Reasoning about Web Applications: An Operational Semantics for HOP

GÉRARD BOUDOL, ZHENGQIN LUO, TAMARA REZK, and MANUEL SERRANO,
INRIA Sophia Antipolis-Mediterranée

We propose a small-step operational semantics to support reasoning about Web applications written in the multitier language HOP. The semantics covers both server side and client side computations, as well as their interactions, and includes creation of Web services, distributed client-server communications, concurrent evaluation of service requests at server side, elaboration of HTML documents, DOM operations, evaluation of script nodes in HTML documents and actions from HTML pages at client side. We also model the browser same origin policy (SOP) in the semantics. We propose a safety property by which programs do not get stuck due to a violation of the SOP and a type system to enforce it.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.2 [**Programming Languages**]: Language Classifications—*Design languages*

General Terms: Languages, Theory

Additional Key Words and Phrases: Web programming, functional languages, Multitier languages

ACM Reference Format:
Boudol, G., Luo, Z., Rezk, T., and Serrano, M. 2012. Reasoning about Web applications: An operational semantics for HOP. ACM Trans. Program. Lang. Syst., 34, 2, Article 10 (June 2012), 40 pages.
DOI = 10.1145/2220365.2220369 http://doi.acm.org/10.1145/2220365.2220369

10

1. INTRODUCTION

The Web is built atop an heterogeneous set of technologies. Traditional Web development environments rely on different languages for implementing specific parts of the applications. Graphical user interfaces are declared with HTML/CSS or Flash. Client-side computations are programmed with JavaScript augmented with various APIs such as the Document Object Model (DOM) API. Communications between servers and clients involve many different protocols such as HTTP for the low level communication, XmlHttpRequest for implementing remote procedure calls, and JSON for serializing data. Server sides are frequently implemented with languages such as PHP, Java, Python, or Ruby. Using so many different tools and technologies makes it difficult to develop and maintain robust applications. It also makes it difficult to understand their precise semantics.

Semantics of Web applications has not been studied globally but rather component by component. In a precursor paper Queinnec [2000] has studied the interaction model of Web applications based on forms submissions. This work has been pursued by Graunke and his colleagues in several publications [Graunke et al. 2003; Matthews et al. 2004]. Several formal semantics for JavaScript have been proposed [Guha et al. 2010; Maffeis et al. 2008] excluding the semantics of the DOM that has been first

This work is supported in part by the French ANR agency, grant ANR-09-EMER-009-01.
Author's address: Z. Luo; email: zhengqin.luo@inria.fr.

DOI 10.1145/2220365.2220369 http://doi.acm.org/10.1145/2220365.2220369

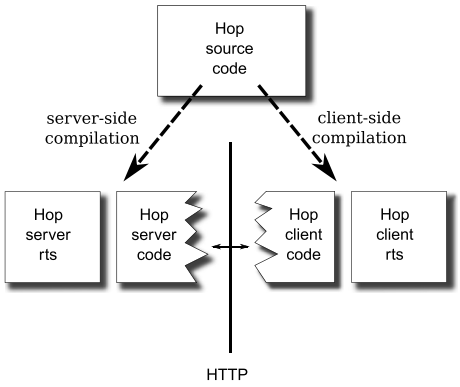

Fig. 1. Hop architecture.

studied [Gardner et al. 2008a, 2008b]. A high-level semantics that focuses on the Remote Procedure Calls (RPCs) of the Links language [Cooper et al. 2006] has also been studied [Cooper and Wadler 2009]. Another formal semantics of a Web browser [Bohannon and Pierce 2010] has been proposed as a framework to further study security problems within the browser. Various formal semantics of programming languages can be used to understand behaviors of server-side code but as precise as they are, none of them can be used to understand applications as a whole as they only cover small parts of the applications.

As a response to the emergent need of simplifying the development process of Web applications, multitier languages have been recently proposed. Examples of such languages include HOP [Serrano et al. 2006], Links [Cooper et al. 2006], Swift [Chong et al. 2009], and Ur [Chlipala 2010]. Multitier languages usually provide a unified syntax, typically based on a mainstream programming language syntax, where Web applications can be fully specified: server and client code. These languages usually also relieve the programmer from the burden of thinking about communication protocols. The HOP programming language pushes this philosophy to the extreme by addressing all aspects of Web applications and totally eliminates the need of any external language in programming these applications.

HOP [Serrano et al. 2006] (`http://hop.inria.fr`) is based on the Scheme programming language [Kelsey et al. 1998] which it extends in several directions. It is multithreaded. It provides many libraries used for implementing modern applications (mail, multimedia, ...). It also extends Scheme with constructs and APIs dedicated to Web programming. The new constructs include: (i) service definitions that are server functions associated with URLs that can be invoked by clients, (ii) service invocations that let clients invoke servers' services, (iii) client-side expressions that are used by servers to create client-side programs, and, (iv) server-side expressions, which are embedded inside client-side expressions. The new APIs are: full HTML and DOM support that let servers and clients define and modify HTML documents and HTML fragments.

When a HOP program is loaded into a HOP *broker* [Serrano 2009], that is, the HOP execution environment, it is split and compiled on the fly. Server-side parts are compiled to a mix of bytecode or native code and client-side parts are compiled to JavaScript [Loitsch and Serrano 2008]. In the source code, a syntactic mark instructs the compiler about the location where the expression is to be evaluated. Figure 1 illustrates the dual compilation.

When the HOP broker starts, it registers all the available programs and waits for client connections. Upon connection, it actually loads the program needed to

fulfill the request it has received and returns a HTML document which contains the client-side part of the program to the client that has emitted the request. That client proceeds with the execution of the program. When needed, the client may invoke server-side services which accept client-side values and returns server-side values. The normal execution of the HOP program keeps flowing from the client to the server and vice-versa.

By covering all aspects of programming Web applications, HOP can then be used to reason globally about these applications. Our contribution in this article is to provide a formal and unified small-step operational semantics that could support such reasoning. A denotational continuation-based semantics was previously given for a core subset of HOP [Serrano and Queinnec 2010]. However, this work did not cope with DOM operations nor multiple clients. It only described the elaboration of client-side code as generated by the server-side code. The semantics given in this article, in addition to being written in the more versatile style of operational semantics, covers a much wider spectrum of the language. Indeed, our semantics can support global formal reasoning about Web applications.

On one hand, the HOP language relies on standard programming constructs. On the other hand, several features of HOP are specific to a multitier language, and therefore require specific semantics that, as far as we can see, have not been previously formalized. These features are mostly related to the stratified design of server codes. In particular, the dynamic client code generation from the server and its installation at client site are prominent features of HOP.

Finally, we propose a type system to enforce that typable programs do not get stuck due to a violation to the same origin policy (codified as an error), and using the semantics we show formally that the system is sound.

Contents. The article is organized as follows. In Section 2 we describe a core of the language HOP and its semantics which is extended in Section 3 with DOM operations. In Section 4 we model the same-origin policy and inlining of code. In Section 5 we propose a safety property based on the same-origin policy, a static analysis, and a soundness proof for it. Finally, we conclude in Section 6 and propose future directions.

Remarks. This article revises and extends an earlier workshop version [Boudol et al. 2010]. We extend the workshop version by adding explanations and examples and modeling the browser same-origin policy and inlining in the semantics. We also propose a static analysis for a safety property that assures that programs are compliant with same-origin policy and prove its soundness using the semantics.

2. CORE HOP

In this section we introduce the syntax, and then the semantics, of the HOP language, or more precisely of the core constructs of the language. More complete versions, involving the DOM part, inlining code and Same Origin Policy, will be considered in Section 3 and 4. Our core language exhibits the most prominent features of the HOP language: service definition and invocation, transfer of code and values from the server to a client, that is, the distributed computing aspect of HOP.

2.1. Syntax

The Core HOP syntax is stratified into *server code s* and *tilde code t*. The former is basically Scheme code enriched with a construct $(\mathsf{service}\,(x)\,s)$ to define a new *service*, that is a function bound to an URL, and a construct $\sim t$ to ship (tilde) code t to the client. The latter may include references $\$x$ to server values, and will be translated into *client code c*, before being shipped to the client. In the actual HOP system, the

$$
\begin{array}{rcll}
u & \in & \mathcal{U}rl & \textit{URL} \\
s & ::= & x \mid w \mid (s_0\, s_1) \mid (\mathsf{set!}\, x\, s) \mid {\sim}t & \textit{server code} \\
 & \mid & (\mathsf{service}\,(x)\, s) \mid (\mathsf{with\text{-}hop}\, s_0\, s_1) & \\
t & ::= & x \mid u \mid u?v \mid (\mathsf{lambda}\,(x)\, t) & \\
 & \mid & (t_0\, t_1) \mid (\mathsf{set!}\, x\, t) \mid \$x \mid (\mathsf{with\text{-}hop}\, t_0\, t_1) & \\
w & ::= & u \mid u?w \mid (\mathsf{lambda}\,(x)\, s) & \textit{server values} \\
 & \mid & {\sim}c \mid () & \\
c & ::= & x \mid v \mid (c_0\, c_1) \mid (\mathsf{set!}\, x\, c) & \textit{client code} \\
 & \mid & (\mathsf{with\text{-}hop}\, c_0\, c_1) & \\
v & ::= & u \mid u?v \mid (\mathsf{lambda}\,(x)\, c) \mid () & \textit{client values}
\end{array}
$$

Fig. 2. Core HOP syntax.

latter is compiled into JavaScript code [Loitsch and Serrano 2008], but here we ignore the compilation phase from HOP to JavaScript, as we provide a semantics at the level of (source) client code.

The syntax is given in Figure 2, where x denotes any variable. We assume that we are given a set $\mathcal{U}rl$, disjoint from the set of variables, of names denoting URLs. These names are values in the Core HOP language, where they are used as denoting a function, or more accurately a service. When provided with an argument, that is a value w, a call $(u\, w)$ to a service is transformed into a value $u?w$ that can be passed around as an argument. In particular, such an argument will be used in the $(\mathsf{with\text{-}hop}\, u?w\, w')$ form, which sends the value w to the service at u somewhere in the Web, and waits for a value to be returned as an argument to the continuation w'.

A server expression usually contains subexpressions of the form ${\sim}t$. As we said, t represents code that will be executed at client site. This code cannot create a service, that is, it does not contain any subexpression $(\mathsf{service}\,(x)\, s)$, but it usually calls services from the server, by means of the $(\mathsf{with\text{-}hop}\, t_0\, t_1)$ construct, and it may use values provided by the server, by means of subexpressions $\$x$. When the latter are absent (that is, when they have been replaced by the value bound to x), a t expression reduces to a client expression c. Notice that for the server an expression ${\sim}c$ is a value, meaning that the code c is frozen and will only be executed at client site. Values also include $()$, which is a shorthand for the unspecified Scheme runtime value. In a more complete description of the language we would include other kinds of values, like for instance Boolean truth values, integers, strings, and so on, as well as some constructs to build and use these values.

As usual $(\mathsf{lambda}\,(x)\, s)$ binds x in the expression s, and the same holds for $(\mathsf{service}\,(x)\, s)$. However, inside tilde code, a $(\mathsf{lambda}\,(x)\, t)$ does not bind x in the subexpressions $\$x$. Then we have to say more precisely what is the set $\mathsf{fv}(s)$ of free variables of an expression s. This set is defined as usual, except that

$$
\begin{aligned}
\mathsf{fv}({\sim}t) &= \mathsf{fv}^{\$}(t) \\
\mathsf{fv}(\$x) &= \emptyset,
\end{aligned}
$$

where $\mathsf{fv}^{\$}(t)$ is given by

$$
\begin{aligned}
\mathsf{fv}^{\$}(x) &= \emptyset \\
\mathsf{fv}^{\$}((\mathsf{lambda}\,(x)\, t)) &= \mathsf{fv}^{\$}(t) \\
\mathsf{fv}^{\$}(\$x) &= \{x\}
\end{aligned}
$$

(the remaining cases are defined in the obvious way). A HOP program is a *closed* expression s, meaning that it does not contain any free variable (but it may contain names u for services that are provided from outside of the program). We shall consider expressions up to α-conversion, that is up to the renaming of bound variables, and we denote by $s\{y/x\}$ the expression resulting from substituting the variable y for x in s, possibly renaming y in subexpressions where this variable is bound, to avoid captures. Again, since the variables occurring in a subexpression $\$x$ are not bound by a lambda in tilde code, we have to define more precisely what $s\{y/x\}$ is. The definition is standard, except that:

$$\begin{aligned}(\sim t)\{y/x\} &= \sim(t\{y//x\})\\ (\$z)\{y/x\} &= \$z,\end{aligned}$$

where for tilde code $t\{y//x\}$ is given by

$$\begin{aligned}z\{y//x\} &= z\\ \$z\{y//x\} &= \begin{cases}\$y & \text{if } z = x\\ \$z & \text{otherwise}\end{cases}\\ (\mathsf{lambda}\,(z)\,t)\{y//x\} &= (\mathsf{lambda}\,(z')\,t\{z'/z\}\{y//x\}),\end{aligned}$$

where $z' \notin \{x, y\} \cup \mathsf{fv}(t)$.

