Extraction of Strong Typing Laws from Action Semantics Definitions

Kyung-Goo Doh* and David A. Schmidt*
Department of Computing and Information Sciences
Kansas State University, Manhattan, Kansas 66506, U.S.A.
{doh,schmidt}@cis.ksu.edu

Abstract: We describe a method that automatically extracts a type checking semantics, encoded as a set of type inference rules, from an action semantics definition of a programming language. The type inference rules are guaranteed to enforce strong typing, since they are based on an underlying metasemantics for action semantics, which uses typing functions and natural transformations to give meaning. Next, we use the type checking semantics to extract a dynamic semantics definition from the original action semantics definition. We present an example.

1 Introduction

The key component of a compiler-based programming language is its typing system. A compiler-based language should have a static semantics (hereafter, called a *typing semantics*) that matches the structure of the data types and operations that underlie the language. A typing semantics is accompanied by a dynamic semantics, which gives meaning to the well typed programs in the language.

A language is *statically typed* if the typing annotations of the phrases in a program can be calculated without running the program. The language is *strongly typed* if every program that is completely annotated with typings (such a program is *well typed*) will not produce an operator-operand incompatibility error (a *typing error*) when it is run. A statically typed language should be implemented by a compiler that annotates programs with typings, and the typing annotations make the language strongly typed. (Algol60 is an example of a statically typed language that is not strongly typed, due to imprecise typing of procedure parameters [27].)

These goals place upon the language designer the burden of designing a typing semantics that enforces strong typing. Since the design of a strongly typed language is nontrivial, a language designer would do best to follow a methodology based on a formal semantics; the methodology should support a method for deriving the typing semantics and showing that it is a strong typing. But this is surprisingly difficult to do with existing semantics methods. Consider a definition in denotational semantics [33]; here is a sample clause of a definition:

*Partially supported by NSF Grants CCR-8822378 and CCR-9102625.

$$\begin{array}{ll}\mathcal{E}[\![\text{is\_zero E}]\!] = \lambda e.\lambda s.\ \textit{cases}\ (\mathcal{E}[\![\text{E}]\!]e\ s)\ \textit{of} & \\ \qquad\qquad isInteger(i) \rightarrow (i =_{int} 0) & \\ \qquad\qquad [\!]\ isReal(r) \rightarrow (r =_{real} 0.0) & \\ \qquad\qquad [\!]\ isBoolean(b) \rightarrow error & \\ \qquad\qquad \cdots end & \end{array}$$

The clause suggests the typing rule:

$$\frac{typings\_in(e) \vdash \text{E} : \tau}{typings\_in(e) \vdash \text{is\_zero E} : Boolean} \quad \tau \in \{Integer, Real\}$$

but the formal derivation of the rule from the clause is intricate. Noteworthy attempts are by Barbuti and Martelli [1] and Montenyohl and Wand [19], where a separate typing semantics is handwritten and proved to enforce strong typing with respect to the original semantics. Then, hand transformations are performed on the original semantics to derive a dynamic semantics.

An approach that is often relevant is the application of a partial evaluator to the semantics definition [11, 14, 15, 35]. When supplied with a semantics definition, a partial evaluator produces a compiler. The generated compiler takes a source program as input, translates the program into an expression in the semantic metalanguage, and evaluates the static parts of the expression. The result is a compiled program that contains only dynamic operations. If a typing semantics is encoded within a semantics definition, and it is static, then the compiler performs type checking. This occurs in Montenyohl and Wand's example [19], as demonstrated by Consel and Danvy [6]. Of course, there is no guarantee that the semantics definition contains a static typing semantics. Also, a partial evaluator does not extract the typing semantics and present it separately, which is our goal here.

Finally, a language designer might apply operational or axiomatic semantics techniques and hand code the typing and dynamic semantics. Then, hand proofs must be done for static and strong typing properties [3, 10, 34]. This task is daunting.

None of the above approaches are completely satisfactory, so we report another approach, based on Mosses and Watt's *action semantics* [20, 21, 25, 24, 26, 36], which surmounts the problems noted above. From an action semantics definition of a programming language, we can mechanically extract a typing semantics that is a static and a strong typing. The proofs of static and strong typing are immediate, from general results about the model for action semantics notation [12]. Further, we show how to calculate the dynamic semantics of a language from its typing semantics and the original action semantics definition. The result is a strong typing semantics and a dynamic semantics, which can be used along the lines suggested by Lee and Pleban [16, 17] and Nielson and Nielson [28, 29] to define a compiler for the language. Since the typing semantics can be represented in inference rule format, it also serves as documentation of the typing structure for the language designer and users.

The remainder of the paper goes as follows. We first introduce action semantics notation; next, we describe the approach for extracting the typing semantics and dynamic semantics; and finally, we apply the algorithms to an example language definition.

Functional facet: Its types are *Proper-functional-type* $\cup$ { *ns* }, where:
$t \in$ *Proper-functional-type*
$t ::= int \mid real \mid bool \mid t_1 \times t_2 \mid t_1 \rightarrow t_2$
The ordering is the smallest reflexive, transitive ordering such that:
$int \leq real$
$t \leq ns$ for all t
$t_1 \times t_2 \leq t_1' \times t_2'$ iff $t_1 \leq t_1'$ *and* $t_2 \leq t_2'$
$t_1 \rightarrow t_2 \leq t_1' \rightarrow t_2'$ iff $t_1' \leq t_1$ *and* $t_2 \leq t_2'$

Declarative facet: Its types are *Proper-declarative-type* $\cup$ { *ns* }, where:
$d \in$ *Proper-declarative-type*
$d ::= \{i : t_i\}_{i \in I}$ where I is a finite set of identifiers.
The ordering is the smallest reflexive, transitive ordering such that:
$d \leq ns$ for all d
$\{i : t_i\}_{i \in I} \leq \{i : t_i'\}_{i \in I}$ iff for all $i \in I$, $t_i \leq t_i'$

Figure 1: Facets

2 Action Semantics

Action semantics is a high level notation for writing modular programming language definitions [21, 25, 24, 26, 36]. The notation consists of combinator-like entities, called *actions*, that operate upon *facets*. A facet is a collection of types, and a type is a collection of values. The *functional facet* contains those data types that can be used as temporary values ("transient information" [21]) in a computation. Types like *int*, *bool*, *real*, *int* × *bool*, and so on, belong to the functional facet. Actions that take arguments and produce answers in the functional facet include arithmetic and logical operations. A second facet is the *declarative facet*, which contains types of identifier, value binding ("scoped information"). The types in the declarative facet are record types [5, 13, 31]; an example type is {A:*int*, B:*bool*}, which describes those binding sets ("records") that map A to an integer value and B to a boolean value. Actions that take arguments and produce answers in the declarative facet include operations for making and finding bindings (in a symbol table). A third facet is the *imperative facet*, which contains types of storage structures. Actions include operations for accessing and updating primary storage. Yet another is the *communicative facet*, which describes structures for communication and has actions for file and message input/output. Due to lack of space, we will not explore the last two facets.

