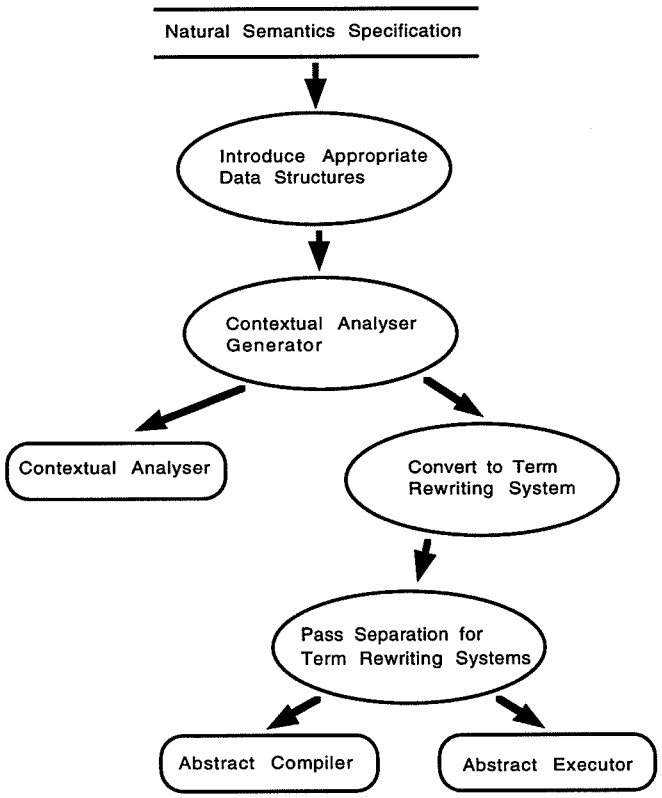
$$interp(prog,data) = exec(comp(prog),data)$$

Hannan presents a series of transformations that automate this split in [Han 91a]. They can separate an interpreter into a translator and corresponding evaluator, each presented as a sequence of rewrite rules, by generating a command language that acts on the given state components. Unfortunately, the technique does not generate new run time data structures automatically or perform any compile time computation (such as replacing identifiers by their memory locations).

We present a method of overcoming these deficiencies by analysing how the environment is used so that appropriate run time data structures are introduced. Followed by performing an initial pass separation to evaluate and encode the compile time computation back into the syntax. We extend the above equation as follows:

$$\begin{aligned} interp(prog,data) &= active\_semantics(contextual\_analyser(prog),data) \\ &= exec(comp(contextual\_analyser(prog)),data) \end{aligned}$$

Along with the corresponding diagram showing the various components of our framework:



We introduce appropriate data structures into the semantics, generating what we call *Implementation Oriented Semantics*, in order to create a distinctive split between compile time binding information and run time objects.
Compile time computation will be carried out by the generated *contextual analyser* that converts syntactical terms into *active syntax*. This allows us to specialise the *Implementation Oriented Semantics* to deal solely with the run time behaviour of a source program described by the *active syntax*.
These residual semantic rules, called the *Active Semantics*, are converted down to term rewriting systems, producing an *Abstract Interpreter*, on which Hannan's pass separation technique is applied.

2 Natural Semantics

An operational semantics is concerned with *how* to execute programs and not merely with what the result of an execution is. It does so by assigning meaning to each language construct in terms of some underlying abstract machine or inference system. A natural semantics describes how the *overall* results of executions are obtained by specifying the relationship between the *initial* and the *final* state for each language construct. Specifications are given in terms of transition relations of the form:

$$env \vdash \langle P, s \rangle \rightarrow_{\mathcal{T}} s'$$

which can be read as in the context *env*, the execution of the phrase P (from the syntactic class $\mathcal{T}$) in state s will terminate and the resulting state will be s'. A rule has the general form:

$$\frac{env \vdash \langle p_1, s_0 \rangle \rightarrow_{\mathcal{T}} s_1 \quad \ldots \quad env \vdash \langle p_n, s_{n-1} \rangle \rightarrow_{\mathcal{T}} s_n}{env \vdash \langle L(p_1, \ldots, p_n), s \rangle \rightarrow_{\mathcal{T}} s'}$$

We shall consider a simple imperative language which has the following syntax:

$$c \in CMD,\ d \in DEC,\ a \in A\_EXP,\ b \in B\_EXP,\ v \in VAR,\ n \in NUM,\ t \in BOOL$$

$$\begin{aligned}
a &::= n \mid v \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
b &::= \mathbf{true} \mid \mathbf{false} \mid a_1 = a_2 \mid b_1 \wedge b_2 \mid \neg b \\
c &::= v := a \mid c_1 \,;\, c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else} c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c \mid \mathbf{begin}\ d \,;\, c\ \mathbf{end} \\
d &::= v := a \,;\, d \mid \varepsilon
\end{aligned}$$

Due to the lack of space we shall concentrate on the rules for assignment and the while loop that demonstrate some of the more interesting features of our approach.

$$\vdots$$

Assign

$$\frac{env \vdash \langle a, M \rangle \rightarrow_a k}{env \vdash \langle v := a, M \rangle \rightarrow_c M[env\ v] \leftarrow k}$$

While_true

$$\frac{\begin{array}{c} env \vdash \langle b, M \rangle \rightarrow_b \mathbf{tt} \\ env \vdash \langle c, M \rangle \rightarrow_c M' \\ env \vdash \langle \mathbf{while}\ b\ \mathbf{do}\ c, M' \rangle \rightarrow_c M'' \end{array}}{env \vdash \langle \mathbf{while}\ b\ \mathbf{do}\ c, M \rangle \rightarrow_c M''}$$

While_false

$$\frac{env \vdash \langle b, M \rangle \rightarrow_b \mathbf{ff}}{env \vdash \langle \mathbf{while}\ b\ \mathbf{do}\ c, M \rangle \rightarrow_c M}$$

3 Implementation Oriented Semantics

The initial phase of our framework is similar to a partial evaluator's, namely that we analyse the semantics in order to deduce what computation can be undertaken at compile time. However, the main thrust of our analysis is to decide how best to implement the language at run time as opposed to a partial evaluator's which will attempt to undertake as much computation at compile time as possible.

Thus, we are concerned with the flow of declarative and transient information described by the semantics. For the former, we are interested in splitting environments into symbol tables and associated run time memories. In our simple imperative language all bindings are static such that variables can be allocated memory cells at compile time. However, if we add procedures with static bindings to the language then our symbol table will consist of mappings from identifiers to *level* and *displacement* pairs, along with a run time stack of activation records. Alternatively, if our procedures have dynamic bindings then we are forced to leave the environment as a run time data structure and introduce *dumps* to maintain the flow of information.

For transient information, such as the results of expressions, we need to introduce suitable data structures to maintain the values of intermediate results for as long as they are required. This will normally consist of either *register* or *stack* introduction.

In our example language we introduce a stack to evaluate both boolean and arithmetic expressions. We model the environment with a symbol table and a pointer, *top*, to the next free location in the memory.

$$\vdots$$

Assign

$$\frac{(\underline{sym},\underline{top}) \vdash \langle a,(S,M)\rangle \rightarrow_a (k{:}S,M)}{(\underline{sym},\underline{top}) \vdash \langle v := a,(S,M)\rangle \rightarrow_c (S,M[\underline{sym}\ v] \leftarrow k)}$$

While_true

$$\frac{\begin{array}{c}(\underline{sym},\underline{top}) \vdash \langle b,(S,M)\rangle \rightarrow_b (\mathbf{tt}{:}S,M) \\ (\underline{sym},\underline{top}) \vdash \langle c\ ,(S,M)\rangle \rightarrow_c (S,M') \\ (\underline{sym},\underline{top}) \vdash \langle \mathbf{while}\ b\ \mathbf{do}\ c,(S,M')\rangle \rightarrow_c (S,M'')\end{array}}{(\underline{sym},\underline{top}) \vdash \langle \mathbf{while}\ b\ \mathbf{do}\ c,(S,M)\rangle \rightarrow_c (S,M'')}$$

While_false

$$\frac{(\underline{sym},\underline{top}) \vdash \langle b,(S,M)\rangle \rightarrow_b (\mathbf{ff}{:}S,M)}{(\underline{sym},\underline{top}) \vdash \langle \mathbf{while}\ b\ \mathbf{do}\ c,(S,M)\rangle \rightarrow_c (S,M)}$$

Compile time data structures and computations are underlined.

4 Contextual Analyser Generator

Using the annotated semantics of the previous section we can perform our first pass separation by generating functions that convert source programs into *active syntax* terms along with a *specialised* version of the semantics.

The *Contextual Analyser* will consist of a series of functions mapping syntactic phrases of a source program onto their corresponding active syntactic representations. The reason for which contextual information is passed to each function is so that compile time evaluation can be undertaken and inserted into the new active syntactic term. Typical examples of this will be to replace strings representing basic values by the basic values themselves and to replace identifiers by their run time locations (stored in the compile time symbol table).

Contextual Analyser

$$\vdots$$

$$\begin{aligned} \mathcal{C}([\![v := a]\!],(sym,top)) &= \textbf{Assign}(\text{addr}, \text{arg}) \\ &\quad \text{where} \\ &\quad\quad \text{addr} = sym\ v \\ &\quad\quad \text{arg} = \mathcal{A}([\![a]\!],(sym,top)) \end{aligned}$$

$$\begin{aligned} \mathcal{C}([\![\textbf{while}\ b\ \textbf{do}\ c]\!],(sym,top)) &= \textbf{While}(\text{arg1}, \text{arg2}) \\ &\quad \text{where} \\ &\quad\quad \text{arg1} = \mathcal{B}([\![b]\!],(sym,top)) \\ &\quad\quad \text{arg2} = \mathcal{C}([\![c]\!],(sym,top)) \end{aligned}$$

The compile time behaviour of a source program will have been computed by the Contextual Analyser so that a residual semantic description will be sufficient to describe the remaining run time behaviour.

Active Semantics

$$\vdots$$

Assign

$$\frac{\langle a,(S,M)\rangle \rightarrow (k{:}S,M)}{\langle \textbf{Assign}(addr,a),(S,M)\rangle \rightarrow (S,M[addr] \leftarrow k)}$$

While_true

$$\frac{\langle b,(S,M)\rangle \rightarrow (\textbf{tt}{:}S,M) \quad \langle c,(S,M)\rangle \rightarrow (S,M') \quad \langle \textbf{While}(b,c),(S,M')\rangle \rightarrow (S,M'')}{\langle \textbf{While}(b,c),(S,M)\rangle \rightarrow (S,M'')}$$

While_false

$$\frac{\langle b,(S,M)\rangle \rightarrow (\textbf{ff}{:}S,M)}{\langle \textbf{While}(b,c),(S,M)\rangle \rightarrow (S,M)}$$

5 Converting Active Semantics into Term Rewriting Systems

If we consider that an inference rule has a simple conclusion $\mathcal{A}$ and possibly many premises $\mathcal{A}_1$ ... $\mathcal{A}_n$ then we might read a rule as saying: "to prove $\mathcal{A}$ we should prove $\mathcal{A}_1$ and ... and $\mathcal{A}_n$". The aim of this section is to show how we can derive an Abstract Interpreter for active syntax terms that corresponds to a depth first left to right proof search through the Active Semantics. However, to be able to translate inference rules to term rewriting rules we need to eliminate the need for backtracking. Consider the case when we have a proof state $\langle \textbf{While}(b,c),(S,M)\rangle$ for which both *While_true* and *While_false* are candidates, but at most one is applicable. We deal with the problem in a similar manner to the factorization of context free production rules with common initial segments. The two rules used to define the while loop are factored by introducing a new language constructor, **Loop**, which is "activated" after the boolean test is accomplished [daSilva 90].

Factorized Semantics

$$\vdots$$

Assign
$$\frac{\langle a,(S,M)\rangle \rightarrow (k{:}S,M)}{\langle \textbf{Assign}(addr,a),(S,M)\rangle \rightarrow (S,M[addr] \leftarrow k)}$$

While
$$\frac{\langle b,(S,M)\rangle \rightarrow (bv{:}S,M) \quad \langle \textbf{Loop}(b,c),(bv{:}S,M)\rangle \rightarrow (S,M')}{\langle \textbf{While}(b,c),(S,M)\rangle \rightarrow (S,M')}$$

Loop_true
$$\frac{\langle c\ ,(S,M)\rangle \rightarrow (S,M') \quad \langle \textbf{While}(b,c),(S,M')\rangle \rightarrow (S,M'')}{\langle \textbf{Loop}(b,c),(\textbf{tt}{:}S,M)\rangle \rightarrow (S,M'')}$$

Loop_false
$$\langle \textbf{Loop}(b,c),(\textbf{ff}{:}S,M)\rangle \rightarrow (S,M)$$

We translate each factorized rule into a rewrite rule by following the left to right ordering of the premises; inserting suitable *instructions* when the output state of one transition does not match the input state of the subsequent one. If we consider the rule for assignment then the output state of the arithmetic sub expression does not match the output state of the conclusion. Thus, we introduce a new instruction, *STORE(addr)*, which will place the result of the expression into the memory cell belonging to that particular variable.

Abstract Machine

$$
\begin{array}{lll}
 & \vdots & \\
< ev(\textbf{Assign}(addr,a)):C,\ (S,M) > & \Rightarrow & < ev(a):STORE(addr):C,\ (S,M) > \\
< STORE(addr):C,\ (k:S,M) > & \Rightarrow & < C,\ (S,M[addr]\leftarrow k) > \\
< ev(\textbf{While}(b,c)):C,\ (S,M) > & \Rightarrow & < ev(b):ev(\textbf{Loop}(b,c)):C,\ (S,M) > \\
< ev(\textbf{Loop}(b,c)):C,\ (\textbf{tt}:S,M) > & \Rightarrow & < ev(c):ev(\textbf{While}(b,c)):C,\ (S,M) > \\
< ev(\textbf{Loop}(b,c)):C,\ (\textbf{ff}:S,M) > & \Rightarrow & < C,\ (S,M) >
\end{array}
$$

These rules form part of the abstract interpreter for active syntax terms and should be viewed as an evaluation model for the Active Semantics.

6 Pass Separation on Abstract Machines

The second pass separation that we apply aims to generate an *abstract compiler* that lifts the active syntax terms out of the abstract machine by rewriting them to a sequence of *instructions*, along with an *abstract executor* that evaluates these instructions on some initial state. Transformations which achieve this are presented in [Han 91a] and are completely mechanical and automatic.

The rules of the abstract machine are of the form $< p,s > \Rightarrow_{\mathcal{R}} < p',s' >$. Pass separation involves constructing two sets $\mathcal{R}_C,\mathcal{R}_X$ of rewrite rules such that $< u,v > \hat{\Rightarrow}_{\mathcal{R}} < u',v' >$ iff $u \hat{\Rightarrow}_{\mathcal{R}_C} u_C$ and $< u_C,v > \hat{\Rightarrow}_{\mathcal{R}_X} < u',v' >$. Where we view the rules $\mathcal{R}_C$ as forming an abstract compiler, while the rules $\mathcal{R}_X$ form the corresponding abstract executor.

Abstract Compiler

$$
\begin{array}{ll}
 & \vdots \\
ev(\textbf{Assign}(addr,a)):C & \Rightarrow_C\ ev(a):STORE(addr):C \\
ev(\textbf{While}(b,c)):C & \Rightarrow_C\ ev(b):ev(\textbf{Loop}(b,c)):C \\
ev(\textbf{Loop}(b,c)):C & \Rightarrow_C\ LOOP(ev(b):nil,ev(c):nil):C
\end{array}
$$

Abstract Executor

$$
\begin{array}{lll}
 & \vdots & \\
< STORE(addr):C,\ (k:S,M) > & \Rightarrow x & < C,\ (S,M[addr]\leftarrow k) > \\
< LOOP(b,c):C,\ (\textbf{tt}:S,M) > & \Rightarrow x & < c@b@LOOP(b,c):C,(S,M) > \\
< LOOP(b,c):C,\ (\textbf{ff}:S,M) > & \Rightarrow x & < C,\ (S,M) >
\end{array}
$$

An interesting product of Hannan's staging transformations is the construction of a semantics-directed machine architecture. Other approaches require the language designer to either specify their language using a fixed and sufficiently powerful combinator language, such as Action Semantics [Mosses 92], or to transform their semantics into a compiler and executor pair by choosing special purpose combinators themselves [Wand 82].

7 Summary

We have presented a framework for generating compilers based on the notion that the various constructs of a programming language have times of meanings as well as meanings. We have achieved this by extending Hannan's pass separation technique to include the contextual analysis phase and the conversion from inference rules to term rewriting rules. However, much work still remains. We have yet to formalise a suitable binding time analysis that would enable us to introduce the appropriate data structures; or looked at how to map the resulting abstract executors on to real hardware by extending the refinements given in [Han 91b].

References

[daSilva 90] da Silva,F., *Towards a Formal Framework for Evaluation of Operational Semantics Specifications*. LFCS Report ECS-LFCS-90-126, Edinburgh University (1990).

[Han 91a] Hannan,J., *Staging Transformations for Abstract Machines*. Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation (1991), 130-141.

[Han 91b] Hannan,J., *Making Abstract Machines Less Abstract*. Fifth ACM Conference on Functional Programming Languages and Computer Architecture, LNCS 523 (1991), 618-635.

[JGS 93] Jones,N., Gomard,C., Sestoft,P., *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science (1993).

[JS 86] Jørring,U., Scherlis,W., *Compilers and Staging Transformations*. Sixteenth ACM Symposium on Principles of Programming Languages (1986), 281-292.

[Kahn 87] Kahn,G., *Natural Semantics*. Fourth Annual Symposium On Theoretical Aspects of Computer Science, LNCS 247 (1987), 22-39.

[Mosses 92] Mosses,P., *Actions Semantics*. Cambridge Tracts in Theoretical Computer Science (1992).

[NN 92] Nielson,H., Nielson,F., *Semantics with Applications*. John Wiley & Sons (1992).

[Wand 82] Wand,M., *Semantics-Directed Machine Architecture*. Ninth ACM Symposium on Principles of Programming Languages (1982), 234-241.

A Demonstration of ASD
The Action Semantic Description Tools

Arie van Deursen* Peter D. Mosses†

Introduction *Action Semantics* is a framework for describing the semantics of programming languages [4, 6]. One of the main advantages of Action Semantics over other frameworks is that it scales up smoothly to the description of larger practical languages, such as Standard Pascal [5]. An increasing number of researchers and practitioners are starting to use action semantics in preference to other frameworks.

The ASD tools include facilities for parsing, syntax-directed (and textual) editing, checking, and interpretation of action semantic descriptions. Such facilities significantly enhance accuracy and productivity when writing large specifications, and are also particularly useful for students learning about the framework. The notation supported by the ASD tools is a direct ASCII representation of the standard notation used for action semantic descriptions in the literature, as defined in [4, Appendices B–F].

Action Semantic Descriptions The notation used in action semantic descriptions can be divided into four kinds:

Meta-Notation, used for introducing and specifying the other notations;

Action Notation, a fixed notation used for expressing so-called *actions*, which represent the semantics of programming constructs;

Data Notation, a fixed notation used for expressing the data processed by actions; and

*Email: arie@cwi.nl. Address: CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands. Supported by the EC under ESPRIT project 2177 *Generation of Interactive Programming Environments* and the Netherlands Organization for Scientific Research NWO project *Incremental Program Generators*

†Email: pdmosses@daimi.aau.dk. Address: BRICS (Basic Research in Computer Science, a Centre of the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade Bldg. 540, DK-8000 Aarhus C, Denmark.

Specific Notation, introduced in particular action semantic descriptions to specify the abstract syntax of the programming language, the semantic functions that map abstract syntax to semantic entities, and the semantic entities themselves (extending the fixed action and data notation with new sorts and operations).

Compared with conventional frameworks for algebraic specification, the meta-notation is unusual in that it allows operations on sorts, not only on individual values. Its foundations are given by the framework of Unified Algebras [3]. Moreover, so-called *mix-fix* notation for operations is allowed, thus there is no fixed grammar for terms. This is a crucial feature, because action notation includes many infix combinators (e.g., A_1 **and then** A_2, which expresses sequencing of the actions A_1, A_2) and mix-fix primitive actions (e.g., **bind** I **to** D). The specific notation introduced by users tends to follow the same style.

The Platform The ASD tools are implemented using the ASF+SDF system [1, 2]. In the ASF+SDF approach to tool generation, the syntax of a language is described using the Syntax Definition Formalism SDF, which defines context-free syntax and signature at the same time. Functions operating on terms over such a signature are defined using (conditional) equations in the algebraic specification formalism ASF. Typical functions describe type checking, interpreting, compiling, etc., of programs. These functions are executed by interpreting the algebraic specifications as term rewriting systems. Moreover, from SDF definitions, parsers can be generated, which in turn are used for the generation of syntax-directed editors. ASF+SDF modules allow hiding and mutual dependence. (The ASD demonstration assumes that the basic features of ASF+SDF are already known, so as to focus attention on this application of the system.)

The ASF+SDF system currently runs on, e.g., Sun4 and Silicon Graphics workstations, and uses X-Windows. It is based on the Centaur system (developed by, amongst others, INRIA) so a Centaur licence is required.[1] Once one has installed the ASF+SDF system, all that is needed before using the ASD tools is to get a copy of the ASD modules and user guide, together with a configuration file that specifies the effects of the various buttons in the ASD interface; these items are freely available by FTP.

The Implementation ASD modules written in the Meta-Notation are translated to ASF+SDF modules, using the ASF+SDF system itself. Concerning the unusual features of the Meta-Notation: sort operations are dealt with by generating (in some cases) extra sorts in the ASF+SDF module; and the arbitrary mix-fix operations are catered for by a two-phase generation scheme.

[1] Academic institutions currently pay FF600 for a copy of the complete Centaur/ASF+SDF distribution tape.

The Demonstration The main features of ASD are demonstrated in turn:

Editing: A previously-prepared action semantic description (a.s.d.) is read into the system. A (deliberate) typo prevents it from being parsed immediately, but clicking on the error message moves the cursor to the point where correction is needed. After correction the a.s.d. parses OK, and by clicking at various points, the structural focus is moved around to exhibit the recognised grouping. Changing part of a term requires reparsing only of the changed part, exploiting the incrementality of ASF+SDF. However, when the introduced (mix-fix) symbols of the a.s.d. are changed, the initially-generated term parser becomes obsolete, and terms remain unparsed until a new parser is generated (by pressing a button).

Parser Generation: An a.s.d. module containing a grammar is read in. A button press generates an ASF+SDF module containing an equivalent grammar, allowing the next step.

Program Parsing: An actual program in the described language is edited; If it can be parsed, a button press transforms it into the abstract syntax notation used in action semantics.

Semantics Generation: An a.s.d. module specifying semantic functions (by semantic equations, as in denotational semantics) is read in. A button press generates an ASF+SDF module that can execute the semantic equations as rewrite rules, which allows the next step.

Program Semantics: Given these rewrite rules and an actual program in abstract syntax notation, a button press can map this program to its corresponding action term.

Sort Checking: The ASF+SDF modules generated in the preceding steps incorporate basic sort-checking of the usage of operations in terms, exploiting the so-called functionality axioms specified in a.s.d.s.

Further features are currently being implemented.

References

[1] J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[2] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.

[3] P. D. Mosses. Unified algebras and institutions. In *LICS'89, Proc. 4th Ann. Symp. on Logic in Computer Science*, pages 304–312. IEEE, 1989.

[4] P. D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[5] P. D. Mosses and D. A. Watt. Pascal action semantics. Version 0.6. Available by FTP from ftp.daimi.aau.dk in pub/action/pascal, Mar. 1993.

[6] D. A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

Hardware and Software Requirements

For installing and demonstrating the ASD system:

Disk Space

- 200 Mbytes needed to install and store ASD.

Workstation

- Minimum 32 Mbytes main memory needed for running the demonstration.
- Preferably Silicon Graphics, running with:
  – Operating System IRIX Release 4.05F (or higher),
  – MIPS R4010/R4000.

  Alternatively:
  Sparc running SunOS 4.1 (not Solaris), or
  Silicon Graphics with MIPS R2000/3000.
- Standard Unix software needed: X-Windows, twm.
- Colour screen desirable, but not essential.

PASCAL definition in the system LDL

Günter Riedewald, Ralf Lämmel
Universität Rostock, Fachbereich Informatik
18051 Rostock, Germany
E-mail: (gri | rlaemmel) @ informatik.uni-rostock.de

Abstract. LDL is a system supporting the design of procedural programming languages and generating prototype interpreters directly from language definitions. Language definitions are based on GSFs - a kind of attribute grammars - and the denotational semantics approach. Semantics is defined in a two-level approach more or less similar to action semantics. First a term representing the semantic meaning is constructed and afterwards this term is interpreted. To derive (within the system LDL) a prototype interpreter of a language its language definition must be transformed to Prolog. Language definitions within LDL and the transformations into Prolog are considered using a PASCAL-like language as an example. The underlying approach for language definition, especially semantics definition, is compared with other approaches. Moreover it is sketched how our approach to language definition could be adapted for interconnecting attribute grammars (GSFs) and action semantics which would allow a more appropriate semantics definition (including static semantics) than in the case of pure action semantics.

1 Introduction

This paper is structured as follows. The subsections 1.1, 1.2 establish some basic knowledge about the system LDL, language definitions applied in LDL and the derived LDL prototype interpreters implemented as Prolog programs. In section 1.3 a PASCAL-like language MYPAS serving as running example for this paper is introduced. In sections 2 and 3 we discuss the formalisms for language definition applied in LDL, i.e. GSFs (a kind of attribute grammars) and denotational semantics, and the implementation of such language definitions for purposes of prototyping interpreters. GSFs are considered more in detail, since we want to sketch at the end of the paper (section 5: Conclusion and Future work) how GSFs - especially GSFs of that specific form we are applying in the system LDL - could be useful to be interconnected with action semantics descriptions. In Section 4 we give references to some related work.

1.1 Structure of LDL - *L*anguage *D*evelopment *L*aboratory

Keeping in mind Koskimies' statement ([K91]) "The concept of an attribute grammar is too primitive to be nothing but a basic framework, the 'machine language' of language implementation." LDL offers a higher-level tool supporting the definition of (at least procedural) languages and their implementation in form of a prototype interpreter. For this purpose the LDL library (Fig. 1) contains predefined language constructs together with their static and dynamic semantics and the Prolog implementation of these. The (dynamic) semantics components are correct w.r.t. the usual denotational definition. The knowledge base and the tool

for language design ensure the derivation of correct prototype interpreters from these correct components. Moreover, LDL derives test program generators producing syntactically correct programs satisfying the context conditions of the defined language and possessing certain additional properties.

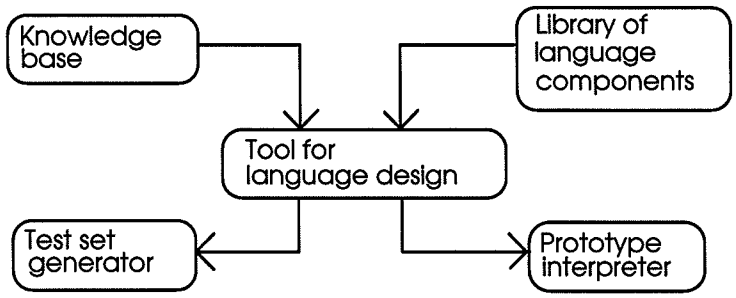


Fig.1: Structure of the LDL system

1.2 Language definitions and prototype interpreters within LDL

The language definitions and the corresponding prototype interpreters in the system LDL are based on the idea from [R91] and exploit GSFs (GSF - Grammar of Syntactical Functions) - a kind of attribute grammars - and denotational semantics descriptions for the language definition.