The operational semantics of the language will be described as a transition system, where at each step a (possibly distributed) *redex* is reduced. As usual, this occurs in specific positions in the code, that are described by means of *evaluation contexts* [Felleisen and Hieb 1992]. In order to describe in a simple way the communications between a client, invoking a service, and the server, which computes the answer to the service request, we shall introduce a new form into the syntax, namely

$$s ::= \cdots \mid (j\,s),$$

where j is a *communication identifier* (or channel), taken from some infinite set, disjoint from the set of variables and the set $\mathcal{U}rl$ of URLs.

The syntax of evaluation contexts is as follows:

$$\begin{aligned}\mathbf{E} ::= {} & [] \mid (\mathbf{E}\,s) \mid (w\,\mathbf{E}) \mid (j\,\mathbf{E}) \mid (\mathsf{set!}\,x\,\mathbf{E})\\ & \mid (\mathsf{with\text{-}hop}\,\mathbf{E}\,s) \mid (\mathsf{with\text{-}hop}\,w\,\mathbf{E}).\end{aligned}$$

Since client code and client values are particular cases of server code and server values respectively, evaluation contexts in client code are particular cases of (server) evaluation contexts. One can see that for the $(\mathsf{with\text{-}hop}\,s_0\,s_1)$ form, one has to evaluate s_0, and then s_1, before actually calling a service. As usual, we denote by $\mathbf{E}[s]$ the result of filling the hole [] in context $\mathbf{E}$ with expression s.

2.2. Semantics

The semantics of a HOP program is represented as a sequence of transitions between configurations. We consider a simple scenario where there is only one server and one client. An extension for many servers and clients will be discussed in Section 4. A configuration consists of the following

— A server configuration S, together with an environment (or store) μ providing the values for the variables occurring in the server configuration. The server

configuration consists in a main thread executing server code, and a number of threads of the form (js) executing clients' requests to services.
— A client configuration C, which is a tuple $\langle c, \mu, W\rangle$ where c is the running client code, typically performing service requests, μ is the local environment for the client (distinct from the one of the server: the client and the server do not share any state), and W is a multiset of pending continuations $(v\ j)$, waiting for a value returned from a service call (which has been named j), or callbacks $(v\, c)$, and more generally client code c waiting to be processed at client site (we shall see another instance of this in Section 3 with the onclick construct).
— A HOP environment ρ, binding URLs to the services they denote. Services bound in ρ are defined by evaluating expressions of the form $(\mathsf{service}\,(x)\,s)$. Environment ρ represents the services that can be called by the client at different URLs.
— An external environment Web, binding URLs to server-side values to be consumed by server-side with-hop invocations. Web represents the external environment of the whole Web with respect to the only server in the configuration.
— A set J of communication identifiers that are currently in use.

Then a configuration Γ has the form $((S, \mu), C, \rho, \mathsf{Web}, J)$. An initial configuration for a given server-side expression s has the following form:

$$((\{s\}, \emptyset), \langle (), \emptyset, \emptyset\rangle, \emptyset, \mathsf{Web}, \emptyset).$$

To simplify a little the semantic rules, and to represent the concurrent execution of the various components, we shall use the following syntax for configurations:

$$\Gamma \ ::= \ \mu \ \mid \ \rho \ \mid \ \mathsf{Web} \ \mid \ J \ \mid \ s \ \mid \ \langle c, \mu, W\rangle \ \mid \ (\Gamma \parallel \Gamma),$$

where μ is a mapping from a finite set $\mathsf{dom}(\mu)$ of variables to values (server values or client values), ρ is a mapping from a finite set $\mathsf{dom}(\rho)$ of URLs to services, that is functions $(\mathsf{lambda}\,(x)\,s)$, Web is a mapping from a set $\mathsf{dom}(\mathsf{Web})$ of URLs to server-side values w, and W is a finite set of expressions of the form $(v\ j)$[1] or $(v\, c)$. Expressions $(v\ j)$ represent a continuation waiting for a callback from a request with communication identifier j. We assume that the domains of ρ and Web are disjoint from each other, that is, $\mathsf{dom}(\rho) \cap \mathsf{dom}(\mathsf{Web}) = \emptyset$. A configuration is *well-formed* if it contains exactly one μ, one C, one ρ, one Web and one J[2]. We only consider well-formed configurations in what follows. We assume that parallel composition $\parallel$ is commutative and associative, so that the rules can be expressed following the "chemical style" of Berry and Boudol [1990], specifying local "reactions" of the form $\Gamma \to \Gamma'$ that can take place anywhere in the configuration. That is, we have a general rule

$$\frac{\Gamma \to \Gamma'}{(\Gamma \parallel \Gamma'') \to (\Gamma' \parallel \Gamma'')}$$

meaning that if the components of Γ are present in the configuration, which can therefore be written $(\Gamma \parallel \Gamma'')$, and if these components interact to produce Γ', then we can replace the components of Γ with those of Γ'.

[1]These expressions $(v\ j)$ do not evaluate, and therefore we do not need to add them to the syntax of client code.
[2]These components are omitted whenever they are empty.

$$
\begin{array}{rcl}
\dagger w & = & \begin{cases} \bot \ \text{if } w = (\mathsf{lambda}\ (x)\ s) \\ c \quad \text{if } w = {\sim}c \\ w \quad \text{otherwise} \end{cases} \\
\Xi(\mu, x) & = & x \\
\Xi(\mu, u) & = & u \\
\Xi(\mu, u?v) & = & u?v \\
\Xi(\mu, (\mathsf{lambda}\ (x)\ t)) & = & (\mathsf{lambda}\ (y)\ \Xi(\mu, t\{y/x\})) \\
 & & \text{where } y \notin \mathsf{dom}(\mu) \\
\Xi(\mu, (\mathsf{set!}\ x\ t)) & = & (\mathsf{set!}\ x\ \Xi(\mu, t)) \\
\Xi(\mu, (t_0\ t_1)) & = & (\Xi(\mu, t_0)\ \Xi(\mu, t_1)) \\
\Xi(\mu, \$x) & = & \begin{cases} \bot \text{ if } \mu(x) = (\mathsf{lambda}\ (y)\ s) \\ \quad \text{or } \mu(x) = {\sim}c \\ \mu(x) \quad \text{otherwise} \end{cases} \\
\Xi(\mu, (\mathsf{with\text{-}hop}\ t_0\ t_1)) & = & (\mathsf{with\text{-}hop}\ \Xi(\mu, t_0)\ \Xi(\mu, t_1))
\end{array}
$$

Fig. 3. Transformation from server value to client value.

Before introducing and commenting on the reaction rules, we need to define an auxiliary function transforming tilde code into client code. As we said a subexpression $\sim t$ in server code is *not* evaluated at server side, but will be shipped to the client, usually as the answer to a service request. Since the expression t may contain references $\$x$ to server values, to define the semantics we introduce an auxiliary function Ξ that takes as arguments an environment μ and an expression t, and transforms it into a client expression c. This is defined in Figure 3, where we also introduce a partial function that transforms a (server) value into a client expression by removing the tilde, provided the value is not a λ-abstraction. The Ξ transformation consists in replacing $\$x$ by the value bound to x in μ, but one should notice that a function, that is a $(\mathsf{lambda}\,(x)\,s)$, or client code c cannot be sent to the client this way, because this would in general result in breaking the bindings of free variables that may occur in such an expression. Then this has to be considered as an error.

The semantics is given in Figure 4, which we now comment on. First notice that we write a compound configuration $(\Gamma \parallel \Gamma')$ as $\Gamma \parallel \Gamma'$. This is not ambiguous, since parallel composition is commutative and associative. When we have to evaluate a variable (rule VARS), we need to look up into the corresponding environment μ, which we express as a reaction from $\mathbf{E}[x] \parallel \mu$, but obviously the environment must remain unchanged as a component of the configuration, which is why we restore it in $\mathbf{E}[w] \parallel \mu$, where $w = \mu(x)$. We can also update the value of a variable in server-side store or client-side store, respectively (rules SETS and SETC, where $\mu[x \mapsto w]$ denotes the updated store). As we said when introducing the syntax, a call $(u\,w)$ to a service is transformed into a value $u?w$ (rules REQS and REQC). Evaluating $(\mathsf{service}\,(x)\,s)$ (rule SERVDEFS) creates a new URL name[3] $u \notin \mathsf{dom}(\rho)$, returns this name to the evaluation context, and updates the service environment ρ by adding a new service (= function) associated with u. We may have service invocations from the server, that is $(\mathsf{with\text{-}hop}\,u?w_0\,w_1)$, where the name u refers to some pre-existing service, that has not been created by the running program. In that case (rule SERVINS), we use the returned value w provided by Web. This value is passed as an argument to the continuation w_1. In this rule $\mathsf{Web}(u?w_0)$ represents a call to an external service available in the Web, and allows for writing mashups using

[3]In the HOP language this is an optional argument to a service definition.

$$\frac{\mu(x) = w}{\mathbf{E}[x] \parallel \mu \to \mathbf{E}[w] \parallel \mu}\ (\textsc{VarS}) \qquad \frac{\mu(x) = v}{\langle \mathbf{E}[x], \mu, W\rangle \to \langle \mathbf{E}[v], \mu, W\rangle}\ (\textsc{VarC})$$

$$\frac{x \in \mathsf{dom}(\mu)}{\mathbf{E}[(\mathsf{set!}\, x\, w)] \parallel \mu \to \mathbf{E}[()] \parallel \mu[x \mapsto w]}\ (\textsc{SetS})$$

$$\frac{x \in \mathsf{dom}(\mu)}{\langle \mathbf{E}[(\mathsf{set!}\, x\, v)], \mu, W\rangle \to \langle \mathbf{E}[()], \mu[x \mapsto v], W\rangle}\ (\textsc{SetC})$$

$$\frac{}{\mathbf{E}[(u\, w)] \to \mathbf{E}[u?w]}\ (\textsc{ReqS}) \qquad \frac{}{\langle \mathbf{E}[(u\, v)], \mu, W\rangle \to \langle \mathbf{E}[u?v], \mu, W\rangle}\ (\textsc{ReqC})$$

$$\frac{y \notin \mathsf{dom}(\mu)}{\mathbf{E}[((\mathsf{lambda}\ (x)\ s)w)] \parallel \mu \to \mathbf{E}[s\{y/x\}] \parallel \mu \cup \{y \mapsto w\}}\ (\textsc{AppS})$$

$$\frac{y \notin \mathsf{dom}(\mu)}{\langle \mathbf{E}[((\mathsf{lambda}\ (x)\ c)v)], \mu, W\rangle \to \langle \mathbf{E}[c\{y/x\}], \mu \cup \{y \mapsto v\}, W\rangle}\ (\textsc{AppC})$$

$$\frac{u \notin \mathsf{dom}(\rho)}{\mathbf{E}[(\mathsf{service}\ (x)\ s)] \parallel \rho \to \mathbf{E}[u] \parallel \rho \cup \{u \mapsto (\mathsf{lambda}\ (x)\ s)\}}\ (\textsc{ServDefS})$$

$$\frac{u \notin \mathsf{dom}(\rho) \qquad \mathsf{Web}(u?w_0) = w}{\mathbf{E}[(\mathsf{with\text{-}hop}\, u?w_0\, w_1)] \parallel \rho \to \mathbf{E}[(w_1\, w)] \parallel \rho}\ (\textsc{ServInS})$$

$$\frac{j \notin J \qquad \rho(u) = w \qquad W' = W \cup \{(v_1\, j)\} \qquad J' = J \cup \{j\}}{\langle \mathbf{E}[(\mathsf{with\text{-}hop}\, u?v_0\, v_1)], \mu, W\rangle \parallel \rho \parallel J \to (j\, (w\, v_0)) \parallel \langle \mathbf{E}[()], \mu, W'\rangle \parallel \rho \parallel J'}\ (\textsc{ServInC})$$

$$\frac{}{(j\, w) \parallel \langle c, \mu, W \cup \{(v\, j)\}\rangle \parallel J \to \langle c, \mu, W \cup \{(v\, (\dagger w))\}\rangle \parallel J - \{j\}}\ (\textsc{ServRetS})$$

$$\frac{}{\langle v, \mu, \{c\} \cup W\rangle \to \langle c, \mu, W\rangle}\ (\textsc{CallbackC}) \qquad \frac{\Xi(\mu, t) = c}{\mathbf{E}[{\sim}t] \parallel \mu \to \mathbf{E}[{\sim}c] \parallel \mu}\ (\textsc{TildeS})$$

$$\frac{j \notin J \qquad \rho(u) = w}{\langle v, \mu, \emptyset\rangle \parallel \rho \parallel J \to (j\, (w\, v)) \parallel \langle (), \emptyset, \{((\mathsf{lambda}\ (x)\ x)\ j)\}\rangle \parallel \rho \parallel J \cup \{j\}}\ (\textsc{InitC})$$

Fig. 4. Core HOP semantics.

