Lucretia - intersection type polymorphism for scripting languages

Lucretia — intersection type polymorphism for scripting languages

Marcin Benke
University of Warsaw*
ben@mimuw.edu.pl

Viviana Bono
Dipartimento di Informatica dell'Università di Torino†
bono@di.unito.it

Aleksy Schubert
University of Warsaw*
alx@mimuw.edu.pl

Scripting code may present maintenance problems in the long run. There is, then, the call for methodologies that make it possible to control the properties of programs written in dynamic languages in an automatic fashion. We introduce Lucretia, a core language with an introspection primitive. Lucretia is equipped with a (retrofitted) static type system based on local updates of types that describe the structure of objects being used. In this way, we deal with one of the most dynamic features of scripting languages, that is, the runtime modification of object interfaces. Judgements in our systems have a Hoare-like shape, as they have a precondition and a postcondition part. Preconditions describe static approximations of the interfaces of visible objects before a certain expression has been executed and postconditions describe them after its execution. The field update operation complicates the issue of aliasing in the system. We cope with it by introducing intersection types in method signatures.

1 Introduction

Dynamic languages optimise the programmer time, rather than the machine time, and are very effective when small programs are constructed [13, 17]. The advantages of the languages that help in development of short programs can be detrimental in the long run. Succinct code, which has clear advantages over short-term programming, gives less information on what a particular portion of code is doing (and figuring this out is critical for software maintenance, see [14, 9]). As a result, productivity of software development can be in certain situations impaired [12]. In particular, strong invariants a programmer can rely on in understanding of statically typed code are no longer valid, e.g., the type of a particular variable can easily change in an uncontrolled way with each function call in the program.

Still, systems that handle complex and critical tasks such as the Swedish pension system [16], developed in Perl, are deployed and maintained. Thus it is desirable to study methodologies which help programmers in understanding their code and keeping it consistent. To this end, *retrofitted* type systems[1] may be an approach to bridge the gap between flexibility and type safety.

Our proposal is a retrofitted type system for a calculus with a reflection primitive. Our type system handles one of the most dynamic features of object-oriented scripting languages, the runtime modification of object interfaces. In particular, the runtime type of an object variable may change in the course of program execution. This feature can be tackled to some extent through the introduction of a single assignment form for local variables. Still, this cannot be applied easily to object fields. On the other hand, the information that statically describes the evolution of the runtime type of a variable cannot be

[*] This work was partly supported by the Polish government grant no N N206 355836.

[†] This work was partly supported by the MIUR PRIN 2010-2011 CINA grant and by the ICT COST Action IC1201 BETTY.

[1] A retrofitted type system is a a type system that was designed after the language. In particular, this is used in the setting of dynamic languages to indicate a static type system flexible enough to accept their most common idioms, that would be ill-typed with a classical type system, but that are run-time correct.

Submitted to:
ITRS 2014

locations	$\mathsf{Loc} \ni$	l	
variables	$\mathsf{Var} \ni$	x,y	$::=$ (identifiers)
value names	$\mathsf{VNames} \ni$	z,w	$::= x \mid l$
field names	$\mathsf{Fnames} \ni$	n,m	$::=$ (identifiers)
constants	$\mathsf{Const}_V \ni$	c	$::=$ (literals)
function value	$\mathsf{FVal} \ni$	v_f	$::= \mathsf{func}(x_1,\cdots,x_n)\{e\}$
values	$\mathsf{Val} \ni$	v	$::= c \mid v_f \mid l$
function expressions		e_f	$::= x \mid v_f$
atomic expressions		a	$::= v \mid z$
expressions	$\mathsf{Expr} \ni$	e	$::= a \mid op_n(a_1,\cdots,a_n)$
			$\mid \mathsf{new} \mid a.n \mid a_1.n = a_2$
			$\mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$
			$\mid \mathsf{if}\ (a)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$
			$\mid e_f(a_1,\cdots,a_n)$
			$\mid \mathsf{ifhasattr}\ (a,n)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$
objects	$\mathsf{Obj} \ni$	o	$::= \{\} \mid \{L_f\}$
fields list		L_f	$::= n\!:\!v \mid n\!:\!v,L$
stores	$\mathsf{Heaps} \ni$	σ	$::= \cdot \mid (l,o)\sigma$

Figure 1: Abstract syntax

just a type in the traditional sense, but must reflect the journey of the runtime type throughout the control flow graph of the program. However, it would be very inconvenient to repeat the structure of the whole control flow graph for each variable in the program. It makes more sense to describe the type of each variable at program points which are statically available and this is the approach we follow in this paper. In our calculus, a variable referring to an object is annotated with a type variable paired with a constraint expressing an approximation (a lower bound) of the actual type of the object. Our type system design draws inspiration from the work on type-and-effect systems [11, 6, 1]. We present our typings in a different manner, i.e., one where an effect is described by two sets of constraints that express type approximations before and after execution of an instruction. The sets of constraints together with the typed expression can be viewed as a triple in a Hoare-style program logic.

An important element of the language design is the way functions (called methods in object-oriented vocabulary) are handled. The function types describe contracts associated with the functions. We obtained a satisfactory level of flexibility of function application due to type polymorphism. We use two kinds of polymorphism here that serve two different purposes. The first one is the parametric polymorphism, similar to the one of System F. Through universal quantifier instantiation we make it possible to adapt the function type to different sets of parameters. The second one is a form of ad-hoc polymorphism obtained through the use of intersection types [3] and its purpose is to provide particular contracts that are for specific aliasing schemes, i.e., one may describe additional possible behaviours of a function that cannot be described by instantiation of a universal type.

2 Overview of the Calculus

The syntax of our calculus is depicted in Figure 1. The elements of the set $\mathsf{VNames} = \mathsf{Var} \cup \mathsf{Loc}$ are called *value names*. The calculus is object-based and our objects are records of pairs *fieldname:value*.