The development of prototype interpreters is based on the following ideas:

- Because GSFs and Prolog programs are closely related, after some modifications a language definition in form of a GSF can directly be used as the core of a prototype interpreter written in Prolog and applicable for syntactical and semantic analysis.
- Denotational semantics descriptions can be implemented as logical programs by defining term representations of elements of any domain and by transforming the functional equations into definite clauses.
- The semantics definition can be a stepwise process. First, we could be interested only in the calling structure of the semantic functions of a given source program. Finally, we are interested in the execution (interpretation) of the source program. Thus our semantics definition consists of two levels:
  1. The meaning of a program is a term consisting of names of semantic functions in the GSF sense which can be considered as the abstract syntactical structure of the program. It can be defined using a GSF with special production rule patterns (see subsection 2.2).
  2. Based on the denotational approach the interpretation of terms is defined.
- Before computing the meaning of a source program according to the two levels of the semantics definition its context conditions are checked (evaluation of the auxiliary syntactical functions in the GSF sense).

The structure of a prototype interpreter can be seen from Fig. 2.

The prototype interpreter operates as follows:

- A source program is read token by token from a text file.
- Each token is classified by a scanner. The scanner is invoked by a special operator preceding each terminal within the Prolog version of the GSF.
- The parsing and checking of context conditions is interconnected with scanning. If the context-free basic grammar of the GSF describing the source language is an LL(k)-grammar the Prolog system itself can be used straightforwardly for parsing, whereas LR(k)-grammars require to include a special parser into the prototype interpreter.
- Recognizing a language construct its meaning in form of a term is constructed by connecting the meanings of its subconstructs.
- The term representing the meaning of the whole program is interpreted, i.e. the function names of the term are associated with functions transforming a given program state into a new one, where a state is usually an assignment of values to program variables.

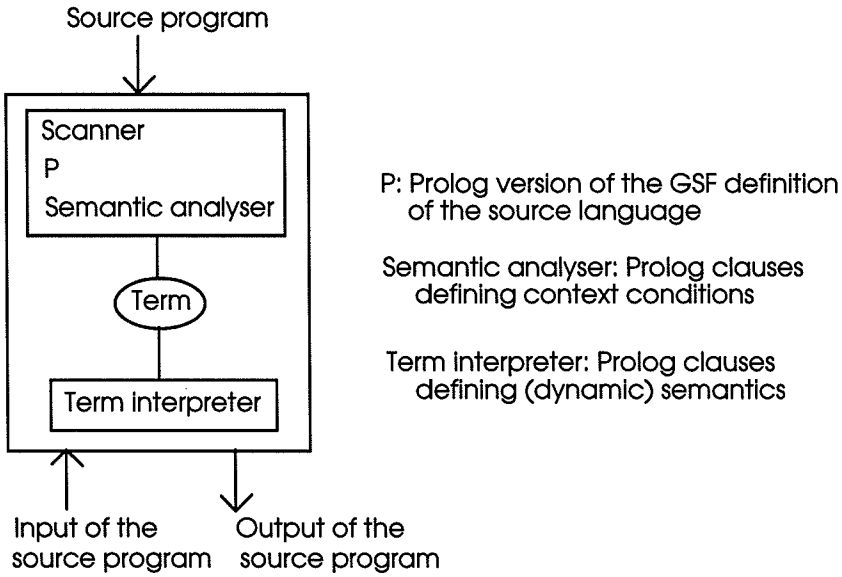


Fig.2: Structure of a prototype interpreter

1.3 MYPAS - a PASCAL-like language developed by LDL

MYPAS is a PASCAL-like language which has been designed to consider a non-trivial example of an imperative language in the system LDL. MYPAS was an experiment to explore possibilities for the transformation of denotational semantics into logical programs / Prolog programs ([Lä93]).

MYPAS does not contain the following PASCAL-constructs:

- sets, enumerated / subrange types
- records with variants
- CASE statement
- some standard procedures / functions
- forward

Additionally to PASCAL the following constructs have been included:

- structured result types for functions
- break / continue statements

The LDL prototype interpreter of MYPAS can be applied to interprete non-trivial MYPAS programs and has passed several tests, for example the heavy scope test for static binding from [WG84]), standard algorithms (e.g. for sorting or matrix calculations) and little applications (e.g. file-oriented data management programs). Up to now we have not compared the speed of the prototype interpreters with the speed known for other systems dealing with prototype interpreters derived from formal language definitions. However we expect the term interpretation (i.e. the implementation of the denotational semantics) to be comparable in speed with the approach of executing denotational semantics in a functional language like SML. In general the speed of interpretation and the size of inputs which can be processed by LDL prototype interpreters strongly depends on the fact whether features of Prolog are exploited within the term interpreter (we did so for MYPAS) in difference to deriving pure logical programs with a clean declarative meaning (allowing provable correctness as for the LDL prototype interpreter of the language VSPL, see [LR94]).

Only to give an idea on the size of the obtained prototype interpreter definition we mention numbers of clauses and file sizes (including comments for any clause) for all its parts in Table 1.

Part of definition	**Clauses**	**KB ASCII**
GSF	234	28
static semantics	269	21
dynamic semantics	201	24
applied modules	161	26
Σ	865	99

Table 1: Size of parts of the MYPAS definition

Figure 3 represents the structure of the prototype interpreter definition more in detail. The immediate parts are derived from the language definition consisting of a GSF and a denotational semantics description. The abstract data types offer (reusable) services for the semantic analysis and the term interpretation.

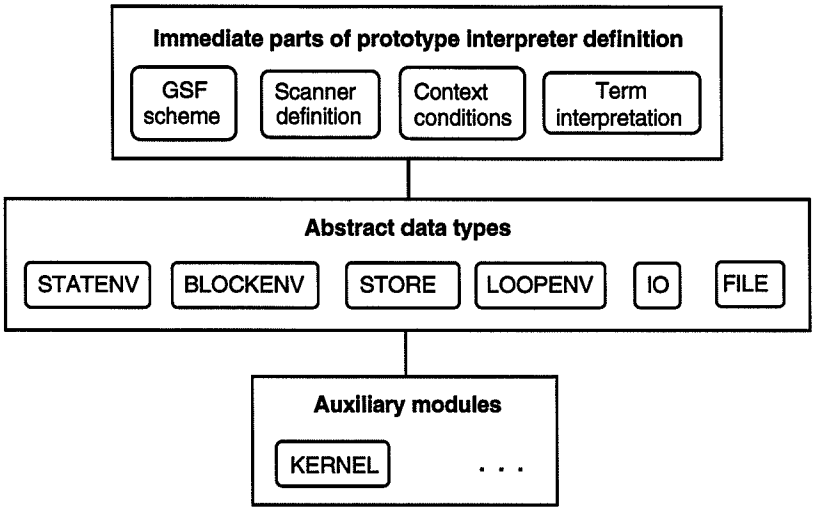


Fig.3: Structure of the MYPAS prototype interpreter definition

2 GSFs - Grammars of Syntactical Functions

2.1 Definition of GSFs

The GSF formalism ([R91]) is closely related to the DCG ([PW80]) and RAG ([CD87]) formalisms, but, other than these, it has been derived from two-level grammars during 1971-1972 with the aim to obtain an executable and more readable form of two-level grammars. A GSF definition consists of two parts:

- a GSF scheme defining the rough structure of the syntax and semantics of a language
- a GSF interpretation refining the GSF scheme.

Roughly speaking a GSF is a parametrized context-free grammar extended by relations over the parameters. For historical and practical reasons, in the following definitions these relations are classified into auxiliary syntactical functions and semantic functions. Defining a programming language auxiliary syntactical functions and semantic functions can be used to define the static and dynamic semantics, respectively.

Definition 1 (GSF scheme):
A *GSF scheme* is a tuple S = <B, A, SF, V, C, AR, R>,
where B = <N, T, R´,ST> is a reduced context-free grammar (N set of nonterminals - here called *names of syntactical functions*, T set of terminals - *names of basic syntactical functions*, R´ set of production rules, ST ∈ N start symbol) - the *basic grammar* of the GSF, and A, SF, V and C are finite sets of *names of auxiliary syntactical functions, names of semantic functions, variables and constants* resp.. V ∪ C is the set of *parameters.* R is a finite set of *production rule patterns,* each of the form

$f_0(P_{f0,1},...,P_{f0,nf0})$:
$f_1(P_{f1,1},...,P_{f1,nf1}),...,\ f_r(P_{fr,1},...,P_{fr,nfr}),$
$h_1(P_{h1,1},...,P_{h1,nh1}),...,\ h_s(P_{hs,1},...,P_{hs,nhs}).$ (1)

where $f_0 \in N$, $f_1,...,f_r \in N \cup T$, $h_1,...,h_s \in A \cup SF$,
$P_{f0,1},...,P_{hs,nhs} \in V \cup C$ and
$f_0: f_1,...,f_r \in R'$ (2)

N, T, A, and SF are pairwise disjoint. V and C are disjoint too. The *arity* AR maps each function name (element of N U T U A U SF) into the set of integers (number of parameters of a function). $g(P_1,...,P_n)$ is called *syntactical function, basic syntactical function, semantic function* or *auxiliary syntactical function* if $g \in N$, $g \in T$, $g \in SF$, $g \in A$ resp. Each syntactical function $ST(P_1,...,P_n)$ occurring on the left-hand side of some production rule pattern is a *start element* of the GSF.

■

Example 1 (Excerpt of the GSF scheme of MYPAS):

```
% concatenation of statements
sm_list(S,ST) : statement(S1,ST),";",sm_list(S2,ST),
                CONCAT(S,S1,S2).
sm_list(S,ST) : SKIP(S).

% concrete statements
statement(S,ST) : assign_statement(S,ST).
statement(S,ST) : if_statement(S,ST).
. . .

% assign statement
assign_statement(S,ST) : left_value(E1,T1,ST),
                         ":=",expression(E2,T2,ST),
                         CHECK_ASSIGN_TYPES(T1,T2,ST),
                         ASSIGN(S,E1,E2).

% if statement with optional else-part
if_statement(S,ST) : "if",expression(E,T,ST),
                     "then",sm_list(S1,ST),
                     else_part(S2,ST),
                     "fi",
                     IS_BOOLEAN_TYPE(T,ST),
                     IF(S,E,S1,S2).

else_part(S,ST) : "else",sm_list(S,ST).
else_part(S,ST) : SKIP(S).
```

sm_list, statement, assign_statement, if_statement, left_value, expression, else_part ∈ N,
";", ":=", "if", "then", "else", "fi" ∈ T,
CHECK_ASSIGN_TYPES, IS_BOOLEAN_TYPE ∈ A,
CONCAT, SKIP, ASSIGN, IF ∈ SF,
$S,S_1,S_2,ST,E,E_1,E_2,T,T_1,T_2 \in V$.

■

From the definition it can be seen that a GSF scheme defines the context-free basic structure of a language and the dependencies between auxiliary syntactical and/or

semantic functions. To determine the meaning of a language construct we need to know concrete parameter domains and the meaning of auxiliary syntactical and semantic functions.

Definition 2 (GSF, interpretation):
Suppose S = <B, A, SF, V, C, AR, R> is a GSF scheme as introduced in the previous definition. A *GSF* is a pair <S, IP>, where IP = <M, D, I, F> is an *interpretation* consisting of a family D of domains, a function I associating with each element f ϵ A ∪ SF an n-ary relation on the domains from D (n=AR(f)), a function M assigning to the i-th parameter position of a function name f a particular domain M(f,i) ϵ D and the *forbidden symbol* F.
$f(v_1,...,v_n)$ with v_i ϵ M(f,i) is called an *instance* of the function $f(P_1,...,P_n)$.
Moreover, the following conditions must be satisfied:

- A variable occurring on the i-th parameter position of a function $f(P_1,...,P_n)$ stands for a value from M(f,i). It represents the same value whenever it occurs in a given production rule pattern.
- A constant occurring on the i-th parameter position of f is an element from the domain M(f,i).
- If f ϵ A ∪ SF, AR(f)=n, and a_i ϵ M(f,i), i=1,...,n,
  then $f(a_1,...,a_n)$ = { ε, if $(a_1,...,a_n)$ ϵ I(f)
  { F, else
  where ε denotes the empty string.
- For each production rule pattern there are variables occurring as well in the syntactical functions as in auxiliary or semantic functions.

■

Example 2 (Continuation of Example 1):
If S, E denote the sets of meanings of statements and expressions resp., ST is the set of all possible symbol tables, T the set of all types, then the function M (domains of parameter positions of function names) can be defined by the Table 2.

f \ i	1	2	3	4
sm_list	S	ST	-	-
statement	S	ST	-	-
assign_statement	S	ST	-	-
if_statement	S	ST	-	-
left_value	E	T	ST	-
expression	E	T	ST	-
else_part	S	ST	-	-
CHECK_ASSIGN_TYPES	T	T	ST	-
IS_BOOLEAN_TYPE	T	ST	-	-
CONCAT	S	S	S	-
SKIP	S	-	-	-
ASSIGN	S	E	E	-
IF	S	E	S	S

Table 2: Domains of parameter positions of GSF of MYPAS

The relations associated with the names of auxiliary syntactical and semantic functions are described here only informally.

- I(CHECK_ASSIGN_TYPES) = { (t_1,t_2,st) | t_1,t_2 ϵ T are types valid for the left-hand and right-hand side of an assignment; st ϵ ST defines user types}
- I(IS_BOOLEAN_TYPE) = { (t,st) | t ϵ T denotes the boolean type; st ϵ ST defines user types}
- I(SKIP) = { s | s ϵ S is the meaning of the empty statement }
- I(CONCAT) = { (s,s_1,s_2) | s ϵ S is the meaning of a concatenation of statements with the meanings s_1,s_2 ϵ S }
- I(ASSIGN) = { (s, e_1, e_2) | s ϵ S is the meaning of an assignment depending on the meanings e_1,e_2 ϵ E of the left-hand and the right-hand side}
- I(IF) = { (s,e,s_1,s_2) | s ϵ S is the meaning of an IF-statement where e ϵ E is the meaning of the conditional expression and s_1,s_2 ϵ S are the meanings of the THEN/ELSE-path resp. }

■

To generate a word by a GSF first suitable production rule patterns must be turned into context-free production rules replacing each variable occurring in the given production rule pattern by a value from its corresponding domain. This substitution process is controlled by the relations also occurring in the production rule pattern.

Definition 3 (Derived context-free production rule):
Suppose G = <S, IP> is a GSF with the GSF scheme S = <B,A,SF,V,C,AR,R> and the interpretation IP = <M, D, I, F> as introduced above. Then

$$F_0 : F_1,...,F_r. \qquad (3)$$

is a context-free *production rule derived from the production rule pattern* (1) if (3) is a result of consistently replacing each variable occurring in (1) by a value from its corresponding domain (F_i, i=0,1,...,n is an instance of $f_i(...)$) in such a way that the instances $H_1,...,H_s$ of the auxiliary syntactical functions and semantic functions occurring in (1) yield ε.

■

Definition 4 (Word, Language):
Let G be a GSF defined as above. A string w consisting of terminals of G (instances of basic functions) is a *word* generated by G iff there is an instance $ST(v_1,...,v_n)$ of some start element $ST(P_1,...,P_n)$ of G such that $ST(v_1,...,v_n)$ $=\overset{+}{=}>$ w applying a suitable set of context-free production rules derived from the set of production rule patterns of G. In an analogous way *subwords* can be defined.
The *language* L(G) generated by the GSF G is the set of words generated by G.

■

Based on the language generated by a GSF it is possible to associate relations with the names of syntactical functions and a meaning with each word.

Definition 5 (Relation associated with a syntactical function):
Let G be a GSF defined as above and f a given name of a syntactical function. Rel(f) is a n-ary *relation* (n=AR(f)) and ER(f) a (n+1)-ary *relation associated with* f, where $(v_1,...,v_n) \in Rel(f)$ and $(w, v_1,...,v_n) \in ER(f)$ iff $f(v_1,...,v_n) \overset{*}{\Longrightarrow} w$, $uwu' \in L(G)$, $v_i \in M(f,i)$, i=1,...,n.

■

Now the meaning of a word (subword) w can be defined as the tuple $(v_1,...,v_n)$ iff $(w, v_1,...,v_n) \in ER(ST)$ ($(w, v_1,...,v_n) \in ER(f)$, $f \in N$, $f \neq ST$). It can also be identified with a subtuple of this tuple.

Example 3:
With the syntactical function statement(S, ST) from Example 1 we can associate the relation ER(statement) = { (w, s, st) | s is the meaning of the statement w generated by statement(s, st), where st is a symbol table containing the declarations visible in w} .

■

2.2 Specific application of GSFs in the system LDL

In our applications usually the *first* parameter of each syntactical function is assumed to denote the meaning of subwords generated by the syntactical function. Then the meaning of the syntactical function of the left-hand side of a production rule pattern is computed by a semantic function from the meanings of the right-hand side syntactical functions. Thus, only the following two kinds of production rule patterns are possible:

- First kind:
  $f(c,...) : b.$ $\quad c \in C, f \in N,$
  b sequence of parameterless basic syntactical functions.
  Remark: We suppose that basic syntactical functions with non-empty parameter lists are defined by implicitly given rules of the first kind, e.g. identifier(x) : 'x'. This mirrors the situation in compilers that identifiers and other classes of terminals are recognized by lexical analysers.

- Second kind (The meaning p_0 is computed from the meanings $p_1,...,p_r$.):
  $f_0(p_0,...) :$ $\quad b_0,f_1(p_1,...),b_1,f_2(p_2,...),...,f_r(p_r,...),b_r,$
  $a_1(...),...,a_s(...),SM(p_0,p_1,...,p_r).$
  $f_0 \in N$, $f_1,...,f_r \in N \cup T$,
  $b_0,...,b_r$ sequences of parameterless basic syntactical functions,
  $a_1,...,a_s \in A$,
  $SM \in SF$, $p_0 \in V$, $p_1,...,p_r \in V \cup C$
  and $I(SM) = \{ (y,x_1,...,x_r) \mid F_{SM}(x_1,...,x_r) = y,$
  $x_i \in M(f_i,1), i = 1,...,r, y \in M(f_0,1) \}$,
  F_{SM} suitable function.

(4)

Interested readers may refer to [RL93], where the nice property of GSFs with such production rule patterns to define the meaning of a word (subword) as a homomorphic image of the structure of the word (subword) is considered.

Because of our two-level approach exploiting
- GSFs for syntactical / semantical analysis and term generation
- denotational semantics for dynamic semantics (term interpretation)

formally we want to use a GSF for associating with each word generated by the GSF as its meaning its syntactical structure in form of a term. Therefore after the consideration of relations between context-free grammars / GSFs and algebras we introduce the notion of a *syntactical algebra associated with a GSF*.

It is well-known that a context-free grammar can be considered as a heterogeneous algebra ([ADJ77]). Let be G=(N,T,P,S) a context-free grammar, where N is the set of nonterminals, T is the set of terminals, P is the set of production rules and S ∈ N is the start symbol. Considering G as algebra N can be identified with the set of sorts. To each production rule p ∈ P

$X_0 \rightarrow a_0 X_1 a_1 \ldots X_n a_n$,

where $X_i \in N$, $i = 0,\ldots,n$, $a_j \in T^*$, $j = 0,\ldots,n$, an operator o_p with the profile $\langle X_1 \ldots X_n, X_0 \rangle$ is assigned in a one-to-one manner. Let be O the set of all operators constructed. O together with the profiles of the operators is an N-sorted signature. Now a *syntactical algebra* SA can be defined as follows:

$SA = \langle \{SA_s\}_{s \in N}, \{o_{pSA}\}_{op \in O} \rangle$, where

$SA_s = L(s)$ (the set of all strings from T^* generated by s)

$o_{pSA}(x_1,\ldots,x_n) = a_0 x_1 a_1 \ldots x_n a_n$

if o_p is defined as above

A more abstract syntactical algebra is the N-sorted term algebra T(O). But if G is reduced and unambigous then both algebras are isomorphic.

It is possible to construct a syntactical algebra associated with a GSF as the syntactical algebra of its context-free basic grammar. But GSFs enable a second approach. The parameters occurring in the parameter lists of the syntactical functions can be used to refine the sorts and the signature given by the context-free basic grammar of the GSF scheme.

Definition 6 (Syntactical algebra associated with a GSF):
Let G be a GSF as defined above. A *syntactical algebra associated with* G is a term algebra based on the following signature Sig (*signature associated with* G):

1. Suppose Ref maps each name of a syntactical function into the set of integers (Ref selects a refinement parameter; $0 \leq Ref(f) \leq AR(f)$; Ref(f)=0 means there is no refinement parameter). Then the set of sorts of Sig is

   $\{f_v\}_{f \in N \cup T' \wedge v \in M(f,Ref(f))}$ (f_v is f if Ref(f)=0), where

   T' is the set of names of basic syntactical functions with non-empty parameter lists and M(f, Ref(f)) is the domain of the Ref(f)-th parameter position of f.

2. Suppose

$f_0(...,P_{f0},Ref(f0),...)$: $f_1(...,P_{f1},Ref(f1),...),...$ (*)
$f_r(...,P_{fr},Ref(f_r),...)$,
$h_1(...),...,h_s(...)$.

is a production rule pattern of the GSF scheme of G,

$f_0(...,v_0,...)$: $f_1(...,v_1,...),...,f_r(...,v_r,...)$, (**)
$h_1(...),...,h_s(...)$.

is a result of substituting $v_i \in M(f_i, Ref(f_i))$ for the $Ref(f_i)$-th parameter position of f_i and there is a context-free production rule derived from (**). Then we introduce an operator o: $f_{1v1} \times ... \times f_{rvr} \rightarrow f_{0v0}$. The operators are associated with the rules (**) in a one-to-one manner. The set of operators constructed as above for each production rule pattern (*) of the GSF scheme of G is the set of operators of the signature Sig.

■

We obtain a GSF associating with each word generated by the GSF as its meaning its syntactical structure in form of a term of the syntactical algebra associated with the GSF by combining production rule patterns of form (4) with the refinement concept. From the refinement concept it follows that it would be useful if the semantic functions $SM(p_0,p_1,...,p_r)$ occurring in production rule patterns of second kind would be refined by the refinement parameters of the syntactical functions too. Then the production rule patterns and their interpretation are defined as follows:

Let G=<S,IP> be the GSF, O a signature and T(O) the term algebra associated with G. Remember that O denotes both the signature and the set of operators of the signature.

- First kind of production rule patterns:

  $f(o_b,...)$: b.

  b sequence of parameterless basic syntactical functions,
  $o_b \in O$ operator associated with the rule

- Second kind of production rule patterns: (5)

  $f_0(p_0,...,P_{f0},Ref(f0),...)$: $b_0,f_1(p_1,...,P_{f1},Ref(f1),...),b_1,...$
  $f_r(p_r,...,P_{fr},Ref(f_r),...),b_r$,
  $a_1(...),...,a_s(...)$,
  $SM(p_0,p_1,...,p_r,P_{f0},Ref(f0),...,P_{fr},Ref(f_r))$.

  $f_0 \in N$, $f_1,...,f_r \in N \cup T$,
  $b_0,...,b_r$ sequences of parameterless basic syntactical functions,
  $a_1,...,a_s \in A$, $SM \in SF$, $p_1,...,p_r \in V \cup C$, $p_0 \in V$.

  Let $O_p = \{O_{p,w}\}$, where $w = v_1 ... v_r\ v_0$ with $v_i \in M(f_i,Ref(f_i))$, i = 0,1,...,r, is the set of operators of Sig associated with the production rule pattern p. The interpretation IP=<M,D,I,F> must possess the following properties:

  1. $M(f_i(...,v_i,...),1)$, $v_i \in M(f_i,Ref(f_i))$, i = 0,1,...,r, is the set of all terms which are syntactical structures of the subwords derived from instances of $f_i(...,v_i,...)$, i.e. $M(f_i(...,v_i,...),1) \subseteq T(O)_{fivi}$

Remark: Because $f_i(...,v_i,...)$ with a constant (v_i) as $Ref(f_i)$-th parameter can be considered as a function derived from $f_i(...)$ $M(f_i(...,v_i,...),1)$ makes sense.

2. $M(SM(p_0,...,p_r,v_0,v_1,...,v_r),i+1) = M(f_i(...,v_i,...),1)$
   $v_i \in M(f_i,Ref(f_i))$, $i = 0,1,...,r$

   Remark: Because $v_0,v_1,...,v_r$ are constants, $SM(p_0,...,p_r,v_0,v_1,...v_r)$ can be considered as a new semantic function with r+1 parameters which is a specialization of the original semantic function $SM(p_0,...,p_r)$ in (4).

   $M(SM,i+1) = \bigcup_{v_i \in M(f_i,Ref(f_i))} M(f_i(...,v_i,...),1)$, $i = 0,1,...,r$

3. $I(SM) = \{ (O_{p,w}(p_1,...,p_r),p_1,...,p_r,v_0,v_1,...,v_r) \mid$
   $O_{p,w}(p_1,...,p_r) \in M(f_0(...,v_0,...),1)$,
   $p_i \in M(f_i(...,v_i,...),1)$, $i = 1,...,r$ $\}$

   Remark: The interpretation I(SM) defines the meaning p_0 as term constructed from the meanings of the subwords derived from the syntactical functions of the right-hand side by using an operator which is unique for the actual combination of values for the refinement parameters.

2.3 Prolog-Implementation of GSFs

The production rule patterns of GSFs together with the interpretation of auxiliary syntactical and semantic functions resemble Prolog clauses. The question arises whether a GSF rule can be interpreted as a Prolog clause. Using the following interpretation the answer is ´YES´:

For each variable occurring in the production rule pattern (1) the following holds

If $(F_1 \overset{*}{=\!=\!=>} t_1 \wedge ... \wedge F_r \overset{*}{=\!=\!=>} t_r \wedge H_1 = \varepsilon \wedge ... \wedge H_s = \varepsilon)$
then $(F_0 \overset{\pm}{=\!=>} t_1 \ ... \ t_r)$, where $F_0 : F_1,...,F_r$. with the instances $H_1,...,H_s$ of the auxiliary syntactical functions and the semantic functions is a context-free production rule derived from the production rule pattern (1).

To exploit given Prolog systems for treating GSFs as Prolog programs requires some modifications of the production rule patterns of a GSF and some additional expense:

- The notation must be changed into the Prolog notation.
- Basic syntactical functions must be preceded by the special symbol @. This symbol will be used as an operator realizing the scan of terminals.
- Auxiliary syntactical functions and semantic functions will be labelled by # and & resp. which are considered as operators realizing the evaluation of the functions.

- Because the right-hand sides of Prolog clauses are treated from left to right the auxiliary syntactical functions and semantic functions must be rearranged in correspondence with this order. To avoid infinite loops the order of clauses must be rearranged too.
- Prolog clauses realizing the operators @, #, & must be added.

Moreover the following properties arise from this approach:

- If the context-free basic grammar of the GSF is an LL(k)-grammar then there is no need of a parser because the Prolog system itself can be exploited. Otherwise a parser must be added.
- The implementation of # and & can be done in several steps ([R91]).
- Under certain circumstances some parameter positions can be omitted. For example the parameter positions for symbol tables usually can be omitted and the symbol table can be managed in the Prolog database (as done in the Examples 4 and 5).

Note the difference between these modifications and transformations of DCG rules into Prolog clauses. The latter approach causes that terminals are changing from a syntactical element of the DCG into an argument of a predicate. In our approach a terminal of the GSF is still at the same place in the Prolog clause but now preceded by a special operator which realizes the scan of terminals.

