Electronic Notes in Theoretical Computer Science 82 No. 8 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 21 pages

Alias types for "environment-aware" computations

Ferruccio Damiani [1,3]

Dipartimento di Informatica
Università di Torino
Torino, Italia

Paola Giannini [2,4]

Dipartimento di Informatica
Università del Piemonte Orientale
Alessandria, Italia

Abstract

In previous work with Bono we introduced a calculus for modelling "environment-aware" computations, that is computations that adapt their behavior according to the capabilities of the environment. The calculus is an imperative, object-based language (with extensible objects and primitives for discriminating the presence or absence of attributes of objects) equipped with a small-step operational semantics.

In this paper we define a type and effect system for the calculus. The typing judgements specify, via constraints, the shape of environments which guarantees the correct execution of expressions and the typing rules track the effect of expression evaluation on the environment. The type and effect system is sound w.r.t. the operational semantics of the language.

1 Introduction

The growth of the Internet has popularized scripting languages. In particular JavaScript, see [10], has become popular for scripting Web pages. JavaScript

[1] Partially supported by IST-2001-33477 DART and MIUR Cofin'01 NAPOLI projects. The funding bodies are not responsible for any use that might be made of the results presented here.
[2] Partially supported by IST-2001-33477 DART and MIUR Cofin'02 McTati projects. The funding bodies are not responsible for any use that might be made of the results presented here.
[3] Email: damiani@di.unito.it
[4] Email: giannini@di.unito.it

is a powerful object-based language that can be embedded directly in HTML pages. It allows to create dynamic, interactive applications that runs completely within a Web browser.

The *object-based* paradigm (see [1,8]) is a rather natural choice to model Web scripting since Web browsers represents the content of an HTML page, and the container in which the page is shown, via a set of objects called the Document Object Model (DOM). The DOM defines the building-block objects which make up a Web page, and the characteristic properties which belong to each of those objects. In particular, the object document represents the content of a page, and the container in which such page is shown is represented by the object window. All objects are accessible in JavaScript through these two objects. The displaying attributes of objects are monitored by the browser, and changing some of their values may result in a different appearance of the Web page.

JavaScript is an *imperative language with extensible objects* (see e.g. [5] and [9]): modifying an *attribute* (i.e. a *field* or a *method*) that is not already defined causes the definition of that attribute. Attempting to read the value of an attribute that does not exist results in a special undefined value. In this way, the interpreter can detect the existence of attributes (and objects), and keep going as far as possible. Interestingly enough, by relying on such features, some techniques have been developed to write Web pages with scripting code that can adapt its behavior according to the browser where they are running (see [12]). These features can be used, in a more general setting, to write code that adapts its behavior to the attributes defined for the objects in the environment. We refer to this kind of capability as *environment awareness*.

The flexibility of a language such as JavaScript has, however, a major drawback, that is the extreme difficulty of reasoning about the behavior of programs and of doing any kind of check on their consistency.

In [4] Bono and we introduced an imperative object-based language in which objects are extensible and it is possible to test the existence of attributes of objects in the environment. In this paper we propose a type system for the language of [4]. The type system aims to capture the effects on the objects manipulated. In an imperative setting this may be quite complex due to the need of tracking aliasing. Work on typing for low level languages by Smith, Walker, and Morrisett (see [11,13]) focus on the concept of using *aliasing constraints* on locations for describing the layout of the store that is expected for the correct execution of programs. Changes to the memory layout are tracked in the type system by modifying the constraints accordingly. Anderson *et al.* (see [2]) adapted this technique to δ, an imperative object-based language with extensible objects and delegation introduced by Anderson and Drossopoulou in [3]. In this paper we use the technique of [2] to define a type system for the environment-aware calculus of [4]. Besides the fact that our language does not deal with attribute removal and delegation and that the language δ does not have conditional expressions and primitives for attribute

$$e \in \mathbf{Exp} ::= \mathsf{n} \mid e_1 \ \mathsf{op} \ e_2 \mid e_1; e_2 \mid \mathsf{iszero}(e) \,?\, e_1 : e_2 \mid \mathsf{isdef}(e, a) \,?\, e_1 : e_2 \mid \mathsf{root} \mid \mathsf{self} \mid \langle\rangle \mid e.a \mid e_1 \leftarrow a = e_2 \mid e_1 \Leftarrow a = e_2$$

Fig. 1: Expressions

detection, a main difference between the type system of [2] and our type system is that we consider constraints and rules to track not just *presence*, but also *absence* or *possible presence* of attributes. The idea of having types expressing presence or absence of fields has been already considered by Cardelli and Mitchell (see [7]) in a functional context. A main difference between [7] and our proposal is that we have also constraints expressing that if a given attribute is present then it must have a given type.

The paper is structured as follows. In Section 2 we introduce syntax and operational semantics of the language. In Section 3 we define the type system and in Section 4 we state the soundness result. We conclude by outlining some directions for further work. Some examples are presented in Appendix A.

2 The calculus

In this section we introduce syntax and operational semantics for a core language that allows the definition and manipulation of objects. [5]

2.1 Syntax

The language corresponds to the core part of JavaScript [10]. The main features of the language are:

- it is object based and imperative,
- objects may be extended, and
- there is a primitive for discriminating the presence/absence of attributes of objects (i.e., for detecting the capabilities of the environment).

The only primitive values considered are integers. The syntactic category of *expressions*, defined by the grammar in Fig. 1, is parametric in an infinite set of *attributes names* $a \in \mathbf{A}$. The first two clauses define integer expressions ($\mathsf{n} \in \mathbf{Z}$ ranges over integer literals and op ranges over binary operations on integers). Sequential composition of expressions, $e_1; e_2$, is evaluated from left to right and its value is the value of e_2. There are two kinds of conditional expressions:

[5] The language is a small variant of the one introduced in [4]: to simplify the presentation of the type and effect system (in Section 3) we have integrated the expression isdef, that detects the presence/absence of attributes, with the conditional expression. This simplifies the definition of the typing rules for conditional expressions in which we make assumptions depending on the presence/absence of attributes of objects in the environment.
For simplicity, as in [4], the language does not contain method's parameters and local variables.

$\mathsf{iszero}(e)\,?\,e_1 : e_2$ and $\mathsf{isdef}(e,a)\,?\,e_1 : e_2$. Expressions $\mathsf{iszero}(e)\,?\,e_1 : e_2$ are evaluated by first evaluation e and then either e_1 (when e evaluates to the integer 0) or e_2 (when e evaluates to an integer different from 0). An error occurs when e does not evaluate to an integer. Expressions $\mathsf{isdef}(e,a)\,?\,e_1 : e_2$ allow to discriminate the presence/absence of attibutes of objects (and so to program environment dependent behaviour): the expression e_1 is evaluated if the object denoted by e has the attribute a whereas e_2 is evaluated if the object denoted by e does not have the attribute a. An error occurs when e does not evaluates to an object refererence.

The expression root denotes the top-level object of the environment, whereas self is a metavariable denoting the current object during the execution of a method (outside of method bodies its value is not defined and its evaluation produces an error). The expression $\langle\rangle$ is the empty object and $e.a$ is the selection of the attribute a from the object denoted by the expression e. If a is bound to a *value* (that is either an integer or the address of an object) such value is returned. If a is bound to a *method body* (a non-evaluated expression) such method body is evaluated. An error occurs when e does not denote an object or when the object denoted has not the attribute a.

The expression $e_1 \leftarrow a = e_2$ is the overriding or adding of an attribute to an object depending on the fact that, for the object denoted by e_1, the attribute a is defined or not. The attribute a is set to the value resulting from the evaluation of e_2. Similarly, the expression $e_1 \Leftarrow a = e_2$ sets the attribute a to the method body represented by the (non-evaluated) expression e_2 where the metavariable self has been replaced by the address of the object denoted by e_1.

2.2 Operational semantics

The semantics is presented in the style advocated in [14] (see also [6]). An expression evaluates to a value that can be either an integer or the address of an object. Stores maps addresses to objects. To account for the imperative nature of the language, the reduction relation rewrites in addition to expressions also stores.