(Let-Propag)	if $\sigma, e_1 \leadsto \sigma', e_1'$ then $\sigma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \leadsto \sigma', \mathsf{let}\ x = e_1'\ \mathsf{in}\ e_2$
(Let-Reduce)	$\sigma, \mathsf{let}\ x = v\ \mathsf{in}\ e \leadsto \sigma, e[x := v]$
(Op-Eval)	$\sigma, op_n(v_1, \cdots, v_n) \leadsto \sigma, \delta_n(op_n, v_1, \cdots, v_n)$
(βv)	$\sigma, \mathsf{func}(x_1, \cdots, x_n)\{e\}(v_1, \cdots, v_n) \leadsto \sigma, e[x_1 := v_1, \cdots, x_n := v_n]$
(If-True)	$\sigma, \mathsf{if}\ (\mathsf{true})\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \leadsto \sigma, e_2$
(If-False)	$\sigma, \mathsf{if}\ (\mathsf{false})\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \leadsto \sigma, e_3$
(Ifhtr-True)	$\sigma, \mathsf{ifhasattr}\ (l, n)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \leadsto \sigma, e_1$ when $a \in \mathrm{dom}(\sigma(l))$
(Ifhtr-False)	$\sigma, \mathsf{ifhasattr}\ (l, n)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \leadsto \sigma, e_2$ when $a \notin \mathrm{dom}(\sigma(l))$
(New)	$\sigma, \mathsf{new} \leadsto (l, \{\})\sigma, l \quad l$ fresh
(SetAttr)	$\sigma, l.n = v \leadsto \sigma[l := \sigma(l)[n := v]], v$
(GetAttr)	$\sigma, l.n \leadsto \sigma, \sigma(l)(n) \quad$ when $n \in \mathrm{dom}(\sigma(l))$

Figure 2: Semantic rules of Lucretia

Moreover, it is imperative, that is, it has side-effects, therefore we have a heap where objects are stored. Methods are modelled by fields containing functions. There is no built-in concept of *self*, but it can be encoded (see the examples in Section 3). Values are either constants, functions, locations (the latter do not appear in source programs, only in the semantics).

Expressions include value names, primitive operation application, an object creation operation, field access, field update, let-assignment, function application, a conditional expression, an introspection-based conditional expression checking if a certain field belongs to an object.

The operational semantics is presented in Figure 2. The construct let is the only possible evaluation context of the calculus, and rule (Let-Propag) takes care of the propagation of the reduction, while (Let-reduce) performs the appropriate substitution of the computed value v, once this is obtained. Rule (Op-Eval) applies the semantical counterpart of the operation symbol to the given arguments. Rule (β_v) is the call-by-value function application. Rules (If-True) and (If-False) are self-documented. Rules (Ifhtr-True) and (Ifhtr-False) check whether a certain field belongs or not to an object allocated in the heap, and choose a computation branch accordingly. Rule (New) allocates a fresh address in the heap. Rule (SetAttr): either adds the field n to the object allocated at location l, initialised with value v, if n does not exist in the object; or updates n with v, otherwise. Rule (GetAttr) extracts the value of the field n from the object at location l, if n belongs to the object. Note that the semantics is deterministic.

The usage of an object field depends on its type, and since the type clearly depends on the computation flow, we need to update the constraints via static analysis of the computation flow; to keep track of the knowledge about the current fieldset, we use judgements which are a combination of usual typing judgements, and Hoare-style triples: $\Psi_1;\Gamma \vdash e : t;\Psi_2$, where Ψ_i are constraint sets representing type information about the objects in expression e, respectively before and after considering the effects of expression. We call them the *precondition* and the *postcondition*. The type information associated with an expression is, then, a combination of two items: a representation of its actual type and a set of constraints on objects in the relevant part of the heap.

New fields can be added dynamically to our objects, moreover any existing field can be assigned with values of different types during the computation, as it happens in dynamic languages (e.g., Python, JavaScript, Ruby). An object type, then, is not fixed once and forever. We decided, therefore, to type an object with a *constrained* type variable, written $X <\# \{\overline{n : q}\}$, describing some type information for the listed fields of an object of type X (we write $\overline{\alpha}$ for a sequence $\alpha_1, \ldots, \alpha_k$).

$$\begin{array}{rlll}
\mathsf{Const}_T \ni & t_b & (\mathsf{int},\mathsf{bool},\mathsf{str},\ldots) & \qquad \mathcal{V}_T \ni X \\
\mathsf{Types}_c \ni & t_c & ::= & t_f \mid t_f \wedge t_c \\
\mathsf{Types}_f \ni & t_f & ::= & [\overline{t};\Psi] \Rightarrow [t;\Psi] \mid \forall X.t_f \\
\mathsf{Types} \ni & t,u & ::= & t_b \mid X \mid t_c \mid t \vee t \\
\mathsf{Types}_a \ni & q & ::= & t \mid \bot \mid t \vee \bot \\
\mathsf{Recs} \ni & r & ::= & \{\overline{n:q}\} \mid \{\} \\
\mathsf{Constr} \ni & \Psi & ::= & X <\# \, r, \Psi \mid \emptyset
\end{array}$$

Figure 3: Types

One group of challenges in the design of the type system is posed by forks and joins in the control flow. Consider, for instance,

$$\mathsf{if}\ (b)\ \mathsf{then}\ x.n = 1\ \mathsf{else}\ x.n = \text{”}\texttt{hello}\text{”}$$

Statically, we do not know whether x has field n of type int or of type string. To keep track of both possibilities, we introduce *union* types: we type x with type X, where $X <\# \{n : \mathsf{int} \vee \mathsf{string}\}$. Another example is $\mathsf{if}\ (b)\ \mathsf{then}\ x.n = 1\ \mathsf{else}\ 0$: statically, we do not know whether x has field n, but if it does, it is of type int. To be able to track the possible absence of a field, we introduce a *bottom* type: we type x with type X, where $X <\# \{n : \mathsf{int} \vee \bot\}$. Moreover, the constraint $X <\# \{n : \bot\}$ means that the field is definitely absent.

Field access is allowed only if the types indicate the field is definitely present; we can then check whether x has field n as in

$$\mathsf{ifhasattr}\ (x,n)\ \mathsf{then}\ x.n + 1\ \mathsf{else}\ 0$$

to decide whether it is possible to access n or not.