The set of types for each facet also includes an error type, which we call *ns* (for "nonsense"). An output of *ns* type occurs when an action receives an argument whose type is incompatible with the action, that is, when a typing error arises. For example, an *ns*-typed output occurs when a boolean value is given to an addition action.

The types in a facet can be ordered to express subtyping relationships. For example, we might have $int \leq real$ in the functional facet, that is, *int* is a subtype of *real* [13, 31], or {A:*int*} $\leq$ {A:*real*} in the declarative facet. Figure 1 shows the internal structure

Action	Kind	Typing function	Meaning
copy	$\mathtt{F}\to\mathtt{F}$	$\lambda t:\mathtt{F}.\,t$	$\lambda t:\mathtt{F}.\lambda v:t.\,v$
give(n : t$_0$)	$\mathtt{1}\to\mathtt{F}$	$\lambda t:\mathtt{1}.\,t_0$	$\lambda t:\mathtt{1}.\lambda v:t.\,n$
eqzero	$\mathtt{F}\to\mathtt{F}$	$\lambda t:\mathtt{F}.\,\mathit{if}\ t\leq \mathit{real}$ $\mathit{then\ bool}$	$\lambda t:\mathtt{F}.\,\mathit{cases}\ t\ \mathit{of}$ $\mathit{int} : \lambda v:\mathit{int}.(v =_{\mathit{int}} 0)$ $\mathit{real} : \lambda v:\mathit{real}.(v =_{\mathit{real}} 0.0)$ $\mathit{else} : \lambda v.\mathit{error}$ end
add	$\mathtt{F}\to\mathtt{F}$	$\lambda t:\mathtt{F}.$ $\mathit{if}\ \mathit{fst}(t)\leq \mathit{real}$ $\mathit{and}\ \mathit{snd}(t)\leq \mathit{real}$ $\mathit{then}\ \mathit{fst}(t)\sqcup \mathit{snd}(t)$	$\lambda t:\mathtt{F}.$ $\mathit{cases}\ t\ \mathit{of}$ $\mathit{int}\times\mathit{int} : \lambda(v_1,v_2):\mathit{int}\times\mathit{int}.\ v_1 +_{\mathit{int}} v_2$ $\mathit{int}\times\mathit{real} : \lambda(v_1,v_2):\mathit{int}\times\mathit{real}.\ \mathit{coerce\text{-}real}(v_1) +_{\mathit{real}} v_2$ $\mathit{real}\times\mathit{int} : \lambda(v_1,v_2):\mathit{real}\times\mathit{int}.\ v_1 +_{\mathit{real}} \mathit{coerce\text{-}real}(v_2)$ $\mathit{real}\times\mathit{real} : \lambda(v_1,v_2):\mathit{real}\times\mathit{real}.\ v_1 +_{\mathit{real}} v_2$ $\mathit{else} : \lambda(v_1,v_2).\ \mathit{error}$ end
bind I	$\mathtt{F}\to\mathtt{D}$	$\lambda t:\mathtt{F}.\,\{\mathrm{I}:t\}$	$\lambda t:\mathtt{F}.\lambda v:t.\,\{\mathrm{I}=v\}$
find I	$\mathtt{D}\to\mathtt{F}$	$\lambda d:\mathtt{D}.\,\mathit{if}\ \{\mathrm{I}:t\}\in d$ $\mathit{then}\ t$	$\lambda d:\mathtt{D}.\lambda r:d.\,r.\mathrm{I}$

Note: "*if C then T*" abbreviates "*if C then T else ns*"
$\mathit{fst}(t_1\times t_2) = t_1$, $\mathit{snd}(t_1\times t_2) = t_2$
$\mathit{fst}(\mathit{int}) = \mathit{fst}(\mathit{real}) = \mathit{fst}(\mathit{bool}) = \mathit{ns} = \mathit{snd}(\mathit{bool}) = \mathit{snd}(\mathit{real}) = \mathit{snd}(\mathit{int})$
"r.I" is record indexing

Figure 2: Actions

of the two facets we use in this paper. (Note that the type sets of the two facets form sup-semilattices. Also, the typing ordering for the declarative facet is simpler than the version that is normally used; see [13].)

An action is a mapping whose domain and codomain are facets. For example, the action copy : $\mathtt{F} \to \mathtt{F}$ is the identity mapping on the functional facet. ("F" stands for the functional facet, and "D" stands for the declarative facet.) Since the functional facet contains many types, copy is in fact a family of identity functions: an identity function for integer inputs, an identity for boolean inputs, an identity for real inputs, and so on. Thus, copy $= \{\lambda v{:}t.v\}_{t\in\mathtt{F}}$, which we also write as $\lambda t{:}\mathtt{F}.\lambda v{:}t.v$. We can summarize copy's behavior with the *typing function* $\mathbf{T}_{\mathsf{copy}} = \lambda t{:}\mathtt{F}.t$, which states that, whenever copy receives an input of type t, its output is of type t.

Each action, a, has a typing function, $\mathbf{T}_{\mathsf{a}}$, that characterizes its behavior. In analogy with the typing system in Automath [8], a typing function $\lambda t{:}\mathtt{F}.f(t)$ encodes the second order type $\forall t{:}\mathtt{F}.f(t)$. Further, the typings are "shallow," in the sense of ML types [18]. This makes actions into polymorphic functions, where the polymorphism can be parametric, inclusive, or ad-hoc. A mathematical view is that an action, a, is a natural transformation in $\mathcal{I} \dot{\to} \mathcal{I} \circ \mathbf{T}_{\mathsf{a}}$, where $\mathcal{I}$ is the interpretation functor that maps the type names in the facets to the value sets they represent and where $\mathbf{T}_{\mathsf{a}}$ is treated as an endofunctor on the facet. Details are found in [12].[1]

[1]A related model, which is based on *unified algebra* rather than category theory, is described in