Example 4 (Prolog version of Example 1):

```
% concatenation of statements
sm_list(S) :- statement(S1),@";",
              sm_list(S2),
              & concat(S,S1,S2).
sm_list(S) :- & skip(S).

% concrete statements
statement(S) :- assign_statement(S).
statement(S) :- if_statement(S).
. . .

% assign statement
assign_statement(S) :- left_value(E1,T1),
                       @":=",expression(E2,T2),
                       # check_assign_types(T1,T2),
                       & assign(S,E1,E2).

% if statement with optional else-part
if_statement(S) :- @"if",expression(E,T),
                   # is_boolean_type(T),
                   @"then",sm_list(S1),
                   else_part(S2),
                   @"fi",
                   & if(S,E,S1,S2).

else_part(S) :- @"else",sm_list(S).
else_part(S) :- & skip(S).
```

■

The straightforward approach to implement the auxiliary syntactical functions is to implement the interpretation I of a name f of an auxiliary syntactical function by a predicate named f. Then the operator # working within the Prolog version of GSF as a marker of auxiliary syntactical functions must be implemented simply by the following clause:

(# X) :- call(X).

Example 5 (Prolog clauses defining context conditions for Example 4):

```
% checking types for assign statement
check_assign_types(realtype,integertype) :- !.
check_assign_types(T1,T2) :-
 match_types(T1,T2),!.
check_assign_types(usertype(Id),T) :-
 type_def(usertype(Id),TD),
 check_assign_types(TD,T),!.

% translating user types
type_def(usertype(Id),Type) :- !,
 statenv(Id,typedef(T)), % querying the global symbol table
 type_def(T,Type).
type_def(T,T).

% defining an equality between types
match_types(T,T) :- !.
match_types(T,pointertype(undeftype)) :- !,
 type_def(T,pointertype(_)).
match_types(pointertype(undeftype),T) :-
 type_def(T,pointertype(_)).

% testing a type to be the boolean type
is_boolean_type(T) :- type_def(T,booleantype).
```

■

It remains to discuss the implementation of the semantic functions which are marked by &. In subsection 2.2 (see (5)) we have described an interpretation (for names of semantic functions) guaranteeing the meaning of a word to be its syntactical structure in form of a term of the syntactical algebra associated with the GSF. The reader may remember the last remark of (5) in order to realize that the following implementation of & is a proper solution:

(& X) :- X =.. [OP|[P0|PIs_and_Refs]],P0 =.. [OP|PIs_and_Refs].

So the resulting term is indeed a term constructed from the meanings of the subconstructs. As operator symbol the name of the semantic function itself is applied which is obviously not unique for any combination of values for the refinement parameters. However, by taking the values of the refinement parameters into the term, the uniqueness is still satisfied, since the operator symbol may be considered as beeing qualified by the refinement parameter positions within the term.

Example 6 (A simple MYPAS program):

```
1.          PROGRAM Count;
2.           VAR x : INTEGER;
            BEGIN
3.           Read(x);
4.           WHILE
5.                x > 0
                  DO
6.                    x := x - 1;
7.                    WriteLn(x);
                  OD;
            END.
```

■

Example 7 (The term generated for the program from Example 6):

```
1. prog(input,output,
    block(
     concatd(
2.    concatd(vardec([x],integertype),nodec),
      nodec),
     concat(
3.    read(default(input), [id(x)],[integertype]),
      concat(
4.     while(
5.      op(id(x), const(0), agt),
        concat(
6.       assign(id(x), op(id(x), const(1), minus)),
         concat(
7.        writeln(default(output),[id(x)],[integertype]),
          skip))),
       skip))))
```

■

3 Denotational semantics

3.1 Preface

Denotational semantics descriptions as required in the system LDL and the approach to their implementation are considered in [Lä93], [RL93], [LR94]. We will only sketch some ideas concerning this topic in subsections 3.2 and 3.3.

Concerning the implementation of denotational semantics as a logical program it should pointed out, that there are two extrema. First it is possible to use a well-defined transformation from recursive function definitions into pure logical programs with a clean declarative semantics beeing a good basis for stating about correctness. Second an analysis of the language and its language definition together with standard techniques from compiler construction may result in more sophisticated transformations to derive a logical program (not a pure logical program, rather a real Prolog program) offering the term interpreter.

3.2 Demands on the denotational semantics definitions

In our language definitions we apply denotational semantics definitions to assign dynamic semantics to the terms generated by the corresponding GSF. Thus a basic requirement is that the syntactical domains of the denotational description can be identified with the sets of terms beeing the underlying sets of the syntactical algebra associated with the GSF.

The derivation of a logical program from a recursive function definition becomes easier when functional domains have been flattened. Indeed, our constructive approach sketched in [LR94] assumes the absence of functional domains. The following transformations are applied to reach this goal:

- *Representing functions by sequences*: Certain (so-called semi-finite) functions may be considered as sequences of tupels. These are functions like f with f: D_1 -> (D_2 + { undef }$_\perp$), where D_1, D_2 are arbitrary semantic domains, { x | x ϵ D_1 ∧ f(x) ≠ undef } is a finite set. f can be considered as f′ ϵ (D_1 × D_2)* consisting of all pairs <x_i,y_i>, x_i ϵ D_1, y_i ϵ D_2, y_i = f(x_i) ≠ undef. It is straightforward to define auxiliary operators for application and modification.
- *Freezing computation of meanings*: The denotational semantics of certain non-trivial language constructs demands to pass meanings through the semantics description or to consider them as intermediate results which e.g. have to be bound in an environment. Functional domains introduced this way can be omitted by working on (syntactical) terms. For example, the meaning of an imperative procedure declaration (as bound in the environment) usually would be a function mapping the actual parameter to a state-transition function. To avoid this functional domain we consider (meanings of) procedure declarations simply as terms beeing more or less equal to the syntactical procedure declarations. So we can achieve a late application of semantic functions to syntactical elements at positions where all arguments of the semantic functions are known instead of working on intermediate meanings. This approach is also sufficient to implement continuation semantics.
- *Avoiding higher-order operators*: Some functional domains are introduced only for parameter/result domains of auxiliary higher-order operators which are used to figure out common parts of semantic definitions into operators. A usual example are composition operators with error propagation. Functional domains introduced this way can be omitted by using the definition of a operator in positions where this operator is applied instead of the operator itself.

- *Curried -> Uncurried functions*: Since we are interested rather in execution (interpretation) of programs than in computation of meanings we can apply uncurried versions of semantic functions instead of their curried original versions. By this simple transformation we can avoid domains for meanings which are often functional entities (as described by semantic functions of usual denotational semantics descriptions).

3.3 Implementation as logical program

The transformation of a recursive function definition into a logical program consists of two subtasks:

1. *Definition of representations for elements of domains*: For any element of any syntactical and semantic domain we need a representation within the logical program. For that purpose we apply ground term representations. In [RL93] we systematically define a bijective function mapping elements of domains to terms of a term algebra w.r.t. a signature derived from the domain equations.
2. *Transformation of functional equations into definite clauses*: To model functions we derive predicates (definite clauses) having input and output parameter positions. Some basic transformation ideas are considered:
   - *Ad hoc implementations*: For basic operations ad hoc implementations can be developed.
   - *Term construction instead of functions*: For some simple functions the corresponding predicate need not to be derived, since the applications of these functions can be modelled within the logical program by term construction. For example, because of the term representation of elements projections, injections w.r.t. sum domains and list operation as head, tail, cons w.r.t. domains of sequences can be simulated by term construction.
   - *Composition by conjunction of atomic formulae*: A nested application like $f_n(f_{n-1}(...f_2(f_1(x_0))...))$ is modelled as conjunction of atomic formulae $pf_1(X_0,X_1),pf_2(X_1,X_2),...,pf_{n-1}(X_{n-2},X_{n-1}),pf_n(X_{n-1},X_n)$ where the predicates pf_i are intended to implement the functions f_i and X_0 should be bound to the representation of x_0. We start with the innermost application taking the left-to-right processing of right-hand sides of clauses by Prolog systems into consideration. The intermediate results are passed from left to right from the output positions to the input positions.

Example 8
(Excerpt of Prolog clauses defining the term interpretation for MYPAS):

```
% interpretation of commands
interpret(skip) :- !.
interpret(concat(S1,S2)) :- !,ccom(S1,S2,Cont),
                            interpret(Cont).
interpret(Term) :- ccom(Term,skip,Cont),
                   interpret(Cont).
```

```
% semantics of concatenation
ccom(concat(S1,S2),C,R) :-
 !,ccom(S1,concat(S2,C),R).

% ... of if statement
ccom(if(E,S1,S2),C,concat(S,C)) :-
 !,reval(Val,E),
   ( Val == true,!,
     S = S1
   ; S = S2
   ).

% commands defined by direct semantics
ccom(Com,C,C) :- com(Com).

% assignment
com(assign(LHS,RHS)) :- leval(LVal,LHS),
                        reval(RVal,RHS),
                        update(LVal,RVal).
```

Remark: This term interpretation has been derived from a denotational semantics description written in continuation style. Continuations are modelled here as terms representing statements. ■

4 Related work

Paulson's semantic grammars ([Pau82]) give also a descriptional formalism exploiting attribute grammars and denotational semantics. In difference to our approach semantic grammars define semantics rather in an one-level manner by allowing attributes to be elements from arbitrary semantic domains (Functional entities are written in lambda notation.). Our two-level approach is very similar to Lee's High-Level semantics ([L89]) consisting of a macro semantics and at least one micro semantics. The construction of terms is there described by semantic equations instead of using attribute grammars as in our case. Other related work is considered in [RL93].

5 Conclusion and Future work

For the language / prototype interpreter definitions as discussed in this paper the semantic functions of a GSF were assumed to be interpreted in such a way that they describe the semantic meaning of an input word as a term constructed from the meanings of its subwords.

Although it is allowed (and useful) to enrich the term structure by results of the semantic analysis (refinement concept) such a term will be considered more likely as an abstract syntactical structure of the input word. Thus it would be straightforward to take the syntactical algebra associated with a GSF as abstract syntax definition for an action semantics description. The action semantics

description could take profit from the semantic analysis described within the GSF, since the results of the analysis could be made available within the generated terms using the refinement concept.

A different approach to the interconnection of GSFs and action semantics would be to define the generation of actions by the semantic functions. Relating GSFs with action semantics descriptions this way would allow to describe static semantics separately at the level of the attribute grammar. The specific form of GSFs as described in subsection 2.2 always constructs meanings directly from submeanings eventually applying refinement parameters. This compositional behaviour of the semantic functions of those GSFs seems to be useful also in the context of action generation, since one basic demand on action semantics descriptions is their compositional form.

Although it is possible to exploit action semantics itself for definition of static semantics (see e.g. [WM87]), there is no obvious way to take profit from the static analysis within the dynamic semantics description when describing both (static and dynamic semantics) by a separate action semantics description. Therefore we suggest to interconnect GSFs and action semantics as motivated above. The details of this interconnection and the consideration of other alternatives (for example other kinds of attribute grammars) are points for future work.

References

[ADJ77] Goguen, J.A.; Thatcher, J.W.; Wagner, E.G.; Wright, J.B.:
Initial algebra semantics and continous algebras
JACM 24 (1977) 1, 68-95

[AM91] Alblas, H.; Melichar, B. (Eds.) : Attribute grammars, Applications and Systems, Proc. of the International Summer School SAGA, Prague, Czechoslowakia, June 1991, LNCS # 545, Springer-Verlag

[CD87] Courcelle, B.; Deransart, P.: Proofs of partial correctness for attribute grammars with application to recursive procedures and logic programming, RR No. 322, INRIA Rocquencourt, 1984

[K91] Koskimies, K.: Object-orientation in attribute grammars, In: [AM91], 297-329

[L89] Lee, P.: Realistic compiler generation, MIT Press 1989

[Lä93] Lämmel, R.: Prolog-Implementation denotationaler Semantikbeschreibungen, Diplomarbeit, Universität Rostock, FB Informatik, Jan. 1993

[LR94] Lämmel, R.; Riedewald, G.:
Provable Correctness of Prototype Interpreters in LDL
In: Fritzson, P.A. (Ed.): Compiler Construction
5th International Conference, CC '94, Edinburgh, U.K., April 1994,
Proceedings, LNCS # 786, Springer-Verlag, 218 - 232

[Pau82] Paulson, L.: A semantics-directed compiler generator,
In: Proceedings of the Ninth Annual ACM Symposium
on Principles of Programming Languages, 1982, Albuquerque,
New Mexico, 224-239

[PW80] Pereira, F.N.G.; Warren, D.H.D.: Definite Clause Grammars for
language analysis: a survey of the formalism and comparison
with augmented transition networks
Artificial Intelligence, 13 - 3 (1980), 231-278

[R91] Riedewald, G.: Prototyping by using an attribute grammar as a logic
program, In: [AM91], 401-437

[RL93] Riedewald, G.; Lämmel, R.: Provable Correctness of Prototype Interpreters in LDL
Preprint CS-9-93, Sept. '93, Universität Rostock, FB Informatik

[WG84] Waite, W.M.; Goos, G.: Compiler Construction, Springer-Verlag, 1984

[WM87] Watt, D. A.; Mosses, P.D.: Pascal: Static Action Semantics. Draft,
Version 0.31, 1987

The ACTRESS Compiler Generator and Action Transformations (*Abstract*)

Hermano Moura*
Caixa Economica Federal, Brazil

Actress is a semantics-directed compiler generation system based on action semantics. Its aim is to generate compilers whose performance is closer to hand-written compilers than the ones generated by other semantics-directed compiler generators. Actress generates a compiler for a language based solely on the language's action semantic description. We describe the process by which this is achieved.

A compiler for action notation is the core of the generated compilers. It translates actions to object code. Action notation can be seen as the intermediate language of every generated compiler.

A conventional hand-written compiler eliminates, whenever possible, references to identifiers at compile time. Some storage allocation is often performed at compile-time too. We can see both steps as transformations whose main objective is to improve the quality of the object code. The compiler writer, based on his knowledge of properties of the source language, implements these "transformations" as best as he can. In the context of Actress we adopt a similar approach. We introduce a set of transformations, called action transformations, which allow the systematic and automatic elimination of bindings in action notation for statically scoped languages. They also allocate storage statically whenever possible. We formalise and implement these action transformations. The transformations may be included in generated compilers. We show that this inclusion improves the quality of the object code generated by Actress' compilers.

In general, action transformations are a way to do some static processing of actions. Transforming actions corresponds to partially performing them, leaving less work to be done at performance time. Thus, transformed actions are more efficient.

A full paper on this topic was presented at CC'94, see:

> Moura, H., and Watt, D. A. (1994) Action transformations in the ACTRESS compiler generator, in *Compiler Construction – 5th International Conference CC'94* (ed. Fritzson, P.), vol. 786, Lecture Notes in Computer Science, Springer-Verlag, pages 16-30.

*SQN 206, Bloco I, Apto 103, Brasilia, DF, Brazil. E-mail: hermano@cic.unb.br

Sort Inference in the ACTRESS Compiler Generator

Deryck F. Brown* David A. Watt†

Abstract

ACTRESS accepts the action-semantic description of a source language, and from it generates a compiler. The generated compiler translates its source program to an action, performs sort inference on this action, (optionally) simplifies it by transformations, and finally translates it to object code. The sort inference phase provides valuable information for the subsequent transformation and code generation phases. In this paper we study the problem of sort inference on actions.

1 Introduction

ACTRESS is an *action-semantics directed compiler generator* [4]. That is to say, it accepts a formal description of the syntax and action semantics [8, 14] of a particular programming language, the *source language*, and from this formal description it automatically generates a compiler that translates the source language to C object code.

The generated compiler first translates each source program to an action, which we call the *program action*. Then it sort-checks the program action. Finally, it (optionally) transforms the program action, and translates it to C object code. The program action serves as an intermediate representation of the source program's semantics.

Sort checking is important, not only to discover sort errors in the program action, but also to infer sort information necessary for effective transformation and code generation. Sort inference on action notation is a challenging problem. Records, subsorts, and polymorphism are all involved. Action notation itself is much richer than the various λ-calculi usually studied by type theorists.

This paper describes our work on sort inference. The rest of the paper is structured as follows. Section 3 is a brief description of the ACTRESS compiler generation system. Section 4 explains the general notion of sort in action notation, and the slightly simpler notion of sort adopted in ACTRESS. Section 5 describes our sort inference algorithm, and the sort inference rules that guide it. Section 6 surveys related work, and Section 7 concludes.

*INMOS Ltd, 10 Priory Road, Bristol BS8 1TU, England. E-mail: deryck@pact.srf.ac.uk.

†Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: daw@dcs.glasgow.ac.uk.

2 Action Notation

Action semantics was developed by Mosses and Watt [8, 14]. As compared with other methods, action semantics has unusually good pragmatic qualities: action-semantic descriptions are easy to read, to write, and to modify.

An *action* is a computational entity, which can be *performed.* When performed, an action either *completes* (terminates normally) or *fails* (terminates abnormally) or *diverges* (does not terminate at all). Actions are performed in a designated order (control flow). They can pass data to one another (data flow), in several forms: *transients* (data that disappear unless used immediately), *bindings* (data bound to identifiers, propagating over a designated scope), and *storage* (data stored in cells, remaining stable unless overwritten or deallocated).

Action notation provides a number of *action primitives*, *action combinators*, and *yielders*. An action primitive represents a single computational step, such as giving a transient datum, binding an identifier to a datum, storing a datum in a cell, or immediately completing. An action combinator combines one or two sub-actions into a composite action, and governs the control flow and data flow between these sub-actions. There are action combinators that correspond to sequential composition, functional composition, choice, iteration, and so on. Finally, some primitive actions include yielders, which are used to access data passed to the action (transients, bindings, or storage). The action primitives, action combinators, and yielders of the ACTRESS subset are summarised in Table 1[1].

An *action-semantic description* of a programming language $\mathcal{L}$ specifies a mapping from the phrases of $\mathcal{L}$ (expressions, commands, declarations, etc.) to action notation. An action-semantic description is structured like a denotational description, with semantic functions and semantic equations, but the denotations of phrases are expressed in action notation.

3 The ACTRESS Compiler Generator

ACTRESS is a compiler (and interpreter) generation system developed at the University of Glasgow by Brown, Moura, and Watt [4]. It provides a collection of modules that operate on actions (represented internally as trees). These modules include:

- $Check_{\mathcal{A}}$ is the action notation sort checker. This infers the sorts of the given action and all its sub-actions. The sort of an action includes the sorts of all transients and bindings passed into and out of that action. The sort checker discovers any sub-action that must fail due to a sort error (such as attempting to use an integer where a truth value is expected). The sort checker simply replaces any such ill-sorted sub-action by 'fail'. Finally, the sort checker annotates the action with the inferred sorts.

[1] For historical reasons, this version of action notation differs slightly from that given in Mosses[8] and Watt[14].

Primitive	**Informal meaning**
complete	Completes immediately (i.e., does nothing).
fail	Fails immediately.
give Y	Gives the datum yielded by Y, labelled 0.
give Y label #n	Gives the datum yielded by Y, labelled n.
bind k to Y	Produces a single binding, of identifier k to the datum yielded by Y.
recursively bind k to Y	As 'bind', but allows the binding of k to be used in evaluating Y.
store Y_1 in Y_2	Stores the datum yielded by Y_1 in the cell yielded by Y_2.
allocate a S	Finds an unreserved cell of sort S, reserves it, and gives it.
enact Y	Performs the action incorporated by the abstraction yielded by Y.
Combinator	**Informal meaning**
A_1 or A_2	Performs either A_1 or A_2. If the chosen sub-action fails, the other sub-action is chosen.
A_1 else A_2	Tests a given truth value, and then performs A_1 if it is true or A_2 if it is false.
unfolding A	Performs A iteratively. Dummy action 'unfold', whenever encountered inside A, is replaced by A.
A_1 and A_2	Performs A_1 and A_2 collaterally. Any transients given by A_1 and A_2 are merged. Any bindings produced by A_1 and A_2 are merged.
A_1 and then A_2	Performs A_1 and A_2 sequentially. Otherwise behaves like 'A_1 and A_2'.
A_1 then A_2	Performs A_1 and A_2 sequentially. Transients given by A_1 are given to A_2.
A_1 hence A_2	Performs A_1 and A_2 sequentially. Bindings produced by A_1 propagate to A_2.
A_1 moreover A_2	Performs A_1 and A_2 sequentially. Bindings produced by A_2 override those produced by A_1.
A_1 before A_2	Performs A_1 and A_2 sequentially. Bindings produced by A_1 and A_2 are accumulated.
furthermore A	Performs A. Bindings produced by A override the received bindings.
Yielder	**Informal meaning**
the S	The given transient datum labeled 0. It must be of sort S.
the S#n	The given transient datum labeled n. It must be of sort S.
the S bound to k	The datum currently bound to identifier k. It must be of sort S.
the S stored in Y	The datum currently contained in the cell yielded by Y. It must be of sort S.
abstraction A	The abstraction that incorporates action A.
closure Y	The abstraction yielded by Y, with the current bindings supplied to the incorporated action.
Y_1 with Y_2	The abstraction yielded by Y_1, with the transient datum yielded by Y_2 given to the incorporated action.

Table 1: Action primitives, action combinators and yielders.

- $Encode_{\mathcal{A}}$ is the action notation code generator. This translates the annotated action to C object code.

Other modules are generated by ACTRESS from the formal description of a particular source language $\mathcal{L}$:

- $Parse_{\mathcal{L}}$ is a parser for $\mathcal{L}$. This parser is generated using the standard parser generator, *mlyacc*:

$$parse_{\mathcal{L}} = mlyacc(syntax_{\mathcal{L}}) \tag{1}$$

  where $syntax_{\mathcal{L}}$ is a syntactic description of language $\mathcal{L}$.

- $Act_{\mathcal{L}}$ is an *actioneer* for $\mathcal{L}$. This is a module that translates a parsed $\mathcal{L}$ program to the corresponding program action. This module is generated using the actioneer generator *actgen*:

$$act_{\mathcal{L}} = actgen(semantics_{\mathcal{L}}) \tag{2}$$

  where $semantics_{\mathcal{L}}$ is an action-semantic description of language $\mathcal{L}$. The actioneer generator treats the latter simply as a syntax-directed translation from $\mathcal{L}$ to action notation.

Composition of the generated parser and actioneer for $\mathcal{L}$ with the action notation sort checker and code generator yields a compiler for language $\mathcal{L}$:

$$compile_{\mathcal{L}} = encode_{\mathcal{A}} \circ check_{\mathcal{A}} \circ act_{\mathcal{L}} \circ parse_{\mathcal{L}} \tag{3}$$

Finally, we have recently added a new module to ACTRESS:

- $Transform_{\mathcal{A}}$ is the action transformer, which attempts to simplify a given action by applying action transformations.

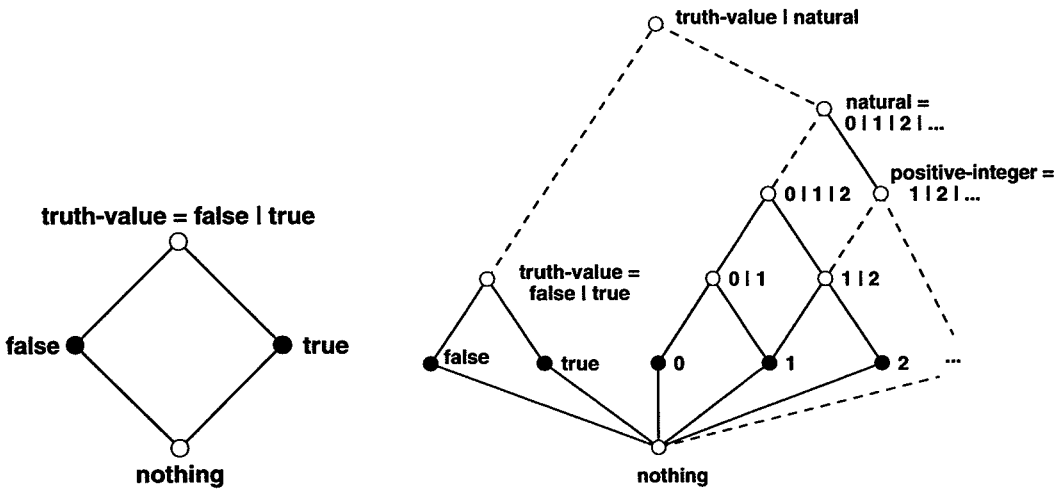
This module may be used to construct compilers that generate smaller and faster object code, at the expense of increased compilation time:

$$compile'_{\mathcal{L}} = encode_{\mathcal{A}} \circ transform_{\mathcal{A}} \circ check_{\mathcal{A}} \circ act_{\mathcal{L}} \circ parse_{\mathcal{L}} \tag{4}$$

4 Sorts

4.1 Data Sorts in Standard Action Notation

The theoretical foundation of action notation is Mosses' *unified algebras* [8]. This algebraic framework elegantly solves some of the problems that beset older algebraic frameworks, by the simple expedient of abandoning the usual sharp distinction between values and sorts.



(a) truth-values

(b) truth-values and naturals

Figure 1: Example sort hierarchies

In a unified algebra, a *sort* is just a classification of *individuals*. No distinction is made between an individual and the singleton sort that classifies just that individual. Sorts are partially ordered by a subsort relation, '$\leq$'. The least sort, nothing, is the classification of no individuals. The *join* of two sorts, $s_1 \mid s_2$, is their least upper bound, and the *meet* of two sorts, $s_1 \,\&\, s_2$, is their greatest lower bound. The notation '$x : s$' asserts that x is an individual and belongs to sort s.

In Figure 1(a), the universe of discourse consists of the truth values. The individuals are false and true. The sorts are nothing, false, true, and false | true. In this example, nothing and truth-value = false | true are the only *proper sorts*, i.e., sorts that are not individuals. The nodes of the graph represent the sorts (individuals being shaded black and proper sorts white); the edges of the graph represent the '$\leq$' relation.

In Figure 1(b), the universe of discourse consists of not only the truth values but also the natural numbers (individuals 0, 1, 2, ...). In this example there are many sorts, of which only a few are shown. Among the interesting proper sorts are 0 | 1 | 2, 1 | 2 | 3 | ... (also known as positive-integer), 0 | 1 | 2 | 3 | ... (also known as natural), and truth-value | natural. There are also some less useful sorts, such as 2 | true.

One benefit of unified algebras is that operations may be defined uniformly over proper sorts as well as individuals. For example, the operation 'successor _' not only maps 0 to 1, 1 to 2, ...; it also maps 0 | 1 to 1 | 2, ..., and positive-integer to natural.[2]

[2]Indeed, these infinite sorts are defined by the recursive equations positive-integer = successor

(data sorts)	s	::=	nothing | bi | bs | $sc[s]$ | s | s | s & s | datum
(basic individuals)	bi	::=	false | true | 0 | 1 | 2 | ...
(basic sorts)	bs	::=	truth-value | integer | ...
(sort constructors)	sc	::=	list | cell | ...

Table 2: Syntax of data sorts in ACTRESS

The operation 'list[_]' maps sorts of data to sorts of lists. For example, list[truth-value] is the sort of all lists of truth values, list[1] is the sort of all lists of ones[3], list[natural] is the sort of all lists of natural numbers, and list[truth-value | natural] is the sort of all lists of truth values and natural numbers (a sort of heterogeneous lists).

For nearly all practical purposes, we may view sorts as sets, nothing as the empty set, ':' as set membership, '$\leq$' as set inclusion, '|' as set union, and '&' as set intersection.

4.2 Data Sorts in ACTRESS Action Notation

The action notation sort checker can deal only with finitely expressible sort terms. Therefore it restricts sort terms to those generated by the BNF grammar in Table 2.