Notation. *We write $f \in S \rightarrow_{\mathbf{fin}} S'$ to mean that f is a a partial function from the set S to the set S' with a finite domain, written $\mathbf{Dom}(f)$. Given $f \in S \rightarrow_{\mathbf{fin}} S'$ we write $f(s) = \text{Undf}$ to mean that $s \notin \mathbf{Dom}(f)$, and given $g \in S \rightarrow_{\mathbf{fin}} (S' \rightarrow_{\mathbf{fin}} S'')$ we write $g(s)(s') = \text{Undf}$ to mean that either $s \notin \mathbf{Dom}(g)$ or $s' \notin \mathbf{Dom}(g(s))$.*

The semantics is defined by a reduction relation

$$\longmapsto \subseteq (\mathbf{Stores} \times \mathbf{EExp}) \times ((\mathbf{Stores} \times \mathbf{EExp}) + \{\mathsf{err}\})$$

defined in terms of another reduction relation

$$\longrightarrow \subseteq (\mathbf{Stores} \times \mathbf{Red}) \times ((\mathbf{Stores} \times \mathbf{EExp}) + \{\mathsf{err}\})$$

and evaluation contexts (see Fig. 2). The special term err is used to model run-time errors. The semantic categories involved in the definition of $\longmapsto$ are:

- *Extended expressions*, $e \in \mathbf{EExp}$, defined by extending the grammar of expressions (see Fig. 1) with the clause
$$\mid \iota$$
where $\iota \in \mathbf{I}$ is an *address* (we assume an infinite set $\mathbf{I}$ of addresses).
- *Values*, $v \in \mathbf{Val}(\subseteq \mathbf{EExp}) ::= \mathsf{n} \mid \iota$
- *Objects*, $\mathbf{o} \in \mathbf{O} \stackrel{\Delta}{=} \mathbf{A} \rightarrow_{\mathbf{fin}} \mathbf{EExp}$, i.e., finite mappings from attribute names to extended expressions, denoted by $\langle a_1 = e_1, \ldots, a_n = e_n \rangle$. Given an object $\mathbf{o}$, let $\mathbf{o}\{a := e\}$ denote the object such that $\mathbf{o}\{a := e\}(a) = e$ and $\mathbf{o}\{a := e\}(a') = \mathbf{o}(a')$, for $a' \neq a$.
- *Stores*, $\sigma \in \mathbf{Stores} \stackrel{\Delta}{=} \mathbf{I} \rightarrow_{\mathbf{fin}} \mathbf{O}$, i.e., finite mappings from addresses to objects, denoted by $[\iota_1 \mapsto \mathbf{o}_1, \ldots, \iota_n \mapsto \mathbf{o}_n]$. For every store σ there is an address ι_{root} (the address of the top-level object in σ) such that $\sigma(\iota_{\mathsf{root}}) \neq \text{Undf}$. Let $\sigma\{\iota := \mathbf{o}\}$ denote the store such that $\sigma\{\iota := \mathbf{o}\}(\iota) = \mathbf{o}$ and $\sigma\{\iota := \mathbf{o}\}(\iota') = \sigma(\iota')$, for $\iota' \neq \iota$.
- *Redexes*,
$$r \in \mathbf{Red}(\subseteq \mathbf{EExp}) ::= v_1 \ \mathsf{op} \ v_2 \mid v; e \mid \mathsf{iszero}(v) \,?\, e_1 : e_2 \mid \mathsf{isdef}(v, a) \,?\, e_1 : e_2 \mid \mathsf{root} \mid \mathsf{self} \mid \langle\rangle \mid v.a \mid v_1 \leftarrow a = v_2 \mid v \Leftarrow a = e$$
- *Evaluation contexts*,
$$C \in \mathbf{C} ::= [\,] \mid C \ \mathsf{op} \ e \mid v \ \mathsf{op} \ C \mid C; e \mid \mathsf{iszero}(C) \,?\, e_1 : e_2 \mid \mathsf{isdef}(C, a) \,?\, e_1 : e_2 \mid C.a \mid C \leftarrow a = e \mid v \leftarrow a = C \mid C \Leftarrow a = e$$

For every binary operation on integers op, there is a ternary relation $\underline{\mathsf{op}} \subseteq \mathbf{Z} \times \mathbf{Z} \times \mathbf{Z}$ such that: $(\mathsf{n}_1, \mathsf{n}_2, \mathsf{n}) \in \underline{\mathsf{op}}$ if and only if n is the value of $\mathsf{n}_1 \ \mathsf{op} \ \mathsf{n}_2$. To simplify the presentation we assume that, for all operations op and for all $\mathsf{n}_1, \mathsf{n}_2 \in \mathbf{Z}$, there exists $\mathsf{n} \in \mathbf{Z}$ such that $(\mathsf{n}_1, \mathsf{n}_2, \mathsf{n}) \in \underline{\mathsf{op}}$.[6] The *substitution of the free occurrences of* self *in the extended expression* e *by the address* ι, is defined in the usual way (in a method definition expression $e_1 \Leftarrow a = e_2$, the occurrences of self in e_2 are bound).

The transition system rewrites *configurations* that are pairs (σ, e) where e is an extended expression. An extended expression e *is closed* if it does not contain free occurrences of self. A store σ *is closed* if, for all ι and a, $\sigma(\iota)(a) = e$ implies

- e does not contain free occurrences of self, and
- for all ι' occurring in e, $\sigma(\iota') \neq \text{Undf}$.

A configuration (σ, e) *is closed* if σ and e are closed and, for all ι occurring in

[6] Aritmetic errors, like "division by zero", can be modelled by introducing a new error symbol (indeed, this kind of errors are not interesting for our purposes, since they would not be prevented by static typing).

e, $\sigma(\iota) \neq \text{Undf}$. In the following we will consider only closed configurations.

The reduction rules of Fig. 2 are mostly self-explanatory. Notice only that in the rule ($\mathsf{att}_{\Leftarrow}$) the metavariable self is substituted in e with the address of the object to which the method is added.

The relations $\longrightarrow$ and $\longmapsto$ are deterministic (modulo renaming of addresses introduced by rule ($\mathsf{obj}_{\langle\rangle}$)) and enjoy the progress property.

Lemma 2.1 *(Progress for $\longrightarrow$) Let $(\sigma, r) \in \mathbf{Stores} \times \mathbf{Red}$ be closed. Then either $(\sigma, r) \longrightarrow \mathsf{err}$ or $(\sigma, r) \longrightarrow (\sigma', e')$ with (σ', e') closed and $\mathbf{Dom}(\sigma) \subseteq \mathbf{Dom}(\sigma')$.*

Lemma 2.2 *(Unique decomposition) Let $e \in \mathbf{EExp}$. Then either $e \in \mathbf{Val}$, or exists (unique) $C \in \mathbf{C}$ and $r \in \mathbf{Red}$ such that $e = C[r]$.*

Proposition 2.3 *(Progress for $\longmapsto$) Let $(\sigma, e) \in \mathbf{Stores} \times \mathbf{EExp}$ be closed. Then either $e \in \mathbf{Val}$, or $(\sigma, e) \longmapsto \mathsf{err}$, or $(\sigma, e) \longmapsto (\sigma', e')$ with (σ', e') closed and $\mathbf{Dom}(\sigma) \subseteq \mathbf{Dom}(\sigma')$.*

(op) $\quad (\sigma, \mathsf{n}_1\ \mathsf{op}\ \mathsf{n}_2) \longrightarrow (\sigma, \mathsf{n}) \quad$ with $(\mathsf{n}_1, \mathsf{n}_2, \mathsf{n}) \in \underline{\mathsf{op}}$

(seq) $\quad (\sigma, v; e) \longrightarrow (\sigma, e)$

($\mathsf{if}_{\mathsf{zero}}$) $\quad (\sigma, \mathsf{iszero}(\mathsf{0})\,?\,e_1 : e_2) \longrightarrow (\sigma, e_1)$

($\mathsf{if}_{\mathsf{notZero}}$) $\quad (\sigma, \mathsf{iszero}(\mathsf{n})\,?\,e_1 : e_2) \longrightarrow (\sigma, e_2) \quad$ with $\mathsf{n} \neq \mathsf{0}$

($\mathsf{if}_{\mathsf{def}}$) $\quad (\sigma, \mathsf{isdef}(\iota, a)\,?\,e_1 : e_2) \longrightarrow (\sigma, e_1) \quad$ with $\sigma(\iota) = \mathbf{o}$ and $\mathbf{o}(a)$ is defined

($\mathsf{if}_{\mathsf{notDef}}$) $\quad (\sigma, \mathsf{isdef}(\iota, a)\,?\,e_1 : e_2) \longrightarrow (\sigma, e_2) \quad$ with $\sigma(\iota) = \mathbf{o}$ and $\mathbf{o}(a)$ is undefined

($\mathsf{obj}_{\mathsf{root}}$) $\quad (\sigma, \mathsf{root}) \longrightarrow (\sigma, \iota_{\mathsf{root}})$

($\mathsf{obj}_{\langle\rangle}$) $\quad (\sigma, \langle\rangle) \longrightarrow (\sigma\{\iota := \langle\rangle\}, \iota) \quad$ with ι fresh

($\mathsf{att}_{.}$) $\quad (\sigma, \iota.a) \longrightarrow (\sigma, e) \quad$ with $\sigma(\iota) = \mathbf{o}$ and $\mathbf{o}(a) = e$

($\mathsf{att}_{\leftarrow}$) $\quad (\sigma, \iota \leftarrow a = v) \longrightarrow (\sigma\{\iota := \mathbf{o}\{a := v\}\}, \iota) \quad$ with $\sigma(\iota) = \mathbf{o}$

($\mathsf{att}_{\Leftarrow}$) $\quad (\sigma, \iota \Leftarrow a = e) \longrightarrow (\sigma\{\iota := \mathbf{o}\{a := e[\iota/\mathsf{self}]\}\}, \iota) \quad$ with $\sigma(\iota) = \mathbf{o}$

(err) $\quad (\sigma, r) \longrightarrow \mathsf{err} \quad$ if no previous rule can be applied

(cnt) $\quad \dfrac{(\sigma, r) \longrightarrow (\sigma', e')}{(\sigma, C[r]) \longmapsto (\sigma', C[e'])} \qquad (\mathsf{cnt}_{\mathrm{ERR}}) \quad \dfrac{(\sigma, r) \longrightarrow \mathsf{err}}{(\sigma, C[r]) \longmapsto \mathsf{err}}$

Fig. 2: Operational semantics: $\longrightarrow$ and $\longmapsto$ reduction rules

3 Typing

In this section we introduce a type and effect system for the language. Expressions may denote integers or object addresses. The type of an object address ι is the address ι itself, and constraints involving addresses describe

the attributes that the corresponding objects should, should not, or might have.