Action	Kind	Typing function	Meaning
a_1 then a_2	$\mathtt{K}_1 \rightarrow \mathtt{K}_3$	$\lambda k{:}\mathtt{K}_1.\, \mathbf{T}_{\mathsf{a}_2}(\mathbf{T}_{\mathsf{a}_1}(k))$	$\lambda k{:}\mathtt{K}_1.\, \lambda v{:}k.\mathsf{a}_2(\mathbf{T}_{\mathsf{a}_1}(k))(\mathsf{a}_1(k)(v))$
	where $\mathsf{a}_1 : \mathtt{K}_1 \rightarrow \mathtt{K}_2$ and $\mathsf{a}_2 : \mathtt{K}_2 \rightarrow \mathtt{K}_3$		
a_1 and a_2	$\mathtt{K}_1 \rightarrow \mathtt{K}_2$	$\lambda k{:}\mathtt{K}_1.\mathbf{T}_{\mathsf{merge}_{\mathtt{K}_2}}(\mathbf{T}_{\mathsf{a}_1}(k), \mathbf{T}_{\mathsf{a}_2}(k))$	$\lambda k{:}\mathtt{K}_1.\lambda v{:}k.\mathsf{merge}_{\mathtt{K}_2}(\mathbf{T}_{\mathsf{a}_1}(k), \mathbf{T}_{\mathsf{a}_2}(k))$ $(\mathsf{a}_1(k)(v), \mathsf{a}_2(k)(v))$
	where $\mathsf{a}_1 : \mathtt{K}_1 \rightarrow \mathtt{K}_2$ and $\mathsf{a}_2 : \mathtt{K}_1 \rightarrow \mathtt{K}_2$		
a_1 andthen a_2	$\mathtt{D} \rightarrow \mathtt{F}$	$\lambda d{:}\mathtt{D}.\mathbf{T}_{\mathsf{a}_2}(\mathbf{T}_{\mathsf{a}_1}(d), d)$	$\lambda d{:}\mathtt{D}.\lambda r{:}d.\mathsf{a}_2(\mathbf{T}_{\mathsf{a}_1}(d), d)(\mathsf{a}_1(d)(r), r)$
	where $\mathsf{a}_1 : \mathtt{D} \rightarrow \mathtt{F}$ and $\mathsf{a}_2 : \mathtt{F}\times\mathtt{D} \rightarrow \mathtt{F}$		
a_1 else a_2	$\mathtt{F}\times\mathtt{D} \rightarrow \mathtt{F}$	$\lambda(t, d){:}\mathtt{F}\times\mathtt{D}.$ *if* $t \leq$ *bool* *then* $\mathbf{T}_{\mathsf{a}_1}(d) \sqcup \mathbf{T}_{\mathsf{a}_2}(d)$	$\lambda(t, d){:}\mathtt{F}\times\mathtt{D}.$ *if* $t \leq$ *bool* *then* $\lambda(v, r){:}t\times d.$ *if* v *then* $\mathsf{a}_1(d)(r)$ *else* $\mathsf{a}_2(d)(r))$ *else* $\lambda(v, r){:}t\times d.\mathit{error}$
	where $\mathsf{a}_1 : \mathtt{D} \rightarrow \mathtt{F}$ and $\mathsf{a}_2 : \mathtt{D} \rightarrow \mathtt{F}$		
furthermore a	$\mathtt{D} \rightarrow \mathtt{D}$	$\lambda d{:}\mathtt{D}.\mathbf{T}_{\mathsf{a}}(d)@d$	$\lambda d{:}\mathtt{D}.\lambda r{:}d.\mathsf{a}(d)(r)$ *cat* r
	where $\mathsf{a} : \mathtt{D} \rightarrow \mathtt{D}$		
	$\mathtt{F}\times\mathtt{D} \rightarrow \mathtt{D}$	$\lambda(t, d){:}\mathtt{F}\times\mathtt{D}.\mathbf{T}_{\mathsf{a}}(t)@d$	$\lambda(t, d){:}\mathtt{F}\times\mathtt{D}.\lambda(v, r){:}t\times d.\mathsf{a}(t)(v)$ *cat* r
	where $\mathsf{a} : \mathtt{F} \rightarrow \mathtt{D}$		

where $\mathbf{T}_{\mathsf{merge}_{\mathtt{F}}}(t_1, t_2) = t_1 \times t_2$
$\mathsf{merge}_{\mathtt{F}}(t_1, t_2)(v_1, v_2) = (v_1, v_2)$
$\mathbf{T}_{\mathsf{merge}_{\mathtt{D}}}(d_1, d_2) =$ *if have-disjoint-fields*(d_1, d_2) *then* $d_1@d_2$
$\mathsf{merge}_{\mathtt{D}}(d_1, d_2) =$ *if have-disjoint-fields*(d_1, d_2) *then* $\lambda(r_1, r_2).r_1$ *cat* r_2 *else* $\lambda(r_1, r_2).\mathit{error}$
"@" represents record type concatenation
(e.g., $\{\mathrm{A} : \mathit{int}, \mathrm{B} : \mathit{bool}\}@\{\mathrm{B} : \mathit{int}, \mathrm{C} : \mathit{real}\} = \{\mathrm{A} : \mathit{int}, \mathrm{B} : \mathit{bool}, \mathrm{C} : \mathit{real}\}$)
"*cat*" represents record concatenation
(e.g., $\{\mathrm{A}{=}0, \mathrm{B}{=}\mathit{false}\}$ *cat* $\{\mathrm{B}{=}1, \mathrm{C}{=}2.2\} = \{\mathrm{A}{=}0, \mathrm{B}{=}\mathit{false}, \mathrm{C}{=}2.2\}$)

Figure 3: Action Combinators

The relationship between the typing function and the action it describes is exact: if $\mathbf{T}_{\mathsf{a}}(t) = t'$, then for all arguments v of type t, $\mathsf{a}(t)(v)$ has type t'. In particular, if $\mathbf{T}_{\mathsf{a}}(t) = ns$, then $\mathsf{a}(t)(v)$ is an error.[2] This exact relationship is no accident. It is demanded by the mathematical model we use, for the typing functions and facets constitute the operator names and sort names, respectively, of a signature of a category-sorted algebra [30, 32]. The actions and value sets form the operations and carriers, respectively, of the category-sorted algebra. This relationship lets us extract the strong typing laws from a language definition. The formalities of category sorted algebra and action semantics are described in [12].

Actions exist for all the fundamental operations of programming languages: value passing, arithmetic, binding creation and lookup, storage allocation and updating, and so on [20, 24, 26, 36]. Our version of action notation is combinator-based, for technical and historical reasons, but it is interconvertible with the notation in Mosses' book [25].

Figure 2 presents the actions we use in this paper. We have already seen action **copy**. **give**(n : t_0) emits n as its output. Since **give** requires no input, we use a degenerate facet,

[21, 22, 23].

[2]Also, typing functions are "ns-strict": $\mathbf{T}_{\mathsf{a}}(ns) = ns$, for all actions a.

Abstract Syntax

D ∈ Declaration
E ∈ Expression
N ∈ Int-numeral
R ∈ Real-numeral

D ::= val I = E | D_1, D_2
E ::= N | R | $E_1 + E_2$ | if E_1 then E_2 else E_3 | is-zero E | I | let D in E end

Action Semantics

declare : Declaration → $\text{Action}_{D \to D}$
  declare⟦val I = E⟧ = evaluate⟦E⟧ then bind I
  declare⟦D_1, D_2⟧ = declare⟦D_1⟧ and declare⟦D_2⟧
evaluate : Expression → $\text{Action}_{D \to F}$
  evaluate⟦N⟧ = give(⟦N⟧ : int)
  evaluate⟦R⟧ = give(⟦R⟧ : real)
  evaluate⟦$E_1 + E_2$⟧ = (evaluate⟦E_1⟧ and evaluate⟦E_2⟧) then add
  evaluate⟦if E_1 then E_2 else E_3⟧ = evaluate⟦E_1⟧ andthen (evaluate⟦E_2⟧ else evaluate⟦E_3⟧)
  evaluate⟦is_zero E⟧ = evaluate⟦E⟧ then eqzero
  evaluate⟦I⟧ = find I
  evaluate⟦let D in E end⟧ = (furthermore declare⟦D⟧) then evaluate⟦E⟧

Figure 4: An Example Expression Language

called 1, for its domain. **eqzero** checks if a number is zero; **add** adds a pair of numbers; **bind** I maps a value into a binding set with a single binding for I to the value; and **find** I maps a binding set to the value bound to I in the set.[3]

Actions are composed into compound actions with combinators. For example, actions $a_1 : K_1 \to K_2$ and $a_2 : K_2 \to K_3$ can be sequentially composed into a_1 **then** $a_2 : K_1 \to K_3$ by the **then** combinator. (The codomain facet of a_1 must match the domain facet of a_2.) Combinators possess typing functionals that map the typing functions of the component actions into a typing function for the compound action. Figure 3 gives the definitions and typing functions for the combinators we use.