This class of sorts has the following useful properties [3]:

- The basic individuals are partitioned into a number of *basic sorts*, such that every basic individual belongs to a unique basic sort. Thus we can talk about *the* basic sort of a given basic individual.
- Individuals of constructed sorts are not expressible. This is because such individuals are constructed using ordinary data operations, and the sort checker makes no attempt to evaluate such terms. For example, the individual list of 1 and 2 might be represented by the term: concatenation of (list of 1, list of 2). The resulting individual value cannot be determined by the sort checker, without making use of the definition of concatenation.
- Every sort term can be reduced to a finite canonical sort term, which is of the form s_1 | ... | s_n, where $n > 0$ and each s_i is either a basic individual or a basic sort or a sort constructor applied to a canonical sort term. In particular, '&' can always be eliminated.

natural; natural = 0 | positive-integer (*disjoint*).

[3]Note that 'list[_]' maps an individual to a sort.

- There are algorithms to compute '$x : s$', '$s_1 \leq s_2$', and '$s_1 \,\&\, s_2$', for arbitrary basic individuals x and arbitrary sort terms s, s_1, s_2.

4.3 Action Sorts in Standard Action Notation

In standard action notation, the sort action classifies all actions. A subsort of actions is characterised either by restricting its *incomes* (the data it may use), or by restricting its *outcomes* (e.g., whether it completes, fails, or diverges, and what data it passes out if it does complete), or by restricting both its incomes and its outcomes. For example: the sort 'action [using current bindings]' classifies actions that use the bindings propagated into them; the sort 'action [giving a value]' classifies actions that each gives a datum of sort value; the sort 'action [binding]' classifies actions that produce bindings; the sort 'action [storing]' classifies actions that effect changes in storage; and the sort 'action [binding] [using current bindings]' classifies actions that both use and produce bindings.

4.4 Action Sorts in ACTRESS Action Notation

The action notation sort checker is not concerned with all possible classifications of actions. For one thing, it cannot concern itself with whether an action may diverge or not.[4] Also, it does not concern itself with storage, for reasons to be explained shortly. It does concern itself with transients and bindings.

The sort of a set of bindings may be represented by a *record sort*. For example, the record sort {x: integer, y: truth-value}, represents the fact that x is bound to an unknown datum of sort integer and y is bound to an unknown datum of sort truth-value. Other examples of record sorts are {x: integer, y: true}, where in this case y is known to be bound to true, and {x: 6, y: true}, where in this case both x and y are bound to known data. This notation is legitimate, because the individuals 6 and true are themselves sorts. It is also convenient, because the sort of a set of bindings informs us concisely which identifiers are bound to known data (those whose sorts are individuals) and which are bound to unknown data (those whose sorts are proper sorts).

These record sorts are similar to the record types studied by Wand, Cardelli, Mitchell and others [5, 12, 13]. The domain of each record sort must be known, i.e., there must be no variables ranging over the domain of a record sort. We can use record sorts to represent the sorts of bindings, during sort inference of a particular action, since the domain of each set of bindings (a set of identifiers) will be known statically. Similarly, we can use record sorts to represent the sorts of transients, since the domain of each set of transients (a set of labels) will also be known statically. However, we cannot use record sorts to represent the sorts of stores, since the domain of a store (a set of cells) will, in general, be determined only dynamically.

[4] In other words, it cannot solve the halting problem!

(sort schemes)	ss	::=	$\ldots \mid \theta$
(action sort schemes)	as	::=	$(t, b) \hookrightarrow (t', b') \mid$ nothing
(yielder sort schemes)	ys	::=	$(t, b) \rightsquigarrow ss \mid$ nothing
(fields)	f	::=	$ss \mid$ **absent** $\mid \Delta$
(transients)	t	::=	$\{l_1\colon f_1, \ldots, l_m\colon f_m\}[\rho \mid \gamma]$
	t'	::=	$\{l_1\colon f_1, \ldots, l_m\colon f_m\}[\rho]$
(bindings)	b	::=	$\{k_1\colon f_1, \ldots, k_n\colon f_n\}[\rho \mid \gamma]$
	b'	::=	$\{k_1\colon f_1, \ldots, k_n\colon f_n\}[\rho]$

Table 3: Syntax of action sorts in ACTRESS

While record sorts can be used to specify the sorts of individual actions, they are not sufficient to describe sort inference over actions. For this, we need to extend the record sorts into record sort schemes. This extension follows the extension of record types into record schemes given in [13]. We extend the notation as follows. We extend sorts to sort schemes which include sort variables, denoted by θ, which range over sorts. Also, instead of mapping field names to sorts directly, record sort schemes map names to *field schemes*. A field scheme can be: a sort scheme, ss, indicating that the field is definitely present in the record sort scheme; **absent**, indicating that the field is definitely absent from the record scheme; or a field variable, Δ, giving no information about the presence or absence of the field. Finally, a record sort scheme may have a row variable (ρ or γ) affixed to it. Row variables are used to represent possible extra unknown fields.

Combining these ideas with those of Even and Schmidt [7], we write the sort scheme of an action A as follows:

$$A : (t, b) \hookrightarrow (t', b') \tag{5}$$

where t and b are the record sort schemes of the transients and bindings used by A, and where t' and b' are the record sort schemes of transients and bindings passed out of A (assuming that it completes). If an action is ill-sorted, we write A : nothing.

More precisely, the action notation sort checker deals with action sort schemes generated by the BNF grammar in Table 3. Not specified are l (labels, used to identify transients) and k (tokens, or identifiers).

For example:

- bind "n" to 7 : $(\{\}\gamma_1, \{\}\gamma_2) \hookrightarrow (\{\}, \{\text{n: 7}\})$
- bind "n" to the integer #1 : $(\{\text{1: integer}\}\gamma_3, \{\}\gamma_4) \hookrightarrow (\{\}, \{\text{n: integer}\})$

- give sum (the integer #1, the integer #2) label #3 :
  $(\{1\text{: integer}, 2\text{: integer}\}\gamma_5, \{\}\gamma_6) \hookrightarrow (\{3\text{: integer}\}, \{\})$
- rebind : $(\{\}\gamma_7, \{\}\rho_1) \hookrightarrow (\{\}, \{\}\rho_1)$
- furthermore bind "x" to the integer bound to "y" :
  $(\{\}\gamma_8, \{\text{y: integer}\}\rho_2) \hookrightarrow (\{\}, \{\text{x: integer, y: integer}\}\rho_2)$

Unlike Wand [13], we use two different classes of row variables. This reflects the different uses of the information given to an action. Firstly, any transients or bindings explicitly used within the body of an action appear explicitly in its sort scheme.

The row variables γ_i represent the sorts of any transients or bindings that are passed into actions but not used ('garbage'). Since no action is obliged to use all the transients or bindings passed into it, most actions have γ-variables affixed to the t and b parts of their sorts.

The row variables ρ_i represent transients or bindings that are passed into actions but just propagated out of them. The action 'rebind' (which propagates the bindings it receives) is the simplest example of this. This action is *polymorphic*, and its sort contains a ρ-variable to reflect this polymorphism.[5] Actions derived from 'rebind', such as 'furthermore A', are also polymorphic. When writing a sort, we assume that all free variables are implicitly universally quantified.

When a record sort scheme has a row variable affixed to it, the row variable may be instantiated to any record sort with a disjoint domain. For example, consider the following action sort:

$$\text{give the cell bound to "v"} : (\{\}\gamma_1, \{\text{v: cell}\}\gamma_2) \hookrightarrow (\{\text{0: cell}\}, \{\})$$

In $\{\text{v: cell}\}\gamma_2$, the row variable γ_2 could be instantiated to {x: integer, y: truth-value}, representing the possibility that the action may receive (but ignores) bindings for x and y. There are, of course, many other possibilities. However, {v: integer} is *not* a possibility, because of the duplicate v field. In $\{\}\gamma_1$, the row variable γ_1 could be instantiated to *any* record sort scheme, representing the possibility that the above action may receive *any* transients (but ignores them).

4.5 Abstraction Sorts in ACTRESS Action Notation

An abstraction incorporates an action, which is performed whenever the abstraction is enacted. It follows that abstraction sorts are isomorphic to action sorts. Since abstractions are classified as data, we augment the data sorts of Table 2 with abstraction sorts:

$$\text{(data sorts)} \quad s \ ::= \ \ldots \mid \text{abstraction}(t, b) \hookrightarrow (t', b')$$

[5] Compare the polymorphic function $(\lambda x.x)$, whose type may be written $\tau \rightarrow \tau$, where τ is a type variable.

5 Sort Inference in Action Notation

5.1 Sort Inference Rules

Using the action sort notation introduced in Section 4.4, our next step is to write down sort inference rules for ACTRESS action notation. We use the following judgments for assigning sorts to actions A and yielders Y, respectively:

$$\mathcal{E} \vdash A : (t,b) \hookrightarrow (t',b') \tag{6}$$

$$\mathcal{E} \vdash Y : (t,b) \rightsquigarrow s \tag{7}$$

Here $\mathcal{E}$ is the *sort environment*, containing (among other things), sort information about constants such as false, true, truth-value, and integer, and about operations such as sum(_,_).

The following are typical rules for basic actions and combinators:

(COMPLETE) $$\frac{}{\mathcal{E} \vdash \textsf{complete} : (\{\}\gamma_i, \{\}\gamma_j) \hookrightarrow (\{\}, \{\})}$$

(AND-THEN) $$\frac{\mathcal{E} \vdash A_1 : (t_1,b_1) \hookrightarrow (t_1',b_1'); \quad \mathcal{E} \vdash A_2 : (t_2,b_2) \hookrightarrow (t_2',b_2')}{\begin{array}{c}\mathcal{E} \vdash A_1 \textsf{ and then } A_2 : \\ (union\ t_1\ t_2, union\ b_1\ b_2) \hookrightarrow (merge\ t_1'\ t_2', merge\ b_1'\ b_2')\end{array}}$$

(OR) $$\frac{\mathcal{E} \vdash A_1 : (t_1,b_1) \hookrightarrow (t_1',b_1'); \quad \mathcal{E} \vdash A_2 : (t_2,b_2) \hookrightarrow (t_2',b_2')}{\begin{array}{c}\mathcal{E} \vdash A_1 \textsf{ or } A_2 : \\ (combine\ t_1\ t_2, combine\ b_1\ b_2) \hookrightarrow (join\ t_1'\ t_2', join\ b_1'\ b_2')\end{array}}$$

The auxiliary operation *union* unites two record sort schemes, taking the pairwise meet of any sorts associated with the same fields. For example:

- $union\{\textsf{w:}\ s_1, \textsf{x:}\ s_2\}\ \{\textsf{x:}\ s_3, \textsf{y:}\ s_4, \textsf{z:}\ s_5\} = \{\textsf{w:}\ s_1, \textsf{x:}\ (s_2\ \&\ s_3), \textsf{y:}\ s_4, \textsf{z:}\ s_5\}$
- $union\{\textsf{w:}\ s_1, \textsf{x:}\ s_2\}\rho_1\ \{\textsf{x:}\ s_3, \textsf{y:}\ s_4\} = \{\textsf{w:}\ s_1, \textsf{x:}\ (s_2\ \&\ s_3), \textsf{y:}\ s_4\}\rho_2$, where ρ_1 is instantiated to $\{\textsf{y:}\ s_4\}\rho_2$
- $union\{\textsf{w:}\ s_1, \textsf{x:}\ s_2\}\rho_1\ \{\textsf{x:}\ s_3, \textsf{y:}\ s_4\}\rho_2 = \{\textsf{w:}\ s_1, \textsf{x:}\ (s_2\ \&\ s_3), \textsf{y:}\ s_4\}\rho_3$, where ρ_1 is instantiated to $\{\textsf{y:}\ s_4\}\rho_3$, and ρ_2 is instantiated to $\{\textsf{w:}\ s_1\}\rho_3$

In these examples, if the sort $s_2\ \&\ s_3$ is nothing, then the action is ill-sorted.

The auxiliary operation *merge* concatenates two record sort schemes, insisting that their domains are disjoint. For example:

- $merge\{\textsf{w:}\ s_1\}\ \{\textsf{x:}\ s_2, \textsf{y:}\ s_3\} = \{\textsf{w:}\ s_1, \textsf{x:}\ s_2, \textsf{y:}\ s_3\}$
- $merge\{\textsf{w:}\ s_1\}\rho_1\ \{\textsf{x:}\ s_2, \textsf{y:}\ s_3\} = \{\textsf{w:}\ s_1, \textsf{x:}\ s_2, \textsf{y:}\ s_3\}\rho_2$, where ρ_1 is instantiated to $\{\textsf{x:}\ \textbf{absent}, \textsf{y:}\ \textbf{absent}\}\rho_2$

- $merge\{\mathsf{w}\colon s_1\}\rho_1\ \{\mathsf{x}\colon s_2, \mathsf{y}\colon s_3\}\rho_2 = \{\mathsf{w}\colon s_1, \mathsf{x}\colon s_2, \mathsf{y}\colon s_3\}\rho_3$,
  where ρ_1 is instantiated to $\{\mathsf{x}\colon \textbf{absent}, \mathsf{y}\colon \textbf{absent}\}\rho_3$,
  and ρ_2 is instantiated to $\{\mathsf{w}\colon \textbf{absent}\}\rho_3$

The auxiliary operations *combine* and *join* are peculiar to the (OR) rule, and reflect the fact that only one sub-action is performed by this combinator. The *combine* operation is similar to the *union* operation, except that it takes the pairwise join of any sorts associated with the same fields. For example:

- $combine\{\mathsf{w}\colon s_1, \mathsf{x}\colon s_2\}\ \{\mathsf{x}\colon s_3, \mathsf{y}\colon s_4, \mathsf{z}\colon s_5\} = \{\mathsf{w}\colon s_1, \mathsf{x}\colon (s_2 \mid s_3), \mathsf{y}\colon s_4, \mathsf{z}\colon s_5\}$

The *join* operation also forms the pairwise join of the sorts, but it insists on the domains of the record sorts being identical. Its use in rule (OR) enforces a deliberate restriction, that the transients and bindings passed out of the two sub-actions of 'or' must have identical domains – we forbid *conditional* transients and bindings. For example:

- $join\{\mathsf{x}\colon s_1, \mathsf{y}\colon s_2\}\ \{\mathsf{x}\colon s_3, \mathsf{y}\colon s_4\} = \{\mathsf{x}\colon (s_1 \mid s_3), \mathsf{y}\colon (s_2 \mid s_4)\}$

The following are the most important rules that deal with bindings:

$$\text{(BIND)}\ \frac{\mathcal{E} \vdash Y : (t, b) \rightsquigarrow s; \qquad s\ \&\ \mathsf{bindable} \neq \mathsf{nothing}}{\mathcal{E} \vdash \mathsf{bind}\ k\ \mathsf{to}\ Y : (t, b) \hookrightarrow (\{\}, \{\mathsf{k}\colon s\ \&\ \mathsf{bindable}\})}$$

$$\text{(BOUND)}\ \frac{s\ \&\ \theta \neq \mathsf{nothing}}{\mathcal{E} \vdash \mathsf{the}\ s\ \mathsf{bound\ to}\ k : (\{\}\gamma_i, \{\mathsf{k}\colon \theta\}\gamma_j) \rightsquigarrow s\ \&\ \theta}$$

$$\text{(HENCE)}\ \frac{\mathcal{E} \vdash A_1 : (t_1, b_1) \hookrightarrow (t_1', b_1'); \qquad \mathcal{E} \vdash A_2 : (t_2, b_2) \hookrightarrow (t_2', b_2'); \qquad b_1' = b_2}{\mathcal{E} \vdash A_1\ \mathsf{hence}\ A_2 : (union\ t_1\ t_2, b_1) \hookrightarrow (merge\ t_1'\ t_2', b_2')}$$

Rule (BIND) is straightforward. Rule (BOUND) infers that the yielder '**the** s **bound to** k' expects to receive a binding of k to a datum of sort θ. Here θ is a *sort variable*, which is to be instantiated to some actual sort that satisfies the stated constraint that $s\ \&\ \theta \neq$ **nothing**. The sort variable θ will be instantiated to a particular sort, depending on the received bindings. For example, if θ is instantiated to a subsort $s' \leq s$, then the yielder's result sort will be narrowed to $s\ \&\ s' = s'$. (In the extreme case θ might be instantiated to an individual, whereupon the yielder's result sort will be instantiated to that individual.) However, if θ is instantiated to a supersort of s, then the inference rule indicates a place where a run-time sort check is required, since the datum bound to k might turn out at run-time not to be of sort s.

In rule (HENCE), the antecedent '$b_1' = b_2$' insists that the sort of bindings produced by A_1 be unified with the sort of bindings received by A_2. For example:

- $b_1' = \{\mathsf{x}\colon s_1, \mathsf{y}\colon s_2\}$ and $b_2 = \{\mathsf{x}\colon s_1, \mathsf{y}\colon s_3\}$. These can be made equal by replacing both b_1' and b_2 by their meet, $\{\mathsf{x}\colon s_1, \mathsf{y}\colon (s_2\ \&\ s_3)\}$. If $s_2\ \&\ s_3 =$ **nothing**, b_1' and b_2 cannot be made equal, therefore we have inferred that 'A_1 **hence** A_2' is ill-sorted.

To be concrete, if s_2 = 7 and s_3 = integer, then s_2 & s_3 = 7. In other words, having already inferred that A_2 expects a binding of "y" to an unknown integer, we have now inferred from the context of A_2 that the integer is, in fact, 7.

Or if s_2 = truth-value and s_3 = integer, then s_2 & s_3 = nothing. In other words, A_1 binds "y" to a truth value, but A_2 expects a binding of "y" to an integer. Clearly 'A_1 hence A_2' is ill-sorted.

- $b'_1 = \{\mathsf{x}\text{: } s_1, \mathsf{y}\text{: } s_2\}$ and $b_2 = \{\mathsf{x}\text{: } s_1\}\gamma_1$. These can be made equal by instantiating γ_1 to $\{\mathsf{y}\text{: } s_2\}$. In other words, we have inferred that A_2 receives a binding of "y" to a datum of sort s_2 (which it ignores) as well as a binding of "x" to a datum of sort s_1.

- $b'_1 = \{\mathsf{x}\text{: } s_1, \mathsf{y}\text{: } s_2\}$ and $b_2 = \{\mathsf{x}\text{: } s_1, \mathsf{z}\text{: } s_3\}\gamma_1$. These cannot be made equal, however we instantiate γ_1. Therefore we have inferred that 'A_1 hence A_2' is ill-sorted.

The following rules show how we infer the sorts of 'unfolding' actions:

$$\text{(UNFOLDING)}\ \frac{[\mathsf{unfold} : as_u]\mathcal{E} \vdash A : as; \qquad as = as_u}{\mathcal{E} \vdash \mathsf{unfolding}\ A : as}$$

$$\text{(UNFOLD)}\ \frac{}{[\mathsf{unfold} : as]\mathcal{E} \vdash \mathsf{unfold} : as}$$

We insist that, inside 'unfolding A', every occurrence of 'unfold' has the same sort as_u, which can be unified with the sort as of A itself. This restriction excludes polymorphic 'unfolding' actions[6]. However, it does not exclude the 'unfolding' actions that occur in practical situations, such as the semantics of loops in programming languages.

The following are some of the rules that deal with abstractions:

$$\text{(ABSTRACTION)}\ \frac{\mathcal{E} \vdash A_0 : (t_0, b_0) \hookrightarrow (t'_0, b'_0)}{\mathcal{E} \vdash \mathsf{abstraction}\ A_0 : \text{abstraction}(t_0, b_0) \hookrightarrow (t'_0, b'_0)}$$

$$\text{(ENACT)}\ \frac{\mathcal{E} \vdash Y : (t, b) \rightsquigarrow \text{abstraction}(\{\}, \{\}) \hookrightarrow (t'_0, b'_0)}{\mathcal{E} \vdash \mathsf{enact}\ Y : (t, b) \hookrightarrow (t'_0, b'_0)}$$

$$\text{(CLOSURE)}\ \frac{\mathcal{E} \vdash Y : (t, b) \rightsquigarrow \text{abstraction}(t_0, b_0) \hookrightarrow (t'_0, b'_0)}{\mathcal{E} \vdash \mathsf{closure}\ Y : (t, \mathit{union}\ b\ b_0) \rightsquigarrow \text{abstraction}(t_0, \{\}) \hookrightarrow (t'_0, b'_0)}$$

$$\text{(WITH)}\ \frac{\begin{array}{c}\mathcal{E} \vdash Y_1 : (t_1, b_1) \rightsquigarrow \text{abstraction}(t_0, b_0) \hookrightarrow (t'_0, b'_0); \\ \mathcal{E} \vdash Y_2 : (t_2, b_2) \rightsquigarrow s_2; \qquad t_0 = \{\mathsf{0}\text{: } s_0\}; \qquad s_0\ \&\ s_2 \neq \mathsf{nothing}\end{array}}{\mathcal{E} \vdash Y_1\ \mathsf{with}\ Y_2 : (\mathit{union}\ t_1\ t_2, \mathit{union}\ b_1\ b_2) \rightsquigarrow \text{abstraction}(\{\}, b_0) \hookrightarrow (t'_0, b'_0)}$$

[6]For which sort inference is undecidable [11].

Rule (ABSTRACTION) shows the isomorphism between the sort of 'abstraction A_0' and the sort of the incorporated action A_0. Rule (ENACT) insists that the transients and bindings required by the abstraction's incorporated action are empty. Suppose that this is not the case, e.g., that the incorporated action expects to receive non-empty bindings; then the 'enact' action will fail, since this action does not itself supply any bindings to the incorporated action. (Only the 'closure' operation does so.)

Rule (CLOSURE) infers the sort of bindings required to form the closure of an abstraction, principally the bindings b_0 required by the incorporated action (united with the bindings b required to evaluate Y). The sort of the resulting abstraction indicates that it requires no bindings (as required for use in an 'enact' action).

Rule (WITH) is slightly more complicated. Firstly, the abstraction must receive a single transient datum labelled 0 (the antecedent $t_0 = \{0\text{: } s_0\}$). Secondly, the sort s_0 of this transient must be consistent with the sort s_2 of the datum actually supplied ($s_0 \,\&\, s_2 \neq$ nothing). Again, the sort of the resulting abstraction is made to have empty input transients.

Space does not permit us to present all the sort inference rules here. They are presented in full in [3].

5.2 Sort Inference Algorithm

Our sort inference algorithm is based on the Even–Schmidt algorithm [7], but is improved in several important respects. Our algorithm achieves a greater measure of internal uniformity, by using record schemes for both transients and bindings. It infers exactly which transients and bindings an action uses, using γ-variables to represent transients and bindings passed to the action but not used. It infers action sorts more precisely, by using a more refined sort hierarchy (Figure 1). Not least, it handles a much larger and more representative subset of action notation, including choice, iteration, and abstractions, all of which are essential for writing useful action-semantic descriptions.

Our algorithm consists of three passes. The first pass annotates the given action with record schemes, in accordance with the sort inference rules. The second pass uses reduces all sorts to canonical form, in particular, eliminating all occurrences of '&'. It also removes all field and row variables. Field variables are no longer required, and may be eliminated by instantiating them to **absent**, and then simplifying the record sorts. Row variables can be eliminated if it is assumed that no transients and bindings are passed to the program action. In effect, any ρ-variable at the top-level is instantiated to $\{\}$, and any γ-variables can be safely discarded[7]. The third pass marks places where run-time sort checks are required, replaces ill-sorted actions by 'fail', and simplifies the program action.

[7] An unusual form of 'garbage collection'!

5.3 An Example of Sort Inference

Consider the following little program in a simple imperative language:

let const b ~ $true$;
  var x: **int**
in while b **do** $x := -x$

This would be mapped to the following program action:

```
 1  furthermore
 2  | | give true then bind "b" to the value
 3  | before
 4  | | allocate a cell then bind "x" to the cell
 5  hence
 6  | unfolding
 7  | | | give the value bound to "b" or
 8  | | | give the value stored in the cell bound to "b"
 9  | | then
10  | | | | | give the value bound to "x" or
11  | | | | | give the value stored in the cell bound to "x"
12  | | | | then give negation (the integer)
13  | | | | then store the value in the cell bound to "x"
14  | | | and then unfold
15  | | | else complete
```

First consider the action on line 2. Application of rules (GIVE)[8] and (BIND) to the sub-actions gives:

$$\begin{aligned} \text{give true} &: (\{\}\gamma_1, \{\}\gamma_2) \hookrightarrow (\{\text{0: true}\}, \{\}) \\ \text{bind "b" to the value} &: (\{\text{0: } \theta_1\}\gamma_3, \{\}\gamma_4) \hookrightarrow (\{\}, \{\text{b: (value } \& \ \theta_1\})) \end{aligned}$$

Application of rule (THEN)[9] now forces unification of the first sub-action's outgoing transients record sort $\{\text{0: true}\}$ with the second sub-action's incoming transients record sort $\{\text{0: } \theta_1\}\gamma_3$. Thus the sort variable θ_1 is instantiated to true. The resulting sort assignments are:

$$\begin{aligned} \text{bind "b" to the value} &: (\{\text{0: true}\}\gamma_3, \{\}\gamma_4) \hookrightarrow (\{\}, \{\text{b: true}\}) \\ \text{give true then bind "b" to the value} &: (\{\}\gamma_1, \{\}\gamma_5) \hookrightarrow (\{\}, \{\text{b: true}\}) \end{aligned}$$

The action on line 4 is assigned the following sort:

$$\text{allocate a cell then bind "x" to the cell} : (\{\}\gamma_6, \{\}\gamma_7) \hookrightarrow (\{\}, \{\text{x: cell}\})$$

[8](GIVE) is not shown in this paper, but is analogous to (BIND).
[9](THEN) also is not shown in this paper, but is analogous to (HENCE).

and the 'before' action on lines 2–4 is assigned the following sort:

$$\ldots \text{ before} \ldots \;:\; (\{\}\gamma_8, \{\}\gamma_9) \hookrightarrow (\{\}, \{\text{b: true, x: cell}\})$$

Application of rules (BOUND), (STORED), and (GIVE) to the actions on lines 7 and 8 gives:

give the value bound to "b"
: $(\{\}\gamma_{10}, \{\text{b: } \theta_2\}\gamma_{11}) \hookrightarrow (\{\text{0: } (\theta_2 \;\&\; \text{value})\}, \{\})$
give the value stored in the cell bound to "b"
: $(\{\}\gamma_{12}, \{\text{b: } \theta_3\}\gamma_{13}) \hookrightarrow (\{\text{0: } (\theta_4 \;\&\; \text{value})\}, \{\})$

subject to the constraints $\theta_2 \,\&\, \text{value} \neq \text{nothing}$, $\theta_3 \,\&\, \text{cell} \neq \text{nothing}$, and $\theta_4 \,\&\, \text{value} \neq \text{nothing}$. Application of rule (OR) to the 'or' action on lines 7–8 now gives:

give the value bound to "b" or
give the value stored in the cell bound to "b"
: $(\{\}\gamma_{14}, \{\text{b: } (\theta_2 \mid \theta_3)\}\gamma_{15}) \hookrightarrow (\{\text{0: } (\theta_2 \;\&\; \text{value}) \mid (\theta_4 \;\&\; \text{value})\}, \{\})$

Eventually, application of rule (HENCE) will instantiate the sort variables θ_2 and θ_3 to true. Thus the antecendent $\theta_3 \,\&\, \text{cell} \neq \text{nothing}$ is not satisfied, and the action on line 8 is ill-sorted. This action can be replaced by 'fail', and the identity 'A or fail $= A$' can be used to simplify the 'or' action to 'give the value bound to "b"'.