HOP in later extensions for the semantics. Observe that service invocation from the server behaves like a RPC, whereas service invocation from a client is asynchronous: evaluating a $(\mathsf{with\text{-}hop}\, u?v_0\, v_1)$ from client side (rule SERVINC) creates a new communication name j, spawns a thread $(j(w\, v_0))$ at server side to evaluate the request to the service, and terminates the invocation at client side while adding a continuation $(v_1\, j)$ that waits for the value returned from the server. This returned value is transformed into client code (or value) by means of the † function, and then provided as an argument to the continuation v_1 (rule SERVRETS); the communication identifier j is

then recovered. Concluding the semantics of the with-hop construct, a callback $(v_1\, c)$ from the set W is evaluated when the client's code has terminated. Indeed, in the CALLBACKC rule, client code c (representing a continuation with a return value from a service) is randomly chosen from the set W to start execution in the client. In server code, a subexpression $\sim t$ is translated (rule TILDES) into a server value $\sim c$ by means of the transformation Ξ. Finally, we have a last rule INITC, similar to SERVINC, that models the situation where a client has finished its own computation (with an empty W in its configuration), and sends a service request to the server, initiating a new thread of computation at server side. In this rule the client's continuation is simply the identity (lambda $(x)\, x$) (In Section 3, this continuation is used to set up an HTML page at client side). Let us illustrate this semantics with an example, where we use the form (let $((x\, s_0))\, s_1$) as an abbreviation for ((lambda $(x)\, s_1)\, s_0$).

Example 1 *Core Hop Semantics*. Let

$$\begin{aligned} s &= (\text{let } ((x\, (\text{service } (y)\, y)))\ \sim t) \\ t &= (\text{with-hop } (\$x\, ())\, (\text{lambda } (x)\, x)) \end{aligned}$$

We start with a configuration where there is a service (lambda $(z)\, s$) available at URL u_0, that is with $\rho_0 = \{u_0 \mapsto (\text{lambda } (z)\, s)\}$ (and $\mu = \emptyset = J$, so we omit these components). Then we have the transitions shown in Figure 5, which displays service definition and the interactions between clients and a server.

In Example 1, a client will invoke the service defined at u_0, obtaining a client code which invokes another service by with-hop. In the first step the continuation is the identity. Note that in μ_0 there is a fresh variable z' introduced as the effect of lambda application. One can see (in the last steps) that server threads compute concurrently with clients. However, one should observe that, since the server and the clients do not share any common state, there is no conflict between the server and client computations, nor among client computations. This means that, when reasoning about the behavior of a HOP program, we do not have to consider all the possible interleavings, since many steps are actually independent from each other. In fact, we could have presented the semantics using a synchronous style, where a client always waits for the answers from the server before resuming its own computations. That is, we could have restricted the VARC, REQC, APPC and SERVINC rules to the case where the set W only contains callbacks of the form $(v\, c)$, and no pending continuation $(v\, j)$. This is not the way a HOP program actually behaves, but this restriction to the semantics does not change it in an essential manner, if the services always return. In any case, one should be able to use local reasoning for server and client code.

3. DOM EXTENSION

In Section 2 we have seen how distributed computations are built and run in HOP. In this section we consider another part of the HOP language, which allows one to build HTML trees, that will be interpreted and displayed by the client's browser. The client can manipulate the host HTML page by means of the DOM (Document Object Model [Hors et al. 2000]) interface of the browser. Then we enrich the syntax with some basic HTML constructs, written in Scheme style, and operations supported by the DOM. Here we consider only the HTML and DIV tags, and the (dom-appchild! $s_0\, s_1$) construct; the other ones are similar [Gardner et al. 2008a]. The HOP syntax is

$$
\begin{array}{lll}
\rho_0 \;\rightarrow\; (j\,((\mathsf{lambda}\,(z)\,s)\,()))\\
\quad \parallel \langle (), \emptyset, ((\mathsf{lambda}\,(x)\,x)\,j)\rangle \parallel \rho_0 \parallel \{j\} & (\textsc{InitC})\\
\quad \stackrel{*}{\rightarrow}\; (j, (\mathsf{let}\ ((x\,u_1))\ {\sim}t)) \parallel \mu_0 & (\textsc{AppS},\\
\quad \parallel \langle (), \emptyset, ((\mathsf{lambda}\,(x)\,x)\,j)\rangle \parallel \rho_1 \parallel \{j\} & \textsc{ServDefS})\\
\quad \text{where } \mu_0 = \{z' \mapsto ()\}\\
\qquad\qquad \rho_1 = \rho_0 \cup \{u_1 \mapsto (\mathsf{lambda}\,(y)\,y)\}\\
\quad \stackrel{*}{\rightarrow}\; (j\,{\sim}c) \parallel \mu_1 & (\textsc{AppS},\\
\quad \parallel \langle (), \emptyset, ((\mathsf{lambda}\,(x)\,x)\,j)\rangle \parallel \rho_1 \parallel \{j\} & \textsc{TildeS})\\
\quad \text{where } c = (\mathsf{with\text{-}hop}\,(u_1\,())\,(\mathsf{lambda}\,(x)\,x))\\
\qquad\qquad \mu_1 = \mu_0 \cup \{x' \mapsto u_1\}\\
\quad \stackrel{*}{\rightarrow}\; \mu_1 \parallel \langle ((\mathsf{lambda}\,(x)\,x)\,c), \emptyset, \emptyset\rangle \parallel \rho_1 & (\textsc{ServRetS},\\
& \textsc{CallbackC})\\
\quad \stackrel{*}{\rightarrow}\; (j\,((\mathsf{lambda}\,(y)\,y)\,())) \parallel \mu_1 & (\textsc{ReqC},\\
\quad \parallel \langle (((\mathsf{lambda}\,(x)\,x)\,()), \emptyset, ((\mathsf{lambda}\,(x)\,x)\,j)\rangle & \textsc{ServInC})\\
\quad \parallel \rho_1 \parallel \{j\}\\
\quad \stackrel{*}{\rightarrow}\; (j\,()) \parallel \mu_2 & (\textsc{AppS}, \textsc{VarS},\\
\quad \parallel \langle (), \mu_0', ((\mathsf{lambda}\,(x)\,x)\,j)\rangle \parallel \rho_1 \parallel \{j\} & \textsc{AppC}, \textsc{VarC})\\
\quad \text{where } \mu_2 = \mu_1 \cup \{y' \mapsto ()\}\\
\qquad\qquad \mu_0' = \{x' \mapsto ()\}\\
\quad \stackrel{*}{\rightarrow}\; \mu_2 \parallel \langle (), \mu_1', \emptyset\rangle \parallel \rho_1 & (\textsc{ServRetS},\\
\quad \text{where } \mu_1' = \mu_0' \cup \{z \mapsto ()\} & \textsc{CallbackC},\\
& \textsc{AppC}, \textsc{VarC})
\end{array}
$$

Fig. 5. HOP operational semantics: An example.

extended as shown in Figure 6, where we assume given an infinite set $\mathcal{P}ointer$ of *pointers*, that will be used to denote nodes in HTML trees. The pointers are runtime values. Notice that instead of writing $\langle tag\rangle \cdots \langle/tag\rangle$ as in HTML, we write $(\langle tag\rangle\ \cdots)$ in HOP, which means that a *tag* is a function that is used to build an HTML node. The general form in HOP is

$$(\langle tag\rangle\ [{:}attr]\ s_0 \ldots s_n),$$

where *attr* is an optional list of attributes, and $s_0 \ldots s_n$ are the list of children of this node to be created. However, in the extension for the syntax, we consider a simpler form $(\langle tag\rangle\ [{:}attr])$ which creates a node with no child. More general forms of creating a node with arbitrary number of children can be defined as syntactic sugar. For example, creating a node with one child can be defined as follows:

$$(\langle tag\rangle\ [{:}attr]\ s) \triangleq ((\mathsf{lambda}\,(x)\,(\mathsf{dom\text{-}appchild!}\,(\langle tag\rangle\ [{:}attr])\,x))\,s).$$

We consider here only the cases where there is no attribute, or where this attribute is $\mathsf{onclick}\,s$, but we sometimes write $(\langle tag\rangle\ [{:}attr])$ for any of the two forms, when the

$$
\begin{array}{rcl}
p,\ q,\ r\ldots & \in & \mathit{Pointer} \\
s & ::= & \cdots \mid (\langle tag\rangle) \mid (\langle tag\rangle\ \mathsf{:onclick}\ s_0) \\
 & \mid & (\mathsf{dom\text{-}appchild!}\ s_0\ s_1) \\
t & ::= & \cdots \mid (\langle tag\rangle) \mid (\langle tag\rangle\ \mathsf{:onclick}\ t_0) \\
 & \mid & (\mathsf{dom\text{-}appchild!}\ t_0\ t_1) \\
w & ::= & \cdots \mid p \\
c & ::= & \cdots \mid (\langle tag\rangle) \mid (\langle tag\rangle\ \mathsf{:onclick}\ c_0) \\
 & \mid & (\mathsf{dom\text{-}appchild!}\ c_0\ c_1) \\
v & ::= & \cdots \mid p \\
tag & ::= & \mathsf{HTML} \mid \mathsf{DIV} \mid \cdots
\end{array}
$$

Fig. 6. DOM extension for HOP syntax.

attribute is irrelevant. The optional onclick s attribute offers to the client the possibility of running some code (namely c if $s = \sim c$), by clicking on the node. (In HOP there are other similar facilities.)

The semantics of the $(\langle tag\rangle\ [:attr])$ construct is that it builds a node of a tree in a *forest*. In order to define this, we assume given a specific null pointer, denoted α, which is not in $\mathcal{P}ointer$. We use π to range over $\mathcal{P}ointer \cup \{\alpha\}$. Then a forest maps (non null) pointers to pairs made of a (possibly null) pointer and an expression of the form $(\langle tag\rangle\ [:attr]\ c_1 + \cdots + c_n)$. The pointer $q \in \mathcal{P}ointer$ assigned to p is the *ancestor* of the node, if it exists. If it does not, this pointer is α. Such a node is labeled *tag* and has n children, which are either leaves (labeled with some client code or value) or pointers to other nodes in the tree. For simplicity we consider here the forest as joined to the environment providing values for variables. That is, we now consider that μ is a mapping from a set $\mathsf{dom}(\mu)$ of variables and (nonnull) pointers, that maps variables to values, and pointers to pairs made of a (possibly null) pointer and a *node* expression. The syntax for node expressions a is as follows:

$$
\begin{array}{rcl}
a & ::= & (\langle tag\rangle\ \ell) \mid (\langle tag\rangle\ \mathsf{:onclick}\ c\ \ell) \\
\ell & ::= & \varepsilon \mid c \mid (\ell_0 + \ell_1),
\end{array}
$$

where ε is the empty list. In what follows we assume that + is associative, and that $\varepsilon + \ell = \ell = \ell + \varepsilon$. We shall also use the following notations in defining the semantics, assuming that the pointers occurring in the list ℓ are distinct:

$$
\begin{array}{rcl}
(\langle tag\rangle\ [:attr]\ \ell) + p & = & (\langle tag\rangle\ [:attr]\ \ell + p) \\
(\langle tag\rangle\ [:attr]\ \ell_0 + p + \ell_1) - p & = & (\langle tag\rangle\ [:attr]\ \ell_0 + \ell_1).
\end{array}
$$

Given a forest μ, and $p \in \mathsf{dom}(\mu)$, we denote by $\mu[p \mapsto (\pi, a)]$ the forest obtained by updating the value associated with p in μ. More generally, we define $\mu[\mu']$ as follows:

$$
\begin{array}{rcl}
\mathsf{dom}(\mu[\mu']) & = & \mathsf{dom}(\mu) \cup \mathsf{dom}(\mu') \\
(\mu[\mu'])(p) & = & \begin{cases} \mu'(p) & \text{if } p \in \mathsf{dom}(\mu') \\ \mu(p) & \text{otherwise.} \end{cases}
\end{array}
$$

$$\frac{j \notin J \qquad \rho(u) = w}{\langle v, \mu, \emptyset, r\rangle \parallel \rho \parallel J \to (j\,(w\,v)) \parallel \langle (), \emptyset, \{(\mathsf{setdoc}\, j)\}, \alpha\rangle \parallel \rho \parallel J \cup \{j\}} \ (\textsc{InitC})$$

$$\frac{\mu(r) = (\alpha, (\langle \mathsf{HTML}\rangle\, [:attr]\ \ell))}{(j\,r) \parallel \mu \parallel \langle (), \emptyset, \{(\mathsf{setdoc}\, j)\}, \alpha\rangle \parallel J \to \mu \parallel \langle r, \mu \lceil r, \emptyset, r\rangle \parallel J - \{j\}} \ (\textsc{ServRet1S})$$

$$\frac{\mathsf{dom}(\mu_0 \lceil w) \cap \mathsf{dom}(\mu_1) = \emptyset \quad W' = W \cup \{(v\,(\dagger w))\}}{(j\,w) \parallel \mu_0 \parallel \langle c, \mu_1, W \cup \{(v\,j)\}, r\rangle \parallel J \to \langle c, \mu_1[\mu_0 \lceil w], W', r\rangle \parallel J - \{j\}} \ (\textsc{ServRet2S})$$

Fig. 7. Core HOP semantics modified for DOM.