In addition to sequential composition, we have the parallel composition a_1 **and** a_2, which gives its input to both a_1 and a_2 and allows them to evaluate in parallel; the results are "merged." The compound action a_1 **andthen** a_2 is a combination of **then** and **and**. The action a_1 **else** a_2 models choice: the input, a Boolean, selects a_1 or a_2 for evaluation. Finally, **furthermore** a concatenates the binding set produced by a to the input binding set.

Figure 4 gives a language definition in action semantics.

[3]A reader familiar with Montenyohl and Wand's work [19] will notice that the structure of the meanings of the actions in Figure 2 match the structure of denotations in [19] following factorization and static replacement. This is not surprising, since the structure is a natural one for a language with static and dynamic stages. What is significant is that the meanings in Figure 2 *must* have proper structure because they are natural transformations – the category theory model makes the representations correct.

3 Derivation of the Typing Semantics

We derive the typing semantics for a language definition by deriving a typing rule for each semantic equation in the definition. This is done in two steps: (i) calculate the typing function for the semantic equation; (ii) translate the equation into an inference rule. For example, the semantic equation:

$$\textsf{evaluate}\ [\![\text{is\_zero E}]\!] = \textsf{evaluate}\ [\![\text{E}]\!]\ \textsf{then eqzero}$$

has as its typing function:

$$\textsf{typing}\ [\![\text{is\_zero E}]\!] = \lambda d{:}\mathtt{D}.\, \textit{if}\ \textsf{typing}\ [\![\text{E}]\!](d) \leq \textit{real then bool}$$

This typing function is translated into the rule:

$$\frac{d \vdash \text{E} : t \leq \textit{real}}{d \vdash \text{is\_zero E} : \textit{bool}}$$

We now present the details.

3.1 Calculation of Typing Functions

Typing functions have the syntax:

$F \in$ Typing-function-expression $\quad C \in$ Constraint
$T \in$ Type-expression $\quad I \in$ Type-identifier
$A \in$ Atomic-expression $\quad O \in$ Primitive-operator

$$\begin{aligned}
F &::= \lambda I.T \\
T &::= A \mid FT \mid \textit{if } C \textit{ then } T \mid O(T_1, \ldots, T_n), n \geq 0 \\
A &::= I \mid O(A_1, \ldots, A_n), n \geq 0 \\
C &::= A_1 \leq A_2 \mid C_1 \textit{ and } C_2 \mid O(A_1, \ldots, A_n), n \geq 0
\end{aligned}$$

(Recall that "*if C then T*" abbreviates "*if C then T else ns*.") Primitive operators, O, include constants like *int* and operators like *fst* from the third column of Figure 2.

A typing function expression is normalized by these rules:

(1) $(\lambda I.T)A \Rightarrow [A/I]T$
(2) $F(\textit{if } C \textit{ then } T) \Rightarrow \textit{if } C \textit{ then } FT$
(3) $O(T_1, \ldots, (\textit{if } C \textit{ then } T_i), \ldots, T_n) \Rightarrow \textit{if } C \textit{ then } O(T_1, \ldots, T_i, \ldots, T_n)$
(4) $\textit{if } C_1 \textit{ then if } C_2 \textit{ then } T \Rightarrow \textit{if } C_1 \textit{ and } C_2 \textit{ then } T$
(5) $\textit{fst}(T_1 \times T_2) \Rightarrow T_1$
(6) $\textit{fst}(\textit{int}) \Rightarrow \textit{ns}$

Rules similar to (5) and (6) are used for the other primitive operators.

The rules are confluent and strongly normalizing [9]; they remove nested lambda abstractions and "flatten" a typing function expression with nested occurrences of "*if C then T*" into an expression with at most one occurrence. Importantly, normal forms must have the format "$\lambda I.$ *if C then A*", which proves crucial for building the inference rules.

Here is an example rewriting:

$$\begin{array}{lll}
\mathbf{T}_{\textsf{eqzero then copy}} & = \lambda t.\mathbf{T}_{\textsf{copy}}(\mathbf{T}_{\textsf{eqzero}}(t)) & \\
& = \lambda t.(\lambda t'.t')((\lambda t'.\mathit{if}\ t' \leq \mathit{real\ then\ bool})(t)) & \\
& \Rightarrow \lambda t.(\lambda t'.t')(\mathit{if}\ t \leq \mathit{real\ then\ bool}) & \text{by (1)} \\
& \Rightarrow \lambda t.\mathit{if}\ t \leq \mathit{real\ then}\ (\lambda t'.t')\mathit{bool} & \text{by (2)} \\
& \Rightarrow \lambda t.\mathit{if}\ t \leq \mathit{real\ then\ bool} & \text{by (1)}
\end{array}$$

A semantic equation's typing function is derived in a similar fashion. Given a semantic equation: **valuate** $[\![\text{op } E_1 \ldots E_n]\!] = \cdots$ **valuate** $[\![E_1]\!] \cdots$ **valuate** $[\![E_n]\!] \cdots$

we wish to calculate $\mathbf{T}_{\textsf{valuate}}$ $[\![\text{op } E_1 \ldots E_n]\!]$, which we also call **typing** $[\![\text{op } E_1 \ldots E_n]\!]$. We replace each action, a, by $\mathbf{T}_{\textsf{a}}$; occurrences of **valuate** $[\![E_i]\!]$ are replaced by "primitive operators" $\mathbf{T}_{\textsf{valuate}}[\![E_i]\!]$. We then apply the rewriting rules to normalize.

Here is an example. For the equation:

$$\textsf{evaluate}[\![E_1 + E_2]\!] = (\textsf{evaluate}[\![E_1]\!] \textsf{ and evaluate}[\![E_2]\!]) \textsf{ then add}$$