We use intersection types to capture possible different aliasing scenarios (cf. Section 3).

2.1 Types

The syntax of types is shown in Figure 3. We use an abbreviation $\{m : u, r\}$ for $\{m : u, \overline{n:q}\}$ where $r = \{\overline{n:q}\}$, $m \notin \overline{n}$. We impose additional, natural restrictions on the shape of the records and constraints. We require that in a record of the form $\{\overline{n:q}\}$ the labels in $\overline{n}$ are unique. For a constraint $\Psi = \overline{X <\# \, r}$ we require that $r \in \mathsf{Recs}$ and that the variables $\overline{X}$ are also unique.

The shape of all types but function types is self-explanatory. A function type is made of: domain information, that is, the type of its arguments and a set of constraints that can be read as preconditions to the function application; and codomain information, the return type and a set of constraints which are the postconditions holding after the function body has been executed.

We say that $m \in \mathrm{dom}(\{\overline{n:q}\})$ when m is an element of $\overline{n}$. Similarly, we say that $Y \in \mathrm{dom}(\Psi)$ when $\Psi = \overline{X <\# \, r}$ and Y is one of the elements of $\overline{X}$. We define the set of free variables $\mathsf{FTV}(\Psi)$ in a set of constraints Ψ so that when $\Psi = X <\# \, r, \Psi'$ we have $X \in \mathsf{FTV}(\Psi)$, $\mathsf{FTV}(r) \subseteq \mathsf{FTV}(\Psi)$ and $\mathsf{FTV}(\Psi') \subseteq \mathsf{FTV}(\Psi)$. Moreover, we consider $\forall$ to be a binding operator so that $\mathsf{FTV}(\forall X.t) = \mathsf{FTV}(t) - \{X\}$.

Judgements are of the form $\Psi_1;\Gamma \vdash e : t;\Psi_2$, where e is an expression, t is a type, Γ is an environment, and Ψ_1 and Ψ_2 are type variable constraint sets, as described earlier.

We use type variable renaming, indicated with θ, to adapt universally quantified types to different situations they can be used in.

Its formal definition follows.

Definition 1 (Renaming) *A bijection $\theta : \mathscr{X} \to \mathscr{Y}$ where $\mathscr{X} \cup \mathscr{Y}$ is a finite subset of* Var *is called* renaming. *We extend it structurally to types, expressions, environments and constraints with avoiding name clashes for bound variables. We use the notation* $\mathrm{dom}(\theta) = \mathscr{X}$ *and* $\mathrm{img}(\theta) = \mathscr{Y}$. *When sequences $\overline{X}, \overline{Y}$ of unique variables have the same length we write $[\overline{X} := \overline{Y}]$ for a renaming θ such that $\theta(X_i) = Y_i$ for $X_i \in \overline{X}$. We assume that $\theta(Z) = Z$ for $Z \notin \overline{X}$. We apply $[\overline{X} := \overline{Y}]$ as a suffix, i.e. $t[\overline{X} := \overline{Y}] = \theta(t)$. We write $A \parallel \theta$ when $A \cap (\mathrm{img}(\theta) - \mathrm{dom}(\theta)) = \emptyset$.*

Observe that these renamings, unlike type instantiation in System F, cannot substitute two universally quantified variables with the same variable. This is an important design choice as we believe that the form of types should not hide other information. The standard convention that makes it possible to glue together two different variables puts on type readers the burden of checking if different uniting schemes do not lead to unexpected situations, that is, unexpected aliasing, in our case.

2.2 Weakening Woes

Since the type information changes with the control flow, a constraint update operation plays a central role in our system. For compositionality, the following "knowledge monotonicity" with respect to the constraint update operation must hold.

Monotonicity principle *For every set of constraints Ψ and derivable judgement $\Psi_1; \Gamma \vdash e : t; \Psi_2$, such that variable names for objects created in e are fresh with respect to Ψ, we can derive*

$$\Psi \hookleftarrow \Psi_1; \Gamma \vdash e : t; \Psi \hookleftarrow \Psi_2$$

Intuitively $\Psi \hookleftarrow \Psi_1$ means the set of constraints Ψ is updated with constraints from Ψ_1; it is formally defined in Figure 5.

We observe that our conditional typing rules must have the same postconditions for the two branches (see rules (if) and rule (ifhttr) in Figure 6). In Hoare logic, equalising branches' postconditions is obtained via weakening, which in our case might be formulated more or less like this:

$$\frac{\Psi_1; \Gamma \vdash e : t; \Psi' \quad \Psi' \preceq_c \Psi}{\Psi_1; \Gamma \vdash e : t; \Psi}$$

where $\Psi' \preceq_c \Psi$ means that Ψ is weaker than Ψ'. We need, however, to be careful that weakening obeys monotonicity, lest the system be unsound (we have the scars to show for it).

One example of weakening pitfall is forgetting a constraint, i.e.:

$$\Psi, X <^{\#} r \preceq_c \Psi$$

Let's say we can infer

$$X <^{\#} \{\}; \Gamma \vdash x.m = 1 : \mathsf{int}; X <^{\#} \{m : \mathsf{int}\}$$

Forgetting the constraint would allow us to infer

$$X <^{\#} \{\}; \Gamma \vdash x.m = 1 : \mathsf{int}; \emptyset$$

while monotonicity with $\Psi = X <^{\#} \{m : \mathsf{str}\}$ requires that

$$X <^{\#} \{m : \mathsf{str}\}; \Gamma \vdash x.m = 1 : \mathsf{int}; X <^{\#} \{m : \mathsf{str}\}$$

which is not sound.