For $P \subseteq \mathsf{dom}(\mu)$, we also define $\mu \lceil P$ to be the least subset of μ satisfying

$$\begin{array}{l} P \subseteq \mathsf{dom}(\mu \lceil P) \\ q \in \mathsf{dom}(\mu \lceil P)\ \&\ \mu(q) = (q', (\langle tag\rangle\ [:attr]\ \ell)\ \Rightarrow\ q' \in \mathsf{dom}(\mu \lceil P) \\ q \in \mathsf{dom}(\mu \lceil P)\ \&\ \mu(q) = (\pi, (\langle tag\rangle\ [:attr]\ \ell_0 + q' + \ell_1)\ \Rightarrow\ q' \in \mathsf{dom}(\mu \lceil P) \\ q \in \mathsf{dom}(\mu \lceil P)\ \Rightarrow\ (\mu \lceil P)(q) = \mu(q). \end{array}$$

We overload this notation by writing $\mu \lceil c$ for $\mu \lceil P$ where P is the set of pointers that occur in c. This is the forest that is the part of μ relevant for the expression c.

The syntax of evaluation contexts needs to be extended, but we now also have to distinguish client evaluation contexts $\mathbf{C}$ from server's evaluation contexts $\mathbf{S}$:

$$\begin{array}{rcl} \mathbf{S} & ::= & \cdots \mid (\langle tag\rangle\ \mathsf{:onclick}\ \mathbf{S}) \\ & \mid & (\mathsf{dom\text{-}appchild!}\ \mathbf{S}\ s) \\ & \mid & (\mathsf{dom\text{-}appchild!}\ w\ \mathbf{S}) \\ \mathbf{C} & ::= & \cdots \mid (\mathsf{dom\text{-}appchild!}\ \mathbf{C}\ c) \\ & \mid & (\mathsf{dom\text{-}appchild!}\ v\ \mathbf{C}). \end{array}$$

The main difference between server context and client context is that at client side we do not evaluate c_0 in $(\langle tag\rangle\ \mathsf{:onclick}\ c_0)$, because this is the code that will be executed at client side when an "onclick" action is performed. Finally, as regards configurations, we now assume that clients are *rooted*. That is, a client configuration now has the form

$$\langle c, \mu, W, r\rangle,$$

where the pointer r is the root of the HTML page that is displayed on the client site by the browser.

The semantic rules given in Figure 4 still hold, except for SERVRETS, which we redefine shortly. We also have to extend the $\Xi(\mu, t)$ function in rule TILDES to take into account the new constructs. This is done in the obvious way, preserving the structure of the expression (the function Ξ only has an effect on the $\$x$ subexpressions), with $\Xi(\mu, p) = p$ for any $p \in \mathcal{P}ointer$. The modified rules are given in Figure 7, while the new ones are in Figure 8. The VARC, SETC, REQC, APPC, CALLBACKC, SERVINC and INITC rules of Figure 4 have to be adapted to suit the new form of a client configuration, which involves a root. This is done in the obvious way—so we omit the adapted rules—except that in the INITC rule, after initiating a request (when no computation in client is available), the client is not yet rooted, or more precisely its root is α: the

$$\frac{\mathsf{R}(r,p,\mu) \quad \mu(p) = (q, (\langle tag\rangle\ [:attr]\ \ell_0 + c + \ell_1))}{\langle v,\mu,W,r\rangle \to \langle c, \mu[p \mapsto (q, (\langle tag\rangle\ [:attr]\ \ell_0 + \ell_1))], W, r\rangle}\ (\textsc{ScriptC})$$

$$\frac{\mathsf{Q}(r,p,\mu) \quad \mu(p) = (q, (\langle tag\rangle\ \mathsf{:onclick}\ c_0\ \ell))}{\langle c_1,\mu,W,r\rangle \to \langle c_1,\mu,W \cup \{c_0\}, r\rangle}\ (\textsc{OnclickC})$$

$$\frac{p \notin \mathsf{dom}(\mu)}{\mathbf{S}[(\langle tag\rangle)] \parallel \mu \to \mathbf{S}[p] \parallel \mu \cup \{p \mapsto (\alpha, (\langle tag\rangle))\}}\ (\textsc{Tag1S})$$

$$\frac{p \notin \mathsf{dom}(\mu)}{\mathbf{S}[(\langle tag\rangle\ \mathsf{:onclick}\ w)] \parallel \mu \to \mathbf{S}[p] \parallel \mu \cup \{p \mapsto (\alpha, (\langle tag\rangle\ \mathsf{:onclick}\ \dagger w))\}}\ (\textsc{Tag2S})$$

$$\frac{p \notin \mathsf{dom}(\mu)}{\langle \mathbf{C}[(\langle tag\rangle\ [:attr])], \mu, W, r\rangle \to \langle \mathbf{C}[p], \mu \cup \{p \mapsto (\alpha, (\langle tag\rangle\ [:attr]))\}, W, r\rangle}\ (\textsc{TagC})$$

$$\frac{\mu(p) = (\pi, a_0) \quad \mu(q) = (q', a_1) \quad \mu(q') = (\pi', a_2) \quad \neg\mathsf{R}(q,p,\mu)}{\mathbf{S}[(\mathsf{dom\text{-}appchild!}\, p\, q)] \parallel \mu \to \mathbf{S}[()] \parallel \mu \begin{bmatrix} p \mapsto (\pi, a_0 + q), \\ q \mapsto (p, a_1), \\ q' \mapsto (\pi', a_2 - q) \end{bmatrix}}\ (\textsc{Append1S})$$

$$\frac{\mu(p) = (\pi, a_0) \quad \mu(q) = (\alpha, a_1) \quad \neg\mathsf{R}(q,p,\mu)}{\mathbf{S}[(\mathsf{dom\text{-}appchild!}\, p\, q)] \parallel \mu \to \mathbf{S}[()] \parallel \mu \begin{bmatrix} p \mapsto (\pi, a_0 + q), \\ q \mapsto (p, a_1) \end{bmatrix}}\ (\textsc{Append2S})$$

$$\frac{\mu(p) = (\pi, a_0) \quad w \notin \mathcal{P}ointer}{\mathbf{S}[(\mathsf{dom\text{-}appchild!}\, p\, w)] \parallel \mu \to \mathbf{S}[()] \parallel \mu[p \mapsto (\pi, a_0 + \dagger w)]}\ (\textsc{Append3S})$$

$$\frac{\mu(p) = (\pi, b_0) \quad \mu(q) = (q', b_1) \quad \mu(q') = (\pi', b_2) \quad \neg\mathsf{R}(q,p,\mu)}{\langle \mathbf{C}[(\mathsf{dom\text{-}appchild!}\, p\, q)], \mu, W, r\rangle \to \langle \mathbf{C}[()], \mu \begin{bmatrix} p \mapsto (\pi, b_0 + q), \\ q \mapsto (p, b_1), \\ q' \mapsto (\pi', b_2 - q) \end{bmatrix}, W, r\rangle}\ (\textsc{Append1C})$$

$$\frac{\mu(p) = (\pi, b_0) \quad \mu(q) = (\alpha, b_1) \quad \neg\mathsf{R}(q,p,\mu)}{\langle \mathbf{C}[(\mathsf{dom\text{-}appchild!}\, p\, q)], \mu, W, r\rangle \to \langle \mathbf{C}[()], \mu \begin{bmatrix} p \mapsto (p', b_0 + q), \\ q \mapsto (p, b_1) \end{bmatrix}, W, r\rangle}\ (\textsc{Append2C})$$

Fig. 8. HOP semantics extended for DOM.

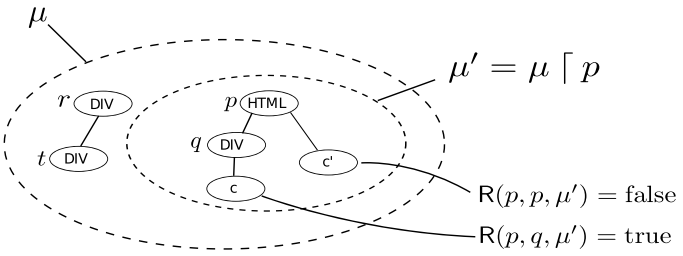

Fig. 9. Example: The predicate R.

client is waiting for an HTML tree (with a root) to be provided by the server. This is formalized in rule SERVRET1S: the server sends a root r, together with the associated tree $\mu \lceil r$, which should satisfy some well-formedness condition to be displayed by the browser. Here we only require that the node at r denotes an $\langle$HTML$\rangle$ node, without any ancestor. In this rule the evaluation of (setdoc r), which is supposed to have the (here invisible) side effect of displaying something of $\mu \lceil r$, immediately returns r. The SERVRET2S rule is the same as SERVRETS of Core HOP, except that some forest may also be returned, which should not conflict with the current client's HTML forest. The reader will notice the asymmetry between the rules for passing a value from the server to a client (SERVRET1S & 2), which "drags" a tree with it, and for passing a value from a client to the server, as an argument for a service call (SERVINC), which does not pass a tree. This is because we have found no interesting use for that and modern browsers does not naturally support that. Consequently, in the current version of HOP it is an error to use a client's node on the server.

The document sent by the server to the client upon initialization may contain some code to execute, and also opportunities for interactions with the user, which in our simplifying presentation of the HOP language only consists in onclick c expressions. Then there is a phase in which the browser, while interpreting the document sent by the server, will execute client code that is contained in the document. This is expressed by the SCRIPTC rule, where the predicate $\mathsf{R}(r, p, \mu)$ means that p is a descendant of r in μ, and that the code that we find at node p, and which is to be triggered, is the leftmost one in the tree $\mu \lceil r$ determined by r. An example of the predicate R is given in Figure 9. (We should also check that this tree is still a valid HTML document. We do not formally define this predicate here; this is straightforward.) When this has been done, the client may interact with the server, by clicking on an active node. This is expressed by the rule ONCLICKC, where again we have a precondition $\mathsf{Q}(r, p, \mu)$, meaning that p is a descendant from r in μ, and that there is no code left to execute (by means of the SCRIPTC rule) in the tree (again we do not formally define this predicate here).

We have already explained the next three rules, from TAG1S to TAGC, that describe the construction of the server (resp. client) node in the forest from the ($\langle tag \rangle$) and ($\langle tag \rangle$:onclick s_0) (resp., ($\langle tag \rangle$ [:*attr*])) expressions. For each case we just create a corresponding new node in the forest. In the rules for the server we see that the server values are transformed into client values or client code by means of the † function.

The remaining rules describe how the DOM operation (dom-appchild! $s_0\, s_1$) computes: first the expressions s_0 and s_1 have to be evaluated. They are supposed to return pointers p and q (or pointer p and value w) pointing to nodes in the forest. Then one updates the node at p, moving q as a new child of p (or simply adding w as a new child of p, respectively), as well as the node at the ancestor of q (if any) which loses its q child, and we update q's ancestor to be p. It is easy to formalize the other DOM constructs in a similar way [Gardner et al. 2008a, 2008b].

Let us illustrate the semantics with DOM extension with a few examples. For all the following examples, we start with a configuration where there is a service $(\mathsf{lambda}\,(z)\,s_0)$ available at URL u_0, that is with $\rho_0 = \{u_0 \mapsto (\mathsf{lambda}\,(z)\,s_0)\}$.

Example 2 *Tree Transmission*. This example demonstrates how the DOM tree is manipulated and transmitted from server-side to client-side. Let

$$\begin{aligned}
s_0 &= (\mathsf{let}\,((x\,(\mathsf{service}\,(y)\,(\langle\mathsf{DIV}\rangle\,y))))\,s_1)\\
s_1 &= (\mathsf{let}\,((d\,(\langle\mathsf{DIV}\rangle\,()))\,s_2)\\
s_2 &= (\mathsf{let}\,((h\,(\langle\mathsf{HTML}\rangle\,d)))\,s_3)\\
s_3 &= (\mathsf{let}\,((k\,(\mathsf{dom\text{-}appchild!}\,h\,(\langle\mathsf{DIV}\rangle\sim t))))\,h)\\
t &= (\mathsf{with\text{-}hop}\,(\$x\,())\\
&\qquad (\mathsf{lambda}\,(x)\,(\mathsf{dom\text{-}appchild!}\,\$d\,x)))
\end{aligned}$$

The transitions are shown in Figure 10, where the service ships an HTML tree containing a piece of client node. The client code, evaluated in client-side, requests a new tree from the server and appends it to the current document.