The derivation goes:

$$\begin{array}{l}
\mathbf{T}_{\textsf{evaluate}\,[\![E_1+E_2]\!]} \\
= \textsf{typing}[\![E_1 + E_2]\!] \\
= \mathbf{T}_{(\textsf{evaluate}\,[\![E_1]\!]\ \textsf{and evaluate}\,[\![E_2]\!])\ \textsf{then add}} \\
= \lambda d.\mathbf{T}_{\textsf{add}}(\mathbf{T}_{\textsf{evaluate}\,[\![E_1]\!]\ \textsf{and evaluate}\,[\![E_2]\!]}(d)) \\
= \lambda d.\mathbf{T}_{\textsf{add}}(\mathbf{T}_{\textsf{evaluate}\,[\![E_1]\!]}(d) \times \mathbf{T}_{\textsf{evaluate}\,[\![E_2]\!]}(d)) \\
= \lambda d.\mathbf{T}_{\textsf{add}}(\textsf{typing}[\![E_1]\!](d) \times \textsf{typing}[\![E_2]\!](d)) \\
= \lambda d.(\lambda t.\ \mathit{if}\ \mathit{fst}(t) \leq \mathit{real\ and\ snd}(t) \leq \mathit{real\ then\ fst}(t) \sqcup \mathit{snd}(t)) \\
\qquad\qquad\qquad\qquad (\textsf{typing}[\![E_1]\!](d) \times \textsf{typing}[\![E_2]\!](d)) \\
\Rightarrow \lambda d.\ \mathit{if}\ \mathit{fst}(\textsf{typing}[\![E_1]\!](d) \times \textsf{typing}[\![E_2]\!](d)) \leq \mathit{real\ and} \\
\qquad \mathit{snd}(\textsf{typing}[\![E_1]\!](d) \times \textsf{typing}[\![E_2]\!](d)) \leq \mathit{real} \\
\quad \mathit{then\ fst}(\textsf{typing}[\![E_1]\!](d) \times \textsf{typing}[\![E_2]\!](d)) \sqcup \mathit{snd}(\textsf{typing}[\![E_1]\!](d) \times \textsf{typing}[\![E_2]\!](d)) \\
\Rightarrow^* \lambda d.\ \mathit{if}\ \textsf{typing}[\![E_1]\!](d) \leq \mathit{real\ and}\ \textsf{typing}[\![E_2]\!](d) \leq \mathit{real} \\
\quad \mathit{then}\ \textsf{typing}[\![E_1]\!](d) \sqcup \textsf{typing}[\![E_2]\!](d)
\end{array}$$

3.2 Translation into Inference Rules

We can simply translate a semantic equation's typing function into an inference rule. A first step is to rewrite the typing function so that occurrences of phrases **typing** $[\![E_i]\!](T)$ have explicit names. This is done by the rewriting rule:

$\lambda I.\mathcal{C}[\textsf{typing}\ [\![E_i]\!](A)] \Rightarrow \lambda I.\ \mathit{let}\ I' = \textsf{typing}\ [\![E_i]\!](A)\ \mathit{in}\ \mathcal{C}[I']$

where $\mathcal{C}[\ ]$ is a context,

A contains no occurrences of any **typing** $[\![E_j]\!]$-phrases, and

I' is a fresh identifier

Here is the derivation of the new form of typing function for the example from the previous section:

$$
\begin{aligned}
\textsf{typing}\,[\![\mathrm{E}_1 + \mathrm{E}_2]\!] = {} & \lambda d.\mathit{if}\ \textsf{typing}\,[\![\mathrm{E}_1]\!](d) \leq \mathit{real}\ \mathit{and}\ \textsf{typing}\,[\![\mathrm{E}_2]\!](d) \leq \mathit{real} \\
& \qquad \mathit{then}\ \textsf{typing}\,[\![\mathrm{E}_1]\!](d) \sqcup \textsf{typing}\,[\![\mathrm{E}_2]\!](d) \\
\Rightarrow {} & \lambda d.\mathit{let}\ t_1 = \textsf{typing}\,[\![\mathrm{E}_1]\!](d) \\
& \qquad \mathit{in}\ \mathit{if}\ t_1 \leq \mathit{real}\ \mathit{and}\ \textsf{typing}\,[\![\mathrm{E}_2]\!](d) \leq \mathit{real} \\
& \qquad\quad \mathit{then}\ t_1 \sqcup \textsf{typing}[\![\mathrm{E}_2]\!](d) \\
\Rightarrow {} & \lambda d.\mathit{let}\ t_1 = \textsf{typing}\,[\![\mathrm{E}_1]\!](d)\ \mathit{and} \\
& \qquad\quad t_2 = \textsf{typing}\,[\![\mathrm{E}_2]\!](d) \\
& \qquad \mathit{in}\ \mathit{if}\ t_1 \leq \mathit{real}\ \mathit{and}\ t_2 \leq \mathit{real}\ \mathit{then}\ t_1 \sqcup t_2
\end{aligned}
$$

The general form of typing function is now:

$$
\begin{aligned}
\textsf{typing}\,[\![\mathrm{op}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n]\!] = \lambda t.\ & \mathit{let}\ t_1 = \textsf{typing}\,[\![\mathrm{E}_1]\!](t'_1)\ \mathit{and} \\
& \qquad \vdots \qquad\quad \vdots \\
& \qquad t_n = \textsf{typing}\,[\![\mathrm{E}_n]\!](t'_n) \\
& \mathit{in}\ \ \mathit{if}\ C\ \mathit{then}\ A
\end{aligned}
$$

where C and A contain no occurrences of any $\textsf{typing}[\![\mathrm{E}_i]\!]$. This format can always be obtained, since the normalization of the original typing function removed all nested λ-abstractions, hence there is no danger in violating binding scopes by moving a $\textsf{typing}[\![E_i]\!](A)$ to the front of a typing function.

Now, the typing function is simply reformatted into the rule:

$$
\frac{t'_1 \vdash \mathrm{E}_1 : t_1 \quad \cdots \quad t'_n \vdash \mathrm{E}_n : t_n}{t \vdash \mathrm{op}\ \mathrm{E}_1 \ldots \mathrm{E}_n : A} \qquad \mathit{if}\, C
$$

which is the typing rule for "op $\mathrm{E}_1\ \ldots\ \mathrm{E}_n$".

In the case of the above example, we find that the typing rule for addition is:

$$
\frac{d \vdash \mathrm{E}_1 : t_1 \qquad d \vdash \mathrm{E}_2 : t_2}{d \vdash \mathrm{E}_1 {+} \mathrm{E}_2 : t_1 \sqcup t_2} \qquad \mathit{if}\ t_1 \leq \mathit{real}\ \mathit{and}\ t_2 \leq \mathit{real}
$$

We can reformat the rule more attractively by moving the constraints on t_1 and t_2 to the antecedents:

$$
\frac{d \vdash \mathrm{E}_1 : t_1 {\leq} \mathit{real} \qquad d \vdash \mathrm{E}_2 : t_2 {\leq} \mathit{real}}{d \vdash \mathrm{E}_1 {+} \mathrm{E}_2 : t_1 \sqcup t_2}
$$

Since the transformation steps in this and the previous section are purely syntactic in nature, the characterization property of the typing functions is preserved in the inference rules. Thus, not only have we derived static typing rules, we have derived strong typing rules, and indeed, the rules are the "strongest" that they can be, in the sense that they state *exactly* the conditions under which a program phrase will not produce a typing error.

4 Dynamic Semantics

The typing semantics defines a sublanguage of the original language. We should "specialize" the semantics definition to the sublanguage defined by the typing semantics. The result is the dynamic semantics.