Where an object with some fields is required, an object having these fields and also some others is allowed, according to the Liskov substitution principle [10]. One way of achieving this would be allowing weakening by forgetting fields:

$$X <\# \{m : u, \overline{n : q}\} \preceq_c X <\# \{\overline{n : q}\}$$

Alas, this is not sound either, since it allows to infer $X <\# \{\}; \Gamma \vdash x.m = 1 : \mathsf{int}; X <\# \{\}$ and, by monotonicity,

$$X <\# \{m : \mathsf{str}\}; \Gamma \vdash x.m = 1 : \mathsf{int}; X <\# \{m : \mathsf{str}\}.$$

This hints to the fact that our $\preceq_c$ (defined over relation $<\#$) does not coincide with subtyping. Subtyping (at least in width) is nevertheless essential in an object-oriented setting and we actually permit it in function calls (see the explanation about rule (fapp) in Section 2.3 and examples in Section 3).

Equalising branches' postconditions might be done using union types: if an attribute has type t_1 after one branch, and t_2 after the other, we say it has type $t_1 \vee t_2$. However, we need to take special care; when trying to handle the case where an attribute is set in one branch of the conditional, e.g.,

```
if (b) then x.m = 1 else 0
```

it may be tempting to use a weakening schema similar to

$$\{\overline{n : t}\} \preceq_c \{\overline{n : t}, m : u \vee \bot\} \quad m \notin \overline{n}$$

This turns out to be unsound, too, as shown by the following:

```
func(x) {ifhasattr(x, m) then x.m + 1 else 0}
```

Using the weakening schema above, we can give it the type

$$[X; X <\# \{\}] \Rightarrow [\mathsf{int}; X <\# \{\}]$$

whereas calling this function with an argument containing field $m : \mathsf{str}$ leads to a crash.

Therefore we propose a notion of type weakening as formulated in Figure 4. This allows us to avoid the pitfall presented previously and give the function mentioned there a correct type

$$[X; X <\# \{m : \bot\}] \Rightarrow [\mathsf{int}; X <\# \{m : \bot \vee \mathsf{int}\}]$$

which ensures that the field m is absent from its argument.

2.3 Typing Rules

The typing rules of our system are presented in Figure 6. A freshly created object has no fields, hence the form of rule (new). We impose an injective map from the set of type variables present in the program to memory locations, therefore the type variable needs to be fresh. The consequence is that any relevant type variable occurring in the postcondition, but not in the precondition of a judgement, refers to an object created within the expression under consideration. More precisely, whenever

$$\Psi_1; \Gamma \vdash e : t; \Psi_2, \qquad X \in \mathrm{dom}(\Psi_2) \setminus \mathrm{dom}(\Psi_1, \Gamma)$$

X is the type of an object created within e (or phantom). Then we also know that all its fields not mentioned in the postcondition for X are definitely absent, which is why the rule (bot) is sound.

Rule (acc) governs field access. A field is accessible from an object (value) if the field's type is a type belonging to the set Types. Intuitively, a field can be accessed only if its type does not contain type $\bot$, that is, the field is actually present in the object.

$$\frac{}{q \preceq_c q}\ (\preceq_c \text{rfl}) \qquad\qquad \frac{}{q \preceq_c q \vee q'}\ (\preceq_c \text{r}\vee)$$

$$\frac{}{\{\} \preceq_c \{\}}\ (\preceq_c \text{rrfl}) \qquad\qquad \frac{}{\emptyset \preceq_c \emptyset}\ (\preceq_c \text{crfl})$$

$$\frac{q_1 \preceq_c q_2 \quad q_2 \preceq_c q_3}{q_1 \preceq_c q_3}\ (\preceq_c \text{trns}) \qquad \frac{\begin{array}{c}\Psi_1 \preceq_c \Psi_2 \\ X \notin \mathsf{FTV}(\Psi_1)\end{array}}{\Psi_1 \preceq_c \Psi_2, X <\# r}\ (\preceq_c \text{cevlv})$$

$$\frac{q \preceq_c q' \quad \{\overline{n:u}\} \preceq_c \{\overline{n:u'}\}}{\{b:q,\overline{n:u}\} \preceq_c \{b:q',\overline{n:u'}\}}\ (\preceq_c \text{strct})$$

$$\frac{\begin{array}{c}r_1 \preceq_c r_2 \quad \Psi_1 \preceq_c \Psi_2 \\ X \notin \mathsf{FTV}(\Psi_1,\Psi_2)\end{array}}{\Psi_1, X <\# r_1 \preceq_c \Psi_2, X <\# r_2}\ (\preceq_c \text{cstrct})$$

Figure 4: Order over constraints

Record update

$$\begin{array}{c}
r \hookleftarrow \{\} = r \\
\{a:u,r\} \hookleftarrow \{a:u'\} = \{a:u',r\} \\
\{\overline{n:q}\} \hookleftarrow \{a:u'\} = \{a:u',\overline{n:q}\} \quad \text{if } a \notin \overline{n} \\
r \hookleftarrow \{a:u'',r'\} = (r \hookleftarrow \{a:u''\}) \hookleftarrow r'
\end{array}$$

Constraint update

$$\begin{array}{l}
\Psi \hookleftarrow \emptyset = \Psi \\
\Psi, X <\# r \hookleftarrow X <\# r' = \Psi, X <\# (r \hookleftarrow r') \\
\Psi, X <\# r \hookleftarrow \Psi', X <\# r' = (\Psi \hookleftarrow \Psi'), X <\# (r \hookleftarrow r') \\
\Psi \hookleftarrow \Psi', X <\# r' = (\Psi \hookleftarrow \Psi'), X <\# r' \quad \text{if } X \notin \mathrm{dom}(\Psi)
\end{array}$$

Figure 5: The update operation $\hookleftarrow$.

Rule (updt) describes field update and works whether the field m is already present in the object or not. The postcondition is updated accordingly, by using the operation $\hookleftarrow$ from Figure 5. The constraint related to X in the postcondition will record either the presence of a new field, or the (possible) change of type of an already present fields (notice that most of the rules defining $\hookleftarrow$ are for the propagation of additions/changes and for bookkeeping).