3.1 Types, effects, and constraints

In the following we define the syntactic categories used for the definition of the type and effect system.

- *Value types*, the types for values, defined by the following clauses
$$t \in \mathcal{V} ::= int \mid \iota$$
where $\iota \in \mathbf{I}$.
- *Attribute types*, the types for fields and methods, are defined by:
$$\tau \in \mathcal{T} ::= t \mid (\Gamma, \nu\overrightarrow{\iota}.(t, \phi))$$
Fields have value types t. Methods have types of the shape $(\Gamma, \nu\overrightarrow{\iota}.(t, \phi))$, where: Γ is a *store constraint* (defined below) representing the constraints that the store must satisfy before the execution of the method, t is the type of the result, ϕ is an *effect* (defined below) representing the effect that the evaluation of the expression has on the store, $\overrightarrow{\iota}$ is a sequence of addresses, and the binder ν binds the occurrences of the addresses $\overrightarrow{\iota}$ in the value type t and in the effect ϕ. We require that the set of the free addresses occurring in $\nu\overrightarrow{\iota}.(t, \phi)$ must be a subset of the set of the free addresses occurring in Γ.
- *Attribute constraints*,
$$\alpha \in \mathcal{A} \triangleq \mathcal{T} \cup \{!\} \cup \{\tau? \mid \tau \in \mathcal{T}\}.$$
Given an attribute constraint α of the form τ or $\tau?$, the *underlying type* of α, denoted by $\lceil\alpha\rceil$, is the attribute type τ.
- *Object constraints*,
$$\rho \in \mathcal{R} \triangleq \mathbf{A} \rightarrow_{\mathbf{fin}} \mathcal{A},$$
are finite mappings from addresses to attribute constraints. They specify the shape required for objects: for an attribute a, if $\rho(a) = \text{Undf}$ then nothing is required for the attribute a, otherwise if $\rho(a)$ is
· τ, then the object must have the attribute a of type τ,
· !, then the object must not have the attribute a,
· $\tau?$, then if the object has the attribute a, then it must be of type τ.
Recording the type of attributes that may be present allows to type more expression in presence of the requirement that overriding an attribute must preserve its type. An object constraint is denoted by $[a_1 : \alpha_1, \ldots, a_n : \alpha_n]$. Let $\rho\{a := \alpha\}$ denote the object constraint such that $\rho\{a := \alpha\}(a) = e$ and $\rho\{a := \alpha\}(a') = \rho(a')$, for $a' \neq a$.
- *Store constraints*,
$$\Gamma \in \mathcal{K} \triangleq \mathbf{I} \rightarrow_{\mathbf{fin}} \mathcal{R},$$

are finite mappings from locations to object's constraints. They specify the restrictions on the shape of the objects in the store: for an address ι, if $\Gamma(\iota) = \text{Undf}$ then nothing is required for the object at the address ι, otherwise the object at the address ι must satisfy the object constraint $\Gamma(\iota)$. For all Γ, we assume that $\Gamma(\iota_{\mathsf{root}})$, the constraint corresponds to what is required for the root object, is always defined. A store constrains is denoted by $[\iota_1 \mapsto \rho_1, \ldots, \iota_n \mapsto \rho_n]$. Let $\Gamma\{\iota := \rho\}$ denote the store constraint such that $\Gamma\{\iota := \rho\}(\iota) = \mathbf{o}$ and $\Gamma\{\iota := \rho\}(\iota') = \Gamma(\iota')$, for $\iota' \neq \iota$.

- *Effects* of expression evaluation specify the changes to the store. They can be either the definition of an attribute or the *possible* definition of an attribute. So, even though they have a different meaning, effects can be represented by a subset of store constraints, as follows.

$$\phi \in \mathbf{Eff} \triangleq \mathbf{I} \rightarrow_{\mathbf{fin}} \{\rho \in \mathcal{R} \mid \text{ for all } \ a \in \mathbf{A},\ \rho(a) \neq !\}.$$

Example 3.1 The store constraint

$$\Gamma = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} :!, \mathtt{cpoint} : \iota], \iota \mapsto [\mathtt{xcoord} : \mathit{int}?]].$$

asserts that the top level object root must have an attribute cpoint that is an object (address) which, in case it has the attribute xcoord, then its type is *int*. Moreover, the object root must not have the attribute cp.

The effect $\phi = [\iota \mapsto [\mathtt{x} : \mathit{int}, \mathtt{z} : \mathit{int}?]]$ asserts that if the store contains the object $\mathbf{o}$ at the address ι then $\mathbf{o}$ is modified by associating to the attribute x a value of type *int* and might be modified by associating to the attribute z a value of type *int*. If the store does not contain an object at the address ι then a new object is added. The new object must have an attribute x of type *int* and, in case it has the attribute z, then its type is *int*.

We give the semantics of store constraints by defining when *a store σ satisfies a store constraint* Γ.

Definition 3.2 (i) We say that the *extended expression e has type τ*, and write $\models e : \tau$, if the following holds.
- If $\tau = \mathit{int}$, then $e = \mathsf{n}$,
- if $\tau = \iota$, then $e = \iota$, and
- if $\tau = (\Gamma, \nu\overrightarrow{\iota}.(t, \phi))$, then the typing judgement $\Gamma \vdash \ e \ : \ t \parallel \phi$ can be derived by the rules in Fig.s 3 and 4 (these rules will be explained in Section 3).

(ii) We say that the *object* $\mathbf{o} = \langle a_1 = e_1, \ldots, a_n = e_n \rangle$ *satisfies the object constraint* ρ, and write $\models \mathbf{o} : \rho$, if for all a such that $\rho(a) \neq \text{Undf}$:
- if $\rho(a) = \tau$, then for some i, $1 \leq i \leq n$, it holds that $a = a_i$ and $\models e_i : \tau$;
- if $\rho(a) = \tau?$, then: $a = a_i$ for some i, $1 \leq i \leq n$, implies that $\models e_i : \tau$;
- if $\rho(a) =!$, then for all i, $1 \leq i \leq n$, it holds that $a \neq a_i$.

(iii) We say that *a store σ satisfies the store constraint* Γ, and write $\sigma \models \Gamma$, if for all $\iota \in \mathbf{Dom}(\Gamma)$ it holds that $\models \sigma(\iota) : \Gamma(\iota)$.

Example 3.3 The store $\sigma = [\iota_{\mathsf{root}} \mapsto [\mathtt{cpoint} : \iota, \mathtt{y} : \iota], \iota \mapsto [\mathtt{xcoord} :$

$0, \texttt{ycoord} : 1]]$ satisfies the constraint Γ of Example 3.1. Instead, we have that the store $\sigma' = [\iota_{\mathsf{root}} \mapsto [\texttt{cp} : 0, \texttt{cpoint} : \iota, \texttt{y} : \iota], \iota \mapsto [\texttt{ycoord} : 1]]$ does not satisfy Γ, since cp is defined for ι_{root} in σ'.

In the following we define two partial orders: the first, $\leq$, on constraints, and the second, $\sqsubseteq$, on effects.

In both definition we use the partial order on attribute constraint, $\preceq$, defined by: $\alpha \preceq \alpha'$ if and only if $\alpha \in \{!, \tau, \tau?\}$ and $\alpha' \in \{\alpha, \tau?\}$.

The partial order on constraints, defined below, is such that $\Gamma \leq \Gamma'$ if Γ imposes more limitations on the objects than Γ', so that a store satisfying Γ satisfies also Γ'.

Definition 3.4 (i) Let $\rho \leq \rho'$, if for all a such that $\rho'(a) \neq \text{Undf}$, it holds that $\rho(a) \preceq \rho'(a)$.

(ii) A store constraint Γ *is more specific than* Γ', $\Gamma \leq \Gamma'$, if for all ι such that $\Gamma'(\iota) \neq \text{Undf}$, it holds that $\Gamma(\iota) \leq \Gamma'(\iota)$.

It is easy to see that: if $\Gamma \leq \Gamma'$, then $\mathbf{Dom}(\Gamma) \supseteq \mathbf{Dom}(\Gamma')$ and, for all ι, $\mathbf{Dom}(\Gamma(\iota)) \supseteq \mathbf{Dom}(\Gamma'(\iota))$.

Example 3.5 Take the constraint

$$\Gamma' = [\iota_{\mathsf{root}} \mapsto [\texttt{cp} : \iota'?], \iota \mapsto [\texttt{xcoord} : \mathit{int}?]]$$

and the constraint $\Gamma$of Example 3.1, it holds that $\Gamma \leq \Gamma'$.

Proposition 3.6 (i) *If* $\rho \leq \rho'$*, then* $\models \mathbf{o} : \rho$ *implies* $\models \mathbf{o} : \rho'$.