In action semantics, each action is a polymorphic function, that is, a collection of monomorphic functions that behave consistently (cf. the definitions in Figure 2). But when actions are composed, not all of the monomorphic functions in an action are needed. For example, the action **copy**, as it appears in the action expression **eqzero then copy**, can be narrowed to a single monomorphic function – the identity map on booleans – since **eqzero** emits only boolean values. A similar phenomenon arises in a language definition: the composition of the actions in a semantics equation limits the domains of the arguments to the actions. For example, the typing rule:

$$\frac{d \vdash \mathrm{E} : t \leq real}{d \vdash \text{is\_zero E} : bool}$$

allows us to specialize the semantic equation for is_zero E to:

$$\textsf{evaluate}\ [\![d \vdash \text{is\_zero E} : bool]\!] = \textsf{evaluate}\ [\![d \vdash \mathrm{E} : t \leq real]\!]\ \textsf{then eqzero}_{\{int,real\}}$$

that is, the semantic equation is specialized to operate on bool-typed phrases, and the action **eqzero** is restricted to a set of just two functions: one that checks integers for zero and one that checks reals for zero. The other functions in the action are discarded.

Specialization proves to be important for compiler construction. In [16, 17], Lee and Pleban propose that actions like **eqzero and copy** should be implemented as code generation routines, and a program is compiled by mapping it through the semantics definition to an action expression. Then, the actions in the action expression translate to target code. Since the meaning of an action is a family of monomorphic functions, the implementation of an action is a table of code generation routines, one for each monomorphic function. For example, the table of code generation routines for the **eqzero** action would be:

integer	⇒	"code to check if fixed point number is zero"
real	⇒	"code to check if floating point number is zero"
bool	⇒	"code to generate exception"
	⋮	

The specialization of **eqzero** in the above semantic equation means that its code generation table need only contain the first two entries.

The specialization step goes as follows: given the typing rule:

$$\frac{t_1 \vdash \mathrm{E}_1 : t_1' \quad \cdots \quad t_n \vdash \mathrm{E}_n : t_n'}{t \vdash \mathrm{op}\ \mathrm{E}_1 \ldots \mathrm{E}_n : t'}$$

for the semantics equation:

$$\textsf{valuate}\ [\![\mathrm{op}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n]\!] = \cdots\ \textsf{valuate}\ [\![\mathrm{E}_1]\!]\ \cdots\ \textsf{valuate}\ [\![\mathrm{E}_n]\!]\ \cdots$$

we label the occurrences of **valuate** $[\![\mathrm{E_i}]\!]$ with t_i, t'_i, giving: **valuate** $[\![t_i \vdash \mathrm{E_i} : t'_i]\!]$. Next, we propagate the t_i, t'_i information throughout the right-hand side of the equation. As a result, every action in the right-hand side is labeled by input-output typing information, which allows us to specialize the actions.

An example shows how this is done. Once again, here is the typing rule for $\mathrm{E_1} + \mathrm{E_2}$:

$$\frac{d \vdash \mathrm{E_1} : t_1 \leq real \quad d \vdash \mathrm{E_2} : t_2 \leq real}{d \vdash \mathrm{E_1}+\mathrm{E_2} : t_1 \sqcup t_2}$$

and its semantic equation:

$$\mathsf{evaluate}\ [\![\mathrm{E_1} + \mathrm{E_2}]\!] = (\mathsf{evaluate}\ [\![\mathrm{E_1}]\!]\ \mathsf{and}\ \mathsf{evaluate}\ [\![\mathrm{E_2}]\!])\ \mathsf{then\ add}$$

If we draw the semantic equation as a tree, then the propagation of typing information can be viewed as a post-order tree traversal algorithm with synthesized and inherited attributes. An action, **a**, that appears as a node in the tree, is of course associated with its typing function, $\mathbf{T}_{\mathsf{a}}$. The input type, i, to the node is an inherited attribute, and $\mathbf{T}_{\mathsf{a}}(i)$ is the synthesized attribute for the node. If we write the inherited attributes and subscripts and the synthesized attributes as superscripts on the tree, we obtain:

$$\mathsf{evaluate}[\![d \vdash \mathrm{E_1} + \mathrm{E_2} : t_1 \sqcup t_2]\!] =$$

$$\begin{tikzcd}
 & & \mathsf{then}_d^{t_1 \sqcup t_2, t_1 \leq real\ and\ t_2 \leq real} \arrow[dl, dash] \arrow[ddr, dash] & \\
 & \mathsf{and}_d^{t_1 \times t_2 \leq real \times real} \arrow[dl, dash] \arrow[d, dash] & & \\
\mathsf{evaluate}[\![d \vdash \mathrm{E_1} : t_1 \leq real]\!] & \mathsf{evaluate}[\![d \vdash \mathrm{E_2} : t_2 \leq real]\!] & & \mathsf{add}_{t_1 \times t_2 \leq real \times real}^{t_1 \sqcup t_2, t_1 \leq real\ and\ t_2 \leq real}
\end{tikzcd}$$

When an inherited attribute, d_i, is passed to a leaf, **evaluate**$[\![d_i \vdash \mathrm{E}_i : t_i]\!]$, we claim that the synthesized attribute for the leaf is t_i. This allows the analysis to proceed throughout the entire tree, even though the value of E_i is unknown.

The linearized version of the above tree is:

$$\begin{aligned}
&\mathsf{evaluate}[\![d \vdash \mathrm{E_1} + \mathrm{E_2} : t_1 \sqcup t_2]\!] = \\
&\qquad (\mathsf{evaluate}[\![d \vdash \mathrm{E_1} : t_1 \leq real]\!]\ \mathsf{and}_d\ \mathsf{evaluate}[\![d \vdash \mathrm{E_2} : t_2 \leq real]\!]) \\
&\qquad \mathsf{then}_d\ \mathsf{add}_{t_1 \times t_2 \leq real \times real}
\end{aligned}$$

where we display the inherited attributes only. As a result of the analysis, we note that the **add** action can be specialized to four entries in its code generation table.

5 An Example Expression Language

We now derive the typing and dynamics semantics for the language in Figure 4.

Typing Functions

$\textsf{typing} : \text{Declaration} \rightarrow \text{Typing-Function}_{\mathrm{D}\rightarrow\mathrm{D}}$

$\quad \textsf{typing}[\![\text{val I} = \text{E}]\!] = \lambda d.\ \{\text{I} : \textsf{typing}[\![\text{E}]\!](d)\}$

$\quad \textsf{typing}[\![\text{D}_1, \text{D}_2]\!] = \lambda d.\ \textit{if have-disjoint-fields}(\textsf{typing}[\![\text{D}_1]\!](d), \textsf{typing}[\![\text{D}_2]\!](d))$
$\qquad\qquad \textit{then}\ \textsf{typing}[\![\text{D}_1]\!](d)@\textsf{typing}[\![\text{D}_2]\!](d)$

$\textsf{typing} : \text{Expression} \rightarrow \text{Typing-Function}_{\mathrm{D}\rightarrow\mathrm{F}}$

$\quad \textsf{typing}[\![\text{N}]\!] = \lambda d.\ \textit{int}$

$\quad \textsf{typing}[\![\text{R}]\!] = \lambda d.\ \textit{real}$

$\quad \textsf{typing}[\![\text{E}_1 + \text{E}_2]\!] = \lambda d.\ \textit{if}\ \textsf{typing}[\![\text{E}_1]\!](d) \leq \textit{real and}\ \textsf{typing}[\![\text{E}_2]\!](d) \leq \textit{real}$
$\qquad\qquad \textit{then}\ \textsf{typing}[\![\text{E}_1]\!](d) \sqcup \textsf{typing}[\![\text{E}_2]\!](d)$