The let instruction provides a form of sequencing and the rule (let) types it accordingly. We use the following notation:

$$\begin{array}{rl}
\mathsf{let}\ x = e;\ es & \mapsto \mathsf{let}\ x = e\ \mathsf{in}\ es \\
e;\ es & \mapsto \mathsf{let}\ \_ = e\ \mathsf{in}\ es
\end{array}$$

Rule (fdcl) types a function declaration. It checks the body against the given preconditions and postconditions, moreover the type is generalised on all possible type variables not appearing in the typing context Γ (in an ML-style). This is done to abstract from the choices of type variable names in types of the objects passed as arguments, as well as objects created in the function body, while protecting the type variables referred to by nonlocal identifiers.

The process of matching formal and actual parameters and preconditions can be seen in the rule (fapp). This rule, given a well-typed function declaration: (i) checks the actual parameters against the formal parameters' types; (ii) checks that the state at the call site (described by Ψ) ensures the callee

$$\Psi;\Gamma,z:t\vdash z:t;\Psi\ (\text{var})\qquad \dfrac{X\notin\mathsf{FTV}(\Psi,\Gamma)}{\Psi;\Gamma\vdash\mathsf{new}:X;X\mathrel{<\!\#}\{\},\Psi}\ (\text{new})$$

$$\Psi;\Gamma\vdash c:t_c;\Psi\ (\text{const})\qquad \dfrac{\begin{array}{c}\Psi_1;\Gamma\vdash e:t;X\mathrel{<\!\#}\{\overline{n:u}\},\Psi_2\\ X\notin\mathsf{FTV}(\Psi_1,\Gamma)\qquad m\notin\overline{n}\end{array}}{\Psi_1;\Gamma\vdash e:t;X\mathrel{<\!\#}\{\overline{n:u},m:\bot\},\Psi_2}\ (\text{bot})$$

$$\dfrac{\begin{array}{c}\Psi;\Gamma\vdash e_1:t_1;\Psi_1\qquad \Psi_1;\Gamma\vdash e_2:t_2;\Psi_2\\ t_1,t_2\preceq_c \mathit{bool}\vee\mathit{int}\vee\mathit{real}\qquad t=t_1\vee t_2\end{array}}{\Psi;\Gamma\vdash +(e_1,e_2):t;\Psi_2}\ (\text{plus})$$

$$\dfrac{\begin{array}{c}\Psi;\Gamma\vdash z:X;\Psi\\ \Psi\ni X\mathrel{<\!\#}\{m:t,\overline{n:q}\}\end{array}}{\Psi;\Gamma\vdash z.m:t;\Psi}\ (\text{acc})\qquad \dfrac{\Psi;\Gamma\vdash z_1:X;\Psi\quad \Psi;\Gamma\vdash z_2:t;\Psi}{\Psi;\Gamma\vdash z_1.m=z_2:t;\Psi\hookleftarrow X\mathrel{<\!\#}\{m:t\}}\ (\text{updt})$$

$$\dfrac{\begin{array}{c}\Psi_1;\Gamma\vdash e_1:t_1;\Psi_2\\ \Psi_2;\Gamma,x:t_1\vdash e_0:t;\Psi_3\end{array}}{\Psi_1;\Gamma\vdash \mathsf{let}\ x=e_1\ \mathsf{in}\ e_0:t;\Psi_3}\ (\text{let})\qquad \dfrac{\begin{array}{c}\Psi;\Gamma\vdash a:\mathit{bool};\Psi\\ \Psi;\Gamma\vdash e_i:t;\Psi'\quad \text{for } i=1,2\end{array}}{\Psi;\Gamma\vdash \mathsf{if}\ (a)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:t;\Psi'}\ (\text{if})$$

$$\dfrac{\Psi;\Gamma\vdash e:t;\Psi_1\quad \vdash\Psi_1\preceq_c\Psi_1'}{\Psi;\Gamma\vdash e:t;\Psi_1'}\ (\text{cnstr}\preceq_c)\qquad \dfrac{\Psi_1;\Gamma\vdash e:t;\Psi_2\quad t\preceq_c t'}{\Psi_1;\Gamma\vdash e:t';\Psi_2}\ (\text{typ}\preceq_c)$$

$$\dfrac{\begin{array}{c}\Psi_s;\Gamma,\overline{x:s}\vdash e:t;\Psi_t\\ \overline{X}=\mathsf{FTV}(\overline{s},\Psi_s,t,\Psi_t)\setminus\mathsf{FTV}(\Gamma)\\ t_f\equiv\forall\overline{X}[\overline{s};\Psi_s]\Rightarrow[t;\Psi_t]\end{array}}{\Psi;\Gamma\vdash\mathsf{func}(\overline{x})\{e\}:t_f;\Psi}\ (\text{fdcl})$$

$$\dfrac{\begin{array}{c}\Psi;\Gamma\vdash e_f:t_f;\Psi\quad t_f\equiv\forall\overline{X}.[\overline{s};\Psi_s]\Rightarrow[r;\Psi_r]\\ \theta:\overline{X}\to\overline{Y}\ \text{is renaming}\qquad \mathsf{FTV}(t_f)\parallel\theta\\ \mathsf{FTV}(\Psi)\cap(\mathrm{dom}(\theta(\Psi_r))-\mathrm{dom}(\theta(\Psi_s)))=\emptyset\\ \Psi;\Gamma\vdash\overline{w}:\theta(\overline{s});\Psi\\ \Psi\preceq_c\Psi\hookleftarrow\theta(\Psi_s)\qquad \mathrm{dom}(\theta(\Psi_s))\subseteq\mathrm{dom}(\Psi)\end{array}}{\Psi;\Gamma\vdash e_f(\overline{w}):\theta(r);\Psi\hookleftarrow\theta_1(\Psi_r)}\ (\text{fapp})$$

$$\dfrac{\begin{array}{c}\Psi;\Gamma\vdash\mathsf{func}(\overline{x})\{e\}:t_1;\Psi\\ \Psi;\Gamma\vdash\mathsf{func}(\overline{x})\{e\}:t_2;\Psi\end{array}}{\Psi;\Gamma\vdash\mathsf{func}(\overline{x})\{e\}:t_1\wedge t_2;\Psi}\ (\text{I}\wedge)\qquad \dfrac{\begin{array}{c}\Psi;\Gamma\vdash a:X;\Psi\\ \Psi[X\leftarrow^{+}\{n\}];\Gamma\vdash e_1:t;\Psi_2\\ \Psi[X\leftarrow^{-}\{n\}];\Gamma\vdash e_2:t;\Psi_2\end{array}}{\Psi;\Gamma\vdash\mathsf{ifhasattr}\ (a,n)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2:t;\Psi_2}\ (\text{ifhttr})$$