A similar argument applies to the other 'or' action, on lines 10–11. Because of the binding x : cell, however, this 'or' action is simplified to 'give the value stored in the cell bound to "x"'.

Finally, consider the sort inferred for the 'unfolding' action on lines 6–15. Initially, in the (UNFOLDING) rule, we take $as_u = (\{\}\rho_1, \{\}\rho_2) \hookrightarrow (\{\}\rho_3, \{\}\rho_4)$, and proceed with sort inference of the sub-action. Rule (ELSE)[10] will cause p_3 and p_4 to be instantiated to $\{\}$ when they are *join*-ed with the sort of complete:

$$\begin{aligned} \text{complete} \;&:\; (\{\}\gamma_{16}, \{\}\gamma_{17}) \hookrightarrow (\{\}, \{\}) \\ (\ldots \text{ and then unfold) else complete} \;&:\; (\ldots, \ldots) \hookrightarrow (\{\}, \{\}) \end{aligned}$$

The variables ρ_1 and ρ_2 are instantiated when applying rule (AND-THEN). When the sort variables are also instantiated, $\{\}\rho_1$ and $\{\}\rho_2$ become $\{\}\gamma_{18}$ and $\{\text{b: true, c: cell}\}\gamma_{19}$, respectively. The final sorts assigned to the actions are:

$$\begin{aligned} \text{unfolding} \ldots \;&:\; (\{\}\gamma_{18}, \{\text{b: true, x: cell}\}\gamma_{19}) \hookrightarrow (\{\}, \{\}) \\ \text{unfold} \;&:\; (\{\}\gamma_{18}, \{\text{b: true, x: cell}\}\gamma_{19}) \hookrightarrow (\{\}, \{\}) \end{aligned}$$

6 Related Work

As well as the work described in this paper, sort inference for action notation is a central theme of the mainly theoretical work of Schmidt's group at Kansas State University

[10]The 'else' combinator is related to the 'or' combinator, and has a similar rule.

[7, 6], and forms part of the more practical compiler-generation work of Palsberg at Aarhus University [10].

In [7], Even and Schmidt study the sort properties of a small dialect of action notation, and present a sort inference algorithm for this dialect. They assign 'kinds' as well as sorts to actions, and allow actions to be composed only if they are kind-compatible. (For example, they do not permit an action that produces bindings to be composed with an action that uses no bindings.) Their action sorts are based on record schemes, similar to those used in this paper. Their dialect of action notation is very small indeed: it lacks yielders, and it lacks some important combinators such as 'or' and 'unfolding'.[11] A fundamental limitation is that they require abstractions to be already annotated by their sorts. Nevertheless, Even and Schmidt's work has strongly influenced our own. Our main contributions have been removal of the unnecessary 'kind' structure, extension to a more representative subset of action notation, proper treatment of abstractions, and formalisation of the sort inference algorithm by a complete set of inference rules [3].

In [6], Doh and Schmidt address a related problem in sort inference. On the *assumption* that the described language is statically typed, they show how to extract static type inference rules from the semantic equations of an action-semantic description. We intend to develop this work and apply it to ACTRESS. At present, every ACTRESS-generated compiler includes the action notation sort checker, which is rather a sledgehammer to crack what might be a small nut (if the source language happens to have a simple type system). Instead, we aim to generate a language-specific sort checker from the action-semantic description.[12] Also, we are currently studying how to *infer* (as opposed to just assuming) whether the source language is statically typed. ACTRESS will not restrict the source language, however. It will continue to accept semantic descriptions of both dynamically-typed and statically-typed languages, but will recognise the latter special case and exploit it to generate compilers that avoid generating run-time sort checks.[13]

Palsberg's compiler generation system CANTOR [10] takes a pragmatic approach to sort inference (which is not a central part of his work). His sort-checker assumes that, for each action and sub-action A, the sorts of the transients and bindings passed into A are initially known, and uses these to infer the sorts of the transients and bindings passed out of A. His algorithm is consequently much simpler than ours, avoiding the heavy machinery of row variables, record schemes, unification, and so on. However, when A is the sub-action of 'unfolding', or the body of an abstraction, the sorts of the transients and bindings passed into A are *not* initially known. In these cases Palsberg's sort-checker resorts to *ad hoc* means to continue.

[11] However, their methods can be extended to remove some of these limitations [11].

[12] An alternative approach would be to generate a conventional type-checker from the language's *static-semantic* description. Of itself this would be straightforward, but there would be no guarantee that the given static semantics is sound with respect to the language's dynamic semantics.

[13] Analogously, ACTRESS will continue to accept both dynamically-scoped and statically-scoped languages, but will recognise the latter special case and exploit it to generate compilers that avoid generating code to manipulate bindings at run-time [9].

Recently, Aiken *et al.* [1, 2] have applied type inference with type constraints to the problem of analysing a dynamically-typed λ-calculus to identify the places where run-time type checks are necessary. The type system they use has several features in common with the sort system of action notation: individuals as types, subtypes, and intersection and union of types, They also have conditional types, and their algorithm, instead of relying on unification, builds and solves a system of constraints of the form $\tau_i \leq \tau_j$, where τ_i and τ_j are certain kinds of types.

7 Conclusion

In this paper, we aimed to show that sort inference in action notation is a difficult problem. In fact, for the unrestricted notation, sort inference is undecidable. However, we believe that the sort inference performed within the ACTRESS compiler generator provides one of the most sophisticated analyses of action notation to date. Indeed, ACTRESS remains unique in its ability to handle actions requiring runtime sort checks. However, we also recognise that better performance could be obtained by generating a language-specific sort checker, using information gained from analysing the action semantics specification of the language. We are currently investigating how our analysis of actions can be extended to deal with an entire specification, and also what properties of the language may be apparent from such an analysis. In particular, we are interested in guaranteeing the absence of runtime sort checks for any program in the specified language.

Acknowledgments

Hermano Moura, the third member of the ACTRESS group, has made innumerable comments and suggestions on the work reported here. We have benefited from numerous interactions with David Schmidt and Kyung-Goo Doh of Kansas State University. Peter Mosses and Phil Wadler have also, at various times, provided valuable inputs. We are happy to acknowledge all their contributions.

References

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming and Computer Architecture '93*, pages 31–41, 1993.

[2] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the 21st Conference on Principles of Programming Languages*, 1994.

[3] D. F. Brown. *Sort inference in action semantic specifications.* PhD thesis, Department of Computing Science, University of Glasgow, 1994. In preparation.

[4] D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an action semantics directed compiler generator. In *Compiler Construction '92*, Lecture Notes in Computer Science, pages 95–109. Springer-Verlag, 1992.

[5] L. Cardelli and J. C. Mitchell. Operations on records. In *Workshop on Mathematical Foundations of Programming Language Semantics*, Lecture Notes in Computer Science. Springer, 1989.

[6] K. Doh and D. A. Schmidt. Extraction of strong typing laws from action semantics definitions. In *ESOP '92*, volume 582 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[7] S. Even and D. A. Schmidt. Type inference for action semantics. In N. Jones, editor, *ESOP '90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 118–133, Copenhagen, Denmark, 1990. Springer-Verlag, Berlin, Germany.

[8] P. D. Mosses. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1993.

[9] H. Moura and D. A. Watt. Action transformations in the ACTRESS compiler generator. In *Compiler Construction 5th International Conference, CC '94*, Lecture Notes in Computer Science, pages 16–30. Springer-Verlag, 1994.

[10] J. Palsberg. A provably correct compiler generator. In *ESOP '92*, volume 582 of *Lecture Notes in Computer Science*, pages 418–434. Springer-Verlag, 1992.

[11] D. A. Schmidt, Apr. 1991. Personal communication.

[12] M. Wand. Complete type inference for simple objects. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, 1987.

[13] M. Wand. Type inference for record concatenation and simple objects. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, 1989.

[14] D. A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, England, 1991.

OASIS: An Optimizing Action-based Compiler Generator

Peter Ørbæk <poe@daimi.aau.dk>*

April 25, 1994

Abstract

Action Semantics is a new and interesting foundation for semantics based compiler generation. In this paper we present several analyses of actions, and apply them in a compiler generator capable of generating efficient, optimizing compilers for procedural and functional languages with higher order recursive functions. The automatically generated compilers produce code that is comparable with code produced by handwritten compilers.

1 Introduction

Semantics based compiler generation has long been a goal in computer science. Automatic generation of compilers from semantic descriptions of programming languages relieves programmers and language theorists from much of the burden of writing compilers.

We describe the OASIS (Optimizing Action-based Semantic Implementation System) compiler generator, and especially the analyses that provide the information enabling the code generator to produce good quality code.

The generated compilers expand a given abstract syntax tree to the equivalent action by way of the action semantics for the language. All analyses are applied to the expanded action. The system is capable of generating compilers for procedural, functional (lazy and eager) and object oriented languages. After analysis, the action is translated to native SPARC code. For further details, see [18].

A short introduction to Action Notation is given first, and in the following section we describe a type-checker for actions, whose *raison d'être* is to allow us to dispense with all run-time type checks.

We then proceed to describe the various analyses that are carried out on the type checked action. Most of the analyses are set up in an abstract interpretation framework. The analyses annotate the action with approximate information about its run-time behavior.

The results of the analyses are used by a code generator, generating code for the SPARC processor. The code generator also employs a couple of optimization techniques on its own, namely a storage cache used to avoid dummy stores and reloads, and a peephole optimizer responsible for filling delayslots and removing no-op code.

*This work was partially funded by BRICS at the Comp. Sci. Dept. of Aarhus University. This article also appears in the CC'94 proceedings, volume 786 of LNCS, all references should point to that article.

Finally we compare the performance of the generated compilers for a procedural and a functional language with handwritten compilers for similar languages, and relate our results to previous approaches to compiler generation.

The results are very encouraging as our automatically generated compilers emit code that performs within a factor 2 of code produced by handwritten compilers. This is a major performance enhancement in relation to earlier approaches to compiler generation based on Action Semantics [20, 19, 3], as well as compared to other semantics based compiler generators.

2 Action Notation

Action Semantics is a formalism for the description of the dynamic semantics of programming languages, developed by Mosses and Watt [17]. Based on an order-sorted algebraic framework, an action semantic description of a programming language specifies a translation from abstract terms of the source language to Action Notation.

Action Notation is designed to allow comprehensible and accessible semantic descriptions of programming languages; readability and modularity are emphasized over conciseness. Action semantic descriptions scale up well, and considerable reuse of descriptions is possible among related languages. An informal introduction to Action Notation, as well as the formal semantics of the notation, can be found in [17].

The semantics of Action Notation is itself defined by a structural operational semantics, and actions reflect the gradual, stepwise, execution of programs. The performance of an action can terminate in one of three ways: It may *complete*, indicating normal termination; it may *fail*, to indicate the abortion of the current alternative; or it may *escape*, corresponding to exceptional termination which may trapped. Finally, the performance of an action may *diverge*, ie. end up in an infinite loop.

Actions may be classified according to which *facet* of Action Notation they belong. There are five facets:

- the *basic* facet, dealing with control flow regardless of data.
- the *functional* facet, processing *transient* information, actions are *given* and *give* data.
- the *declarative* facet, dealing with bindings (*scoped information*), actions *receive* and *produce* bindings.
- the *imperative* facet, dealing with loads and stores in memory (*stable information*), actions may *reserve* and *unreserve* cells of the storage, and change the contents of the cells.
- the *communicative* facet, processing *permanent information*, actions may *send* and *receive* messages communicated between processes.

In general, imperative and communicative actions actions are *committing*, which prevents backtracking to alternative actions on failure. There are also hybrid actions that deal with more than one facet. Below are some example action constructs:

- 'complete': the simplest action. Unconditionally completes, gives no data and produces no bindings. Not committing.

- 'A_1 and A_2': a basic action construct. Each sub-action is given the same data as the combined action, and each receives the same bindings as the combined construct. The data given by the two sub-actions is tupled to form the data given by the combined action, (the construct is said to be *functionally conducting*). The performance of the two sub-actions may be interleaved.

- 'A_1 or A_2': a basic action construct, represents non-deterministic choice between the two sub-actions. Either A_1 or A_2 is performed. If A_1 fails without committing A_2 is performed, and vice versa.

- 'store Y_1 in Y_2': an imperative action. Evaluates the yielder Y_1 and stores the result in the cell yielded by Y_2. Commits and completes when Y_1 evaluates to a storable and Y_2 evaluates to a cell.

An action term consists of constructs from two syntactic categories, there are action constructs like those described above, and there are yielders that we will describe below. Yielders may be evaluated in one step to yield a value. Below are a few example yielders:

- 'sum(Y_1, Y_2)': evaluates the yielders Y_1 and Y_2 and forms the sum of the two numbers.

- 'the given D#n': picks out the n'th element of the tuple of data given to the containing action. Yields the empty sort nothing unless the n'th item of the given data is of sort D.

- 'the D stored in Y': provided that Y yields a cell, it yields the intersection of the contents of that cell and the sort D.

As an example we give below an action semantics for a simple call-by-value λ-calculus with constants.

2.1 Abstract Syntax

needs: **Numbers/Integers**(integer), **Strings**(string) .

grammar:

(1) Expr = ⟦ "lambda" Var "." Expr ⟧ | ⟦ Expr "(" Expr ")" ⟧ |
⟦ Expr "+" Expr ⟧ | integer | Var .

(2) Var = string .

2.2 Semantic Functions

includes: **Abstract Syntax** .

introduces: evaluate _ .

- evaluate _ :: Expr → action .

(1) evaluate I:integer = give I .

(2) evaluate V:Var = give the datum bound to V .

(3) evaluate ⟦ "lambda" *V*:Var "." *E*:Expr ⟧ =
give the closure abstraction of
| furthermore bind *V* to the given datum#1
| hence evaluate *E* .

(4) evaluate ⟦ E_1:Expr "(" E_2:Expr ")" ⟧ =
| evaluate E_1 and evaluate E_2
then enact application the given abstraction#1 to the given datum#2 .

(5) evaluate ⟦ E_1:Expr "+" E_2:Expr ⟧ =
| evaluate E_1 and evaluate E_2
then give the sum of them .

2.3 Semantic Entities

includes: Action Notation .

- datum = abstraction | integer | □ .
- bindable = datum .
- token = string .

3 Type Checking

The main purpose of the type-checker for action notation is to eliminate the need for run-time type-checking. If we hope to gain a run-time performance comparable to traditional compiled languages such as C and Pascal, we need to eliminate run-time type-checks, otherwise values would have to carry tags around identifying their type, and we would immediately suffer the penalty of having to load and store the tags as well as the actual values. We are thus lead to choose a wholly static type system.

Our type-checker is related to the one given by Palsberg in [20], but our type-checker is also capable of handing unfoldings that are not tail-recursive. This imposes some problems, since fixpoints have to be computed in the type lattice. Like Palsberg's type-checker, our type-checker can be viewed as an abstract interpretation of the action over the type lattice.

The type-checker has been proved *safe* with respect to the structural operational semantics of a small subset of Action Notation [18], but we will not go into details about the proof here, we just give the structure of the proof. It should be straightforward, but tedious, to extend the proof to the larger subset accepted by OASIS.

First a type inference system for a small subset of Action Notation is defined. It is shown that the inference system has the Subject Reduction property with respect to the operational semantics of Action Notation. Second, the type checking *algorithm* is proved *sound* with respect to the inference system. Finally subject reduction and soundness are combined to prove the safety of the type checker. Below we state the main safety result:

> *The type-checker is safe in the sense that, if the type-checker infers a certain type for the outcome of an action then, when that action is actually performed, the outcome indeed has the inferred type.*

As a corollary to the safety property, we have proved that all run-time type-checks can be omitted.

4 Analyses

The main reason for the good run-times that we are able to achieve for the produced code, is the analyses that we apply to the action generated from the semantics. The analyses consist of the following stages:

- **forward analysis**, incorporating constant analysis, constant propagation, commitment analysis and termination analysis.
- **backwards flow analysis**, used to shorten the lifetime of registers.
- **heap analysis**, determines which frames can be allocated on the stack, and which need to be allocated on the heap.
- **tail-recursion detection**, checking whether unfoldings are tail-recursive or not.

4.1 Forward Flow Analysis

The forward flow analysis is essentially an abstract interpretation of the action over a product of several complete lattices. The various parts of the analysis are interleaved in order to obtain better results than would be possible had the analyses been done one after the other.

The forward analyses can be divided up into the following parts:

- **constant analysis**, determines whether bound values are static or dynamic.
- **constant propagation and folding**, propagates constants and folds expressions with constant arguments into constants.
- **commitment analysis**, approximates the commitment nature of the action.
- **termination analysis**, approximates the termination mode of the action.

All of the analyses are set up in an abstract interpretation framework [5]. They are all essentially intra-procedural, so each abstraction is not analyzed in relation of all of its enactions (calls).

Constant propagation and folding is a well-known technique often used in compilers to reduce constant expressions to constants. There is nothing special about our constant propagation technique for actions. For example if the two arguments propagated to a "sum" yielder are constant, the sum is folded into a constant at compile time, and propagated further as a constant. The other parts of the forward analysis are more interesting.

4.1.1 Constant Analysis

Since the constant analysis is integrated with the constant propagation, we use the following lattice of abstract values for this part of the analysis:

$$\mathsf{SD} = (\{(\mathsf{Static}, v), \mathsf{Dynamic}\}, \leq)$$

where for all values v : $(\mathsf{Static}, v) \leq \mathsf{Dynamic}$.

The above lattice differs from the traditional lattice used in binding time analyses (eg. in [1]) by incorporating the statically known value with the Static tag. This only buys us a marginal benefit, but it is simply the obvious thing to do when the constant analysis and constant propagation are integrated.

A binding of a constant (Static) value need not have space allocated in the frame of the enclosing abstraction, as the bound value can be inserted statically wherever it is referenced. Bindings of dynamic values are associated with a cell (a memory location) in the relevant frame, and when a value is bound at run-time, it is stored in the cell, and it is retrieved from that cell whenever the bound value is referenced.

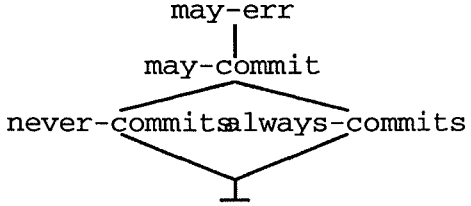
Arguments to abstractions are assumed to be dynamic. We do not attempt to do inter-procedural constant analysis. The need for such an analysis is not too great for ordinary imperative languages, where fewer and larger procedural abstractions dominate. Also, an intra-procedural analysis can be done more efficiently. We avoid the compile-time performance problems often associated with inter-procedural abstract interpretations.

Likewise, the analysis assumes that the contents of memory cells are dynamic. All loads and stores in Action Notation go through pointers, and storing a value to wherever a dynamic pointer points may over-write the contents of any cell of the same type. This problem with dynamic pointers is usually known as *aliasing* problems in traditional compilers. Since the performance of sub-actions may be interleaved, it is hard to guarantee that the contents of a cell has not been over-written by an unknown value at any given point of the performance.

4.1.2 Commitment Analysis

Actions may or may not *commit*. If an action commits it means that it has made some irreversible change to the state of the machine, such as having stored a value in a cell or having sent a message to another process.

We are interested in knowing whether an action may commit and subsequently fail (ie. err) within an "A_1 or A_2" construct. If this is the case, the "or" can't trap the failure, a run-time error should be indicated and the program stopped (in the absence of commitment, an alternative action may be performed). In the CANTOR system [20, 19], a significant amount of run-time is used to check for such committed failures. Our analysis is able to statically determine the possibility of committed failures in most cases, thus much fewer run-time checks need to be inserted. The lattice used by the commitment analysis looks like this:



Commitment of ground actions such as "store Y_1 in Y_2" is determined by the type checker. Commitment of combined actions such as "A_1 and A_2" is determined by commitment of the sub-actions and the results of the termination analysis on the sub-actions. For unfoldings a fixed point is computed.

4.1.3 Termination Analysis

The termination analysis computes approximate knowledge about the termination mode of sub-actions. There are many benefits to be drawn from such knowledge. For example, suppose we have the action "A_1 then A_2". If the termination analysis is able to guarantee that the sub-action A_1 always fails, no code need to be generated for A_2!

The termination analysis abstractly interprets the action over the power-set of the four possible termination modes (*complete, fail, escape* and *diverge*) ordered by subset inclusion:

$$\mathsf{CFED} = (\mathcal{P}\{\mathsf{AC}, \mathsf{AF}, \mathsf{AE}, \mathsf{AD}\}, \subseteq)$$

The termination mode of ground actions is determined by the type checker and influenced by the propagated constants. The termination mode of combined actions is determined by the termination mode of the sub-actions.

The termination- and commitment analyses have been formally specified in [18], although soundness still remains to be proven.

4.2 Backwards Analysis

The backwards analysis is used to shorten the lifetime of transient values. This analysis traces the data-flow backwards and increases counters in the abstract compile-time representations of values each time such values are used, ie. stored to memory, written to standard output or passed as parameters to an abstraction.

During code generation, the same counters are decreased at each point of usage and when the counter reaches zero, the register holding the value can safely be discarded for eventual re-use. This analysis is similar to the computation of *live variables* in traditional optimizing compilers.

4.3 Heap Analysis

Since abstractions are first class values in action notation, they can be given as transient data, returned from abstractions and stored in memory. As we deal with statically scoped languages, we need to provide abstractions with their correct static environment when they are enacted (called).

In traditional languages with first class abstractions, such as Scheme, all frames (or activation records) are typically allocated on the heap and it is up to a garbage collector to release the associated memory when it is no longer used. In order to avoid spending lots of time doing garbage collection, and to avoid heap allocated frames for programs in traditional imperative languages such as Pascal, we employ the heap analysis[1]. The heap analysis is yet another abstract interpretation of the action, this time over the following domain:

$$(\mathcal{P}(\{\mathsf{SA}, \mathsf{PC}\}), \subseteq)$$

[1]This analysis was initially called *closure analysis* for obvious reasons, but that term has a more specific and different meaning in Copenhagen, so it was renamed.

The analysis traverses the action and marks each abstraction with an element from the above domain as explained in the following:

SA stands for *stores abstraction*, it means that the abstraction may store, give, or escape with an abstraction, ie. an abstraction may leak out of scope. PC stands for *provides closure*, it means that the abstraction provides (part of) the closure for another abstraction, ie. it has a syntactically nested abstraction. Only if an abstraction is marked {SA, PC} need the corresponding frame be allocated on the heap. Note that the top-most or global frame can always be allocated on the stack as it will exist until the program terminates.

Thanks to this analysis, an action semantics for full Pascal will never give rise to code needing heap allocated closures, as it is impossible for a procedure in Pascal to leak out of scope.

4.4 Tail Recursion

In order to implement standard `while` loops efficiently by the "unfolding" construct, we need to be able to detect tail-recursive unfoldings, so as not to incur the overhead of a procedure call for each iteration of the loop. The action semantic equation for a `while` construct would typically look something like:

```
execute ⟦ "while" E:Expression "do" S:Statements ⟧ =
    evaluate E then
    | unfolding
    | | | check the given truth-value then execute S then unfold
    | | or
    | | | check not the given truth-value .
```

In full generality, "unfold" may cause a recursive call. The tail-recursion detector traverses the body of the unfolding and marks "unfolds" as tail-recursive or recursive depending on whether any part of the loop-body may be executed after the "unfold". If there is just one recursive "unfold" in the body of a loop, then all "unfolds" in that loop are treated as recursive.

5 Code Generation

The code generator generates assembly code for the SPARC processor from the action tree annotated by the preceding analyses. The assembly code is generated in one pass, and registers are allocated on an as-needed basis.

As much of the code generator as possible is kept machine-independent, to facilitate easy porting of the code generator to other RISC processors. One machine-independent part of the code generator is the storage cache. It serves the purpose of minimizing the number of load and store operations in basic blocks. When a value is loaded from a known memory location into a register, an association between the register and the location is kept, such that a later load from the same address can be coded as a cheap register copy. Storing the contents of a register in a known memory location keeps the association between the location and the register in the same way, and the actual store is delayed until the last possible moment within the same basic block to avoid two stores to the same location just after each other. (This may not be entirely beneficial on a RISC architecture, where loads and stores should be spread out, but it was easier to implement than a full graph-coloring register allocation algorithm).

A machine-dependent part of the code generator is the peephole optimizer. A peephole optimizer is a traditional optimization technique, that is often used to remove dummy instruction sequences and to simplify instructions. Our peephole optimizer does not attempt to eliminate all dummy instructions, but is geared towards fixing deficiencies in the code generated by our specific code generator.

The code generator is pretty intricate, as there are lots of special cases to consider when one tries to generate good code for a realistic machine such as the SPARC. Perhaps a code generator generator such as `iBurg` [8] could be used clean this up.

6 Overview

The action compiler and compiler generator consists of many parts written in different languages[2]. This section gives an overview of the different parts and their interaction.

The compiler generator (`gencomp`) takes an action semantics written as a Scheme [4] program, and produces a compiler written in Perl [24]. Scheme was chosen because it was easy to implement a few macros in Scheme that make it painless to write a semantics using Scheme syntax. Also, it was felt that not too much time should be spent on this part of the compiler generator, as work is in progress that will make it possible to write action semantics in the ASF+SDF system [11].

The generated compiler driver (or front-end) is written in Perl for ease of implementation. The driver parses command-line options, calls the different parts of the compiler and takes care of cleaning up if something goes wrong, such as a syntax error in the given program etc.

The compiler takes a textual representation of an abstract syntax tree (AST) for a program in the source language, and produces, if all goes well, executable code for the SPARC processor. The AST could easily be produced by, say, a YACC or Bison generated parser for the source language.

The first step of the compiler is to massage the input AST into something resembling a Scheme program, and combine it with the Scheme representation of the semantics for the source language.

The second step runs a Scheme interpreter on the semantics and the munged program, and writes a textual representation of the action corresponding to the program to a file. Currently, the free `scm` implementation of the Scheme standard [4] is used.

Step three runs the action compiler on the produced action, and produces assembly code for the SPARC processor. This is the major step of the process. The action compiler consists of approximately 10,000 lines of C++ code [21], plus a lexical analyzer generated by Flex and an action parser generated by Bison.

The fourth step of the compiler assembles the output from the action compiler and links the object module with a small run-time support library providing primitive input/output routines. The run-time library is written in traditional C [10].

The produced object program reads from standard input and writes to standard output.

[2]The OASIS system is available by anonymous ftp from `ftp.daimi.aau.dk` in the directory `/pub/action/systems/`

7 Comparisons

Here we compare the performance of our compiler generator with handwritten compilers and other approaches to compiler generation. We consider two example languages: HypoPL and FunImp.

The procedural language HypoPL contains integers, booleans, arrays, the usual control structures (while-loops and conditionals) and generally nested procedures. The syntax and semantics for HypoPL can be found in [12, 19, 18]. The functional (eager) language FunImp contains higher order recursive functions as well as mutable data. The syntax and semantics for FunImp is derived from the language considered in [22], and is defined in [18].

All our timings, except for the run-times of the generated code, are made on an ordinary 33 MHz 386-based PC with 20 MB RAM, running the Linux operating system (version 0.99). The generated SPARC code was run on a Sun Microsystems SparcStation ELC running SunOS 4.1.1.

Generating a compiler for Lee's HypoPL language [12, 13] takes 0.8 seconds. Using the generated compiler to compile the HypoPL bubblesort program takes 3.9 seconds. As explained in a previous section, this involves running the Perl interpreter, the Scheme interpreter and the action compiler, and the result is an assembly file suitable for the SPARC assembler. The assembly code consists of roughly 250 instructions, ie. 1000 bytes when assembled.