$\quad \textsf{typing}[\![\text{if E}_1 \text{ then E}_2 \text{ else E}_3]\!] = \lambda d.\ \textit{if}\ \textsf{typing}[\![\text{E}_1]\!](d) \leq \textit{bool}$
$\qquad\qquad \textit{then}\ \textsf{typing}[\![\text{E}_2]\!](d) \sqcup \textsf{typing}[\![\text{E}_3]\!](d)$

$\quad \textsf{typing}[\![\text{is\_zero E}]\!] = \lambda d.\ \textit{if}\ \textsf{typing}[\![\text{E}]\!](d) \leq \textit{real then bool}$

$\quad \textsf{typing}[\![\text{I}]\!] = \lambda d.\ \textit{if}\ \{\text{I} : t\} \in d\ \ \textit{then}\ t$

$\quad \textsf{typing}[\![\text{let D in E end}]\!] = \lambda d.\ \textsf{typing}[\![\text{E}]\!](\textsf{typing}[\![D]\!](d)@d)$

Typing (Static) Semantics

Declaration

val I = E : $\dfrac{d \vdash \text{E} : t}{d \vdash \text{val I} = \text{E} : \{\text{I}{:}t\}}$

D₁, D₂ : $\dfrac{d \vdash \text{D}_1 : d_1 \quad d \vdash \text{D}_2 : d_2}{d \vdash \text{D}_1,\text{D}_2 : d_1@d_2}$ $\quad$ *if have-disjoint-fields*(d_1, d_2)

Expression

N : $\dfrac{}{d \vdash \text{N} : \textit{int}}$

R : $\dfrac{}{d \vdash \text{R} : \textit{real}}$

$\text{E}_1 + \text{E}_2$: $\dfrac{d \vdash \text{E}_1 : t_1 \leq \textit{real} \quad d \vdash \text{E}_2 : t_2 \leq \textit{real}}{d \vdash \text{E}_1 + \text{E}_2 : t_1 \sqcup t_2}$

if E_1 then E_2 else E_3 : $\dfrac{d \vdash \text{E}_1 : \textit{bool} \quad d \vdash \text{E}_2 : t_2 \quad d \vdash \text{E}_3 : t_3}{d \vdash \text{if E}_1 \text{ then E}_2 \text{ else E}_3 : t_2 \sqcup t_3}$

is_zero E : $\dfrac{d \vdash \text{E} : t \leq \textit{real}}{d \vdash \text{is\_zero E} : \textit{bool}}$

I : $\dfrac{}{d \vdash \text{I} : t}$ $\quad$ *if* $\{\text{I} : t\} \in d$

let D in E end : $\dfrac{d \vdash \text{D} : d' \quad d'@d \vdash \text{E} : t}{d \vdash \text{let D in E end} : t}$

Specialized (Dynamic) Semantics

declare: Decorated-Declaration $\rightarrow$ Mono-Action$_{\mathrm{D}\rightarrow\mathrm{D}}$

$\mathsf{declare}[\![d \vdash \text{val I} = \text{E} : \{\text{I} : t\}]\!] = \mathsf{evaluate}[\![d \vdash \text{E} : t]\!]\ \mathsf{then}_d\ \mathsf{bind}\ \text{I}_t$

$\mathsf{declare}[\![d \vdash \text{D}_1, \text{D}_2 : d_1@d_2\ \textit{if have-dispoint-fields}(d_1, d_2)]\!] =$
$\quad \mathsf{declare}[\![d \vdash \text{D}_1 : \text{d}_1]\!]\ \mathsf{and}_d\ \mathsf{declare}[\![d \vdash \text{D}_2 : d_2]\!]$

evaluate: Decorated-Expression $\rightarrow$ Mono-Action$_{\mathrm{D}\rightarrow\mathrm{F}}$

$\mathsf{evaluate}[\![d \vdash \text{N} : \textit{int}]\!] = \mathsf{give}([\![\text{N}]\!] : \textit{int})_d$

$\mathsf{evaluate}[\![d \vdash \text{R} : \textit{real}]\!] = \mathsf{give}([\![\text{R}]\!] : \textit{real})_d$

$\mathsf{evaluate}[\![d \vdash \text{E}_1 + \text{E}_2 : t_1 \sqcup t_2]\!] =$
$\quad (\ \mathsf{evaluate}[\![d \vdash \text{E}_1 : t_1 \leq \textit{real}]\!]\ \mathsf{and}_d\ \mathsf{evaluate}[\![d \vdash \text{E}_2 : t_2 \leq \textit{real}]\!]\ )$
$\quad \mathsf{then}_d\ \mathsf{add}_{t_1 \times t_2 \leq real \times real}$

$\mathsf{evaluate}[\![d \vdash \text{if E}_1 \text{ then E}_2 \text{ else E}_3 : t_2 \sqcup t_3]\!] =$
$\quad \mathsf{evaluate}[\![d \vdash \text{E}_1 : \textit{bool}]\!]\ \mathsf{andthen}_d\ (\ \mathsf{evaluate}[\![d \vdash \text{E}_2 : t_2]\!]\ \mathsf{else}_d\ \mathsf{evaluate}[\![d \vdash \text{E}_3 : t_3]\!]\ )$

$\mathsf{evaluate}[\![d \vdash \text{is\_zero E} : \textit{bool}]\!] = \mathsf{evaluate}[\![d \vdash \text{E} : t \leq \textit{real})]\!]\ \mathsf{then}_d\ \mathsf{eqzero}_{t \leq real}$

$\mathsf{evaluate}[\![d \vdash \text{I} : t\ \ \textit{if}\ \{\text{I}{:}t\} \in d]\!] = \mathsf{find}\ \text{I}_d$

$\mathsf{evaluate}[\![d \vdash \text{let D in E end} : t]\!] =$
$\quad (\mathsf{furthermore}_d\ \mathsf{declare}[\![d \vdash \text{D} : d']\!])\ \mathsf{hence}_d\ \mathsf{evaluate}[\![d'@d \vdash \text{E} : t]\!]$

6 Extensions and Future Work .

The result of previous sections can be extended to deal with more complex language features: abstraction and recursive bindings. Higher order constructs can be analyzed:

$$\mathsf{evaluate}\ [\![\text{lam I . E}]\!] = \mathsf{abstract}\ ((\mathsf{furthermore}\ (\mathsf{bind}\ \text{I}))\ \mathsf{then}\ \mathsf{evaluate}\ [\![\text{E}]\!])$$

where **abstract a** converts an action **a** into a functional facet value. Given $\mathbf{T}_{\mathsf{abstract\ a}} = \lambda d.\ t \rightarrow \mathbf{T}_{\mathsf{a}}(t, d)$, the typing function is:

$$\mathsf{typing}\ [\![\text{lam I . E}]\!] = \lambda d.\ t \rightarrow \mathsf{typing}\ [\![\text{E}]\!](\{\text{I} : t\}@d)$$

which gives the rule:

$$\frac{\{\text{I} : t\}@d\ \vdash\ \text{E}\ :\ t'}{d\ \vdash\ \text{lam I . E}\ :\ t \rightarrow t'}$$

This is the typing rule for lambda abstraction in ML [7]. But the typing rule for ML's polymorphic "let" cannot be derived here, since it requires a subphrase to have the quantified type $\forall t.F[t]$, and such types are not included in Figure 1. A facet might include quantified types, cf. page 7 of [2], but we leave this for further exploration.