$$\dfrac{\Psi_1;\Gamma\vdash z:t_1\wedge t_2;\Psi_2}{\Psi_1;\Gamma\vdash z:t_i;\Psi_2}\ (\text{E}\wedge\text{i})\qquad \dfrac{\begin{array}{c}\Psi;\Gamma\vdash a:X;\Psi\quad \Psi;\Gamma\vdash e_*:t;\Psi_2\\ \Psi[X\leftarrow^{*}\{n\}]=\Psi\qquad *\in\{+,-\}\end{array}}{\Psi;\Gamma\vdash\mathsf{ifhasattr}\ (a,n)\ \mathsf{then}\ e_+\ \mathsf{else}\ e_-:t;\Psi_2}\ (\text{ifhttr}^*)$$

Figure 6: Typing rules

In the following * should be understood as any of $^+$ and $^-$:

$$
\begin{gathered}
(\Psi, X <\!\# \, r)[X \leftarrow^* \{a\}] = \Psi, X <\!\# \, (r[X \leftarrow^* \{a\}]) \\
\Psi[X \leftarrow^* \{a\}] = \Psi \text{ when } X \not\in \mathrm{dom}(\Psi) \\
\{a : t, \overline{n : q}\}[X \leftarrow^+ \{a\}] = \{a : t^+, \overline{n : q}\} \\
\{a : t, \overline{n : q}\}[X \leftarrow^- \{a\}] = \{a : \bot, \overline{n : q}\} \\
(t)^+ = t \text{ for } t \in \mathsf{Types} \qquad (t \vee \bot)^+ = t^+
\end{gathered}
$$

Figure 7: Definiteness update $\Psi[X \leftarrow^+ \{a\}]$ and $\Psi[X \leftarrow^- \{a\}]$.

precondition (Ψ_s). Note that this is expressed in terms of updates, because each declared function can be seen as a state updater, as well as inclusion of domains, because the precondition cannot introduce new type variables (that directly correspond to locations on heap). All checks here are done modulo a renaming θ of type variables establishing a one-to-one correspondence between formal and actual arguments and preconditions, which is formally expressed with the $\|$ operator. The other side condition ensures the type variables chosen for the objects created by the function are fresh. Finally, we stipulate that the result type is the formal result type with type variables renamed according to θ, that adapts the type of the function to the site of the function call. The final state corresponds here to the initial state updated according to the callee postcondition Ψ_t.

Renaming is also an instrument we use to deal with aliasing and works together with intersection types and their related rules, (I$\wedge$) and (E$\wedge$i). See the example in Section 3 for an account on how renaming and intersection types work to deal with aliasing scenarios.

Rule (ifhttr) types an introspection expression against two different sets of constraints, *assuming* the presence (resp. absence) of the attribute n. The rule (ifhttr*) is a special case applicable if presence of n can be determined statically.

3 Expressivity of the System

Let us now look at some examples that illustrate the strength of the type system we propose.

An interesting point is what happens when an object is modified and/or created inside a conditional instruction. We present four examples: one where the same attribute is set in both branches, one where the assignment happens in one branch only, one in which an object is created and assigned to a field in one branch only, and one where object creation happens in both branches. We assume we have a variable called `hasarg` of type `bool` and a variable called `arg` of type `string`.

Setting the same attribute in both branches. Consider the code.

```
let x = new in
// x : X; X <# {}
if (ha) then x.m = a else x.m = "help"
// x : X; X <# {m : string}
```

We can type this example as follows:

$$
\frac{\begin{array}{c}\Psi_1;\Gamma_1 \vdash ha : \mathsf{bool} \\ \Psi_1;\Gamma_1 \vdash x.m = a : \mathsf{string};\Psi_2 \\ \Psi_1;\Gamma_1 \vdash x.m = \text{"}\mathtt{help}\text{"} : \mathsf{string};\Psi_2\end{array}}{\Psi_1;\Gamma_1 \vdash \mathsf{if}\ (ha)\ \mathsf{then}\ x.m = a\ \mathsf{else}\ x.m = \text{"}\mathtt{help}\text{"} : \mathsf{string};\Psi_2}
$$

where

$$\begin{aligned}
\Gamma_0 &= \{ha : bool, a : \mathsf{string}\}\\
\Gamma_1 &= \Gamma_0, x : X\\
\Psi_1 &= X <\!\# \,\{\}\\
\Psi_2 &= X <\!\# \,\{m : \mathsf{string}\}
\end{aligned}$$

Note that $\Psi_2 = \Psi_1 \hookleftarrow X <\!\# \,\{m : \mathsf{string}\}$ and, by using the (updt) rule, we can derive

$$\frac{\Psi_1;\Gamma_1 \vdash x : X \qquad \Psi_1;\Gamma_1 \vdash s : \mathsf{string}}{\Psi_1;\Gamma_1 \vdash x.m = s : \mathsf{string}; \Psi_2}$$

for $s \equiv a$ as well as $s \equiv$ ”`help`”.