(ii) *If* $\Gamma \leq \Gamma'$*, then* $\sigma \models \Gamma$ *implies* $\sigma \models \Gamma'$.

The partial order on effects, $\sqsubseteq$, is such that $\phi \sqsubseteq \phi'$ if ϕ describes more precisely than ϕ' the changes to the store, that is the addition of attributes.

Definition 3.7 (i) Let $\rho \sqsubseteq \rho'$, if

- for all a such that $\rho(a) \neq \text{Undf}$, it holds that $\rho(a) \preceq \rho'(a)$, and
- for all a such that $\rho(a) = \text{Undf}$, it holds that either $\rho'(a) = \text{Undf}$ or $\rho'(a) = \tau?$, for some τ.

(ii) An effect ϕ is *approximated* by ϕ', $\phi \sqsubseteq \phi'$, if for all ι such that $\phi(\iota) \neq \text{Undf}$, it holds that $\phi(\iota) \sqsubseteq \phi'(\iota)$.

Observe that, if $\phi \sqsubseteq \phi'$, then $\mathbf{Dom}(\phi) \subseteq \mathbf{Dom}(\phi')$ and, for all ι, $\mathbf{Dom}(\phi(\iota)) \subseteq \mathbf{Dom}(\phi'(\iota))$.

Example 3.8 Let $\phi = [\iota \mapsto [\texttt{x} : \mathit{int}, \texttt{z} : \mathit{int}?]]$, and $\phi' = [\iota \mapsto [\texttt{x} : \mathit{int}?, \texttt{y} : \mathit{int}?, \texttt{z} : \mathit{int}?]]$ then $\phi \sqsubseteq \phi'$. Let $\phi'' = [\iota \mapsto [\texttt{x} : \mathit{int}?, \texttt{y} : \mathit{int}, \texttt{z} : \mathit{int}?]]$, then $\phi \not\sqsubseteq \phi''$.

For effects ϕ and ϕ' we define the notion of *compatibility* meaning that if ϕ and ϕ' modify the same attribute they modify it with values of the same type. Given two compatible effects ϕ and ϕ', the effect $\phi \circ \phi'$ is the *composition of* ϕ *and* ϕ', which behaves like ϕ' followed by ϕ, and the effect $\phi \sqcup \phi'$ is the *best*

approximation of ϕ and ϕ'. (The best approximation is used in the typing rules for conditional expressions.) The notion of *applicability of an effect ϕ to a store constraint* Γ means that if the effect ϕ modifies an attribute having already some constraint, then it does it without changing the type of the attribute. If the effect ϕ is applicable to the store constraint Γ, then the store constraint $\phi@\Gamma$ is the result of the *application of ϕ to* Γ.

Definition 3.9 (i) The effects ϕ *and* ϕ' *are compatible* if for all ι and a, $\phi(\iota)(a) \neq \text{Undf}$ and $\phi'(\iota)(a) \neq \text{Undf}$, imply $\lceil\phi(\iota)(a)\rceil = \lceil\phi'(\iota)(a)\rceil$

(ii) The *composition of the compatible effects of ϕ and ϕ'*, $\phi \circ \phi'$, is defined by:

$$(\phi \circ \phi')(\iota) = \text{Undf}, \text{ if } \phi(\iota) = \phi'(\iota) = \text{Undf}, \text{ and}$$

$$(\phi \circ \phi')(\iota)(a) = \begin{cases} \text{Undf} & \text{if } \phi(\iota)(a) = \phi'(\iota)(a) = \text{Undf} \\ \tau & \text{if either } \phi(\iota)(a) = \tau \text{ or } \phi'(\iota)(a) = \tau \\ \phi(\iota)(a) & \text{if } \phi(\iota)(a) \neq \text{Undf} \\ \phi'(\iota)(a) & \text{otherwise.} \end{cases}$$

(iii) The *best approximation of the compatible effects of ϕ and ϕ'*, $\phi \sqcup \phi'$, is defined by:

$$(\phi \sqcup \phi')(\iota) = \text{Undf}, \text{ if } \phi(\iota) = \phi'(\iota) = \text{Undf}, \text{ and}$$

$$(\phi \sqcup \phi')(\iota)(a) = \begin{cases} \text{Undf} & \text{if } \phi(\iota)(a) = \phi'(\iota)(a) = \text{Undf} \\ \alpha & \text{if } \phi(\iota)(a) = \phi'(\iota)(a) = \alpha \\ \lceil\phi(\iota)(a)\rceil? & \text{if } \phi(\iota)(a) \neq \text{Undf} \\ \lceil\phi'(\iota)(a)\rceil? & \text{otherwise.} \end{cases}$$

(iv) *The effect ϕ is applicable to the store constraint* Γ if, for all ι and a, if $\phi(\iota)(a) \neq \text{Undf}$, then:

$$\Gamma(\iota)(a) = \text{Undf} \quad \text{or} \quad \Gamma(\iota)(a) =! \quad \text{or} \quad \lceil\Gamma(\iota)(a)\rceil = \lceil\phi(\iota)(a)\rceil.$$

(v) Let ϕ be applicable to Γ. Then the *application of ϕ to* Γ, $\phi@\Gamma$, is defined by:

$$(\phi@\Gamma)(\iota) = \text{Undf}, \text{ if } \phi(\iota) = \Gamma(\iota) = \text{Undf}, \text{ and}$$

$$(\phi@\Gamma)(\iota)(a) = \begin{cases} \tau & \text{if either } \phi(\iota)(a) = \tau \text{ or } \Gamma(\iota)(a) = \tau \\ \phi(\iota)(a) & \text{if } \phi(\iota)(a) \neq \text{Undf and } \Gamma(\iota)(a) \neq \tau \\ \Gamma(\iota)(a) & \text{otherwise.} \end{cases}$$

Example 3.10 Consider $\phi = [\iota \mapsto [\mathtt{x} : \mathit{int}, \mathtt{z} : \mathit{int}?]]$, $\phi' = [\iota \mapsto [\mathtt{y} : \mathit{int}, \mathtt{z} : \mathit{int}]]$, and $\phi'' = [\iota \mapsto [\mathtt{x} : \iota']]$. The effect ϕ is compatible with ϕ' but not with ϕ'', while ϕ' and ϕ'' are compatible. We have $\phi \circ \phi' = [\iota \mapsto [\mathtt{x} : \mathit{int}, \mathtt{y} : \mathit{int}, \mathtt{z} : \mathit{int}]]$ and $\phi \sqcup \phi' = [\iota \mapsto [\mathtt{x} : \mathit{int}?, \mathtt{y} : \mathit{int}?, \mathtt{z} : \mathit{int}?]]$. Both ϕ and ϕ' are applicable to

$\Gamma_0 = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} : \iota], \iota \mapsto [\mathtt{x} : \mathit{int}?, \mathtt{z} :!]]$ returning $\phi@\Gamma_0 = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} : \iota], \iota \mapsto [\mathtt{x} : \mathit{int}, \mathtt{z} : \mathit{int}?]]$ and $\phi'@\Gamma_0 = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} : \iota], \iota \mapsto [\mathtt{x} : \mathit{int}?, \mathtt{y} = \mathit{int}, \mathtt{z} : \mathit{int}]]$. Instead ϕ'' is not applicable to Γ_0.

It is straightforward to check that composition and best approximation of compatible effects are both commutative and associative. Moreover, composition is the greatest lower bound and best approximation is the least upper bound of compatible effects w.r.t. $\sqsubseteq$. Other properties of effects and constraints are summarized by the following propositions.

Proposition 3.11 *If ϕ is compatible with ϕ' then, for all Γ,*

(i) *$\phi \circ \phi'$ is applicable to Γ if and only if ϕ' is applicable to Γ and ϕ is applicable to $\phi'@\Gamma$,*

(ii) *$\phi \circ \phi'$ is the greatest lower bound of ϕ and ϕ' with respect to $\sqsubseteq$,*

(iii) *$(\phi \circ \phi')@\Gamma = \phi@(\phi'@\Gamma)$,*

(iv) *$\phi \sqcup \phi'$ is applicable to Γ if and only if both ϕ and ϕ' are applicable to Γ, and*

(v) *$\phi \sqcup \phi'$ is the least upper bound of ϕ and ϕ' with respect to $\sqsubseteq$.*

Proposition 3.12 *If $\phi \sqsubseteq \phi'$, $\Gamma \leq \Gamma'$, ϕ applicable to Γ, ϕ' applicable to Γ', $\mathbf{Dom}(\phi) = \mathbf{Dom}(\phi')$ and, for all ι, $\mathbf{Dom}(\phi(\iota)) = \mathbf{Dom}(\phi'(\iota))$, then $\phi@\Gamma \leq \phi'@\Gamma'$.*

3.2 Type assignment rules

The typing judgment

$$\Gamma \vdash \ e \ : \ t \parallel \phi$$

(to be read "under the store constraints Γ the expression e has type t and effect ϕ"), assert that: in a store σ satisfying the constraints Γ, evaluating the expression e returns a value of type t and modifies the store to satisfy the constraint $\phi@\Gamma$. The type assignment rules are given in Fig.s 3 and 4.