Recursive definitions can also be handled. Action semantics uses a $\mathsf{recursively}_{\text{I}}$ **a** to define scope of recursive binding of I in action **a**. Thus:

$$\mathsf{evaluate}\ [\![\text{fix I . E}]\!] = \mathsf{recursively}_{\text{I}}\ \mathsf{evaluate}\ [\![\text{E}]\!]$$

where $\mathbf{T}_{\mathsf{recursively}_\mathrm{I}\ \mathsf{a}} = \mathit{fix}\ (\lambda f.\lambda d.\mathbf{T}_{\mathsf{a}}(\{\mathrm{I}\!: f(d)\}@d))$. The typing function for the action semantics equation is: $\mathit{fix}\ F$, where $F = \lambda f.\lambda d.\mathsf{typing}\ [\![\mathrm{E}]\!]\ (\{\mathrm{I}\!: f(d)\}@d)$, provided that we make the facets into pointed cpos. We use the fixed point property to derive the typing rule:

$$t' = (\mathit{fix}\ F)d = \mathit{let}\ t = \mathsf{typing}\ [\![\mathrm{E}]\!]\ (\{\mathrm{I} : t'\}@d)\ \mathit{in}\ t$$

But $t = t'$, and we obtain the expected rule:

$$\frac{\{\mathrm{I}\!: t\}@d\ \vdash\ \mathrm{E}\ :\ t}{d\ \vdash\ \mathrm{fix}\,\mathrm{I}\,.\,\mathrm{E}\ :\ t}$$

We are currently implementing the results in this paper as part of an action semantics-directed prototyping system. The system utilizes tools developed by Brown, Moura, and Watt [4], and future integration of their results with ours is likely.

Acknowledgements

Deryck Brown, Olivier Danvy, Erik Meijer, Peter Mosses and the referees made helpful comments on earlier drafts of this paper.

References

[1] R. Barbuti and A. Martelli. A structured approach to static semantics correctness. *Science of Computer Programming*, 3:279–311, 1983.

[2] Heuk Barendregt and Kees Hermerik. Types in lambda calculi and programming languages. In Neil D. Jones, editor, *ESOP90, Proceedings of European Symposium on Programming*, pages 1–35. Lecture Notes in Computer Science 432, Springer-Verlag, 1990.

[3] D. Bjørner and O.N. Oest, editors. *Towards a formal description of ADA*. Lecture Notes in Computer Science 98. Springer, Berlin, 1980.

[4] Deryck Brown, Hermano Moura, and David Watt. Towards a realistic semantics-directed compiler generator. In *Proceedings of Glasgow Functional Programming Workshop, Isle of Skye, Scotland*, 1991.

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

[6] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, 1991.

[7] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[8] N.G. de Bruijn. *The Mathematical Language AUTOMATH, its usage, and some of its extensions*, pages 29–61. Lecture Notes in Mathematics 125. Springer, Berlin, 1970.

[9] Kyung-Goo Doh. *Action semantics-directed prototyping.* PhD thesis, Kansas State University, Manhattan, Kansas, USA, forthcoming.

[10] J. Donahue. *Complementary Definitions of Programming Language Semantics.* Lecture Notes in Computer Science 42. Springer-Verlag, 1976.

[11] A.P. Ershov. On the essence of compilation. In *Formal Description of Programming Concepts*, pages 391–420. IFIP, North-Holland, Amsterdam, 1978.

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

[13] Susan Even and David A. Schmidt. Type inference for action semantics. In Neil D. Jones, editor, *ESOP90, Proceedings of European Symposium on Programming*, pages 118–133. Lecture Notes in Computer Science 432, Springer-Verlag, 1990.

[14] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[15] Neil D. Jones, Peter Sestoft, and Harald Sondergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Journal of LISP and Symbolic Computation*, 2(1):9–50, 1989.

[16] Peter Lee. *Realistic Compiler Generation.* The MIT Press, Cambridge, Massachusetts, 1989.

[17] Peter Lee and Uwe F. Pleban. A realistic compiler generator based on high-level semantics. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 284–295, 1987.

[18] John C. Mitchell and Robert Harper. The essence of ML. In *POPL88, Proceedings of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 28–46. ACM, 1988.

[19] Margaret Montenyohl and Mitchell Wand. Correct flow analysis in continuation semantics. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 204–218, 1988.

[20] Peter D. Mosses. Abstract semantic algebras! In *Formal Description of Programming Concepts II, Proceedings of the IFIP TC2 Working Conference, Garmisch-Partenkirchen, 1982*, pages 45–72. IFIP, North-Holland, Amsterdam, 1983.

[21] Peter D. Mosses. Unified algebras and action semantics. In *STACS89, Proceedings of Symposium on Theoretical Aspects Computer Science, Paderborn.* No. 349, Lecture Notes in Computer Science, Springer-Verlag, 1989.

[22] Peter D. Mosses. Unified algebras and institutions. In *LICS89, Proceedings of the 4th Annual Symposium on Logic in Computer Science*, pages 304–312. IEEE, 1989.

[23] Peter D. Mosses. Unified algebras and modules. In *POPL89, Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 329–343. ACM, 1989.

[24] Peter D. Mosses. An introduction to action semantics. In *Lecture Notes for the Marktoberdorf Summer School*, 1991.

[25] Peter D. Mosses. *Action Semantics.* Tracts in Theoretical Computer Science. Cambridge University Press, Newton, Massachusetts, 1992.

[26] Peter D. Mosses and David A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III.* IFIP, North-Holland, Amsterdam, 1987.

[27] P. Naur. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[28] Flemming Nielson and Hanne Riis Nielson. Code generation from two level denotational meta-languages. In Neil D. Jones, editor, *Programs as Data Objects*, pages 192–205. Lecture Notes in Computer Science 217, Springer-Verlag, 1986.

[29] Flemming Nielson and Hanne Riis Nielson. Pragmatic aspects of two-level denotational metalanguages. In Neil D. Jones, editor, *ESOP86, Proceedings of European Symposium on Programming*, pages 133–143. Lecture Notes in Computer Science 213, Springer-Verlag, 1986.

[30] John Reynolds. Using category theory to design implicit conversions and generic operators. In Neil D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 211–258. Lecture Notes in Computer Science 94, Springer-Verlag, 1980.

[31] John Reynolds. The essence of Algol. In J. deBakker and J.C. vanVliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[32] John Reynolds. *Semantics as a design tool.* Course lecture notes, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA, 1988.

[33] David A. Schmidt. *Denotational Semantics – A Methodology for Language Development.* Allyn and Bacon, Newton, Massachusetts, 1986.

[34] Robert D. Tennent. *Principles of Programming Languages.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1981.

[35] V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.

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