Example 3 *Script Node*. This example shows how script nodes are evaluated in client-side, especially the evaluation order.

$$\begin{aligned}
s_0 &= (\mathsf{let}\,((d\,(\langle\mathsf{DIV}\rangle\sim t_0)))\,s_1)\\
s_1 &= (\mathsf{let}\,((h\,(\langle\mathsf{HTML}\rangle\,d)))\,s_2)\\
s_2 &= (\mathsf{let}\,((c\,(\mathsf{dom\text{-}appchild!}\,h\,(\langle\mathsf{DIV}\rangle\sim t_1))))\,h)\\
t_0 &= ((\mathsf{lambda}\,(y)\,y)\,())\\
t_1 &= ((\mathsf{lambda}\,(x)\,x)\,())
\end{aligned}$$

We then have transitions shown below, where transitions regarding tree construction and transmission in server-side are omitted. The tree transmitted to the client contains two pieces of code. The left one c_0 will be evaluated before the right one c_1.

$$\begin{aligned}
\rho_0 \;\to\;& (j((\mathsf{lambda}\,(z)\,s_0)\,())) && (\textsc{InitC})\\
& \parallel \langle(), \emptyset, \{(\mathsf{setdoc}\,j)\}, \alpha\rangle \parallel \rho_0 \parallel \{j\}\\
\xrightarrow{*}\;& \mu_0 \parallel \langle q, \mu_1, \emptyset, q\rangle \parallel \rho_1 && (\textsc{AppS}, \textsc{VarS},\\
& \text{where } \mu_1 = \{q \mapsto (\alpha, (\langle\mathsf{HTML}\rangle\,p\,r)),\, p \mapsto (q, (\langle\mathsf{DIV}\rangle\,c_0))\} && \textsc{Tag1S},\\
& \qquad\qquad \cup \{r \mapsto (q, (\langle\mathsf{DIV}\rangle\,c_1))\} && \textsc{Append3S},\\
& \qquad c_0 = ((\mathsf{lambda}\,(y)\,y)\,()) \text{ and } c_1 = ((\mathsf{lambda}\,(x)\,x)\,()) && \textsc{Append2S})\\
\xrightarrow{*}\;& \mu_0 \parallel \langle(), \mu_2, \emptyset, q\rangle \parallel \rho_1 && (\textsc{ScriptC},\\
& \text{where } \mu_2 = \mu_1[p \mapsto (q, (\langle\mathsf{DIV}\rangle\,\varepsilon))] \cup \{y' \mapsto ()\} && \textsc{AppS}, \textsc{VarS})\\
\xrightarrow{*}\;& \mu_0 \parallel \langle(), \mu_3, \emptyset, q\rangle \parallel \rho_1 && (\textsc{ScriptC},\\
& \text{where } \mu_3 = \mu_2[r \mapsto (q, (\langle\mathsf{DIV}\rangle\,\varepsilon))] \cup \{x' \mapsto ()\} && \textsc{AppS}, \textsc{VarS})
\end{aligned}$$

Example 4 *Event Handler*. This example demonstrates the ability to run code by invoking an "onclick" attribute.

$$\begin{aligned}
s_0 &= (\langle\mathsf{HTML}\rangle\,(\langle\mathsf{DIV}\rangle\,\mathsf{:onclick}\sim t))\\
t &= ((\mathsf{lambda}\,(x)\,x)\,())
\end{aligned}$$

$$
\begin{array}{lll}
\rho_0 \;\to\; (j\,((\mathsf{lambda}\,(z)\,s_0)\,()))
\;\|\; \langle (), \emptyset, \{(\mathsf{setdoc}\,j)\}, \alpha\rangle \;\|\; \rho_0 \;\|\; \{j\} & & (\textsc{InitC}) \\
\;\xrightarrow{*}\; (j\,(\mathsf{let}\;((x\,u_1))\;s_1)) \;\|\; \mu_0 & & (\textsc{AppS},\\
\;\|\; \langle (), \emptyset, \{(\mathsf{setdoc}\,j)\}, \alpha\rangle \;\|\; \rho_1 \;\|\; \{j\} & & \textsc{ServDefS}) \\
\quad \text{where } \mu_0 = \{z' \mapsto ()\} & & \\
\qquad\quad \rho_1 = \rho_0 \cup \{u_1 \mapsto (\mathsf{lambda}\,(y)\,(\langle \mathsf{DIV}\rangle\, y))\} & & \\
\;\xrightarrow{*}\; (j\,(\mathsf{let}\;((d\,p))\;s_2')) \;\|\; \mu_1 & & (\textsc{AppS}, \textsc{VarS},\\
\;\|\; \langle (), \emptyset, \{(\mathsf{setdoc}\,j)\}, \alpha\rangle \;\|\; \rho_1 \;\|\; \{j\} & & \textsc{Tag1S},\\
\quad \text{where } s_2' = s_2\{x'/x\} & & \textsc{Append3S}) \\
\qquad\quad \mu_1 = \mu_0 \cup \{x' \mapsto u_1, p \mapsto (\alpha, (\langle \mathsf{DIV}\rangle\, ()))\} & & \\
\;\xrightarrow{*}\; (j\,(\mathsf{let}\;((h\,q))\;s_3')) \;\|\; \mu_2 & & (\textsc{AppS}, \textsc{VarS},\\
\;\|\; \langle (), \emptyset, \{(\mathsf{setdoc}\,j)\}, \alpha\rangle \;\|\; \rho_1 \;\|\; \{j\} & & \textsc{Tag1S}\\
\quad \text{where } s_3' = s_3\{x'/x, d'/d\} & & \textsc{Append3S}) \\
\qquad\quad \mu_2 = \mu_1[p \mapsto (q, (\langle \mathsf{DIV}\rangle\, ()))] & & \\
\qquad\qquad \cup \{d' \mapsto p, q \mapsto (\alpha, (\langle \mathsf{HTML}\rangle\, p))\} & & \\
\;\xrightarrow{*}\; (j\,(\mathsf{let}\;((k\,()))\;h')) \;\|\; \mu_3 & & (\textsc{AppS}, \textsc{VarS},\\
\;\|\; \langle (), \emptyset, \{(\mathsf{setdoc}\,j)\}, \alpha\rangle \;\|\; \rho_1 \;\|\; \{j\} & & \textsc{TildeS}, \textsc{Tag1S},\\
\quad \text{where } \mu_3 = \mu_2[q \mapsto (\alpha, (\langle \mathsf{HTML}\rangle\, p\, r))] & & \textsc{Append3S}\\
\qquad\qquad \cup \{h' \mapsto q, r \mapsto (q, (\langle \mathsf{DIV}\rangle \sim c))\} & & \textsc{Append2S}) \\
\qquad\quad c = (\mathsf{with\text{-}hop}\,(u_1\,())\,(\mathsf{lambda}\,(x)\,(\mathsf{dom\text{-}appchild!}\,p\,x))) & & \\
\;\xrightarrow{*}\; (j\,q) \;\|\; \mu_4 & & (\textsc{AppS},\\
\;\|\; \langle (), \emptyset, \{(\mathsf{setdoc}\,j)\}, \alpha\rangle \;\|\; \rho_1 \;\|\; \{j\} & & \textsc{VarS}) \\
\quad \text{where } \mu_4 = \mu_3 \cup \{k' \mapsto ()\} & & \\
\;\to\; \mu_4 \;\|\; \langle q, \mu_5, \emptyset, q\rangle \;\|\; \rho_1 & & (\textsc{ServRetS}) \\
\quad \text{where } \mu_5 = \{q \mapsto (\alpha, (\langle \mathsf{HTML}\rangle\, p\, r)), p \mapsto (q, (\langle \mathsf{DIV}\rangle\, ()))\} & & \\
\qquad\quad \cup \{r \mapsto (q, (\langle \mathsf{DIV}\rangle\, c))\} & & \\
\;\to\; \mu_4 \;\|\; \langle c, \mu_6, \emptyset, q\rangle \;\|\; \rho_1 & & (\textsc{ScriptC}) \\
\quad \text{where } \mu_6 = \mu_5[r \mapsto (q, (\langle \mathsf{DIV}\rangle\, \varepsilon))] & & \\
\;\xrightarrow{*}\; (j\,((\mathsf{lambda}\,(y)\,(\langle \mathsf{DIV}\rangle\, y))\,())) \;\|\; \mu_4 & & (\textsc{ReqC},\\
\;\|\; \langle (), \mu_6, \{((\mathsf{lambda}\,(x)\,(\mathsf{dom\text{-}appchild!}\,p\,x))\,j)\}, q\rangle \;\|\; \rho_1 \;\|\; \{j\} & & \textsc{ServInC}) \\
\;\xrightarrow{*}\; (j\,r') \;\|\; \mu_7 & & (\textsc{AppS},\\
\;\|\; \langle (), \mu_6, \{((\mathsf{lambda}\,(x)\,(\mathsf{dom\text{-}appchild!}\,p\,x))\,j)\}, q\rangle \;\|\; \rho_1 \;\|\; \{j\} & & \textsc{Tag1S}) \\
\quad \text{where } \mu_7 = \mu_4 \cup \{y' \mapsto (), r' \mapsto (\alpha, (\langle \mathsf{DIV}\rangle\, ()))\} & & \\
\;\xrightarrow{*}\; \mu_7 \;\|\; \langle (), \mu_8, \emptyset, q\rangle \;\|\; \rho_1 & & (\textsc{ServRetS},\\
\quad \text{where } \mu_8 = \mu_6[p \mapsto (q, (\langle \mathsf{DIV}\rangle\, ()\, r'))] & & \textsc{CallbackC}, \textsc{AppC},\\
\qquad\quad \cup \{r' \mapsto (p, (\langle \mathsf{DIV}\rangle\, ())), x' \mapsto r'\} & & \textsc{VarC}, \textsc{Append2C})
\end{array}
$$

Fig. 10. DOM extensions example: Tree transmission.

Here are some states in the execution of this program:

$$
\begin{array}{lll}
\rho_0 \;\to\; (j((\mathsf{lambda}\,(z)\,s_0)\,()))) & & (\textsc{InitC}) \\
\quad \parallel \langle (), \emptyset, \{(\mathsf{setdoc}\,j)\}, \alpha\rangle \parallel \rho_0 \parallel \{j\} & & \\
\quad \stackrel{*}{\to}\; \mu_0 \parallel \langle q, \mu_1, \emptyset, q\rangle \parallel \rho_1 & & (\textsc{AppS}, \textsc{VarS}, \\
\quad \text{where } \mu_1 = \{q \mapsto (\alpha, (\langle \mathsf{HTML}\rangle\, p)), & & \textsc{Tag1S}, \textsc{Tag2S}, \\
\qquad\qquad p \mapsto (q, (\langle \mathsf{DIV}\rangle\, \mathsf{:onclick}\, c))\} & & \textsc{Append3S}, \textsc{Append2S}, \\
\qquad c = ((\mathsf{lambda}\,(x)\,x)\,()) & & \textsc{TildeS}, \textsc{ServRetS}) \\
\quad \stackrel{*}{\to}\; \mu_0 \parallel \langle (), \mu_2, \emptyset, q\rangle \parallel \rho_1 & & (\textsc{OnclickC}, \textsc{CallbackC}, \\
\quad \text{where } \mu_2 = \mu_1 \cup \{x' \mapsto ()\} & & \textsc{AppC}, \textsc{VarC})
\end{array}
$$

Example 5. This example shows that only a valid HTML document is meaningful as an answer to an initial client request. Let

$$
\begin{aligned}
s_0 &= (\langle \mathsf{DIV}\rangle\, {\sim}t) \\
t &= ((\mathsf{lambda}\,(x)\,x)\,())
\end{aligned}
$$

Since the returning tree will not be a valid HTML document, the computation will be blocked by rule SERVRET1S.

4. SAME ORIGIN POLICY AND INLINING CODE

We have presented the core HOP semantics and DOM extension in a stand-alone setting where only one server and one client exist. There are some subtleties in extending the semantics so that it can manage many servers and many clients, where the Same Origin Policy restricts communication between servers and clients. This policy is enforced by all modern browsers. However, it is often considered as over-restrictive for contemporary Web applications, where client mashups require integrating contents originating from different Web sites. This is usually done by "cross-site scripting" on purpose, which is an exception to the Same Origin Policy, allowing browsers to dynamically load code from different places. In this section, we discuss how HOP semantics can be extended to handle the Same Origin Policy as well as "cross-site scripting" (inlining code, in HOP terms), two seemingly contradictory features of Web applications.

4.1. Same Origin Policy

In the core HOP semantics we confined ourselves to only one server and one client. In order to express the Same Origin Policy, we have to extend the semantics to allow many servers and clients to coexist in the global configuration. For this extension, no modification of program syntax is necessary.

The Same Origin Policy, in the setting of HOP, meaning that a client code can only invoke services from a server where the code is originated. We shall describe the Same Origin Policy and behavior of inlining code within the semantics rules.

We introduce a new set $\mathcal{D}omain$ for denoting domain names, to distinguish between different servers, and identify the origin of clients. In a more realistic setting, the origin is composed of the protocol (HTTP or HTTPS), a domain name, and a port number. We use only domain names, for ease of representation. We update the definition of components in a global configuration, so that it may contain many servers and clients.