The rules (int), (root), and (addr) do not have any effect. Numerals have type *int*, the metavariable root has type ι_{root} (the address of the root object of the store) and an address ι has as type ι itself. The rule for the empty object, (empty), has the effect of associating the empty object constraints to a new (fresh) address.

In the rules for binary operators, (op), and sequential composition, (seq), the constraints for the typing of the second expressions are obtained by applying the effects of the evaluations of the first expression to the original constraints. So adding new fields or methods is taken into account. The resulting effect is the composition of the effect of the first expression applied to the one of the second.

The rule for typing a conditional expression testing for zero, ($\text{if}_{\mathsf{iszero}}$), is applicable only if the effects caused by the first and second branches are com-

patible, so that their least upper bound is defined. The rules (IF_{def}), ($\text{IF}_{\mathsf{notDef}}$), and ($\text{IF}_{\mathsf{isdef}}$), cover the possibilities for a conditional expression testing the fact that the attribute a be defined for the type of the expression e (the address ι).[7] In rule (IF_{def}), since a is defined, the resulting type is the one of the expression on the first branch. In rule ($\text{IF}_{\mathsf{notDef}}$), since the constraint asserts that a is absent, the resulting type is the type of the expression on the second branch. In rule ($\text{IF}_{\mathsf{isdef}}$) the store constraint says that a may be defined of type τ, so the store constraint for the first branch says that a is defined and of type τ and the one for the second branch says that a is absent. Note that, since the second branch has a constraint asserting the absence of the attribute a, any override would be possible in ϕ'' on such attribute. So it is required that if ϕ'' modifies a it does it with a value of type τ. In this case, it is safe to conclude that, after the evaluation of the conditional expression, the attribute a is present and has type τ. (As for rule ($\text{IF}_{\mathsf{iszero}}$), the rule is applicable only if the effects caused by the first and second branches are compatible.)

Access to an attribute a is specified in the rules ($\text{ATT}_{.}$) and (CALL). The rule ($\text{ATT}_{.}$) requires that the expression has an address type and that in the store constraint the object associated with this address has the attribute a of value type t (i.e. either *int* or an address ι).

Consider the rule for method call, (CALL). In this case the attribute has a method type $(\Gamma', \nu\overrightarrow{\iota}.(t, \phi'))$. The set of addresses $\overrightarrow{\iota}$ are the new addresses generated during the evaluation of the body of the method. The name of such addresses is not important except that it must be different from the name of any other address. Different calls of the method must correspond to the generation of different addresses, otherwise we would create an *incorrect aliasing* between different addresses. To take this into account we define a *fresh renaming of the addresses in* $\overrightarrow{\iota}$ to be an injective mapping, S, from the addresses $\overrightarrow{\iota}$ into $\mathbf{I}$ such that all the addresses in the codomain of S are fresh (this depends on where the renaming is used). The rule (CALL) requires that the constraints of the call site be less specific that the ones of the definition site (so that, according to Proposition 3.6. ii, every store satisfying the call site constraints will also satisfy the definition site constraints) and that the effect $S(\phi')$ is applicable to the constraints for the call site.

The rule for field definition or override, ($\text{ATT}_{\leftarrow}$), has the effect of adding an attribute a to the object constraint associated to the address resulting from the evaluation of the expression e_1. The constraint for the attribute should specify that the attribute is either not present or it must have the type of e_2. Note that the type of the added/overridden attribute must be a value type t.

The rule for method definition or override, ($\text{ATT}_{\Leftarrow}$), requires that the expression that is the method body be given a type from a store constraint asserting that the associated attribute have the right type (to obtain the extended expression which is the body of the method we substitute self with the

[7] The typing rules for conditionals are illustrated in Example A.1 of the appendix.

address ι of the object being updated). This allows to give a type to recursive methods. Notice also that, the store constraint Γ_0 used to give a type to the body of the method is different from the store constraint Γ. In particular, Γ_0 specifies the constraints that must be satisfied when the method is called, and consistency is checked when typing the method call. This allows to type method bodies that require the presence of attributes that are not yet present, that is to deal with *incomplete objects*, but makes quite difficult the design of an inference algorithm for the system. [8] As for fields we require that either the method be not present or be (possible) present with the same type. [9]

The rule for effect approximation, (appr), allows to approximate the inferred effect, provided that applicability of the effect to store constraint is preserved. This rule is essential to prove the soundness result (Theorem 4.4). [10]

Proposition 3.13 *If* $\Gamma \vdash\ e\ :\ t \parallel \phi$ *then* ϕ *is applicable to* Γ.

The type system has a weakening property.

Proposition 3.14 *Let* $\Gamma' \vdash\ e\ :\ t \parallel \phi$. *If* $\Gamma \leq \Gamma'$, *the addresses in* $\mathbf{Dom}(\phi) - \mathbf{Dom}(\Gamma')$ *do not occur in* Γ, *and* ϕ *is applicable to* Γ, *then* $\Gamma \vdash\ e\ :\ t \parallel \phi$.

4 Soundness of the type system

In this section we state the soundness result with the main lemmas needed for the proof.

We say that a configuration (σ, e) *is typable* if $\Gamma \vdash\ e\ :\ t \parallel \phi$ holds for some Γ, τ, and ϕ such that $\sigma \models \Gamma$.

The typing rules enforce the property that a typable configuration, (σ, e), cannot reduce to err. That is either e is a value or (σ, e) reduces to another typable configuration.

Lemma 4.1 *(Soundness for* $\longrightarrow$*)If* $\Gamma \vdash\ r\ :\ t \parallel \phi$ *(with* $r \in \mathbf{Red}$*) then, for every store* σ *such that* $\sigma \models \Gamma$, *there exists* σ', e', Γ', ϕ'' *and* ϕ' *such that:* $(\sigma, r) \longrightarrow (\sigma', e')$, $\Gamma' \vdash\ e'\ :\ t \parallel \phi''$, $\sigma' \models \Gamma'$, $\phi = \phi'' \circ \phi'$, *and* $\Gamma' \leq \phi'@\Gamma$.

Lemma 4.2 *(Reduction inside Evaluation Contexts)If* $\Gamma \vdash\ C[r]\ :\ t \parallel \phi$ *(with* $C \in \mathbf{C}$ *and* $r \in \mathbf{Red}$*) then, for every store* σ *such that* $\sigma \models \Gamma$, *there exist* t_1, ϕ_1 *and* ϕ_2 *such that*

(i) $\Gamma \vdash\ r\ :\ t_1 \parallel \phi_1$, $\phi = \phi_2 \circ \phi_1$, *and*

(ii) *there exists* σ', e', Γ', ϕ_1'' *and* ϕ_1' *such that:* $(\sigma, r) \longrightarrow (\sigma', e')$, $\Gamma' \vdash\ e'\ :\ t_1 \parallel \phi_1''$, $\sigma' \models \Gamma'$, $\phi_1 = \phi_1'' \circ \phi_1'$, $\Gamma' \leq \phi_1'@\Gamma$, *and* $\Gamma' \vdash\ C[e']\ :\ t \parallel \phi_2 \circ \phi_1''$.

[8] We are exploring various restrictions of rule ($\text{att}_{\Leftarrow}$) and a variant of the language where method definition must contain a typing for the body.

[9] The typing rules for method definition/override and method call are illustrated in Example A.2 of the appendix.

[10] In spite of this, we conjecture that rule (appr) can be removed without reducing the class of typable expressions.

$$(\textsc{int})\ \Gamma \vdash \mathsf{n} \ : \ int \parallel [\,] \qquad (\textsc{root})\ \Gamma \vdash \mathsf{root} \ : \ \iota_{\mathsf{root}} \parallel [\,]$$

$$(\textsc{addr})\ \Gamma \vdash \iota \ : \ \iota \parallel [\,] \qquad (\textsc{empty})\ \frac{\iota \quad \text{fresh}}{\Gamma \vdash \langle\rangle \ : \ \iota \parallel [\iota \mapsto [\,]]}$$

$$(\textsc{op})\ \frac{\Gamma \vdash e_1 \ : \ int \parallel \phi \qquad \phi@\Gamma \vdash e_2 \ : \ int \parallel \phi'}{\Gamma \vdash e_1 \,\mathsf{op}\, e_2 \ : \ int \parallel \phi' \circ \phi} \qquad (\textsc{seq})\ \frac{\Gamma \vdash e_1 \ : \ t \parallel \phi \qquad \phi@\Gamma \vdash e_2 \ : \ t' \parallel \phi'}{\Gamma \vdash e_1; e_2 \ : \ t' \parallel \phi' \circ \phi}$$

$$(\textsc{if}_{\mathsf{iszero}})\ \frac{\begin{array}{l}\Gamma \vdash e_0 \ : \ int \parallel \phi \\ \phi@\Gamma \vdash e_1 \ : \ t \parallel \phi' \\ \phi@\Gamma \vdash e_2 \ : \ t \parallel \phi'' \\ \phi' \text{ and } \phi'' \text{ compatible}\end{array}}{\Gamma \vdash \mathsf{iszero}(e_0)\,?\,e_1 : e_2 \ : \ t \parallel (\phi' \sqcup \phi'') \circ \phi}$$