Comparing these figures with what Palsberg obtained with the CANTOR system [20, 19] shows that the compilers we generate are two orders of magnitude faster than his, and that the code size is also two orders of magnitude smaller than his. It should be noted that Palsberg's tests were also run on a Sun SparcStation ELC.

The tables below show some results from using the generated HypoPL compiler to compile some example programs (the same programs as used in [20, 19]):

- **bubble:** A bubblesort program, bubblesorts 500 integers.
- **sieve:** The sieve of Eratosthenes, finds all primes below 512, repeated 400 times.
- **euclid:** Euclid's method of finding the greatest common divisor of two numbers (1023 and 37), repeated 30,000 times.
- **fib:** Computes the 46'th Fibonacci number 10,000 times. (The 46'th number in the series is the largest that will fit in a 32 bit twos complement integer.)

The table below lists the compile times of various programs. The first column lists the time it takes to compile the HypoPL program to assembly, the second column lists the time it takes to compile the action generated from the HypoPL program, and the last column lists the time it takes to compile an equivalent C program with optimization turned on. All times are in seconds.

Program	HypoPL	Action	C-opt
bubble	3.9	0.9	0.6
sieve	3.4	0.9	0.5
euclid	2.3	0.4	0.4
fib	2.1	0.4	0.3

The figures above indicate that something could be gained by integrating the processing of the semantic functions with the action compiler, instead of relying on a Scheme interpreter to expand the program to an action.

The generated HypoPL compiler is on average 6.5 times slower than the hand-written C compiler. Much of this slowdown stems from the Scheme interpreter. The action compiler itself compiles an action within a factor two of the time it takes to compile the equivalent C program. This is what one would expect, as the action is at least twice as large (textually) as the corresponding C program.

The "code size" column in the table below is a simple line count of the generated assembly files. The actual number of instructions is smaller because of a little overhead, such as assembler directives, labels and so forth. All times are in seconds. The fourth column gives the run-time for an equivalent program written in C and compiled with the GNU C compiler (`gcc 2.4.3`). The last column states the run-time for the C program compiled with full optimization turned on.

Program	Code size	Run-time	C-runtime	C-opt
bubble	254	0.4	0.4	0.2
sieve	211	1.2	0.7	0.3
euclid	144	2.1	1.4	0.7
fib	93	0.8	0.7	0.5

The above table shows that the run-times for all four programs are within a factor 1.7 of code generated by a hand-written C compiler. If we let the C compiler do its best at optimizing the program, our code is still at most 4 times slower. The main reason why our code is so much slower than optimized C code is the lack of a global register allocator. Another reason is that HypoPL allows general nesting of procedures, something that C doesn't. This has an impact on performance, since a HypoPL compiler (in the absence of a corresponding analysis) cannot take the same shortcuts as a C compiler can when accessing variables.

Unrolling the `sieve` program a number of times, to obtain a source program ten times as large (539 lines) yield compile times (19 seconds) about ten times longer than for the small program (2 seconds), as one would expect, since the analyses are of linear complexity. Keeping the actual amount of computation constant, we get the same run-times for the small and large program. The code size scales linearly too, of course.

The figures show that code generated with the OASIS system is about two orders of magnitude faster than code generated with the CANTOR system

7.1 FUNIMP versus `scm`

Here we make a performance comparison between Scheme and the generated FUNIMP compiler. The example program is a recursive Fibonacci function.

The first column below shows the number of seconds it takes to compute the result in interpreted Scheme, the second column is for the Scheme program compiled to C and then to machine code by the Hobbit [23] compiler. The last column shows the run-time for the OASIS-compiled FUNIMP program. Again our results are within a factor two of a hand-written compiler.

scm	Compiled Scheme	FUNIMP
84.8	3.4	5.7

The main difference between the code we generate for `fib` and the code that Hobbit/C generates, is that our implementation of the conditional is less than optimal, due to the symmetric nature of the "or" action construct and our non-optimal implementation of the "check" construct. Our code actually computes the truth value, whereas C need only test the condition. Comparing against an optimized, equivalent hand-written C program, shows that our code is about 3 times slower.

7.2 Lee and Pleban

Comparing performance against Lee's system [12, 13] is difficult since it ran on much slower hardware than what is available today. Comparing the time that it took for that system to compile a HypoPL program to the time it takes for OASIS on more modern hardware would be unfair. Also, traditional compiler technology has improved since his comparisons with the traditional compilers of then, making a comparison based on those relative figures difficult.

Lee's system is based on High Level Semantics, where the static semantics is separated from the dynamic semantics, and he explicitly gives a so-called micro semantics tailored for the processor. Giving a new micro-semantics in his system would equal writing a new code generator part to the action compiler. If one were to write a micro-semantics targeting the SPARC processor, then it would be realistic to assume that code produced by a HypoPL compiler generated from the high level semantics system could be as good as the code produced by a HypoPL compiler generated by the OASIS system.

However, it seems that all optimizations in Lee's system happens at the micro-semantic level, hence a new micro semantics will be difficult to write.

7.3 Kelsey and Hudak

In [9] Kelsey and Hudak describe their compiler generator based on denotational semantics. In their system one writes denotational descriptions of languages in a variant of Scheme, and the system then performs several transformations on the resulting Scheme program, eventually arriving at assembly code for the Motorola 68020 processor.

They evaluate the performance of their system by comparing code produced by a generated Pascal compiler with code produced by the standard Pascal compiler on the Apollo workstation they used. The quality of the produced code is as good as what the standard Pascal compiler can generate, all performance figures lie within a factor 1.5 of the Pascal-generated code. Assuming that the standard Pascal compiler on the Apollo is comparable to a standard C compiler, one will have to say that the performance of their system is on a par with the OASIS system.

Apart from being based on denotational semantics, the main difference between their system and OASIS, is that their system *transforms* the meta-language (Scheme) until they reach something that is close enough to assembly to warrant a mechanical substitution from Scheme terms to assembly code. In OASIS the meta-language (Action Notation) is not transformed, but merely annotated by the various phases of analysis, and then ultimately an intricate code generator is invoked to generate assembly.

7.4 Bondorf and Palsberg

Using the same subset of Action Notation as in [20], Bondorf and Palsberg in [2] present another compiler generator based on Action Semantics. The compiler generator partially evaluates a Scheme representation of the action generated from the semantics. The generated compilers are compared with compilers generated by the CANTOR system. In comparison with the CANTOR system, run-times of the produced Scheme code are improved by at most a factor of 4, including a hypothetical factor 5 that the authors think they would achieve, had they used a Scheme compiler instead of an interpreter. Since the produced object code from the OASIS system is two orders of magnitude faster than what the CANTOR generated compilers produce, our system is clearly superior to this partial evaluation approach to compiler generation.

7.5 Actress

Comparing performance against the ACTRESS system is difficult since only one small test program with timings is given in [3]. For the system that they had implemented at the time, they write that the code they produce (C code) is 69 times slower than the equivalent Pascal program compiled with a standard compiler. This is certainly slower than our system. With certain mechanical optimizations, that were not implemented at the time of their article, they improve performance to within a factor two of the Pascal compiler. No timings are given for how long it takes to compile a program with the generated compilers.

The Actress approach to action compilation is closer to the approach by Kelsey and Hudak than it is to ours, in that they *transform* the action generated by the semantics to gain better run-times.

8 Concluding Remarks and Future Work

We have described several analyses based on Action Semantics, and have shown how they can be applied in a compiler generator capable of generating compilers that produce code comparable to code produced by handwritten compilers for similar languages.

Even though Action Semantics was developed from a semantic perspective without regard for compilability and run-time efficiency, we have demonstrated that efficient compilers can be automatically generated from Action Semantic descriptions.

However, there are various shortcomings of the current version of the system. The type system is probably too strict, and some sort of type inference like the system by Even and Schmidt [7] would be an advantage, if it could be modified in a way that would allow us to dispense with – most or all – run-time type checks. Moreover, many useful data-types are not easily expressible in the system, such as lists and records. Again the type system would have to be extended to cater for them.

There is still ample room for improvements of the code quality. The contents of memory cells should be tracked, and loop optimizations such as strength reduction could be applied. One possible way to obtain better code would be to transform the action tree to some other internal form better suited for low level optimizations, such as RTL (Register Transfer Language) [16, 15] or structured RTL [14].

A few experiments have been made with the specification and generation of compilers for object oriented languages. A small language with classes, objects, block structure and

inheritance has been specified and a compiler has been generated. We simply employ the ability of OASIS to handle higher order abstractions to model objects and methods. However, the current system is not capable of resolving non-virtual method-calls at compile time, as further analysis would be needed to accomplish that.

Work is currently going on to formally specify the various analyses described in this paper, and to prove their safety with respect to the operational semantics of Action Notation.

Further work could go in the direction of using the results of analyses on actions to say something about the source program. It would also be useful to analyze the semantic equations themselves (akin to the work by Doh and Schmidt [6]), this could perhaps cut down on the time it takes to compile actions to assembly. Generally it would be advantageous to analyze as much as possible in the compiler *generation* phase, as opposed the *compilation* phase. Typically one will apply the generated compilers more often that the compiler generator.

9 Acknowledgements

I want to thank Peter D. Mosses for encouragements and guidance in the preparation of the article. I also want to thank Ole L. Madsen for reading a draft of the paper. I must also thank the anonymous referees for useful and guiding feedback.

References

[1] A. Bondorf. Automatic Autoprojection of Higher Order Recursive Equations. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming (ESOP 90)*, volume 432 of LNCS, pages 70–87, Copenhagen, May 1990. Springer-Verlag.

[2] A. Bondorf and J. Palsberg. Compiling Actions by Partial Evaluation. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, 1993.

[3] D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an Action Semantics Directed Compiler Generator. In *Proceedings of the 1992 Workshop on Compiler Construction, Paderborn, Germany*, volume 641 of LNCS. Springer-Verlag, 1992.

[4] W. Clinger and J. R. (editors). Revised^4 Report on the Algorithmic Language Scheme. Technical report, MIT, 1991.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, January 1977.

[6] K.-G. Doh and D. A. Schmidt. Extraction of Strong Typing Laws from Action Semantics Definitions. In B. Krieg-Brückner, editor, *Proceedings of the 4'th European Symposium on Programming (ESOP 92)*, volume 582 of LNCS, pages 151–166, Rennes, February 1992. Springer-Verlag.

[7] S. Even and D. A. Schmidt. Type Inference for Action Semantics. In N. Jones, editor, *Proceedings of the 3rd European Symposium on Programming (ESOP 90)*, volume 432 of LNCS, pages 118–133, Copenhagen, May 1990. Springer-Verlag.

[8] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a Simple, Efficient Code Generator Generator. Technical report, AT&T Bell Labs, 1992.

[9] R. Kelsey and P. Hudak. Realistic Compilation by Program Transformation. In *Proceedings of the 16'th ACM Symposium on Principles of Programming Languages*, pages 281–292, January 1989.

[10] B. W. Kernighan and D. M. Richie. *The C Programming Language*. Prentice-Hall, 1978.

[11] P. Klint. A meta-environment for generating programming environments. In J. A. Bergstra and L. M. G. Feijs, editors, *Algebraic Methods II: Theory, Tools and Applications*, volume 490 of LNCS, pages 105–124, Mierlo, September 1989. Springer-Verlag.

[12] P. Lee and U. F. Pleban. A Realistic Compiler Generator on High-Level Semantics. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 284–295, 1987.

[13] P. Lee and U. F. Pleban. An Automatically Generated, Realistic Compiler for an Imperative Programming Language. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 222–232, June 1988.

[14] C. McConnel. *Tree-Based Code Optimization*. PhD thesis, University of Illinois Urbana Champaign, March 1992. Draft.

[15] C. McConnel and R. E. Johnson. Using SSA Form in a Code Optimizer. Technical report, UIUC, 1991.

[16] C. McConnel, J. D. Roberts, and C. B. Schoening. The RTL System. Technical report, UIUC, October 1990.

[17] P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992. Number 26 in the Cambridge Tracts in Theoretical Computer Science series.

[18] P. Ørbæk. Analysis and Optimization of Actions. M.Sc. dissertation, Computer Science Department, Aarhus University, Denmark, September 1993. URL: `ftp://ftp.daimi.aau.dk/pub/empl/poe/index.html`.

[19] J. Palsberg. A Provably Correct Compiler Generator. In B. Krieg-Brückner, editor, *Proceedings of the 4th European Symposium on Programming (ESOP 92)*, volume 582 of LNCS, pages 418–434, Rennes, February 1992. Springer-Verlag.

[20] J. Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Computer Science Department at Aarhus University, January 1992.

[21] B. Stroustrup and M. A. Ellis. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[22] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[23] T. Tammet. Hobbit: A Scheme to C compiler. Unpublished, (available by ftp from `cs.indiana.edu:/pub/scheme-repository`), 1993.

[24] L. Wall and R. L. Schwartz. *Programming Perl.* O'Reilly and Associates, 1991.

Towards Partial Evaluation of Actions (Abstract with Appendix)

Kyung-Goo Doh
The University of Aizu*

Abstract

Partial evaluation technique has been proven to be useful in many aspects in computing, e.g., program transformation, compiler generation, etc. In this paper, we present an action tranformation technique based on partial evaluation. A binding-time analysis transforms a program action into an equivalent two-level action, annotated either "static" or "dynamic". Then the two-level action is partially evaluated (reduced) to a residual action by blindly following annotation.

Appendix

Action transformation by partial evaluation is described in inference rule format. The syntax and semantics of action notation are given in Figures 1-4, and the syntax of some two-level action notation is shown in Figure 5. Partial evaluation is done in 2 stages: (1) binding-time analysis transforms an action into an quivalent two-level action by annotating the action with binding-time information – either static or dynamic (see Figures 6-9); (2) the two-level action is partially evaluated into a residual action (see Figures 10-13).

*Fukushima 965-80, Japan, kg-doh@u-aizu.ac.jp

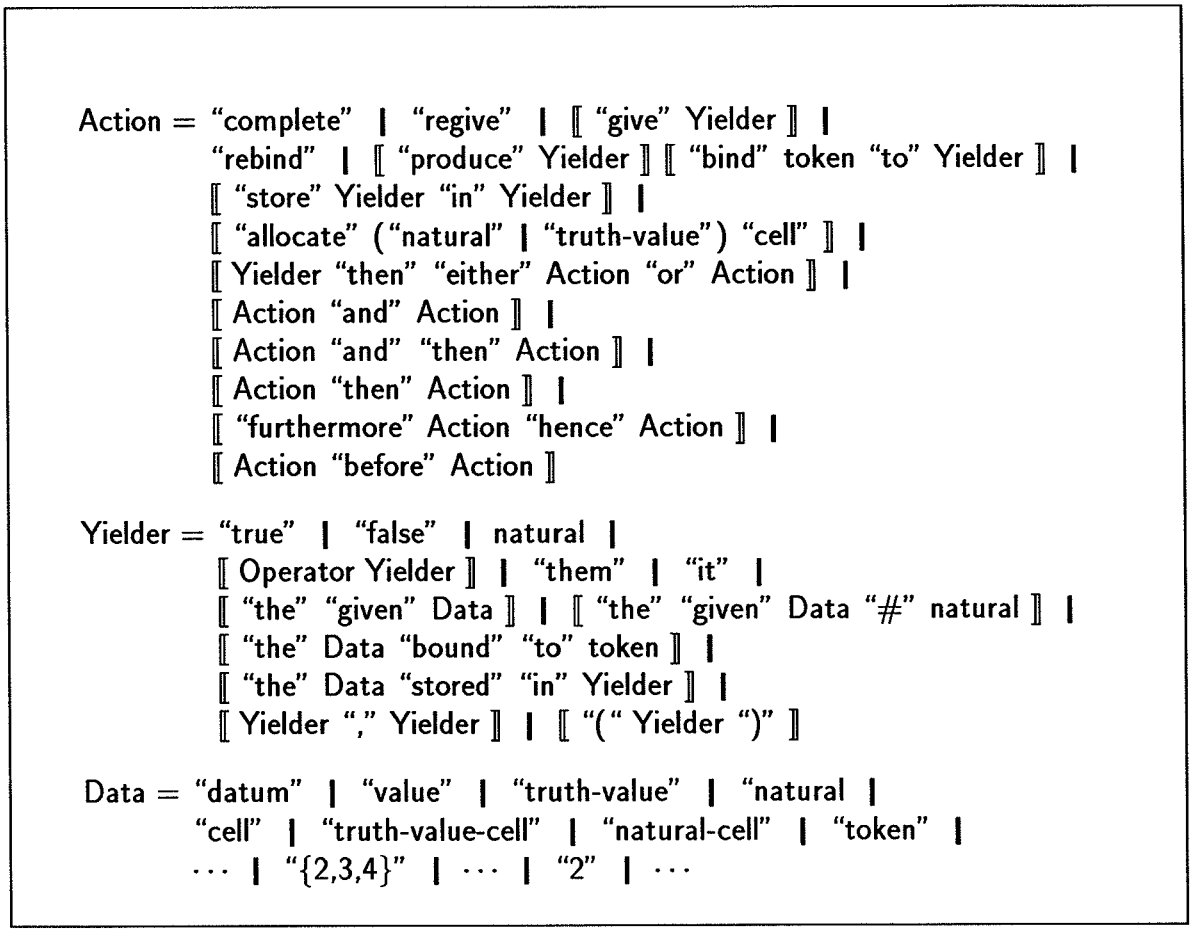


Figure 1: Syntax of Action Notation

$$\frac{}{() \vdash \textsf{true} \bullet \mathit{true}} \qquad \frac{}{() \vdash \textsf{false} \in \mathit{false}} \qquad \frac{}{() \vdash n : \textsf{natural} \in n}$$

$$\frac{\Gamma \vdash Y \in \tau \quad \tau \vdash O \in \tau'}{\Gamma \vdash [\![O : \textsf{Operator}\ Y : \textsf{Yielder}]\!] \in \tau'}$$

$$\frac{}{\tau \vdash \textsf{them} \in \tau} \qquad \frac{}{\tau \vdash \textsf{it} \in \tau}$$

$$\frac{\tau \leq D}{\tau \vdash [\![\textsf{the given}\ D : \textsf{Data}]\!] \in \tau}$$

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

$$\frac{\rho \leq \{t = \tau\} \qquad \tau \leq D}{\rho \vdash [\![\textsf{the}\ D : \textsf{Data bound to}\ t : \textsf{token}]\!] \in \tau}$$

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

$$\frac{\Gamma \vdash Y_1 \in \gamma_1 \qquad \Gamma \vdash Y_2 \in \gamma_2}{\Gamma \vdash [\![Y_1 : \textsf{Yielder}, Y_2 : \textsf{Yielder}]\!] \in (\gamma_1, \gamma_2)}$$

$$\frac{\Gamma \vdash Y \in \gamma}{\Gamma \vdash [\![(\, Y : \textsf{Yielder}\,)]\!] \in \gamma}$$

$$\frac{\Gamma_1 \leq \Gamma_2 \qquad \Gamma_2 \vdash Y \in \gamma_2 \qquad \gamma_2 \leq \gamma_1}{\Gamma_1 \vdash Y : \textsf{Yielder} \in \gamma_1} \qquad (\mathit{weakening\ rule})$$

Figure 2: Operational Semantics for Yielders

$$\frac{}{() \vdash \textsf{complete} \in ()} \qquad \frac{}{\tau \vdash \textsf{regive} \in \tau} \qquad \frac{\Gamma \vdash Y \in \tau}{\Gamma \vdash [\![\, \textsf{give } Y\textsf{:Yielder} \,]\!] \in \tau}$$

$$\frac{}{\rho \vdash \textsf{rebind} \in \rho} \qquad \frac{\Gamma \vdash Y \in \rho}{\Gamma \vdash [\![\, \textsf{produce } Y\textsf{:Yielder} \,]\!] \in \rho}$$

$$\frac{\Gamma \vdash Y \in \tau \quad \tau \leq \textsf{value}}{\Gamma \vdash [\![\, \textsf{bind } t\textsf{:token to } Y\textsf{:Yielder} \,]\!] \in \{\textsf{t} = \tau\}}$$

$$\frac{\Gamma, \sigma \vdash Y_1 \in \tau_1 \quad \tau_1 \leq \textsf{value} \quad \Gamma, \sigma \vdash Y_2 \in \tau_2 \quad \textit{compatible?}(\tau_1, \tau_2) \quad \textit{allocated?}(\tau_2, \sigma)}{\Gamma, \sigma \vdash [\![\, \textsf{store } Y_1\textsf{:Yielder in } Y_2\textsf{:Yielder} \,]\!] \in \textsf{overlay(map } \tau_2 \textsf{ to } \tau_1, \sigma)}$$

$$\frac{\tau = \textsf{new natural cell}(\sigma)}{\sigma \vdash [\![\, \textsf{allocate natural cell} \,]\!] \in \tau, \textsf{overlay(map } \tau \textsf{ to uninitialized}, \sigma)}$$

$$\frac{\tau = \textsf{new truth value cell}(\sigma)}{\sigma \vdash [\![\, \textsf{allocate truth value cell} \,]\!] \in \tau, \textsf{overlay(map } \tau \textsf{ to uninitialized}, \sigma)}$$

Figure 3: Operational Semantics for Actions (1)

$$\frac{\Gamma \vdash Y \in \mathsf{true} \quad \Gamma \vdash A_1 \in \gamma}{\Gamma \vdash [\![\, Y\!:\!\mathsf{Yielder\ then\ either}\ A_1\!:\!\mathsf{Action\ or}\ A_2\!:\!\mathsf{Action}\ ]\!] \in \gamma}$$

$$\frac{\Gamma \vdash Y \in \mathsf{false} \quad \Gamma \vdash A_2 \in \gamma}{\Gamma \vdash [\![\, Y\!:\!\mathsf{Yielder\ then\ either}\ A_1\!:\!\mathsf{Action\ or}\ A_2\!:\!\mathsf{Action}\ ]\!] \in \gamma}$$

$$\frac{\Gamma \vdash A_1 \in \tau_1 \quad \Gamma \vdash A_2 \in \tau_2}{\Gamma \vdash [\![\, A_1\!:\!\mathsf{Action\ and}\ A_2\!:\!\mathsf{Action}\ ]\!] \in (\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash A_1 \in \gamma_1, \sigma_1 \quad \Gamma, \sigma_1 \vdash A_2 \in \gamma_2, \sigma_2}{\Gamma \vdash [\![\, A_1\!:\!\mathsf{Action\ and\ then}\ A_2\!:\!\mathsf{Action}\ ]\!] \in \gamma_2, \sigma_2}$$

$$\frac{\Gamma \vdash A_1 \in \tau \quad \Gamma, \tau \vdash A_2 \in \gamma}{\Gamma \vdash [\![\, A_1\!:\!\mathsf{Action\ then}\ A_2\!:\!\mathsf{Action}\ ]\!] \in \gamma}$$

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

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

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

Figure 4: Operational Semantics for Actions (2)

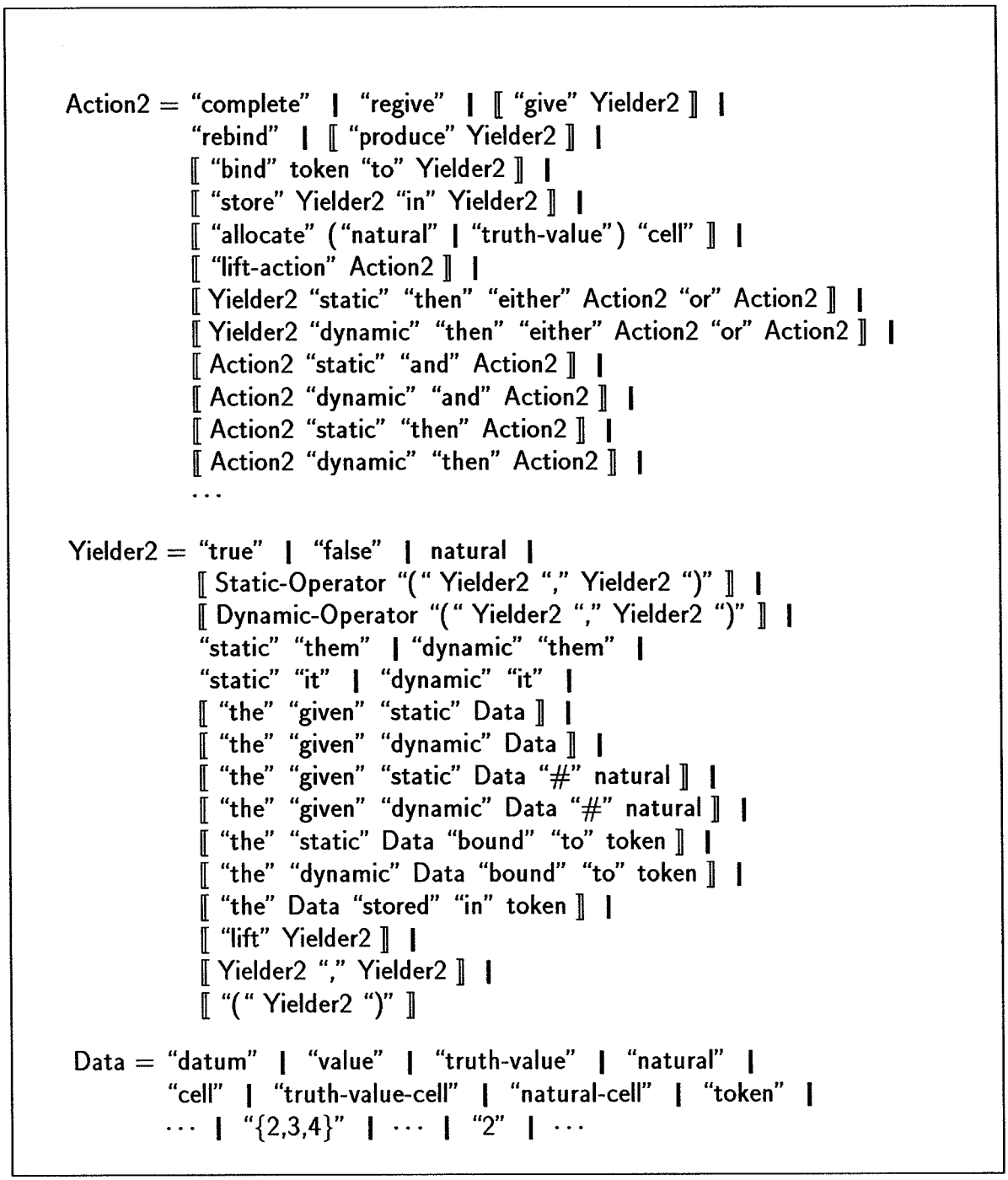


Figure 5: Syntax of 2-Level Action Notation

$$\frac{}{() \vdash \mathsf{true} \in \mathsf{true}, \mathsf{known}}$$

$$\frac{}{() \vdash \mathsf{false} \in \mathsf{false}, \mathsf{known}}$$

$$\frac{}{() \vdash n : \mathsf{natural} \in \mathsf{natural\ to\ string}(n), \mathsf{known}}$$

$$\frac{\Gamma \vdash Y \in \tau_1, \mathsf{known} \quad \Gamma \vdash Y_2 \in \tau_2, \mathsf{known}}{\Gamma \vdash [\![\, \mathsf{sum}\ (Y_1, Y_2)\ ]\!] \in [\![\, \mathsf{static\ sum}(\tau_1, \tau_2)\ ]\!], \mathsf{known}}$$

$$\frac{\Gamma \vdash Y_1 \in \tau_1, \mathsf{known} \quad \Gamma \vdash Y_2 \in \tau_2, \mathsf{unknown}}{\Gamma \vdash [\![\, \mathsf{sum}\ (Y_1, Y_2)\ ]\!] \in [\![\, \mathsf{dynamic\ sum}([\![\mathsf{lift}\ \tau_1]\!], \tau_2)\ ]\!], \mathsf{unknown}}$$

$$\frac{\Gamma \vdash Y_1 \in \tau_1, \mathsf{unknown} \quad \Gamma \vdash Y_2 \in \tau_2, \mathsf{known}}{\Gamma \vdash [\![\, \mathsf{sum}\ (Y_1, Y_2)\ ]\!] \in [\![\, \mathsf{dynamic\ sum}(\tau_1, [\![\mathsf{lift}\ \tau_2]\!])\ ]\!], \mathsf{unknown}}$$

$$\frac{\Gamma \vdash Y_1 \in \tau_1, \mathsf{unknown} \quad \Gamma \vdash Y_2 \in \tau_2, \mathsf{unknown}}{\Gamma \vdash [\![\, \mathsf{sum}\ (Y_1, Y_2)\ ]\!] \in [\![\, \mathsf{dynamic\ sum}(\tau_1, \tau_2)\ ]\!], \mathsf{unknown}}$$

$$\frac{\sqcup\tau = \mathsf{known}}{\tau \vdash \mathsf{them} \in [\![\mathsf{static\ them}]\!], \tau} \qquad \frac{\sqcup\tau = \mathsf{unknown}}{\tau \vdash \mathsf{them} \in [\![\mathsf{dynamic\ them}]\!], \tau}$$

$$\frac{\tau = \mathsf{known}}{\tau \vdash \mathsf{it} \in [\![\mathsf{static\ it}]\!], \mathsf{known}} \qquad \frac{\tau = \mathsf{unknown}}{\tau \vdash \mathsf{it} \in [\![\mathsf{dynamic\ it}]\!], \mathsf{unknown}}$$