— Each runtime server-side thread s (possibly in the form of (js')) is now coupled with a domain $d \in \mathcal{D}omain$, and thus becomes (d, s). Similarly, server-side stores are

$$\frac{\mu(x) = w}{(d, \mathbf{E}[x]) \parallel (d, \mu) \to (d, \mathbf{E}[w]) \parallel (d, \mu)} \ (\textsc{VarS})$$

$$\frac{x \in \mathsf{dom}(\mu)}{\langle d, \mathbf{E}[(\mathsf{set!}\, x\, v)], \mu, W\rangle \to \langle d, \mathbf{E}[()], \mu[x \mapsto v], W\rangle} \ (\textsc{SetC})$$

$$\frac{u \notin \mathsf{dom}(\rho)}{(d, \mathbf{E}[(\mathsf{service}\,(x)\, s)]) \parallel \rho \to (d, \mathbf{E}[u]) \parallel \rho \cup \{u \mapsto (d, (\mathsf{lambda}\,(x)\, s))\}} \ (\textsc{ServDefS})$$

$$\frac{\begin{array}{ll} c = \mathbf{E}[(\mathsf{with\text{-}hop}\, u?v_0\, v_1)] & \rho(u) = (d, w) \\ j \notin J \qquad W' = W \cup \{(v_1\, j)\} & J' = J \cup \{j\} \end{array}}{\langle d, c, \mu, W\rangle \parallel \rho \parallel J \to (d, (j\,(w\, v_0))) \parallel \langle d, \mathbf{E}[()], \mu, W'\rangle \parallel \rho \parallel J'} \ (\textsc{ServInC})$$

$$\frac{j \notin J \qquad \rho(u) = (d, w)}{\rho \parallel J \to (d, (j\,(w\, v))) \parallel \langle d, (), \emptyset, \{(\mathsf{setdoc}\, j)\}\rangle \parallel \rho \parallel J \cup \{j\}} \ (\textsc{InitC})$$

Fig. 11. Core HOP semantics modified for SOP.

paired with a domain $d \in \mathcal{D}omain$. A server thread in domain d, that is, (d, s), can only interact with the store associated with the same domain;

— Each client configuration now has the form $\langle d, c, \mu, W\rangle$, where d denotes the origin (i.e., from which server) of the contents in the client;

— The environment ρ now binds URLs to a pair made of a domain name and the service it denotes, that is, $(d, (\mathsf{lambda}\,(x)\, s))$.

The other components are left unchanged, and therefore the syntax for configurations with multiple servers and clients is as follows:

$$\Gamma \ ::= \ (d, \mu) \ \mid \ \rho \ \mid \ \mathsf{Web} \ \mid \ \mathcal{J} \ \mid \ (d, s) \ \mid \ \langle d, c, \mu, W\rangle \ \mid \ (\Gamma \parallel \Gamma).$$

As before, we only consider *well-formed* configurations that contain exactly one ρ, one Web and one $\mathcal{J}$.

The semantics rules given in Figure 4 still hold except for InitC, ServDefS and ServInC. The modified rules are given in Figure 11. This figure also shows two examples, namely VarS and SetC, of rules that are adapted to incorporate the new origin components, which are left unchanged along the corresponding transitions. When initiating a new client, its origin is set as the domain of the corresponding server. This is formalized by the rule InitC. The domain of a service is the one of the server creating it, as expressed by rule ServDefS. The rule ServInC exhibits the Same Origin Policy: the client can only send requests to services that are in the same domain as the client.

Implementation Remarks. Although the Same Origin Policy is often portrayed as a unified security concept implemented in modern browsers, there are subtle differences between policies on different browser resources, as spotted by Google Browser Security Handbook[4]. For example, the origin of XMLHttpRequest is determined by protocol,

[4]http://code.google.com/p/browsersec/wiki/Main

domain name, and port number of a URL. However, the origin for cookies is *not* bound to a protocol and port number. That is, a server with address `http://server:8080` can share cookies with a server with address `https://server:80`. But a client from one of these servers cannot send XMLHttpRequest to the other server. When extending the semantics for covering aspects such as cookies, one has to take these subtleties into consideration.

4.2 Inlining Code

The Same Origin Policy imposes a strict restriction on what (data or code) can be requested by a client, and this is usually seen as a bottleneck for developing modern Web applications, where data are mashed up from different Web sites in a single application. Therefore, there are exceptions to this policy. In particular, code inlining (or cross-site scripting) is widely used as a solution to escape the Same Origin Policy. By inlining code, a client can dynamically load code from *any source*, and execute it in its own execution environment. We extend the language considered up to now to dynamically load mashup code from clients. Namely, we add a new kind of server value:

$$w \ ::= \ \cdots \ | \ (\langle \mathsf{INLINE} \rangle \, u).$$

The server value $(\langle \mathsf{INLINE} \rangle \, u)$ is used to generate client mashups, where executable code downloaded from the URL u on the Web is executed in the client's environment. The syntax for node expressions a is also extended as follows:

$$\ell \ ::= \ \cdots \ | \ (\langle \mathsf{INLINE} \rangle \, u).$$

The semantics of code inlining is based on the semantics of the DOM constructs. Recall that the predicate $\mathsf{R}(r, p, \mu)$ means that p is a descendant of r in μ, and that the code that we find at node p, and which is to be triggered, is the leftmost one in the tree $\mu \lceil r$ determined by r. Now we extend the meaning of "code" in the definition of R, where it can be client code c or inlining node $(\langle \mathsf{INLINE} \rangle \, u)$. When we encounter a inlining node to execute, we shall pull the code from the Web, as formalized by the following rule:

$$\frac{\mathsf{Web}(u) = \sim c \quad \mathsf{R}(r, p, \mu) \quad \mu(p) = (q, (\langle tag \rangle \ \ \ell_0 + (\langle \mathsf{INLINE} \rangle \, u) + \ell_1))}{\mathsf{Web} \parallel \langle d, v, \mu, W \rangle \rightarrow \mathsf{Web} \parallel \langle d, c, \mu[p \mapsto (q, (\langle tag \rangle \ \ \ell_0 + \ell_1))], W \rangle} \ (\textsc{InlineC}).$$

One can observe that, as opposed to SERVINC, this rule does not confine the origin of dynamically loaded code.

5. STATIC REASONING ABOUT REQUEST SAFETY

In this section we demonstrate how the semantics can be used to formally reason about request safety in Web applications.

5.1. Request-Safety Property

We introduce a request-safety property for core HOP, namely clients will not issue a service request for a URL which is not defined in ρ (that is, a URL that is not bound to any server-side programs). For example, assuming $\rho = \{u \mapsto (\mathsf{lambda}\,(x)\,s)\}$, where

$$s = \sim(\mathsf{with\text{-}hop}\,(u'\,())\,c) \quad \text{and} \quad u \neq u'.$$

If the client starts by invoking the service at u, then the shipped client code will invoke the service at u'. Since u' is not defined in ρ, the client will get stuck. To formally express this property, we modify the semantics (one rule added) such that calling a nonexistent service is a runtime error (represented by err):

$$\frac{u \notin \mathsf{dom}(\rho)}{\langle \mathbf{E}[(\text{with-hop}\, u?v_0\, v_1)], \mu, W\rangle \parallel \rho \to \mathsf{err} \parallel \langle \mathbf{E}[\emptyset], \mu, W\rangle \parallel \rho} \ (\textsc{ServInCErr}).$$

In the rest of this section, we use $\to$ to refer to the core HOP semantics with rule ServInCErr.

Definition 6 *Request-Safety*. A closed server-side expression s is *request-safe* if for any global configuration Γ' reachable from the initial configuration $\Gamma = ((\{s\}, \emptyset), \emptyset, \emptyset, \mathsf{Web}, \emptyset)$, that is $\Gamma \to^* \Gamma'$, we have $\mathsf{err} \notin \Gamma'$.

In this definition we assume that every server starts with a single closed server-side expression, with other components empty in the global configuration.

5.2. A Type System for Request-Safety

We present a type system (for server-side expressions) to ensure that generated client code will never send requests to nonexistent URLs. The type system we use is quite standard. The syntax of types is:

$$T, T_0, T_1, \ldots \triangleq E \mid U \mid R \mid T_0 \to T_1,$$

where E is the type for primitive values (not functions). Type U is a special type for URLs such that a URL in type U is generated by evaluating an expression $(\mathsf{service}\,(x)\,s)$. Type R is a type for expressions that evaluates to requests on a valid URL. $T_0 \to T_1$ is a type for functions. The typing judgment has the form:

$$\Lambda \vdash s : T \quad \text{or} \quad \Lambda \vdash_{\Lambda_s} t : T,$$

where $\vdash$ is used to type server-side expressions, and $\vdash_{\Lambda_s}$ is used to type client code confined by the $\sim$-operator.

Finally, Λ is a typing context, assigning types to variables. Specially, Λ_s in $\vdash_{\Lambda_s}$ represents the typing context for server variables when typing tilde code.

We also formally define the update for typing contexts:

$$\Lambda \lhd \Lambda'(x) = \begin{cases} \Lambda'(x) & \text{if } x \in \mathsf{dom}(\Lambda') \\ \Lambda(x) & \text{otherwise.} \end{cases}$$

We define the subtyping relation to express that type E is the more general and least precise type. There is also a coercion between type E and type $E \to E$. The reason for that is that arrow types which do not contain type U or R are not important for preservation of the safety property we are considering.

$$\frac{}{T \preceq T} \quad \frac{}{U \preceq E} \quad \frac{}{R \preceq E} \quad \frac{}{E \to E \preceq E}$$

$$\frac{}{E \preceq E \to E} \quad \frac{T_0' \preceq T_0 \quad T_1 \preceq T_1'}{T_0 \to T_1 \ \preceq \ T_0' \to T_1'}.$$

The type system of server code is given in Figure 12. The type system of client code is given in Figure 13. Explanations for some of the typing rules in Figure 12 follow (the rest of the typing rules are standard and their explanation omitted).

$$\frac{}{\Lambda \vdash x : \Lambda(x)}\ (\text{TVAR}) \quad \frac{}{\Lambda \vdash () : E}\ (\text{TUNSPEC})$$

$$\frac{}{\Lambda \vdash u : E}\ (\text{TURL}) \quad \frac{}{\Lambda \vdash u?w : E}\ (\text{TREQ1}) \quad \frac{\emptyset \vdash_\Lambda t : T}{\Lambda \vdash {\sim}t : E}\ (\text{TTILDE})$$

$$\frac{\Lambda \vdash s : T_0 \quad T_0 \preceq T_1}{\Lambda \vdash s : T_1}\ (\text{TSUB}) \quad \frac{\Lambda \lhd x : T_0 \vdash s : T}{\Lambda \vdash (\mathsf{lambda}\ (x)\ s) : T_0 \to T}\ (\text{TFUN})$$

$$\frac{\Lambda \vdash s_0 : T_0 \to T \quad \Lambda \vdash s_1 : T_0}{\Lambda \vdash (s_0\ s_1) : T}\ (\text{TAPP})$$

$$\frac{\Lambda \vdash s : T \quad \Lambda(x) = T}{\Lambda \vdash (\mathsf{set!}\ x\ s) : E}\ (\text{TSET}) \quad \frac{\Lambda \lhd x : E \vdash s : T}{\Lambda \vdash (\mathsf{service}\ (x)\ s) : U}\ (\text{TSERVDEF})$$

$$\frac{\Lambda \vdash s_0 : U \quad \Lambda \vdash s_1 : T}{\Lambda \vdash (s_0\ s_1) : R}\ (\text{TREQ2}) \quad \frac{\Lambda \vdash s_0 : T_0 \quad \Lambda \vdash s_1 : E \to T}{\Lambda \vdash (\mathsf{with\text{-}hop}\ s_0\ s_1) : T}\ (\text{TWHOP})$$

Fig. 12. Type system of server code for request-safety.

TUNSPEC. The value unspecified is not a URL. Hence it is typed with type E.
TURL. A URL value that does not come from a service definition expression cannot be considered of type U.
TREQ1. A URL value with an argument that does not come from a service definition expression cannot be considered of type U.
TTILDE. A server side tilde expression is typable if its client expression is typable.
TSET. A set expression is always typed as E since it does not return a valid URL by semantics. Type T binds the types for x and s to be the same.
TSERVDEF. The service definition expression is the only expression typed as U since at evaluation it generates a URL that will not violate the SOP.
TREQ2. In order to distinguish an application $(s_0\ s_1)$ that generates a correct argument for with-hop from other kinds of applications, we use the type R. This kind of application requires that s_0 is of type U, that is, s_0 evaluates to a URL that does not violate SOP.
TWHOP. This expression as server code has the meaning of calling other services from the server side. Therefore the SOP is not applicable in this case. The typing system only restricts the type of the continuation s_1 to be $E \to T$ for some T, since there is no guarantee that the returned value from the service is of a certain type other than the fact that the returned value is typable. The with-hop expression has type T since the invocation is synchronous, and the continuation s_1 will be applied after the service returned.