Setting an attribute in one branch only. Consider the code.

```
let x = new in
// x:X;X <# {m:⊥}
if (ha) then x.m = a else ""
// x:X;X <# {m:string ∨ ⊥}
```

Typing the `then` branch looks like:

$$\frac{\dfrac{\Psi_1;\Gamma_1 \vdash x.m = a : \mathsf{string};\ X_m <\!\# \,\{m : \mathsf{string}\}}{\Psi_1;\Gamma_1 \vdash x.m = a : \mathsf{string};\ X <\!\# \,\{m : \mathsf{string}\}} \\ \{m : \mathsf{string}\} \preceq_c \{m : \mathsf{string} \vee \bot\}}{\Psi_1;\Gamma_1 \vdash x.m = a : \mathsf{string};\ \Psi_2} \quad (1)$$

Typing the `else` branch looks like:

$$\frac{\dfrac{\mathsf{string} \preceq_c \mathsf{string} \vee \bot \qquad \{\} \preceq_c \{\}}{\{m : \bot\} \preceq_c \{m : \mathsf{string} \vee \bot\}}}{\Psi_1; \vdash \text{””} : \mathsf{string};\ X <\!\# \,\{m : \mathsf{string} \vee \bot\}} \quad (2)$$

Then, by putting the two branches together, we get:

$$\frac{\Psi_1;\Gamma_1 \vdash ha : \mathsf{bool}; \Psi_1 \quad (1) \quad (2)}{\Psi_1;\Gamma_1 \vdash \mathsf{if}\ (ha)\ \mathsf{then}\ x.m = a\ \mathsf{else}\ \text{””} : \mathsf{string};\ \Psi_2}$$

where

$$\begin{aligned}
\Gamma_0 &= \{ha : bool, a : \mathsf{string}\}\\
\Gamma_1 &= \Gamma_0, x : X\\
\Psi_1 &= X <\!\# \,\{m : \bot\}\\
\Psi_2 &= X <\!\# \,\{m : \mathsf{string} \vee \bot\}
\end{aligned}$$

To type the whole let we need to derive

$$\frac{\vdash \mathsf{new} : X; X <\!\# \,\{\}}{\vdash \mathsf{new} : X; X <\!\# \,\{m : \bot\}}$$

which is easily done using the (new) and (bot) rules; the side-conditions of (bot) are obviously respected here, however they are needed to prevent the problems with weakening described in Section 2.2.

Creating an object in one branch only. Let b be of type bool, and x an object not containing field a. Consider:

```
if b then
  x.a = new; 0
else 0
```

Typing the then branch (with type Y for the new object) looks like:

$$\frac{\Psi;\Gamma \vdash x.a = \mathsf{new};0 : \mathsf{int}; X <\# \{a : Y\}, Y <\# \{\}}{\Psi;\Gamma \vdash x.a = \mathsf{new};0 : \mathsf{int}; X <\# \{a : Y \vee \bot\}, Y <\# \{\}}$$

by applying rule (cnstr $\preceq_c$), with $\Gamma = \{b : \mathsf{bool}, x : X\}$, $\Psi = \{X <\# \{a : \bot\}\}$.

Similarly, for the else branch we want to prove

$$\Psi;\Gamma \vdash 0 : \mathsf{int}; X <\# \{a : Y \vee \bot\}, Y <\# \{\}$$

We can easily infer $\Psi;\Gamma \vdash 0 : \mathsf{int};\Psi$. Using rule ($\preceq_c$ cevlv) we get

$$X <\# \{a : \bot\} \preceq_c X <\# \{a : \bot\}, Y <\# \{\}$$

Then with some applications of $\preceq_c$ bookkeeping rules we can get

$$\Psi \preceq_c X <\# \{a : Y \vee \bot\}, Y <\# \{\}$$

which leads us to the desired conclusion.

Function declaration and application. Consider a function that adds a field named m with a value provided as its second argument (of an arbitrary type t) to an object being its first argument:

```
func(self,x) { self.m = x }
```

Let $t_{add} = \forall X_s.[X_s, t; X_s <\# \{\}] \Rightarrow [t; X_s <\# \{m : t\}]$. Observe now that the type ensures an important property of the object graph in the heap that holds each time the function is called, therefore it is invariant. The type $\forall \vec{X}.[\ldots;\Psi_1] \Rightarrow [\ldots;\Psi_2]$ may be read as "for all graphs such that Ψ_1 holds before the call, Ψ_2 holds afterwards". In Figure 7, the boxes represent objects in the heap and have names (X_s, t) that stem from the types. In addition they are marked with variables that reference them (self, x). The scribbles in the boxes hide the types that such an object can assume during the computation. The initial input graph does not have an explicit connection between X_s and t, but in the result there is such a connection, from the now explicit field m to t. This is a simple example, but the invariants in real programs may involve complicated graphs that can be expressed straightforwardly in this way.

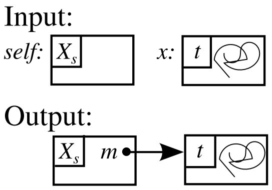

Figure 8: Graph fragments for input and output of the function type in the example.

We can derive t_{add}, by rule (fdcl)

$$\frac{\mathit{self} : X_s, x : t; X_s <\# \{\} \vdash \mathit{self}.m = x : t; X_s <\# \{m : t\}}{\vdash \mathsf{func}(\mathit{self}, x)\{\ldots\} : t_{add}; \emptyset}$$

Now we apply the above function to a newly created object:

```
let init = func(self,x) { self.m = x }
let o = new // o : Xo; Xo <# {}
init(o,42)  // o : Xo; Xo <# {m:int}
```

The renaming θ connects the formal and actual parameters, sending X_s to X_o; putting

$$\Gamma_0 = \{o : X_o; init : t_{add}\}, \qquad \Psi_0 = X_o <\!\# \{\}$$

We can infer, by (fapp),

$$\Psi_0;\, \Gamma_0 \vdash init(o, 42) : \mathsf{int};\, \Psi_0 \hookleftarrow \theta(X <\!\# \{m : \mathsf{int}\})$$

Here, we can see our form of subtyping in width at work (see Section 2.2 and rule (fapp)); observe that the same function may be called on an object containing some fields already: if $\Psi_0 = X_o <\!\# \{n : u\}$, then $\Psi_0 \hookleftarrow \theta(X_s <\!\# \{\}) = \Psi_0$. In both cases, the renaming is a witness that the program state satisfies the function precondition.

Intersection types. As hinted, the idea behind intersection types in our system is that they capture allowed aliasing scenarios. Consider the following function:

```
    func(x,y) { x.m = 1; y.m }
```

This function can work in either of the following scenarios:

(i) the actual parameter for y has field m before it is passed to the function,

(ii) the actual parameters for x and y are the same object.