$$(\textsc{if}_{\mathsf{def}})\ \frac{\begin{array}{l}\Gamma \vdash e \ : \ \iota \parallel \phi \\ (\phi@\Gamma)(\iota)(a) = \tau \\ \phi@\Gamma \vdash e_1 \ : \ t \parallel \phi'\end{array}}{\Gamma \vdash \mathsf{isdef}(e,a)\,?\,e_1 : e_2 \ : \ t \parallel \phi' \circ \phi} \qquad (\textsc{if}_{\mathsf{notDef}})\ \frac{\begin{array}{l}\Gamma \vdash e \ : \ \iota \parallel \phi \\ (\phi@\Gamma)(\iota)(a) =! \\ \phi@\Gamma \vdash e_2 \ : \ t \parallel \phi'\end{array}}{\Gamma \vdash \mathsf{isdef}(e,a)\,?\,e_1 : e_2 \ : \ t \parallel \phi' \circ \phi}$$

$$(\textsc{if}_{\mathsf{isdef}})\ \frac{\begin{array}{l}\Gamma \vdash e \ : \ \iota \parallel \phi \\ \Gamma_0 = \phi@\Gamma \\ \Gamma_0(\iota)(a) = \tau? \\ \Gamma_0\{\iota := \Gamma_0(\iota)\{a := \tau\}\} \vdash e_1 \ : \ t \parallel \phi' \\ \Gamma_0\{\iota := \Gamma_0(\iota)\{a :=!\}\} \vdash e_2 \ : \ t \parallel \phi'' \\ \phi''(\iota)(a) \neq \text{Undf} \ \text{ implies } \ \lceil \phi''(\iota)(a) \rceil = \tau \\ \phi' \text{ and } \phi'' \text{ compatible} \\ \phi''' = \begin{cases} ((\phi' \circ [\iota \mapsto [a : \tau]]) \sqcup \phi'') \circ \phi & \text{if } \phi''(\iota)(a) = \tau \\ (\phi' \sqcup \phi'') \circ \phi & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash \mathsf{isdef}(e,a)\,?\,e_1 : e_2 \ : \ t \parallel \phi'''}$$

Fig. 3: Typing rules - part I

$$(\text{ATT}_{\cdot})\ \frac{\Gamma \vdash\ e\ :\ \iota \parallel \phi \qquad (\phi@\Gamma)(\iota)(a) = t}{\Gamma \vdash\ e.a\ :\ t \parallel \phi}$$

$$(\text{CALL})\ \frac{\begin{array}{l}\Gamma \vdash\ e\ :\ \iota_0 \parallel \phi \\ (\phi@\Gamma)(\iota_0)(a) = (\Gamma', \nu\overrightarrow{\iota}.(t, \phi')) \\ \phi@\Gamma \leq \Gamma' \\ S \text{ fresh renaming of } \overrightarrow{\iota} \\ S(\phi') \text{ applicable to } \phi@\Gamma\end{array}}{\Gamma \vdash\ e.a\ :\ S(t) \parallel (S(\phi')) \circ \phi}$$

$$(\text{ATT}_{\leftarrow})\ \frac{\begin{array}{l}\Gamma \vdash\ e_1\ :\ \iota \parallel \phi \\ \phi@\Gamma \vdash\ e_2\ :\ \tau \parallel \phi' \\ \phi'' = \phi' \circ \phi \\ (\phi''@\Gamma)(\iota)(a) \neq \text{Undf}\ \text{ implies }\ (\phi''@\Gamma)(\iota)(a) \in \{!, \tau, \tau?\}\end{array}}{\Gamma \vdash\ e_1 \leftarrow a = e_2\ :\ \iota \parallel [\iota \mapsto [a : \tau]] \circ \phi''}$$

$$(\text{ATT}_{\Leftarrow})\ \frac{\begin{array}{l}\Gamma \vdash\ e_1\ :\ \iota_0 \parallel \phi \\ \Gamma_0 \vdash\ e_2[\iota_0/\mathsf{self}]\ :\ t_0 \parallel \phi_0 \\ \tau_0 = (\Gamma_0, \nu\overrightarrow{\iota}.(t_0, \phi_0))\ \text{ where }\ \overrightarrow{\iota} = \mathbf{Dom}(\phi_0) - \mathbf{Dom}(\Gamma_0) \\ \Gamma_0(\iota_0)(a) = \tau_0 \\ (\phi@\Gamma)(\iota)(a) \neq \text{Undf}\ \text{ implies }\ (\phi@\Gamma)(\iota_0)(a) \in \{!, \tau_0, \tau_0?\}\end{array}}{\Gamma \vdash\ e_1 \Leftarrow a = e_2\ :\ \iota_0 \parallel [\iota_0 \mapsto [a : \tau_0]] \circ \phi}$$

$$(\text{APPR})\ \frac{\begin{array}{l}\Gamma \vdash\ e\ :\ t \parallel \phi \\ \phi \sqsubseteq \phi' \\ \phi' \text{ applicable to } \Gamma\end{array}}{\Gamma \vdash\ e\ :\ t \parallel \phi'}$$

Fig. 4: Typing rules - part II

Lemma 4.3 *(Soundness for $\longmapsto$)If $\Gamma \vdash\ e\ :\ t \parallel \phi$ then, for every store σ such that $\sigma \models \Gamma$,*

(i) *either $e \in \mathbf{Val}$ and $\models e : t$*

(ii) *or there exists σ', e', Γ', ϕ'' and ϕ' such that: $(\sigma, e) \longmapsto (\sigma', e')$, $\Gamma' \vdash e'\ :\ t \parallel \phi''$, $\sigma' \models \Gamma'$, $\phi = \phi'' \circ \phi'$, and $\Gamma' \leq \phi'@\Gamma$.*

We can now state the soundness result.

Theorem 4.4 *If $\Gamma \vdash\ e\ :\ t \parallel \phi$ then, for every store σ such that $\sigma \models \Gamma$, the configuration (σ, e) cannot reduce to* err.

Conclusions and Future Work

In this paper we introduced a type and effect system for an imperative object-based calculus with extensible objects that models some of the features of Web-oriented scripting languages, such as JavaScript [10]. The main novelty in the system is the use of alias constraints tracking not only the definition of attributes but also their absence or possible presence (with a given type).

Future work includes adding an explicit notion of subtyping, and extending the type and effect system to a language including primitives for attribute removal and/or delegation (see e.g. [3] and [2]). We are also planning the definition of an inference algorithm for some restriction of the system.

Acknowledgments

We thank Mariangiola Dezani, Viviana Bono, Christopher Anderson, Sophia Drossopoulou, and the anonymous WOOD referees for useful comments and criticisms.

References

[1] Abadi, M. and L. Cardelli, "A Theory of Objects," Monographs in Computer Science, Springer, 1996.

[2] Anderson, C., F. Barbanera, M. Dezani-Ciancaglini and S. Drossopoulou, *Can addresses be types? (a case study: objects with delegation)*, in: *Proc. of WOOD'03*, ENTCS **82.8** (2003).

[3] Anderson, C. and S. Drossupoulou, *δ an imperative object based calculus* (2002), presented at the workshop USE, Malaga.

[4] Bono, V., F. Damiani and P. Giannini, *A calculus for "environment aware" computations*, in: *Proc. of FWAN'02*, ENTCS **66.3** (2002).

[5] Bono, V. and K. Fisher, *An Imperative, First-Order Calculus with Object Extension*, in: *Proc. of ECOOP'98*, LNCS **1445**, 1998, pp. 462–497, a preliminary version already appeared in Proc. of 5th Annual FOOL Workshop.

[6] Calcagno, C., E. Moggi and T. Sheard, *Closed Types for a Safe Imperative MetaML*, Journal of Functional Programming (TO APPEAR).

[7] Cardelli, L. and J. Mitchell, *Operations on records*, in: C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, Foundations of Computing Series, The MIT Press, Cambridge, MA, 1994 pp. 295–350.

[8] Fisher, K., F. Honsell and J. C. Mitchell, *A Lambda Calculus of Objects and Method Specialization*, Nordic Journal of Computing **1** (1994), pp. 3–37, a preliminary version appeared in *Proc. of IEEE Symp. LICS'93*.

[9] Fisher, K. and J. Mitchell, *On the relationship between classes, objects, and data abstraction*, Theory and Practice of Object Systems **4(1)** (1998), pp. 3–25.

[10] Flannanghan, D., "JavaScript: The definitive guide," O'Reilly, 1999.

[11] Smith, F., D. Walker and G. Morrisett, *Alias types*, in: *Proceedings of ESOP'00*, LNCS **1782** (2000), pp. 366–381.

[12] Steinman, D., *The Dynamic Duo - Cross Browser Dynamic HTML. Tutorial available at* http://www.dansteinman.com/dynduo/.

[13] Walker, D. and G. Morrisett, *Alias types for recursive data structures*, in: *Worshop on Types in Compilation*, 2000.

[14] Wight, K. and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation (1994).

A Some examples

In this section we give two examples of the application of the type system. The first example is an expression whose behavior depends on the attributes defined for the objects in the environment, and is meant to illustrates the typing rules for conditionals.