Typing rules in Figure 13 for tilde client code are similar to server rules (for convenience in defining typing rules for configurations we keep the two sets of rules separated). The main differences appear in the rule for the with-hop expression, TWHOP, where t_0 should be typed as a correct argument for with-hop (see explanation of rule TREQ2 above) and TCDOLLAR is typed using the server context Λ_s, since a variable with a dollar in front is a variable that belongs to the server. Note that in rule TWHOP, the with-hop expression has type E, since on the client-side with-hop behave asynchronously, returning immediately with a unit value ().

$$\frac{}{\Lambda \vdash_{\Lambda_s} x : \Lambda(x)}\ (\text{TCVAR}) \qquad \frac{}{\Lambda \vdash_{\Lambda_s} () : E}\ (\text{TCUNSPEC})$$

$$\frac{}{\Lambda \vdash_{\Lambda_s} u : E}\ (\text{TCURL}) \qquad \frac{}{\Lambda \vdash_{\Lambda_s} u?v : E}\ (\text{TCREQ1})$$

$$\frac{\Lambda \vdash_{\Lambda_s} s : T_0 \quad T_0 \preceq T_1}{\Lambda \vdash_{\Lambda_s} s : T_1}\ (\text{TCSUB}) \qquad \frac{\Lambda \lhd x : T_0 \vdash_{\Lambda_s} s : T}{\Lambda \vdash_{\Lambda_s} (\mathsf{lambda}\,(x)\,s) : T_0 \to T}\ (\text{TFUN})$$

$$\frac{\Lambda \vdash_{\Lambda_s} s_0 : T_0 \to T \quad \Lambda \vdash_{\Lambda_s} s_1 : T_0}{\Lambda \vdash_{\Lambda_s} (s_0\, s_1) : T}\ (\text{TCAPP})$$

$$\frac{\Lambda \vdash_{\Lambda_s} s : T \quad \Lambda(x) = T}{\Lambda \vdash_{\Lambda_s} (\mathsf{set!}\, x\, s) : E}\ (\text{TCSET}) \qquad \frac{}{\Lambda \vdash_{\Lambda_s} \$x : \Lambda_s(x)}\ (\text{TCDOLLAR})$$

$$\frac{\Lambda \vdash_{\Lambda_s} t_0 : U \quad \Lambda \vdash_{\Lambda_s} t_1 : T}{\Lambda \vdash_{\Lambda_s} (t_0\, t_1) : R}\ (\text{TCREQ2}) \qquad \frac{\Lambda \vdash_{\Lambda_s} t_0 : R \quad \Lambda \vdash_{\Lambda_s} t_1 : E \to T}{\Lambda \vdash_{\Lambda_s} (\mathsf{with\text{-}hop}\, t_0\, t_1) : E}\ (\text{TWHOP})$$

Fig. 13. Type system of client code for request-safety.

Example 7 *Typable and not Typable Expressions.* Consider two independent server expressions:

$$(a)\ \ \sim(\mathsf{with\text{-}hop}\,(\$(\mathsf{service}\,(x)\,(\mathsf{lambda}\,(x)\,s))\,())\,(\mathsf{lambda}\,(x)\,()))$$
$$(b)\ \ \sim(\mathsf{with\text{-}hop}\,(u\,())\,(\mathsf{lambda}\,(x)\,()))$$

If expression s is typable, the expression (a) is typable since the service called is defined by a service definition. However, the expression (b) is not typable because URL u cannot be proved to have been generated from a service definition.

5.3. Lemmas and Proofs

In this section we prove the theorem of type soundness, which states that typable code complies to the safety-request property. The proof follows the classical approach of proving type safety [Wright and Felleisen 1994].

Runtime Typing System. We modify the type system to type runtime expressions and configurations. The runtime typing judgment of expressions has the following form:

$$\Lambda, \rho \Vdash s : T \quad \text{or} \quad \Lambda, \rho \Vdash_c c : T.$$

For runtime server code typing, each rule is augmented with an additional component ρ as typing condition. All rules except TURL and TREQ1 are modified in a minor way—the addition of ρ as context for the typing—that has no impact. Rules TURL and TREQ1 are replaced by the following set of rules:

$$\frac{u \in \mathsf{dom}(\rho)}{\Lambda, \rho \Vdash u : U}\ (\text{TURL}) \qquad \frac{u \notin \mathsf{dom}(\rho)}{\Lambda, \rho \Vdash u : E}\ (\text{TURL}')$$

$$\frac{u \in \mathsf{dom}(\rho)}{\Lambda, \rho \Vdash u?w : R}\ (\text{TREQ1}) \qquad \frac{u \notin \mathsf{dom}(\rho)}{\Lambda, \rho \Vdash u?w : E}\ (\text{TREQ1}').$$

In a runtime configuration, a URL coming from environment ρ is considered correct (evaluated by a service definition expression in the server) and hence typed as U.

Similarly, for runtime client code typing, all rules except TCDOLLAR, TCURL and TCREQ1 are unmodified in an essential way. Notice that $\Vdash_c$ can be used to type standalone client code. Therefore the rule TCDOLLAR is removed and the typing judgment is replaces by $\Vdash_c$ instead of $\vdash_{\Lambda_s}$, where Λ_s is used to provide typing information in rule TCDOLLAR. Rules TCURL and TCREQ1 are replaced by the following set of rules:

$$\frac{u \in \mathsf{dom}(\rho)}{\Lambda, \rho \Vdash_c u : U}\ (\text{TCURL}) \quad \frac{u \notin \mathsf{dom}(\rho)}{\Lambda, \rho \Vdash_c u : E}\ (\text{TCURL}')$$

$$\frac{u \in \mathsf{dom}(\rho)}{\Lambda, \rho \Vdash_c u?v : R}\ (\text{TCREQ1}) \quad \frac{u \notin \mathsf{dom}(\rho)}{\Lambda, \rho \Vdash_c u?v : E}\ (\text{TCREQ1}').$$

Definition 8 *Typable Store.* We say that μ is typable, denoting $\Lambda, \rho \Vdash_* \mu$, if and only if $\forall x.x \in \mathsf{dom}(\mu) \Rightarrow \Lambda, \rho \Vdash_* \mu(x) : \Lambda(x)$.

We also add two rules for typing runtime configurations. We overload $\Vdash$ and $\Vdash_c$ for typing global configurations and client configurations.

$$\frac{\begin{array}{c}\Lambda, \rho \Vdash \mu \\ \forall u \in \mathsf{dom}(\rho).\ \Lambda, \rho \Vdash \rho(u) : T \text{ for some } T \\ \forall s \in S.\ \Lambda, \rho \Vdash s : T' \ \text{ or } \ s = (js') \wedge \Lambda, \rho \Vdash s' : T' \text{ for some } T' \\ \forall b \in C.\ \exists \Lambda_c.\ \Lambda_c, \rho \Vdash_c b : T'' \text{ for some } T''\end{array}}{\Lambda \Vdash ((S, \mu), C, \rho, \mathsf{Web}, J)}$$

$$\frac{\begin{array}{c}\Lambda, \rho \Vdash_c \mu \\ \Lambda, \rho \Vdash_c c : T \text{ for some } T \\ \forall c' \in W.\ \Lambda, \rho \Vdash_c c' : T' \ \text{ or } \ c' = (c''\ j) \wedge \Lambda, \rho \Vdash_c c'' : T' \text{ for some } T'\end{array}}{\Lambda, \rho \Vdash_c \langle c, \mu, W \rangle.}$$

Essentially, a configuration is typable if all of its components are. Typing of store μ is defined in Definition 8. Typing of a HOP environment is possible only if all server expressions that it defines are typable. A server configuration is typable if all its threads are typable. The same applies to a client configuration. Finally, a client thread is typable if the client expression is typable, its store is typable as defined in Definition 8, and all continuations in W are typable.

LEMMA 1 REPLACEMENT. *If* $\Lambda, \rho \Vdash_* \mathbf{E}[s] : T$*, then there exist* Λ' *and* T' *such that* $\Lambda', \rho \Vdash_* s : T'$*, and for any* s' *such that* $\Lambda', \rho \Vdash_* s' : T'$ *we have* $\Lambda, \rho \Vdash_* \mathbf{E}[s'] : T$.

PROOF. The proof proceeds by induction on the definition of evaluation context $\mathbf{E}$. We show two important cases. Other cases follow the same reasoning.

Case $\mathbf{E} = []$. By the assumption $\Lambda, \rho \Vdash_* \mathbf{E}[s] : T$, we have $\Lambda, \rho \Vdash_* s : T$. Therefore for any s' such that $\Lambda, \rho \Vdash_* s' : T$, we have $\Lambda, \rho \Vdash_* \mathbf{E}[s'] : T$.

Case $\mathbf{E} = (\mathbf{E}'\, s')$.

By the assumption $\Lambda, \rho \Vdash_* \mathbf{E}[s] : T$ and typing rule TAPP and TCAPP, $\Lambda, \rho \Vdash_* \mathbf{E}'[s] : T'$.

By the inductive hypothesis, $\Lambda', \rho' \Vdash_* s : T''$ for some T'', and for any s'' such that $\Lambda', \rho' \Vdash_* s'' : T''$, $\Lambda, \rho \Vdash_* \mathbf{E}'[s''] : T$.

By typing rule TAPP, we have $\Lambda, \rho \Vdash_* \mathbf{E}[s''] : T$. □

LEMMA 2 SUBSTITUTION. *If* $\Lambda \lhd x : T_0, \rho \Vdash_* s : T_1$, *and* $y \notin \mathsf{dom}(\Lambda)$, *then* $\Lambda \lhd y : T_0, \rho \Vdash_* s\{y/x\} : T_1$.

PROOF. Since the typing rules of client code are a subset of typing rules of server code. We need only to prove the case of server code.

Let us prove by induction of the definition of typing rules of server code. We show a few important cases only. The other cases follow similar reasoning.

Case TVAR.

By the assumption, $\Lambda \lhd x : T_0, \rho \Vdash x' : T'$.

If $x = x'$, then we have $T' = T_0$ and $\Lambda \lhd y : T_0, \rho \Vdash x'\{y/x'\} : T_0$.

If $x \neq x'$, then we have $T' = \Lambda(x')$ and $\Lambda \lhd y : T_0, \rho \Vdash x'\{y/x\} : \Lambda(x')$.

Case TUNSPEC. By the assumption, $\Lambda \lhd x : T_0, \rho \Vdash () : E$. Thus we have $\Lambda \lhd y : T_0, \rho \Vdash ()\{y/x\} : E$.

Case TTILDE.

By the assumption, $\Lambda \lhd x : T_0, \rho \Vdash \sim t : E$.

By Lemma 3, $\Lambda \lhd y : T_0, \rho \Vdash (\sim t)\{y/x\} : E$.

Case TFUN.

By the assumption, $\Lambda \lhd x : T_0, \rho \Vdash (\mathsf{lambda}\,(x')\,s) : T' \to T$.

By the typing rule, $\Lambda \lhd x : T_0 \lhd x' : T', \rho \Vdash s : T$.

If $x = x'$:

— Since $x \notin \mathsf{fv}((\mathsf{lambda}\,(x)\,s))$, $\Lambda, \rho \Vdash (\mathsf{lambda}\,(x)\,s) : T' \to T$ by Lemma 4.
— By the assumption $y \notin \mathsf{dom}(\Lambda)$, $y \notin \mathsf{fv}((\mathsf{lambda}\,(x)\,s))$.
— By Lemma 4, $\Lambda \lhd y : T_0, \rho \Vdash (\mathsf{lambda}\,(x)\,s)\{y/x\} : T' \to T$.

If $x \neq x'$:

— By the definition of updating typing context, $\Lambda \lhd x' : T' \lhd x : T_0, \rho \Vdash s : T$.
— By the inductive hypothesis, $\Lambda \lhd x' : T' \lhd y : T_0, \rho \Vdash s\{x/y\} : T$.
— By the definition of updating typing context, $\Lambda \lhd y : T_0 \lhd x' : T', \rho \Vdash s\{x/y\} : T$.
— By the typing rule, $\Lambda \lhd y : T_0, \rho \Vdash (\mathsf{lambda}\,(x')\,s)\{y/x\} : T' \to T$.

Case TAPP.

By the assumption, $\Lambda \lhd x : T_0, \rho \Vdash (s_0\, s_1) : T$.

By the typing rule, $\Lambda \lhd x : T_0, \rho \Vdash s_0 : T' \to T$ and $\Lambda \lhd x : T_0, \rho \Vdash s_1 : T'$.

By the inductive hypothesis, $\Lambda \lhd y : T_0, \rho \Vdash s_0\{y/x\} : T' \to T$ and $\Lambda \lhd y : T_0, \rho \Vdash s_1\{y/x\} : T'$.

By the typing rule, $\Lambda \lhd y : T_0, \rho \Vdash (s_0\, s_1)\{y/x\} : T$

Case TSET.

By the assumption, $\Lambda \lhd x : T_0, \rho \Vdash (\mathsf{set!}\, x'\, s) : E$.

If $x = x'$:

— By the typing rule, $\Lambda \lhd x : T_0, \rho \Vdash s : T_0$.

— By the inductive hypothesis, $\Lambda \lhd y : T_0, \rho \Vdash s\{y/x\} : T_0$.
— By the typing rule, $\Lambda \lhd y : T_0, \rho \Vdash (\mathsf{set!}\, x\, s)\{y/x\} : E$.

If $x \neq x'$:

— By the typing rule, $\Lambda \lhd x : T_0, \rho \Vdash s : T$ and $T = \Lambda(x')$.
— By the inductive hypothesis, $\Lambda \lhd y : T_0, \rho \Vdash s\{y/x\} : T$.
— By the typing rule, $\Lambda \lhd y : T_0, \rho \Vdash (\mathsf{set!}\, x\, s)\{y/x\} : E$.

We can conclude by similar reasoning for other cases. □

LEMMA 3 SUBSTITUTION - TILDE CODE. *If* $\Lambda \lhd x : T_0, \rho \Vdash {\sim}t : E$, *and* $y \notin \mathsf{dom}(\Lambda)$, $\Lambda \lhd y : T_0, \rho \Vdash ({\sim}t)\{y/x\} : E$.

PROOF. By the typing rule, $\emptyset, \rho \Vdash_{\Lambda \lhd x:T_0} t : T$. It is sufficient to prove $\emptyset, \rho \Vdash_{\Lambda \lhd y:T_0} t\{y//x\} : T$. We prove by induction on the typing rules for tilde code. By the definition of substitution of server-side variables in tilde code, only the case of rule TCDOLLAR needs to be examined.