Hence the type of the function will be written as $t_1 \wedge t_2$, where t_i represent types corresponding to scenarios (i) and (ii):

$$t_1 = \forall X, Y.[X, Y; \Psi_1] \implies [u; \Psi_1 \hookleftarrow X <\!\# \{m : \mathsf{int}\}]$$

with $\Psi_1 = X <\!\# \{\}, Y <\!\# \{m : u\}$, and

$$t_2 = \forall X.[X, X; X <\!\# \{\}] \implies [\mathsf{int}; X <\!\# \{m : \mathsf{int}\}]$$

In practice one does not need to write intersection types, but instead write multiple contracts for a function (and add more as needed), for example (with a fair dose of syntactic sugar):

```
    f : [X,Y;Y.m:U] => [U;X.m:int]
    f : [X,X] => [int;X.m:int]
    func f(x,y) { x.m = 1; y.m }
```

Point, ColorPoint. The following example is an encoding of a paradigmatic example in our system. The type of the *mv* method is a function type that takes as a parameter an object containing at least a field x of type `int`. Note that this is an imperative version (without MyType) of an analogous example in [5].

```
let o = new;
o.x = 7;     // r = {x:int}
// Tmv = forall Xs.[Xs,int;Xs<#r] => [Xs,Xs<#r]
o.mv = func(self,dx){ self.x = self.x+dx; self }
// o : Xo; Xo <# { x:int, mv:Tmv }
o.c = "blue";
// o : Xo; Xo <# { x:int, mv:Tmv, c:string }
o.mv(o,3); // can call mv: Xo <# { x:int } holds
o.c        // we can still read the field "c"
```

In this example, we again see our subtyping in width at work. The method *mv* requires an object with a field x, but it works also if the actual parameter contains extra fields, in our case field c. Moreover, the field c is still accessible after the method call.

4 Conclusions and future work

Our contribution is a novel type system for typing dynamic languages in a retrofitted manner, with particular emphasis over the flow of control and with the aim of tracing the changes of object interfaces at runtime. We express this change by means of Hoare-like triples that describe the structure of relevant objects before an expression is executed and after its execution.

The type reconstruction for our system seems undecidable (in fact, a slightly weakened version of the intersection type system can be embedded in the language [8]). However, there is a strong evidence that the type system becomes decidable when type annotations are provided for functions (by a programmer or by a non-complete heuristics), as seen for similar annotation schemes [7].

As a further development, we would like to apply Lucretia's approach to regulate control flow in JavaScript. Moreover, we want to couple our system with the *gradual typing* method [4, 2, 15], that supports evolving an untyped program into a typed one, possibly by using the like-type approach of [18].

References

[1] Torben Amtoft, Hanne Riis Nielson & Flemming Nielson (1999): *Type and effect systems - behaviours for concurrency*. Imperial College Press.

[2] Christopher Anderson & Sophia Drossopoulou (2003): *BabyJ: from object based to class based programming via types*. *ENTCS* 82(7), pp. 53–81. Available at `http://dx.doi.org/10.1016/S1571-0661(04)80802-8`.

[3] Henk Barendregt, Mario Coppo & Mariangiola Dezani-Ciancaglini (1983): *A Filter Lambda Model and the Completeness of Type Assignment*. *J. Symbolic Logic* 48(4), pp. 931–940.

[4] Gilad Bracha & David Griswold (1993): *Strongtalk: Typechecking Smalltalk in a Production Environment*. In: *Proc. of OOPSLA'93*, pp. 215–230.

[5] Kathleen Fisher, Furio Honsell & John C. Mitchell (1994): *A lambda calculus of objects and method specialization*. *Nord. J. Comp.* 1, pp. 3–37.

[6] David K. Gifford & John M. Lucassen (1986): *Integrating functional and imperative programming*. In: *Proc. of LFP'89*, LFP '86, ACM, New York, NY, USA, pp. 28–38.

[7] Barry Jay & Simon Peyton Jones (2008): *Scrap Your Type Applications*. In: *Proceedings of the 9th international conference on Mathematics of Program Construction*, LNCS, Springer-Verlag, Berlin, Heidelberg, pp. 2–27.

[8] A.J. Kfoury & J.B. Wells (2004): *Principality and type inference for intersection types using expansion variables*. *Theor. Comput. Sci.* 311(1–3), pp. 1–70.

[9] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz & Htet Htet Aung (2006): *An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks*. *IEEE Transactions on Software Engineering* 32(12), pp. 971–987.

[10] Barbara Liskov & Jeannette M. Wing (1994): *A Behavioral Notion of Subtyping*. *ACM Trans. Program. Lang. Syst.* 16(6), pp. 1811–1841.

[11] Daniel Marino & Todd Millstein (2009): *A generic type-and-effect system*. In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, ACM, New York, NY, USA, pp. 39–50.

[12] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter & Andreas Stefik (2012): *An empirical study of the influence of static type systems on the usability of undocumented software*. In: *Proceedings of OOPSLA*, ACM, pp. 683–702.

[13] Lutz Prechelt (2000): *An Empirical Comparison of Seven Programming Languages*. Computer 33(10), pp. 23–29.

[14] William Sasso (1996): *Cognitive processes in program comprehension: An empirical analysis in the Context of software reengineering*. Journal of Systems and Software 34(3), pp. 177–189.

[15] Jeremy G. Siek & Walid Taha (2007): *Gradual Typing for Objects*. In: Proc. of ECOOP'07, pp. 2–27.

[16] Ed Stephenson (2001): *A Fallback Application Built in Six Months Earns the Prime Role*. `http://oreilly.com/pub/a/oreilly/perl/news/swedishpension_0601.html`.

[17] Tobias Wrigstad, Patrick Eugster, John Field, Nate Nystrom & Jan Vitek (2009): *Software hardening: a research agenda*. In: Proceedings for the 1st Workshop on Script to Program Evolution, STOP '09, ACM, New York, NY, USA, pp. 58–70.

[18] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund & Jan Vitek (2010): *Integrating typed and untyped code in a scripting language*. In: Proc. of POPL'10, ACM, New York, NY, USA, pp. 377–388.