Example A.1 Consider the following expressions

- $e_1 = \mathsf{root.cp} \leftarrow \mathtt{x} = \mathsf{root.cp.x} + 1$
- $e_2 = \mathsf{isdef}(\mathsf{root.cp}, \mathtt{x})\,?\,e_1 : \mathsf{root.cp}$
- $e'_1 = \mathsf{root.cpoint} \leftarrow \mathtt{xcoord} = \mathsf{root.cpoint.xcoord} + 1$
- $e'_2 = \mathsf{isdef}(\mathsf{root.cpoint}, \mathtt{xcoord})\,?\,e'_1 : \mathsf{root.cpoint}$
- $e'_3 = \mathsf{isdef}(\mathsf{root}, \mathtt{cpoint})\,?\,(e'_2; 0) : 0$
- $e_0 = \mathsf{isdef}(\mathsf{root}, \mathtt{cp})\,?\,(e_2; 0) : e'_3$

The expression e is typable w.r.t. various store constraints. We, now, present the proof that e is typable w.r.t.

$$\Gamma_0 = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} : \iota_1?, \mathtt{cpoint} : \iota_2?], \iota_1 \mapsto [\mathtt{x} : \mathit{int}?], \iota_2 \mapsto [\mathtt{xcoord} : \mathit{int}?]]$$

To type the first branch of e, which is the expression $(e_2; 0)$, take the store constraints:

- $\Gamma_1 = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} : \iota_1, \mathtt{cpoint} : \iota_2?], \iota_1 \mapsto [\mathtt{x} : \mathit{int}?], \iota_2 \mapsto [\mathtt{xcoord} : \mathit{int}?]]$,
- $\Gamma_2 = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} : \iota_1, \mathtt{cpoint} : \iota_2?], \iota_1 \mapsto [\mathtt{x} : \mathit{int}], \iota_2 \mapsto [\mathtt{xcoord} : \mathit{int}?]]$, and
- $\Gamma_3 = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} : \iota_1, \mathtt{cpoint} : \iota_2?], \iota_1 \mapsto [\mathtt{x} :! \,], \iota_2 \mapsto [\mathtt{xcoord} : \mathit{int}?]]$.

Let Δ be the derivation:

$$\frac{\Gamma_2 \vdash \mathsf{root} \;:\; \iota_{\mathsf{root}} \,\|\, [\,] \qquad \Gamma_2(\iota_{\mathsf{root}})(\mathtt{cp}) = \iota_1}{\Gamma_2 \vdash \mathsf{root.cp} \;:\; \iota_1 \,\|\, [\,]}\ (\textsc{att}_{\cdot})$$

we have that

$$\dfrac{\dfrac{\dfrac{\Delta}{\Gamma_2(\iota_1)(\mathtt{x}) = int}\ (\text{ATT}_\cdot)}{\Gamma_2 \vdash \ \mathsf{root.cp.x}\ :\ int \parallel [\,]} \qquad \Gamma_2 \vdash\ 1\ :\ int \parallel [\,]}{\Gamma_2 \vdash\ \mathsf{root.cp.x} + 1\ :\ int \parallel [\,]}\ (\text{OP}) \qquad \Delta}{\Gamma_2 \vdash\ e_1\ :\ \iota_1 \parallel \phi_1}\ (\text{ATT}_{\leftarrow})$$

where $\phi_1 = [\iota_1 \mapsto [\mathtt{x} : int]]$. Substituting Γ_1 or Γ_3 for Γ_2 in Δ we can also prove both $\Gamma_1 \vdash\ \mathsf{root.cp}\ :\ \iota_1 \parallel [\,]$ and $\Gamma_3 \vdash\ \mathsf{root.cp}\ :\ \iota_1 \parallel [\,]$. Therefore,

$$\dfrac{\begin{array}{l}\Gamma_1 \vdash\ \mathsf{root.cp}\ :\ \iota_1 \parallel [\,] \\ \Gamma_1(\iota_1)(\mathtt{x}) = int? \\ \Gamma_2 = (\Gamma_1\{\iota_1 := [\mathtt{x} : int]\}) \vdash\ e_1\ :\ \iota_1 \parallel \phi_1 \\ \Gamma_3 = (\Gamma_1\{\iota_1 := [\mathtt{x} :! \,]\}) \vdash\ \mathsf{root.cp}\ :\ \iota_1 \parallel [\,] \\ \phi_1 \text{ and } [\,] \text{ compatible}\end{array}}{\Gamma_1 \vdash\ e_2\ :\ \iota_1 \parallel \phi_2}\ (\text{IF}_{\mathsf{isdef}})$$

where $\phi_2 = [\iota_1 \mapsto [\mathtt{x} : int?]] = (\phi_1 \sqcup [\,]) \circ [\,]$.
Using the previous derivations we get

$$\dfrac{\begin{array}{l}\Gamma_1 \vdash\ e_2\ :\ \iota_1 \parallel \phi_2 \\ \phi_2@\Gamma_1 = \Gamma_1 \vdash\ \mathtt{0}\ :\ int \parallel [\,]\end{array}}{\Gamma_1 \vdash\ e_2; \mathtt{0}\ :\ int \parallel \phi_2}\ (\text{SEQ})$$

For the second branch of e, which is the expression e_2', taking

- $\Gamma_1' = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} :!, \mathtt{cpoint} : \iota_2], \iota_1 \mapsto [\mathtt{x} : int?], \iota_2 \mapsto [\mathtt{xcoord} : int?]]$,
- $\Gamma_2' = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} :!, \mathtt{cpoint} : \iota_2], \iota_1 \mapsto [\mathtt{x} : int?], \iota_2 \mapsto [\mathtt{xcoord} : int]]$, and
- $\Gamma_3' = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} :!, \mathtt{cpoint} : \iota_2], \iota_1 \mapsto [\mathtt{x} : int?], \iota_2 \mapsto [\mathtt{xcoord} :! \,]]$,

in a similar way we can derive

$$\Gamma_1' \vdash\ e_2'\ :\ \iota_2 \parallel \phi_2'$$

where $\phi_2' = [\iota_2 \mapsto [\mathtt{xcoord} : int?]]$, and so also

$$\Gamma_1' \vdash\ e_2'; \mathtt{0}\ :\ int \parallel \phi_2'.$$

Now, let

$$\Gamma_0' = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} :!, \mathtt{cpoint} : \iota_2?], \iota_1 \mapsto [\mathtt{x} : int?], \iota_2 \mapsto [\mathtt{xcoord} : int?]].$$

We have that

$$\frac{\begin{array}{l}\Gamma'_0 \vdash \ \mathsf{root} \ : \ \iota_{\mathsf{root}} \parallel [\,] \\ \Gamma'_0(\iota_{\mathsf{root}})(\mathtt{cpoint}) = \iota_2? \\ \Gamma'_1 = (\Gamma'_0\{\iota_{\mathsf{root}} := [\mathtt{cp} :!, \mathtt{cpoint} : \iota_2]\}) \vdash \ e'_2; \mathtt{0} \ : \ \mathit{int} \parallel \phi'_2 \\ (\Gamma'_0\{\iota_{\mathsf{root}} := [\mathtt{cp} :!, \mathtt{cpoint} :!\,]\}) \vdash \ \mathtt{0} \ : \ \mathit{int} \parallel [\,] \\ \phi'_2 \text{ and } [\,] \text{ compatible}\end{array}}{\Gamma'_0 \vdash \ e'_3 \ : \ \mathit{int} \parallel \phi'_2} \ (\text{IF}_{\mathsf{isdef}})$$

and we can conclude

$$\frac{\begin{array}{l}\Gamma_0 \vdash \ \mathsf{root} \ : \ \iota_{\mathsf{root}} \parallel [\,] \\ \Gamma_0(\iota_{\mathsf{root}})(\mathtt{cp}) = \iota_1? \\ \Gamma_1 = (\Gamma_0\{\iota_{\mathsf{root}} := [\mathtt{cp} : \iota_1, \mathtt{cpoint} : \iota_2?]\}) \vdash \ e_2; \mathtt{0} \ : \ \mathit{int} \parallel \phi_2 \\ \Gamma'_0 = (\Gamma_0\{\iota_{\mathsf{root}} := [\mathtt{cp} :!, \mathtt{cpoint} : \iota_2?]\}) \vdash \ e'_3 \ : \ \mathit{int} \parallel \phi'_2 \\ \phi_2 \text{ and } \phi'_2 \text{ compatible}\end{array}}{\Gamma_0 \vdash \ e \ : \ \mathit{int} \parallel \phi_0} \ (\text{IF}_{\mathsf{isdef}})$$

where $\phi_0 = [\iota_1 \mapsto [\mathtt{x} : \mathit{int}?], \iota_2 \mapsto [\mathtt{xcoord} : \mathit{int}?]] = (\phi_2 \sqcup \phi'_2) \circ [\,]$. This shows that the expression e is typable from the constraints on the store asserting that: in case the attributes `cp` and/or `cpoint` of the root object are defined they must have a location type, and similarly for the attributes `x`, and `xcoord`. The effects of the evaluation of e will be the possible definition of the attribute `x` of the object which type is ι_1, and of the attribute `xcoord` of the object which type is ι_2.

Among the other store constraints in which e is typable, are the following.