Figure 6: Binding-Time Analysis for Yielders (1)

$$\frac{\sqcup\tau = \textsf{known}}{\tau \vdash [\![\ \textsf{the given } D\textsf{: Data}\ ]\!] \in [\![\ \textsf{the given static } D\ ]\!], \tau}$$

$$\frac{\sqcup\tau = \textsf{unknown}}{\tau \vdash [\![\ \textsf{the given } D\textsf{: Data}\ ]\!] \bullet [\![\ \textsf{the given dynamic } D\ ]\!], \tau}$$

$$\frac{\textsf{component\#}\ n\ \tau = \textsf{known}}{\tau \vdash [\![\ \textsf{the given } D\ \#\ n\ ]\!] \in [\![\ \textsf{the given static } D\#n\ ]\!], \textsf{known}}$$

$$\frac{\textsf{component\#}\ n\ \tau = \textsf{unknown}}{\tau \vdash [\![\ \textsf{the given } D\ \#\ n\ ]\!] \in [\![\ \textsf{the given dynamic } D\#n\ ]\!], \textsf{unknown}}$$

$$\frac{\sqcup(\rho\ \textsf{at}\ t) = \textsf{known}}{\rho \vdash [\![\ \textsf{the } D\ \textsf{bound to}\ t\ ]\!] \in [\![\ \textsf{the static } D\ \textsf{bound to}\ t\ ]\!], \rho\ \textsf{at}\ t}$$

$$\frac{\sqcup(\rho\ \textsf{at}\ t) = \textsf{unknown}}{\rho \vdash [\![\ \textsf{the } D\ \textsf{bound to}\ t\ ]\!] \in [\![\ \textsf{the dynamic } D\ \textsf{bound to}\ t\ ]\!], \rho\ \textsf{at}\ t}$$

$$\frac{\Gamma \vdash Y \in \gamma, \beta}{\Gamma \vdash [\![\ \textsf{the } D\ \textsf{stored in}\ Y\ ]\!] \in [\![\ \textsf{the } D\ \textsf{stored in}\ \gamma\ ]\!], \textsf{unknown}}$$

$$\frac{\Gamma \vdash Y_1 \in \gamma_1, \beta_1 \qquad \Gamma \vdash Y_2 \in \gamma_2, \beta_2}{\Gamma \vdash [\![\ Y_1\ ,\ Y_2\ ]\!] \in [\![\ \gamma_1\ ,\ \gamma_2\ ]\!], \sqcup(\beta_1, \beta_2)}$$

$$\frac{\Gamma \vdash Y \in \gamma, \beta}{\Gamma \vdash [\![\ (\ Y\ )\ ]\!] \in \gamma, \beta}$$

Figure 7: Binding-Time Analysis for Yielders (2)

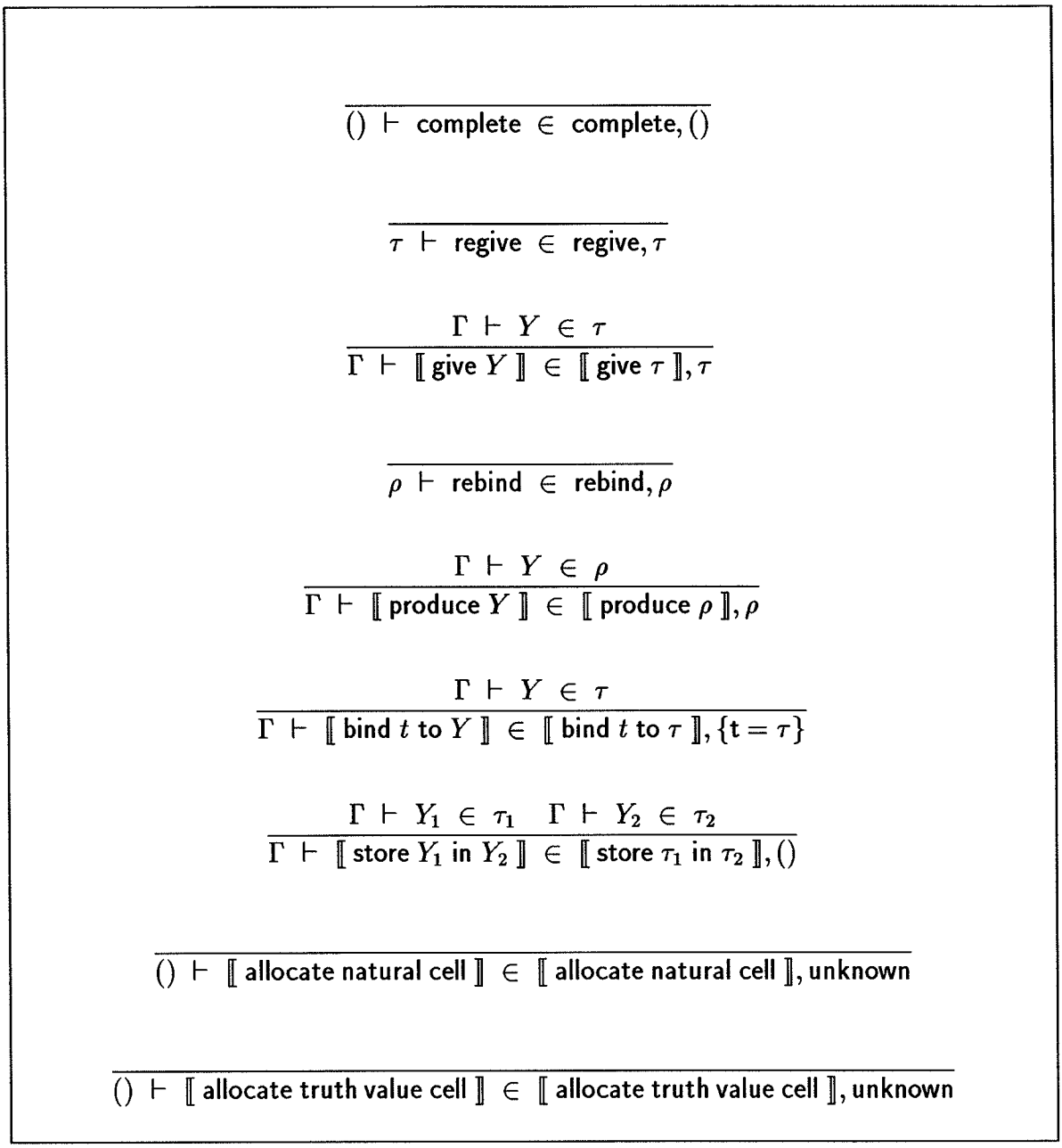


Figure 8: Binding-Time Analysis for Some Actions (1)

$$\frac{\Gamma \vdash Y \in \tau, \textsf{known} \quad \Gamma \vdash A_1 \in \gamma_1, \beta_1 \quad \Gamma \vdash A_2 \in \gamma_2, \beta_2}{\Gamma \vdash [\![Y \textsf{ then either } A_1 \textsf{ or } A_2]\!] \in [\![\tau \textsf{ static then either } \gamma_1 \textsf{ or } \gamma_2]\!], \beta_1 \sqcup \beta_2}$$

$$\frac{\Gamma \vdash Y \in \tau, \textsf{unknown} \quad \Gamma \vdash A_1 \in \gamma_1, \beta_1 \quad \Gamma \vdash A_2 \in \gamma_2, \beta_2}{\Gamma \vdash [\![Y \textsf{ then either } A_1 \textsf{ or } A_2]\!] \in [\![\tau \textsf{ dynamic then either } \gamma_1 \textsf{ or } \gamma_2]\!], \beta_1 \sqcup \beta_2}$$

$$\frac{\Gamma \vdash A_1 \in \tau_1, \beta_1 \quad \sqcup \beta_1 = \textsf{known} \quad \Gamma \vdash A_2 \in \tau_2, \beta_2 \quad \sqcup \beta_1 = \textsf{known}}{\Gamma \vdash [\![A_1 \textsf{ and } A_2]\!] \in [\![\tau_1 \textsf{ static and } \tau_2]\!], (\beta_1, \beta_2)}$$

$$\frac{\Gamma \vdash A_1 \in \tau_1, \beta_1 \quad \sqcup \beta_1 = \textsf{known} \quad \Gamma \vdash A_2 \in \tau_2, \beta_2 \quad \sqcup \beta_1 = \textsf{unknown}}{\Gamma \vdash [\![A_1 \textsf{ and } A_2]\!] \in [\![[\![\textsf{lift action } \tau_1]\!] \textsf{ dynamic and } \tau_2]\!], (\beta_1, \beta_2)}$$

$$\frac{\Gamma \vdash A_1 \in \tau_1, \beta_1 \quad \sqcup \beta_1 = \textsf{unknown} \quad \Gamma \vdash A_2 \in \tau_2, \beta_2 \quad \sqcup \beta_1 = \textsf{known}}{\Gamma \vdash [\![A_1 \textsf{ and } A_2]\!] \in [\![\tau_1 \textsf{ dynamic and } [\![\textsf{lift action } \tau_2]\!]]\!], (\beta_1, \beta_2)}$$

$$\frac{\Gamma \vdash A_1 \in \tau_1, \beta_1 \quad \sqcup \beta_1 = \textsf{unknown} \quad \Gamma \vdash A_2 \in \tau_2, \beta_2 \quad \sqcup \beta_1 = \textsf{unknown}}{\Gamma \vdash [\![A_1 \textsf{ and } A_2]\!] \in [\![\tau_1 \textsf{ dynamic and } \tau_2]\!], (\beta_1, \beta_2)}$$

$$\frac{\Gamma \vdash A_1 \in \tau_1, \beta_1 \quad \Gamma, \beta_1 \vdash A_2 \in \gamma_2, \beta_2 \quad \sqcup \beta_1 = \textsf{known}}{\Gamma \vdash [\![A_1 \textsf{ then } A_2]\!] \in [\![\tau_1 \textsf{ static then } \gamma_2]\!], \beta_2}$$

$$\frac{\Gamma \vdash A_1 \in \tau_1, \beta_1 \quad \Gamma, \beta_1 \vdash A_2 \in \gamma_2, \beta_2 \quad \sqcup \beta_1 = \textsf{unknown} \quad \mathit{static?}(\beta_2)}{\Gamma \vdash [\![A_1 \textsf{ then } A_2]\!] \in [\![\gamma_2]\!], \beta_2}$$

$$\frac{\Gamma \vdash A_1 \in \tau_1, \beta_1 \quad \Gamma, \beta_1 \vdash A_2 \in \gamma_2, \beta_2 \quad \sqcup \beta_1 = \textsf{unknown} \quad \mathit{dynamic?}(\beta_2)}{\Gamma \vdash [\![A_1 \textsf{ then } A_2]\!] \in [\![\tau_1 \textsf{ dynamic then } \gamma_2]\!], \beta_2}$$

Figure 9: Binding-Time Analysis for Some Actions (2)

$$\frac{}{() \vdash \mathsf{true} \in true}$$

$$\frac{}{() \vdash \mathsf{false} \in false}$$

$$\frac{}{() \vdash n \in n}$$

$$\frac{\Gamma \vdash Y_1 \in \tau_1 \quad \Gamma \vdash Y_2 \in \tau_2}{\Gamma \vdash [\![\, \mathsf{static\ sum}(Y_1, Y_2) \,]\!] \in \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash Y_1 \in \tau_1 \quad \Gamma \vdash Y_2 \in \tau_2}{\Gamma \vdash [\![\, \mathsf{dynamic\ sum}(Y_1, Y_2) \,]\!] \in [\![\mathsf{sum}(\tau_1, \tau_2) \,]\!]}$$

$$\frac{}{\tau \vdash [\![\, \mathsf{static\ them} \,]\!] \in \tau}$$

$$\frac{}{\tau \vdash [\![\, \mathsf{dynamic\ them} \,]\!] \in \mathsf{them}}$$

$$\frac{}{\tau \vdash [\![\, \mathsf{static\ it} \,]\!] \in \tau} \qquad \frac{}{\tau \vdash [\![\, \mathsf{dynamic\ it} \,]\!] \in \mathsf{it}}$$

Figure 10: Partial Evaluation for Yielders (1)

$$\frac{}{\tau \vdash [\![\text{the given static } D]\!] \in \tau}$$

$$\frac{}{\tau \vdash [\![\text{the given dynamic } D]\!] \in [\![\text{the given } D]\!]}$$

$$\frac{}{\tau \vdash [\![\text{the given static } D\#n]\!] \in \text{component\# } n\ \tau}$$

$$\frac{}{\tau \vdash [\![\text{the given dynamic } D\#n]\!] \in [\![\text{the given } D\#n]\!]}$$

$$\frac{}{\rho \vdash [\![\text{the static } D \text{ bound to } t]\!] \in \rho \text{ at } t}$$

$$\frac{}{\rho \vdash [\![\text{the dynamic } D \text{ bound to } t]\!] \in [\![\text{the } D \text{ bound to } t]\!]}$$

$$\frac{}{\Gamma \vdash [\![\text{the } D \text{ stored in } Y]\!] \in [\![\text{the } D \text{ stored in } Y]\!]}$$

$$\frac{\Gamma \vdash Y \in \gamma}{\Gamma \vdash [\![\text{lift } Y]\!] \in \text{build data}(\gamma)}$$

$$\frac{\Gamma \vdash Y_1 \in \gamma_1 \qquad \Gamma \vdash Y_2 \in \gamma_2}{\Gamma \vdash [\![Y_1 , Y_2]\!] \in (\gamma_1, \gamma_2)}$$

$$\frac{\Gamma \vdash Y \in \gamma}{\Gamma \vdash [\![(Y)]\!] \in \gamma}$$

Figure 11: Partial Evaluation for Yielders (2)

$$\frac{}{() \vdash \mathsf{complete} \in ()}$$

$$\frac{}{\tau \vdash \mathsf{regive} \in \tau}$$

$$\frac{\Gamma \vdash Y \in \tau}{\Gamma \vdash [\![\, \mathsf{give}\ Y \,]\!] \in \tau}$$

$$\frac{}{\rho \vdash \mathsf{rebind} \in \rho}$$

$$\frac{\Gamma \vdash Y \in \rho}{\Gamma \vdash [\![\, \mathsf{produce}\ Y \,]\!] \in \rho}$$

$$\frac{\Gamma \vdash Y \in \tau}{\Gamma \vdash [\![\, \mathsf{bind}\ t\ \mathsf{to}\ Y \,]\!] \in \{\mathrm{t} = \tau\}}$$

$$\frac{}{\Gamma, \sigma \vdash [\![\, \mathsf{store}\ Y_1\ \mathsf{in}\ Y_2 \,]\!] \in [\![\, \mathsf{store}\ Y_1\ \mathsf{in}\ Y_2 \,]\!]}$$

$$\frac{}{\sigma \vdash [\![\, \mathsf{allocate\ natural\ cell} \,]\!] \in [\![\, \mathsf{allocate\ natural\ cell} \,]\!]}$$

$$\frac{}{\sigma \vdash [\![\, \mathsf{allocate\ truth\ value\ cell} \,]\!] \in [\![\, \mathsf{allocate\ truth\ value\ cell} \,]\!]}$$

$$\frac{\Gamma \vdash A \in \gamma}{\Gamma \vdash [\![\, \mathsf{lift\ action}\ A \,]\!] \in \mathsf{build\ action}(\gamma)}$$

Figure 12: Partial Evaluation for Actions (1)

$$\frac{\Gamma \vdash Y \in \mathsf{true} \quad \Gamma \vdash A_1 \in \gamma}{\Gamma \vdash [\![\, Y \text{ static then either } A_1 \text{ or } A_2 \,]\!] \in \gamma}$$

$$\frac{\Gamma \vdash Y \in \mathsf{false} \quad \Gamma \vdash A_2 \in \gamma}{\Gamma \vdash [\![\, Y \text{ static then either } A_1 \text{ or } A_2 \,]\!] \in \gamma}$$

$$\frac{\Gamma \vdash Y \in \tau_1 \quad \Gamma \vdash A_1 \in \gamma_1 \quad \Gamma \vdash A_2 \in \gamma_2}{\Gamma \vdash [\![\, Y \text{ dynamic then either } A_1 \text{ or } A_2 \,]\!] \in [\![\, \tau_1 \text{ then either } \gamma_1 \text{ or } \gamma_2 \,]\!]}$$

$$\frac{\Gamma \vdash A_1 \in \tau_1 \quad \Gamma \vdash A_2 \in \tau_2}{\Gamma \vdash [\![\, A_1 \text{ static and } A_2 \,]\!] \in (\tau_1, \tau_2)}$$

$$\frac{\Gamma \vdash A_1 \in \gamma_1 \quad \Gamma \vdash A_2 \in \gamma_2}{\Gamma \vdash [\![\, A_1 \text{ dynamic and } A_2 \,]\!] \in [\![\, \gamma_1 \text{ and } \gamma_2 \,]\!]}$$

$$\frac{\Gamma \vdash A_1 \in \tau \quad \Gamma, \tau \vdash A_2 \in \gamma}{\Gamma \vdash [\![\, A_1 \text{ static then } A_2 \,]\!] \in \gamma}$$

$$\frac{\Gamma \vdash A_1 \in \tau \quad \Gamma, \tau \vdash A_2 \in \gamma}{\Gamma \vdash [\![\, A_1 \text{ dynamic then } A_2 \,]\!] \in [\![\, \tau \text{ then } \gamma \,]\!]}$$

Figure 13: Partial Evaluation for Actions (2)

Using ASF+SDF to Interpret and Transform Actions

David A. Watt *

Abstract

ASF+SDF is a very natural tool for prototyping action notation. We can specify the syntax of action notation in the usual way. SDF generates a backtracking parser that discovers ill-formed and ambiguous action terms. We can specify the operational semantics of action notation by rewrite rules that define the effects of performing actions, evaluating yielders, and so on. ASF's term-rewriting engine does the rest. We can go further and add equations that encode the laws of action notation; ASF interprets these as rewrite rules. Thus action terms (whether input by the user, or generated by translation from a source language) are automatically simplified.

This report summarises progress towards building a robust action notation prototyper. It discusses the problems in making it acceptably efficient, the possibility of incorporating the more sophisticated transformations developed in the ACTRESS project, and the prospects of using the prototyper (in conjunction with van Deursen and Mosses' ASD Tools) to test realistic action-semantic specifications.

1 Introduction

Semantic prototyping is a valuable tool for designers and specifiers of programming languages. The idea of prototyping is to take a formal specification of a programming language, and use it to build a quick (but poor-quality) implementation.

Prototyping is an key element of what I call *language engineering* (and what Uwe Pleban has called the *language designer's workbench*). The new language design is first embodied in a formal specification. This exercise is itself an effective way to expose irregularities in the language design. Then the formal specification is used to build a prototype, which is used to run small test programs in the new language. This gives feedback to the language designer, allows the specification itself to be tested, and even allows ordinary programmers to gain some initial experience with the new language. The design, specification, and prototyping stages are then iterated, until the design stabilises. Finally, one or more true compilers are constructed. In this stage, the formal specification is available at least as a guide to the compiler writers, but ideally they could use a semantics-directed compiler generator. It makes sense to defer the most expensive stage, compiler construction, until after the language design has stabilised.

Note the central role of the formal specification in language engineering. The benefits of prototyping and compiler generation give the language designer a strong incentive to write the formal specification – and at an early stage, not as an afterthought.

---

* Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: daw@dcs.gla.ac.uk.

Various approaches to semantic prototyping have been tried, usually based on expressing denotational or natural-semantic specifications in a programming language such as Algol 68 (Pagan 1979), ML (Watt 1986), or Prolog (Despeyroux 1988). I have explored the possibility of prototyping action semantics using ML (Watt 1991). An action can be modelled by an ML function that maps incoming information to a result; an action combinator can be modelled by an ML function that maps actions to actions; and so on. Careful choice of names and priorities[1] for the ML functions allows action terms to be expressed quite recognisably:

```
furthermore
( allocateacell  then
  ( store zero (*in*) it  andthen
    bind "x" (*to*) it ) )
hence
  give (sum (integer storedin (cell boundto "x"), one))
```

I have applied this prototyping method to toy languages, but never to a real language. The effort of manually transcribing a semantic specification from standard action notation to the ML syntax is just too great.[2] Worse, all the data operations (which would normally be specified algebraically) have to be manually implemented in ML.

Recently, Peter Mosses introduced me to ASF+SDF[3] (Klint 1993a, 1993b). This is an extremely powerful and flexible system for processing language specifications. The specified language could be as simple as Boolean expressions or as complex as a real programming language. The specifier introduces sorts, introduces (mixfix) operations over these sorts, and defines these operations by (conditional) equations.

Specifying syntax: In the specification of a programming language's abstract syntax, the sorts are phrase sorts (nonterminals), and the operations are abstract-syntax-tree constructor operations. Appropriate choice of mixfix constructor operations allows the abstract syntax to resemble concrete syntax. Indeed, SDF automatically generates parsers from such syntactic specifications. There are facilities for specifying priorities, and left- or right-associativity.

Specifying semantics: In the specification of a simple language such as Boolean expressions, equations can be used to specify its semantics in the algebraic style. In the denotational specification of a programming language, the equational notation can be used to express semantic equations. In the operational specification of a programming language, conditional equations can be used to express inference rules. In any case, ASF interprets the equations as left-to-right rewrite rules, and a term-rewriting engine allows the specification to be executed.

Mosses expressed the action semantics of a toy programming language, Pico, as an ASF+SDF specification. He specified the syntax of Pico and the syntax of (a subset of) action notation. He specified the semantics of Pico by means of operations corresponding to semantic functions, defined by means of equations corresponding to semantic equations.

---

[1] ML functions are prefix by default. However, a binary function can be declared to be infix, and assigned a priority number. Mixfix functions cannot be declared.

[2] This translation could be automated. My pilot version of the action notation compiler (predating the ACTRESS project) had an option to translate action notation to the ML syntax.

[3] ASF+SDF = Algebraic Specification Formalism + Syntactic Definition Formalism.

Using this specification he could enter Pico programs and translate them to action notation. The only missing feature was a means of executing the action notation!

This prompted me to undertake, as an ASF+SDF familiarisation exercise, development of an action interpreter. I also perceived that ASF+SDF would be ideal for expressing useful algebraic properties of action notation, and I speculated that it would be useful for experimenting with more complicated action transformations. I have largely achieved the first two of these objectives, and have made a start on the third. This report gives an informal account of progress to date. It can be read, either as a detailed discussion of prototyping of action notation, or as a case study of a non-trivial application of ASF+SDF.

2 Action Syntax

My first step was to specify the syntax of actions (and yielders and data terms) in SDF notation. This was quite straightforward – see Box 1. Using this specification I could use ASF+SDF to enter and edit action notation terms. Because operations may be mixfix, I was able to mimic standard action notation very closely. However, only parentheses are available for grouping; the use of vertical lines and indentation for this purpose is not supported. The action shown in Section 1 would be expressed as follows:

```
furthermore
( allocate a cell then
  ( store 0 in it and then
    bind 'x' to it ) )
hence
  give sum (the integer stored in the cell bound to 'x' ), 1)
```

In certain details, I designed my action notation grammar in such a way as to facilitate my later specifications of action interpretation and transformation. In particular, I introduced the sorts SimpleAction, PrefixCombinator, and InfixCombinator for these reasons.

3 Action Interpretation

Information flows from action to action in three facets: transient data (functional facet), bindings (declarative facet), and storage (imperative facet). When an action is performed, its outcome may be to complete, escape, fail, or diverge. Also, the action may or may not commit.

I introduced the sort Info to represent the information flowing into an action, and the sort Result to represent the outcome, commitment, and information flowing out of an action – see Box 2.

No transient data or bindings flow out of an action that fails, so the fields *d* and *b* in the Result term (failed, *com*, *d*, *b*, *s*) are actually redundant. However, I found it simplest to represent failed results uniformly with completed results, on the understanding that in a failed result the fields *d* and *b* will be empty. A similar point applies to escaped results, in which the *b* field will be empty. There is no need to represent the result of a divergent action: performing such an action will be represented by an infinite rewriting.

I introduced the operation "performed" to represent the effect of performing an action with incoming information. This operation will map an action and an Info term to a Result term.

Similarly, I introduced the operation "evaluated" to represent the effect of evaluating a yielder, this time yielding a Data term. (Data itself has a fairly conventional algebraic specification, not shown here.)

Now it is straightforward to specify how simple actions such as "complete", "fail", "give", and "bind" are performed – see Box 2. The performance of an "unfolding" action is specified by means of the auxiliary operation "unfolded (*A*, *A'*)", which substitutes *A'* for each (free) occurrence of "unfold" in *A*.

sorts Action SimpleAction Yielder PrefixCombinator InfixCombinator ...

context-free syntax

SimpleAction	→	Action	
complete	→	SimpleAction	
fail	→	SimpleAction	
give Yielder	→	SimpleAction	
bind Token to Yielder	→	SimpleAction	
store Yielder in Yielder	→	SimpleAction	
...			
unfold	→	Action	
unfolding Action	→	Action	
PrefixCombinator Action	→	Action	
Action InfixCombinator Action	→	Action	{ **left** }
"(" Action ")"	→	Action	{ **bracket** }
furthermore	→	PrefixCombinator	
or	→	InfixCombinator	
and	→	InfixCombinator	
and then	→	InfixCombinator	
hence	→	InfixCombinator	
moreover	→	InfixCombinator	
...			
Data	→	Yielder	
it	→	Yielder	
given Data	→	Yielder	
the Data bound to Token	→	Yielder	
the Data stored in Yielder	→	Yielder	
...			
IdentityOp Yielder	→	Yielder	
PrefixOp Yielder	→	Yielder	
Yielder InfixOp Yielder	→	Yielder	{ **left** }
"(" Yielder ")"	→	Yielder	{ **bracket** }
a	→	IdentityOp	
sum	→	PrefixOp	
is	→	InfixOp	
","	→	InfixOp	
...			