- $\Gamma = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} : \iota_1, \mathtt{cpoint} :!], \iota_1 \mapsto [\mathtt{x} : \mathit{int}]]$, specifying that the root object has the attribute `cp` (containing an object with an attribute `x` containing an integer) and has not the attribute `cpoint`.
- $\Gamma' = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} :!, \mathtt{cpoint} : \iota_2], \iota_2 \mapsto [\mathtt{xcoord} : \mathit{int}]]$, specifying that the root object has the attribute `cpoint` (containing an object with an attribute `xcoord` containing an integer) and has not the attribute `cp`.
- $\Gamma'' = [\iota_{\mathsf{root}} \mapsto [\mathtt{cp} :!, \mathtt{cpoint} :!]]$, specifying that the root object has not the attributes `cp` and `cpoint`.

Note that Γ_0, is a sort of "merge" of Γ and Γ'. Moreover, Γ, Γ', and Γ_0 are not comparable w.r.t. $\leq$.

For what concerns the judgments derivable, we have the following.

- $\Gamma \vdash \ e \ : \ \mathit{int} \parallel [\iota_1 \mapsto [\mathtt{x} : \mathit{int}]]$,
- $\Gamma' \vdash \ e \ : \ \mathit{int} \parallel [\iota_2 \mapsto [\mathtt{xcoord} : \mathit{int}]]$, and
- $\Gamma'' \vdash \ e \ : \ \mathit{int} \parallel [\,]$.

The second example is a simple expression containing the definition of a

method and its call.

Example A.2 Consider the expressions

- $e_1 = \mathsf{self} \leftarrow \mathtt{temp} = \mathsf{self}.\mathtt{x}; \mathsf{self} \leftarrow \mathtt{x} = \mathsf{self}.\mathtt{y}; \mathsf{self} \leftarrow \mathtt{y} = \mathsf{self}.\mathtt{temp}$
- $e_2 = \mathsf{root} \Leftarrow \mathtt{swap} = e_1; \mathsf{root} \leftarrow \mathtt{x} = 0; \mathsf{root} \leftarrow \mathtt{y} = 1$
- $e = e_2; \mathsf{root}.\mathtt{swap}$

Let

- $\Gamma' = [\iota_{\mathsf{root}} \mapsto [\mathtt{swap} : \tau, \mathtt{x} : int, \mathtt{y} : int]]$, and
- $\Gamma_1 = [\iota_{\mathsf{root}} \mapsto [\mathtt{swap} : \tau, \mathtt{x} : int, \mathtt{y} : int, \mathtt{temp} : int]]$,

with $\tau = (\Gamma', \nu\epsilon.(\iota_{\mathsf{root}}, \phi'))$, where ϵ is the empty sequence of addresses, and $\phi' = [\iota_{\mathsf{root}} \mapsto [\mathtt{temp} : int, \mathtt{x} : int, \mathtt{y} : int]]$.

We have that

$$\dfrac{\Gamma' \vdash\ \iota_{\mathsf{root}}\ :\ \iota_{\mathsf{root}} \parallel [\,] \qquad \dfrac{\begin{array}{l}\Gamma' \vdash\ \iota_{\mathsf{root}}\ :\ \iota_{\mathsf{root}} \parallel [\,] \\ \Gamma_1(\iota_{\mathsf{root}})(\mathtt{x}) = int\end{array}}{\Gamma' \vdash\ \iota_{\mathsf{root}}.\mathtt{x}\ :\ int \parallel [\,]}\ (\textsc{att}_{.})}{\Gamma' \vdash\ \iota_{\mathsf{root}} \leftarrow \mathtt{temp} = \iota_{\mathsf{root}}.\mathtt{x}\ :\ \iota_{\mathsf{root}} \parallel [\iota_{\mathsf{root}} \mapsto [\mathtt{temp} : int]]}\ (\textsc{att}_{\leftarrow})$$

Similarly we derive

$$[\iota_{\mathsf{root}} \mapsto [\mathtt{temp} : int]]@\Gamma' = \Gamma_1 \vdash\ \iota_{\mathsf{root}} \leftarrow \mathtt{x} = \iota_{\mathsf{root}}.\mathtt{y}\ :\ \iota_{\mathsf{root}} \parallel [\iota_{\mathsf{root}} \mapsto [\mathtt{x} : int]]$$

and

$$[\iota_{\mathsf{root}} \mapsto [\mathtt{x} : int]]@\Gamma_1 = \Gamma_1 \vdash\ \iota_{\mathsf{root}} \leftarrow \mathtt{y} = \iota_{\mathsf{root}}.\mathtt{temp}\ :\ \iota_{\mathsf{root}} \parallel [\iota_{\mathsf{root}} \mapsto [\mathtt{y} : int]]$$

Applying rule (SEQ) twice we get

$$\Gamma' \vdash\ e_1[\iota_{\mathsf{root}}/\mathsf{self}]\ :\ \iota_{\mathsf{root}} \parallel \phi'$$

So we can apply the rule for method addition/overriding

$$\dfrac{\begin{array}{l}[\iota_{\mathsf{root}} \mapsto [\,]] \vdash\ \mathsf{root}\ :\ \iota_{\mathsf{root}} \parallel [\,] \\ \Gamma' \vdash\ e_1[\iota_{\mathsf{root}}/\mathsf{self}]\ :\ \iota_{\mathsf{root}} \parallel \phi'\end{array}}{[\iota_{\mathsf{root}} \mapsto [\,]] \vdash\ \mathsf{root} \Leftarrow \mathtt{swap} = e_1\ :\ \iota_{\mathsf{root}} \parallel \phi_1}\ (\textsc{att}_{\Leftarrow})$$

where $\phi_1 = [\iota_{\mathsf{root}} \mapsto [\mathtt{swap} : \tau]]$. As we can see we do not require for the attribute $\mathtt{x}$ and $\mathtt{y}$ to be defined when we define $\mathtt{swap}$. The attributes will be required at the time of the call of the method. To type e_2 we apply a (derivable) version of the rule (SEQ) for sequences of three expressions instead of two:

$$\dfrac{\begin{array}{l}[\iota_{\mathsf{root}} \mapsto [\,]] \vdash\ \mathsf{root} \Leftarrow \mathtt{swap} = e_1\ :\ \iota_{\mathsf{root}} \parallel \phi_1 \\ \phi_1@[\iota_{\mathsf{root}} \mapsto [\,]] \vdash\ \mathsf{root} \leftarrow \mathtt{x} = 0\ :\ \iota_{\mathsf{root}} \parallel \phi_2 \\ \phi_2@\phi_1@[\iota_{\mathsf{root}} \mapsto [\,]] \vdash\ \mathsf{root} \leftarrow \mathtt{y} = 1\ :\ \iota_{\mathsf{root}} \parallel \phi_3\end{array}}{[\iota_{\mathsf{root}} \mapsto [\,]] \vdash\ e_2\ :\ \iota_{\mathsf{root}} \parallel \phi_3 \circ \phi_2 \circ \phi_1}\ (\textsc{seq}3)$$

where $\phi_2 = [\iota_{\mathsf{root}} \mapsto [\mathtt{x} : int]]$ and $\phi_3 = [\iota_{\mathsf{root}} \mapsto [\mathtt{y} : int]]$.

Let $\phi = [\iota_{\mathsf{root}} \mapsto [\mathtt{swap} : \tau, \mathtt{x} : \mathit{int}, \mathtt{y} : \mathit{int}]] = \phi_3 \circ \phi_2 \circ \phi_1$ (note that $\Gamma' = \phi@[\iota_{\mathsf{root}} \mapsto [\,]]$). Then:

$$\frac{\begin{array}{l}\Gamma \vdash \ \mathsf{root} \ : \ \iota_{\mathsf{root}} \parallel [\,] \\ \Gamma'(\iota_{\mathsf{root}}(\mathtt{swap}) = \tau = (\Gamma', \nu\epsilon.(\iota_{\mathsf{root}}, \phi')) \\ \Gamma' \leq \Gamma' \\ \phi' \text{ applicable to } \Gamma'\end{array}}{\Gamma' \vdash \ \mathsf{root.swap} \ : \ \iota_{\mathsf{root}} \parallel \phi'} \ (\textsc{call})$$

and

$$\frac{\begin{array}{l}[\iota_{\mathsf{root}} \mapsto [\,]] \vdash \ e_2 \ : \ \iota_{\mathsf{root}} \parallel \phi \\ \phi@[\iota_{\mathsf{root}} \mapsto [\,]] = \Gamma' \vdash \ \mathsf{root.swap} \ : \ \iota_{\mathsf{root}} \parallel \phi'\end{array}}{[\iota_{\mathsf{root}} \mapsto [\,]] \vdash \ e \ : \ \iota_{\mathsf{root}} \parallel \phi' \circ \phi} \ (\textsc{seq})$$

where $\phi' \circ \phi = [\iota_{\mathsf{root}} \mapsto [\mathtt{swap} : \tau, \mathtt{x} : \mathit{int}, \mathtt{y} : \mathit{int}, \mathtt{temp} : \mathit{int}]]$. This shows that for the correct execution of the expression e no constraints on the store are required, and the effect of the execution will be that of adding or overriding the fields `x`, `y`, `temp`, and the method `swap` for the object `root`.