Box 1 Syntax of action notation.

sorts Info Outcome Commitment Result

context-free syntax

	Data "," Bindings "," Storage	→	Info
	completed	→	Outcome
	escaped	→	Outcome
	failed	→	Outcome
	committed	→	Commitment
	uncommitted	→	Commitment
	max "(" Commitment "," Commitment ")"	→	Commitment
	"(" Outcome "," Commitment "," Info ")"	→	Result
	performed Action "(" Info ")"	→	Result
	evaluated Yielder "(" Info ")"	→	Data
	unfolded "(" Action "," Action ")"	→	Action

equations

[complete] performed complete (d, b, s) = (completed, uncommitted, (), {}, s)

[fail] performed fail (d, b, s) = (failed, uncommitted, (), {}, s)

[give] evaluated $Y(d, b, s) = d'$, $d' \neq$ nothing

performed (give Y) (d, b, s) = (completed, uncommitted, d', {}, s)

[bind] evaluated $Y(d, b, s) = d'$, the bindable yielded by $d' \neq$ nothing

performed (bind k to Y) (d, b, s) = (completed, uncommitted, (), $\{k \mapsto d'\}$, s)

[store] evaluated $Y_1(d, b, s) = d'$, the storable yielded by $d' \neq$ nothing,
evaluated $Y_2(d, b, s) = c$, the cell yielded by $c \neq$ nothing,
status of c in $s \neq$ unreserved

performed (store Y_1 in Y_2) (d, b, s) = (completed, committed, (), {}, $s[c \mapsto d']$)

[unfolding] performed (unfolding A) (d, b, s) =
performed unfolded (A, unfolding A) (d, b, s)

... ...

[unfolded-1] unfolded (SA, A') = SA %% $SA \rightarrow$ SimpleAction

[unfolded-2] unfolded (unfold, A') = A'

[unfolded-3] unfolded (unfolding A, A') = unfolding A

[unfolded-4] unfolded ($pc\ A$, A') = pc unfolded (A, A') %% $pc \rightarrow$ PrefixCombinator

[unfolded-5] unfolded ($A_1\ ic\ A_2$, A') = %% $ic \rightarrow$ InfixCombinator
unfolded (A_1, A') ic unfolded (A_2, A')

Box 2 Interpretation of simple actions.

The action notation subset considered here has a natural semantics (Moura 1993). The judgment for performing an action is:

$$(inf) \vdash A \Rightarrow res \tag{1}$$

meaning that *res* is the result of performing action A when information *inf* flows into it. The inference rules for the "and then" infix combinator are reproduced here[4]:

$$\frac{\begin{array}{c}(d, b, s) \vdash A_1 \Rightarrow (\text{completed}, com_1, d_1, b_1, s_1) \\ (d, b, s_1) \vdash A_2 \Rightarrow (\text{completed}, com_2, d_2, b_2, s_2) \\ b' = b_1 \oplus b_2 \qquad b' \neq \text{clash}\end{array}}{(d, b, s) \vdash A_1 \text{ and then } A_2 \Rightarrow (\text{completed}, \max(com_1, com_2), (d_1, d_2), b', s_2)} \tag{2}$$

$$\frac{\begin{array}{c}(d, b, s) \vdash A_1 \Rightarrow (\text{completed}, com_1, d_1, b_1, s_1) \\ (d, b, s_1) \vdash A_2 \Rightarrow (\text{completed}, com_2, d_2, b_2, s_2) \\ b' = b_1 \oplus b_2 \qquad b' = \text{clash}\end{array}}{(d, b, s) \vdash A_1 \text{ and then } A_2 \Rightarrow (\text{failed}, \max(com_1, com_2), (), \{\}, s_2)} \tag{3}$$

$$\frac{\begin{array}{c}(d, b, s) \vdash A_1 \Rightarrow (\text{completed}, com_1, d_1, b_1, s_1) \\ (d, b, s_1) \vdash A_2 \Rightarrow (out_2, com_2, d_2, b_2, s_2) \qquad out_2 \neq \text{completed}\end{array}}{(d, b, s) \vdash A_1 \text{ and then } A_2 \Rightarrow (out_2, \max(com_1, com_2), d_2, b_2, s_2)} \tag{4}$$

$$\frac{(d, b, s) \vdash A_1 \Rightarrow (out_1, com_1, d_1, b_1, s_1) \qquad out_1 \neq \text{completed}}{(d, b, s) \vdash A_1 \text{ and then } A_2 \Rightarrow (out_1, com_1, d_1, b_1, s_1)} \tag{5}$$

The intended purpose of ASF+SDF is for writing and executing specifications. So the most natural method (indeed!) for prototyping an action interpreter is simply to transcribe inference rules (2)–(5) into ASF conditional equations – see Box 3. Judgment (1) is expressed as "performed *A* (*inf*) = *res*".

This method works, but performance is unacceptably slow. I estimate that the time required to perform an action is $O(n^c)$, where n is the action size (after any unfolding) and c is about 4. The reason for this is the straightforward but naive way in which term rewriting is currently implemented in ASF. Given a term of the form "performed (A_1 and then A_2) (*d*, *b*, *s*)", the term-rewriting engine will try each of equations [and-then-1] to [and-then-4] in turn.[5] If any premise in an equation fails, that equation is abandoned and another equation is tried instead. Notice that equations [and-then-1] through [and-then-4] all contain a premise that involves reducing the term "performed A_1 (*d*, *b*, *s*)", and three of them likewise contain a premise that involves reducing the term "performed A_2 (*d*, *b*, s_1)". Thus the sub-actions A_1

[4] I have slightly modified Moura's inference rules to conform to the terminology used in this report, and I have added commitments.

[5] The order in which the equations are tried is implementation-dependent.

and A_2 are performed several times, until the outcome of "A_1 and then A_2" is determined. Similar effects happen with other infix combinators.[6]

I therefore tried an alternative method, based on factoring out the common computations – see Box 4. The term "performed (A_1 and then A_2) (*inf*)" is first rewritten to "continued res_1 and then A_2 (*inf*)", where res_1 is the result of performing A_1. If and only if the outcome field of res_1 is completed, the term "continued res_1 and then A_2 (*inf*)" is in turn rewritten to "concluded res_1 and then res_2", where res_2 is the result of performing A_2. The same auxiliary operations "continued" and "concluded" are used to specify the performance of other action combinations (not shown here).

This method seems to solve the efficiency problem. I estimate that performance time is now $O(n)$, where n is the action size after any unfolding.

Yet another method is possible, based on the style used in the definitive operational semantics of action notation (Mosses 1992). The basic principle of this method is that actions are performed gradually. For example, "A_1 and then A_2" is performed roughly as follows. The 'first' primitive action of A_1 is performed; then A_1' is performed, where A_1' is the remainder of A_1 after removing that primitive action. If A_1 is vacuous (as a consequence of previously removing all primitive actions), then A_2 is performed.

equations

... ...

[and-then-1]
performed A_1 (d, b, s) = (completed, com_1, d_1, b_1, s_1),
performed A_2 (d, b, s_1) = (completed, com_2, d_2, b_2, s_2),
$b' = b_1 \oplus b_2$, $b' \neq$ clash
---
performed (A_1 and then A_2) (d, b, s) =
(completed, max (com_1, com_2), (d_1, d_2), b', s_2)

[and-then-2]
performed A_1 (d, b, s) = (completed, com_1, d_1, b_1, s_1),
performed A_2 (d, b, s_1) = (completed, com_2, d_2, b_2, s_2),
$b' = b_1 \oplus b_2$, $b' =$ clash
---
performed (A_1 and then A_2) (d, b, s) = (failed, com_1 max com_2, (), {}, s_2)

[and-then-3]
performed A_1 (d, b, s) = (completed, com_1, d_1, b_1, s_1),
performed A_2 (d, b, s_1) = (out_2, com_2, d_2, b_2, s_2), $out_2 \neq$ completed
---
performed (A_1 and then A_2) (d, b, s) = (out_2, max (com_1, com_2), d_2, b_2, s_2)

[and-then-4]
performed A_1 (d, b, s) = (out_1, com_1, d_1, b_1, s_1), $out_1 \neq$ completed
---
performed (A_1 and then A_2) (d, b, s) = (out_1, com_1, d_1, b_1, s_1)

... ...

Box 3 Interpretation of composite actions – natural semantics method.

[6] ASF+SDF includes a facility for compiling equations to C. Paul Klint has informed me that a future version of this compiler will automatically factor out common premises from a group of equations. (It is a form of common subexpression elimination.) This should eliminate the problem discussed here.

I have not yet tried this method. It might prove to be impracticable, because splitting a complex action into a primitive action and a remainder must entail a lot of term construction. Unlike the natural semantics method, however, the operational semantics method could be used to specify interleaving and concurrency. Thus I might be forced to adopt the operational semantics method anyway, when I eventually extend the interpreter to the whole of action notation.

context-free syntax

...

continued Result InfixCombinator Action "(" Info ")" → Result

concluded Result InfixCombinator Result → Result

equations

... ...

[and-then-1] performed A_1 (*inf*) = res_1 / performed (A_1 and then A_2) (*inf*) = continued res_1 and then A_2 (*inf*)

[and-then-2] performed A_2 (d, b, s_1) = res_2 / continued (completed, com_1, d_1, b_1, s_1) and then A_2 (d, b, s) = concluded (completed, com_1, d_1, b_1, s_1) and then res_2

[and-then-3] $out_1 \neq$ completed / continued (out_1, com_1, inf_1) and then A_2 (*inf*) = (out_1, com_1, inf_1)

[and-then-4] $b' = b_1 \oplus b_2$, $b' \neq$ clash / concluded (completed, com_1, d_1, b_1, s_1) and then (completed, com_2, d_2, b_2, s_2) = (completed, max (com_1, com_2), (d_1, d_2), b', s_2)

[and-then-5] $b' = b_1 \oplus b_2$, $b' \neq$ clash / concluded (completed, com_1, d_1, b_1, s_1) and then (completed, com_2, d_2, b_2, s_2) = (failed, max (com_1, com_2), (), {}, s_2)

[and-then-6] $out_2 \neq$ completed / concluded (completed, com_1, inf_1) and then (out_2, com_2, inf_2) = (out_2, max (com_1, com_2), inf_2)

... ...

Box 4 Interpretation of composite actions – factored natural semantics method.

4 Action Transformation

The ACTRESS project (Brown *et al.* 1992, Brown & Watt 1994, Moura & Watt 1994) has developed a variety of action transformations. An ACTRESS-generated compiler translates the source program to its denotation, the *program action*; then it sort-checks the program action, transforms it, and finally translates it to (C) object code. Action transformations are essential if the object code is to be acceptably efficient. However, the effort of programming these transformations in ML, the ACTRESS implementation language, proved to be considerable.

Before implementing these transformations in ML, Hermano Moura formalised them, and prototyped them by transcription of his inference rules into Prolog. It would have been much more convenient if ASF+SDF had been available to him at the time. This is because term rewriting is a very natural paradigm for expressing code transformations.

The ACTRESS action transformer (Moura 1993, Moura & Watt 1994) implements four transformations:

- *Algebraic simplification:* application of the algebraic laws of action notation.
- *Transient elimination:* essentially constant propagation, and elimination of redundant "give" actions.
- *Binding elimination:* replacement of applied occurrences of tokens by the (statically known) data to which they are bound, and elimination of redundant "bind" actions.
- *Storage allocation:* replacement of "allocate" actions (dynamic storage allocation) by static storage allocation, where possible.

At the time of writing, I have implemented algebraic simplification in ASF+SDF, and partly implemented binding elimination. These are outlined in the following subsections.

4.1 Algebraic Simplification

Action notation enjoys a variety of nice algebraic laws (Mosses 1992). For example, "fail" is a unit of "or", and "complete" is a unit of "and" and of "and then". These laws, and others, are easily expressed as equations in ASF+SDF – see Box 5.

Now any action term will be automatically simplified by application of the equations of Box 5. This is so whether the term is entered using the ASF+SDF editor, or generated by translation from a source program, or generated by application of other transformations.

I have not expressed all the laws of action notation as equations. In particular, I have not yet encountered any need to exploit the fact that all infix combinators are associative, and that some are commutative. In any case, ASF+SDF has no special facility for specifying associativity and commutativity.[7]

[7] Of course, expressing the commutative property by an ordinary equation results in infinite rewriting.

4.2 Binding Elimination

If an action has been generated by translation from a source program, it is often found that many or all bindings can be eliminated from the action – especially if the source language is statically scoped.

The basic principle of binding elimination is as follows. A term of the form "the *d'* bound to *k*" (an *applied occurrence* of token *k*), in a scope where it is known that *k* is bound to datum *d*, may be replaced by the term "*d*" (more properly, "the *d'* yielded by *d*"). If all applied occurrences of *k* can be replaced in this way, the "bind" action that produced the binding of *k* to *d* may be eliminated (more properly, replaced by "complete").

This can be expressed in ASF+SDF – see Box 6. My method is as follows:

- If an action *A* produces known bindings, replace it by "*A'* producing *b*". Here *b* is the set of known bindings produced by *A*, and action *A'* is obtained from *A* by eliminating the "bind" actions that produced these known bindings.
- Replace "(A_1 producing *b*) hence A_2" by "A_1 hence (A_2 receiving *b*)".
- Simplify "*A* receiving *b*" by using *b* to replace all scoped applied occurrences of tokens bound in *b*.

The specifications of "producing" and "receiving" are shown in Boxes 6(a) and (b).

equations

[or-1]	fail or *A* = *A*
[or-2]	*A* or fail = *A*
[and-1]	complete and *A* = *A*
[and-2]	*A* and complete = *A*
[and-then-1]	complete and then *A* = *A*
[and-then-2]	*A* and then complete = *A*
[and-then-3]	escape and then *A* = escape
[and-then-4]	fail and then *A* = fail
...	...
[give]	give nothing = fail
[bind]	bind *Y* to nothing = fail
...	...

Box 5 Algebraic simplification of actions.

The following is an example of binding elimination, together with algebraic simplification:

```
( bind 'x' to 7
  ( allocate a cell then bind 'y' to it ) and
  bind 'z' to true )
hence
  store the integer bound to 'x' in the cell bound to 'y'
```

$\Rightarrow$ [bind-1] in Box 6(a)

```
( complete binding {'x' ↦ 7} and
  ( allocate a cell then bind 'y' to it ) and
  complete binding {'z' ↦ true} )
hence
  store the integer bound to 'x' in the cell bound to 'y'
```

$\Rightarrow$ [and-1, and-2, producing] in Box 6(a)

```
( complete and
  ( allocate a cell then bind 'y' to it ) and
  complete ) binding {'x' ↦ 7, 'z' ↦ true}
hence
  store the integer bound to 'x' in the cell bound to 'y'
```

context-free syntax

Action producing Bindings → Action

equations

...	...
[bind-1]	bind k to d = complete producing $\{k \mapsto d\}$
[and-1]	$(A_1$ producing $b_1)$ and A_2 = $(A_1$ and $A_2)$ producing b_1
[and-2]	A_1 and $(A_2$ producing $b_2)$ = $(A_1$ and $A_2)$ producing b_2
[hence-1]	$(A_1$ producing $b_1)$ hence A_2 = A_1 hence $(A_2$ receiving $b_1)$
[hence-2]	A_1 hence $(A_2$ producing $b_2)$ = $(A_1$ hence $A_2)$ producing b_2
[moreover-1]	$\dfrac{b' = b_1 - \text{domain (out-bindings } (A_2))}{(A_1 \text{ producing } b_1) \text{ moreover } A_2 = (A_1 \text{ moreover } A_2) \text{ producing } b'}$
[moreover-2]	A_1 moreover $(A_2$ producing $b_2)$ = $(A_1$ moreover $A_2)$ producing b_2
...	...
[producing]	A producing b_1 producing b_2 = A producing $b_1 \oplus b_2$

Box 6(a) Binding elimination in actions – "producing".

⇒ [and-1, and-2] in Box 5

(allocate a cell then bind 'y' to it) producing {'x' ↦ 7, 'z' ↦ true}
hence
  store the integer bound to 'x' in the cell bound to 'y'

⇒ [hence-1] in Box 6(a)

(allocate a cell then bind 'y' to it)
hence
(store the integer bound to 'x' in the cell bound to 'y') receiving {'x' ↦ 7, 'z' ↦ true}

⇒ [store, bound-1, bound-2] in Box 6(b)

(allocate a cell then bind 'y' to it)
hence
  store 7 in the cell bound to 'y'

context-free syntax

Action receiving Bindings → Action
Yielder receiving Bindings → Yielder

equations

[complete] complete receiving *b* = complete

[give] (give *Y*) receiving *b* = give (*Y* receiving *b*)

[given] (given *d*) receiving *b* = given *d*

[bind-2] (bind *k* to *Y*) receiving *b* = bind *k* to (*Y* receiving *b*)

[bound-1] $\dfrac{b \text{ at } k \neq \text{nothing}}{(\text{the } d \text{ bound to } k) \text{ receiving } b = \text{the } d \text{ yielded by } (b \text{ at } k)}$

[bound-2] $\dfrac{b \text{ at } k = \text{nothing}}{(\text{the } d \text{ bound to } k) \text{ receiving } b = \text{the } d \text{ bound to } k}$

[store] (store Y_1 in Y_2) receiving *b* = store (Y_1 receiving *b*) in (Y_2 receiving *b*)

[and-3] (A_1 and A_2) receiving *b* = (A_1 receiving *b*) and (A_2 receiving *b*)

[hence-3] (A_1 hence A_2) receiving *b* = (A_1 receiving *b*) hence A_2

[moreover-3] (A_1 moreover A_2) receiving *b* = (A_1 receiving *b*) moreover (A_2 receiving *b*)

... ...

Box 6(b) Binding elimination in actions – "receiving".

5 Conclusion and Further Work

My experience with ASF+SDF has on the whole been positive. Term rewriting is a powerful computational model, and very natural for language translation, transformation, and interpretation. SDF relieves the specifier of excessive attention to syntactic details, neatly combines abstract and concrete syntactic specification, and supports mixfix operations.

However, ASF+SDF has pitfalls for the unwary (among whom I include myself). The efficiency of term rewriting is highly sensitive to the way in which the equations are written, as discussed in Section 3. I believe that programmers need a mental model of the way in which their programs are executed on a machine. This is in conflict with the deliberate concealment, on methodological grounds, of such a model from the users of ASF+SDF (Klint 1993b).

There are also syntactic pitfalls. It is all too easy for the specifier to introduce ambiguities, especially involving mixfix operations. SDF detects ambiguity only when parsing particular terms; it cannot of course detect ambiguity of the context-free grammar. The specifier can suppress some ambiguities by assigning priorities and associativities to operations, but then there is a risk that some terms will be parsed differently from the specifier's intentions. Finally, the lexical and context-free syntax sometimes interact in unexpected ways. Peter Mosses pointed out a lovely example: if *l* and *t* are variables, "list" can be parsed as "*l* is *t*"!

ASF+SDF is an impressive piece of software engineering. It supports incremental development of modular specifications: if the equations of module *M* are changed, only *M* is re-compiled; if the interface part of *M* is changed, only those modules that import *M* are re-compiled. I took advantage of this to impose an elaborate modular structure on my action interpreter and transformer (not discussed in this report). On the down side, the user interface of ASF+SDF is somewhat eccentric. Also, ASF+SDF requires massive computational power.

The work described here is incomplete. So far I have specified a large subset of action notation, but an important omission is the communicative facet. I have also omitted a few rarely-used action primitives and combinators.

I have specified a restricted form of transient elimination, and binding elimination for known bindings only. As shown in (Moura 1993, Moura & Watt 1994), all bindings can be eliminated from a statically-scoped action. An action that binds a token to an unknown datum is replaced by one that stores the unknown datum in a known cell, and each applied occurrence of that token is replaced by a fetch from that known cell. This works very well in the context of an ACTRESS-generated compiler, where storage is mapped to a global array, but it would be less useful in the context of the prototype described here.

The action interpreter (and transformer) can be coupled to an action semantics of a programming language, also specified as an ASF+SDF specification. Then the user can use the ASF+SDF editor to enter programs in that language. Each program is translated to an action, and the latter may be (transformed and) interpreted.

However, the ASF+SDF specification language is rather different from the specification language of (Mosses 1992), and much editing would have to be done to convert a given action-semantic specification. For this reason, Arie van Deursen and Peter Mosses have written a system that translates a specification from the standard specification language to the ASF+SDF specification language. Their system also performs useful consistency checks on the specification. This system, *Action Semantic Description Tools* (van Deursen & Mosses 1994), is itself implemented in ASF+SDF. ASD Tools has already been used to check and translate the large specifications Data Notation, Action Notation, AD Action

Semantics (Mosses 1992), and the current draft of Pascal Action Semantics (Mosses & Watt 1994).

The way forward, then, is to extend the action interpreter to cover the whole of action notation (including the communicative facet), make it more robust, and integrate it with ASD Tools. This will allow us to test specifications like Pascal Action Semantics thoroughly.[8] The result will be a valuable prototyping tool for language designers and specifiers who use action semantics.

References

Brown, D.F., Moura, H., and Watt, D.A. (1992) ACTRESS: an action semantics directed compiler generator, in *Compiler Construction – 4th International Conference* (ed. Kastens, U., and Pfahler, P.), Springer-Verlag, 95--109.

Brown, D.F., and Watt, D.A. (1994) Sort inference in the ACTRESS compiler generation system, in these proceedings.

Despeyroux, T. (1988) Typol – a formalism to implement natural semantics, Rapports Techniques 94, INRIA, Sophia Antipolis, France.

Klint, P. (1993a) A meta-environment for generating programming environments, *ACM Transactions on Software Engineering and Methodology* 2, 2, 176–201.

Klint, P. (1993b) The ASF+SDF meta-environment – user's guide, CWI, Amsterdam.

Mosses, P.D. (1992) *Action Semantics*, Cambridge University Press.

Mosses, P.D., and Watt, D.A. (1994) Pascal action semantics, version 0.6, Computer Science Department, Aarhus University, and Department of Computing Science, University of Glasgow.

Moura, H. (1993) Action notation transformations, PhD thesis, University of Glasgow.

Moura, H., and Watt, D.A. (1994) Action transformations in the ACTRESS compiler generator, in *Compiler Construction– 5th International Conference* (ed. Fritzson, P.), Springer-Verlag, 16–30.

Pagan, F. (1979) Algol 68 as a metalanguage for denotational semantics, *Computer Journal* 22, 1, 63–66.

van Deursen, A., and Mosses, P.D. (1994) ASD – the Action Semantics Description Tools, in these proceedings.

Watt, D.A. (1986) Executable semantic descriptions, *Software—Practice and Experience* 16, 1, 13–43.

Watt, D.A. (1991) *Programming Language Syntax and Semantics*, Prentice Hall International.

[8] And at the same time subject the prototyping tool itself to a thorough test!

Current and Future Projects

Discussion chaired by Peter D. Mosses

This discussion session took place at the end of a long and exhausting day. There was time only for the participants of the workshop to give brief indications of the topics that they hope to investigate in the near future. More time for coordination of projects should clearly have been allocated in the programme of the workshop.

The following list of topics may give an impression of the work being carried out by the participants, in action semantics and related fields. It is based on rough notes taken during the discussion; apologies to anyone who mentioned topics that didn't get properly recorded. Abbreviations: *a.n.* action notation, *a.s.* action semantics, *a.s.d.* action semantic descriptions.

- publishing up-to-date a.s.d.s of Standard ML, Standard Pascal, ...
- investigating a.s.d.s of logic programming, VHDL, ...
- studying the ANDF-FS, reformulating in standard notation
- analysis of stackability in higher-order cases
- improved type inference for a.n.
- lifting analysis from a.n. to programming languages
- partial evaluation of a.n.
- static action semantics
- use of attribute grammars in a.s.
- evolving algebra semantics for a.n.
- use of a.n. in evolving algebras
- language[1] design based on a.n.
- comparing LDL to a.s., investigating possibility of generating a.n.
- comparing ACP to communicative a.n.
- development of ASD tools, using ASF+SDF
- implementation of interpreters and compilers for a.n.
- tutorial on a.s. (at FME'94)
- software specification using a.n.
- proof techniques for action equivalence
- improved operational semantics for a.n.

Please inform the action semantics mailing list when starting new projects (a footnote in the Preface tells how to subscribe), and when new papers on action semantics and related topics become available.

LIST OF PARTICIPANTS

Dr E. BACON
CIT School
University of Greenwich
Wellington Street
London SE18 6PF
ENGLAND
E-mail:

Deryck F. BROWN
PACT
10 Priory Road
Clifton
Bristol, BS8 1TU
UK
E-mail: deryck@pact.srf.ac.uk

Arie van DEURSEN
CWI
P.O. Box 94079
NL-1090 GB Amsterdam
THE NETHERLANDS
E-mail: arie@cwi.nl

Kyung-Goo DOH
University of Aizu
Fukushima 965-80
JAPAN
E-mail: kg-doh@u-aizu.ac.jp

Bo Stig HANSEN
Department of Computer Science
Building 344
Technical University of Denmark
DK-2800 Lyngby
DENMARK
E-mail: bsh@id.dth.dk

D. HUNT
CIT School
University of Greenwich
Wellington Street
London SE18 6PF
ENGLAND
E-mail:

Ralf LÄMMEL
Universität Rostock
FB Informatik
D-18051 Rostock
GERMANY
E-mail:
rlaemmel@informatik.uni-rostock.de

Søren B. LASSEN
Dept. of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C
DENMARK
E-mail: thales@daimi.aau.dk

Stephen McKEEVER
Programming Research Group
Oxford University
Wolfson Building
Parks Road
Oxford OX1 3QD
ENGLAND
E-mail: swm@comlab.ox.ac.uk

Peter D. MOSSES
BRICS, Dept. of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C
DENMARK
E-mail: pdmosses@daimi.aau.dk

Hermano MOURA
Caixa Economica Federal
SQN 206, Bloco I, Apto 103
Brasilia, DF
BRAZIL
E-mail: hermano@cic.unb.br

Wolfgang MÜLLER
Cadlab – Universität Paderborn
Bahnhofstr. 32
D-33102 Paderborn
GERMANY
E-mail: wolfgang@cadlab.de

Peter ØRBÆK
Dept. of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C
DENMARK
E-mail: poe@daimi.aau.dk

Jens PALSBERG
161 Cullinane Hall
College of Computer Science
Northeastern University
360 Huntington Avenue
Boston, MA 02115
USA
E-mail: palsberg@ccs.neu.edu

Arnd POETZSCH-HEFFTER
Fakultät für Informatik
Technische Universität
D-80290 München
GERMANY
E-mail:
poetzsch@informatik.tu-muenchen.de

Günter RIEDEWALD
Universität Rostock
FB Informatik
D-18051 Rostock
GERMANY
E-mail: gri@informatik.uni-rostock.de

David SCHMIDT
Computing and Info. Sciences Dept.
Kansas State Univ.
Nichols Hall
Manhattan, KS 66506
USA
E-mail: schmidt@cis.ksu.edu

David A. WATT
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ
SCOTLAND
E-mail: daw@dcs.gla.ac.uk

Ms G. WINDALL
CIT School
University of Greenwich
Wellington Street
London SE18 6PF
ENGLAND
E-mail: g.windall@greenwich.ac.uk

Recent Publications in the BRICS Notes Series

NS-94-1 **Peter D. Mosses, editor.** ***Proc. 1st International Workshop on Action Semantics*** **(Edinburgh, 14 April, 1994), number NS-94-1 in BRICS Notes Series, Department of Computer Science, University of Aarhus, May 1994. BRICS. 145 pp.**