Case TCDOLLAR.
By the assumption, $t = \$x'$ for some x'.
If $x = x'$:

— By the assumption, $\emptyset, \rho \Vdash_{\Lambda \lhd x:T_0} t : T_0$.
— By the definition, $t\{y//x\} = \$y$.
— By the typing rule, $\emptyset, \rho \Vdash_{\Lambda \lhd y:T_0} t\{y//x\} : T_0$.

If $x \neq x'$:

— By the assumption, $\emptyset, \rho \Vdash_{\Lambda \lhd x:T_0} t : \Lambda(x')$.
— By the definition, $t\{y//x\} = \$x'$.
— By the typing rule, $\emptyset, \rho \Vdash_{\Lambda \lhd y:T_0} t\{y//x\} : \Lambda(x')$.

We can conclude by similar reasoning for other cases. □

LEMMA 4 TYPING CONTEXT. *Given* s *and* Λ *such that* $y \notin \mathsf{fv}(s)$ *and* $y \notin \mathsf{dom}(\Lambda)$, $\Lambda, \rho \Vdash_* s : T$ *if and only if* $\Lambda, \rho \lhd y : T_0 \Vdash_* s : T$.

PROOF. We prove only for the "only if" direction; the "if" direction is similar.
Since the typing rules of client code are a subset of typing rules of server code. We need only to prove the case of server code.
The proof is straightforward by induction on the typing rules of server code. We show only a few important cases.

Case TVAR.
By the assumption, $\Lambda, \rho \Vdash x : \Lambda(x)$.
By the typing rule, $\Lambda', \rho \Vdash x : \Lambda'(x)$, where $\Lambda' = \Lambda \lhd y : T_0$.
By the definition of updating typing context, $\Lambda'(x) = \Lambda(x)$, therefore $\Lambda \lhd y : T_0, \rho \Vdash s : \Lambda(x)$.

Case TTILDE.
By the assumption, $\Lambda, \rho \Vdash {\sim}t : E$.
By the typing rule, $\emptyset, \rho \Vdash_\Lambda t : T'$ for some T'.
It is sufficient to prove $\emptyset, \rho \Vdash_{\Lambda \lhd y:T_0} t : T'$ for some T'.

We can prove the preceding statement by induction on the typing rules of tilde code. Only the case of TCDOLLAR needs to be examined.

— By the inductive hypothesis, $\emptyset, \rho \Vdash_\Lambda \$x : \Lambda(x)$.
— Since $y \notin \mathsf{dom}(\Lambda)$, $y \neq x$.
— By the typing rule, $\emptyset, \rho \Vdash_{\Lambda \lhd y:T_0} \$x : \Lambda(x)$.

Case TSUB. By the assumption, $\Lambda, \rho \Vdash s : T$.
By the typing rule, $\Lambda, \rho \Vdash s : T'$ and $T' \preceq T$.
By the inductive hypothesis, $\Lambda \lhd y : T_0, \rho \Vdash s : T'$.
By the typing rule, $\Lambda \lhd y : T_0, \rho \Vdash s : T$.

Case TFUN.
By the assumption, $\Lambda, \rho \Vdash (\mathsf{lambda}\,(x)\,s)T'" \to : T$.
By the typing rule, $\Lambda \lhd x : T', \rho \Vdash s : T$.
If $y = x$:

— By the definition of updating typing context, $\Lambda \lhd x : T_0 \lhd x : T', \rho \Vdash s : T$.
— By the typing rule, $\Lambda \lhd x : T_0, \rho \Vdash (\mathsf{lambda}\,(x)\,s) : T' \to T$.
— Since $y = x$, $\Lambda \lhd y : T_0, \rho \Vdash (\mathsf{lambda}\,(x)\,s) : T' \to T$.

If $y \neq x$:

— By the inductive hypothesis, $\Lambda \lhd x : T' \lhd y : T_0, \rho \Vdash s : T$.
— By the definition of updating typing context, $\Lambda \lhd y : T_0 \lhd x : T', \rho \Vdash s : T$.
— By the typing rule, $\Lambda \lhd y : T_0(\mathsf{lambda}\,(x)\,s) : T' \to T$.

We can conclude by similar reasoning for other cases. □

LEMMA 5 TILDE REMOVAL. *If* $\Lambda, \rho \Vdash w : T$ *and* $w \neq \bot$, *then there exists* Λ_c *such that* $\Lambda_c, \rho \Vdash_c w : T'$ *for some* T'.

PROOF. We prove by case analysis on the definition of the function. We show only important cases.

Case $w = ()$.
By the assumption, $\emptyset, \rho \Vdash () : E$.
By the typing rule, $\Lambda', \rho \Vdash_c () : E$ for any Λ'

Case $w = {\sim}c$. By the assumption, $\Lambda, \rho \Vdash {\sim}c : E$.
Since $\mathsf{fv}({\sim}t) = \emptyset$, we have $\emptyset, \rho \Vdash {\sim}c : E$.
By the typing rule, $\emptyset, \rho \Vdash_c c : T$.
By Lemma 4, $\Lambda', \rho \Vdash_c c : T$ for any Λ'. □

LEMMA 6 TILDE CODE TRANSFORMATION. *If* $\Lambda, \rho \Vdash_{\Lambda_s} t : T$; $\Lambda, \rho \Vdash \mu$; *and* $\Xi(\mu, t) = c$, *then* $\Lambda, \rho \Vdash_c c : T$.

PROOF. By the assumption $\Lambda, \rho \Vdash_{\Lambda_s} t : T$. We proceed by induction on the definition of Ξ.

Case $t = x$. By the definition of Ξ, $c = \Xi(\mu, x) = x = t$. Therefore $\Lambda, \rho \Vdash_c c : T$.

Case $t = u$. By the definition of Ξ, $c = \Xi(\mu, u) = u = t$. Therefore $\Lambda, \rho \Vdash_c c : T$.

Case $t = u?v$. By the definition of Ξ, $c = \Xi(\mu, u?v) = u?v = t$. Therefore $\Lambda, \rho \Vdash_c c : T$.
Case $t = (\mathsf{lambda}\,(x)\,t')$.

By the typing rule, $T = T_0 \to T_1$ and $\Lambda \lhd x : T_0, \rho \Vdash_{\Lambda_s} t' : T_1$.
By the inductive hypothesis, $\Lambda \lhd x : T_0, \rho \Vdash_c \Xi(\mu, t') : T_1$.
Hence, by Lemma 2 if $y \notin \Lambda \cup \mathsf{dom}(\mu)$, $\Lambda \lhd y : T_0, \rho \Vdash_c \Xi(\mu, t'\{y//x\}) : T_1$.
By the definition of Ξ, $\Xi(\mu, (\mathsf{lambda}\,(x)\,t')) = (\mathsf{lambda}\,(y)\,\Xi(\mu, t'\{y//x\}))$.
By the typing rule, $\Lambda, \rho \Vdash_c (\mathsf{lambda}\,(y)\,\Xi(\mu, t'\{y//x\})) : T_0 \to T_1$.

Case $t = (\mathsf{set!}\,x\,t')$.
By the typing rule, $T = E$ and $\Lambda, \rho \Vdash_{\Lambda_s} t' : T'$.
By the inductive hypothesis, $\Lambda, \rho \Vdash_c \Xi(\mu, t') : T'$.
By the definition of Ξ, $\Xi(\mu, (\mathsf{set!}\,x\,t')) = (\mathsf{set!}\,x\,\Xi(\mu, t'))$.
By the typing rule, $\Lambda, \rho \Vdash_c (\mathsf{set!}\,x\,\Xi(\mu, t')) : E$.

Case $t = (t_0\,t_1)$. This case follows the same reasoning as previous ones.

Case $t = \$x$.
By the typing rule, $\Lambda_s(x) = T$.
By the assumption $\Lambda_s, \rho \Vdash \mu$, $\Lambda_s, \rho \Vdash \mu(x) : T$.
By the definition of Ξ, $\Xi(\mu, \$x) = \mu(x)$, and $\mu(x)$ must be a primitive value.
By the typing rules, $\Lambda, \rho \Vdash_c \mu(x) : T$.

Case $t = (\mathsf{with\text{-}hop}\,t_0\,t_1)$. This case follows the same reasoning as previous ones. □

LEMMA 7 INITIAL CONFIGURATION. *If* $\Lambda \vdash s : T$, *then* $\Lambda \Vdash ((\{s\}, \emptyset), \emptyset, \emptyset, \mathsf{Web}, \emptyset)$.

PROOF. By the hypothesis $\Lambda \vdash s : T$ and the definition of $\Vdash$, we have $\Lambda, \emptyset \vdash s : T$. Therefore we have $\forall s \in \{s\}.\ \Lambda, \emptyset \Vdash s : T'$ for some T'. We can also straightforwardly conclude the following facts:

— $\Lambda, \emptyset \Vdash \emptyset$ by Definition 8;
— $\forall u \in \mathsf{dom}(\rho).\ \Lambda, \rho \Vdash \rho(u) : T$ for some T, where $\rho = \emptyset$;
— $\forall b \in C.\ \rho \Vdash_c b : T''$ for some T'', where $C = \emptyset$.

Therefore by definition of the typing rule of global configuration, we conclude that $\Lambda \Vdash ((S, \emptyset), \emptyset, \emptyset, \mathsf{Web}, \emptyset)$. □

LEMMA 8 SUBJECT REDUCTION. *If* $\Lambda \Vdash \Gamma$ *and* $\Gamma \to \Gamma'$, *then there exist* Λ' *such that* $\Lambda' \Vdash \Gamma'$.

PROOF. The proof proceeds by case analysis on transition $\Gamma \to \Gamma'$. Let us assume $\Gamma = ((S, \mu), C, \rho, \mathsf{Web}, J) \to \Gamma'$.

Case VARS. By the core semantics, we have $\mathbf{E}[x] \in S$ such that

$$\frac{\mu(x) = w}{\mathbf{E}[x] \parallel \mu \to \mathbf{E}[w] \parallel \mu}\ (\text{VARS}).$$

By the hypothesis of typability of global configurations, $\Lambda, \rho \Vdash \mathbf{E}[x] : T$ and $\Lambda, \rho \Vdash \mu$ hold.
By Lemma 1, $\Lambda, \rho \Vdash x : \Lambda(x)$ holds.
By Definition 8, $\Lambda, \rho \Vdash \mu(x) : \Lambda(x)$ holds.
By Lemma 1, $\Lambda, \rho \Vdash \mathbf{E}[w] : T$ holds.
By the core semantics, $\Gamma' = ((S', \mu), C, \rho, \mathsf{Web}, J)$ where $S' = (S \backslash \{\mathbf{E}[x]\}) \cup \{\mathbf{E}[w]\}$.

By the hypothesis of typability of global configurations, we have $\forall s \in S'.\ \Lambda, \rho \Vdash s : T'$ for some T'.
By the typing rule of global configuration, we have $\Lambda \Vdash \Gamma'$.

Case VARC. By the core semantics, we have:

$$\frac{\mu(x) = v}{\langle \mathbf{E}[x], \mu, W\rangle \to \langle \mathbf{E}[v], \mu, W\rangle}\ (\textsc{VarC}).$$

The proof of this case is similar to the case of VARS.

Case SETS. By the core semantics, we have:

$$\frac{x \in \mathsf{dom}(\mu)}{\mathbf{E}[(\mathsf{set!}\, x\, w)] \parallel \mu \to \mathbf{E}[()] \parallel \mu[x \mapsto w]}\ (\textsc{SetS}).$$

By hypothesis of typability of global configurations, $\Lambda, \rho \Vdash \mathbf{E}[(\mathsf{set!}\, x\, w)] : T$ and $\Lambda, \rho \Vdash \mu$ holds.
By Lemma 1, $\Lambda, \rho \Vdash (\mathsf{set!}\, x\, w) : E$ holds.
By Lemma 1 and typing rule TUNSPEC, $\Lambda, \rho \Vdash \mathbf{E}[()] : T$ holds.
By typing rule TSET, $\Lambda, \rho \Vdash w : \Lambda(x)$.
By the core semantics, $\Gamma' = ((S', \mu'), C, \rho, \mathsf{Web}, J)$ where $S' = (S \backslash \{\mathbf{E}[x]\}) \cup \{\mathbf{E}[w]\}$, and $\mu' = \mu[x \mapsto w]$.
By hypothesis of typability of global configurations, we have $\forall s \in S'.\ \Lambda, \rho \Vdash s : T'$ for some T'.
By Definition 8, $\Lambda, \rho \Vdash \mu$, and $\Lambda, \rho \Vdash \mu'(x) : \Lambda(x)$, we can infer that $\Lambda, \rho \Vdash \mu'$. By typing rule of global configuration, we have $\Lambda \Vdash \Gamma'$.

Case SETC. By the core semantics:

$$\frac{x \in \mathsf{dom}(\mu)}{\langle \mathbf{E}[(\mathsf{set!}\, x\, v)], \mu, W\rangle \to \langle \mathbf{E}[()], \mu[x \mapsto v], W\rangle}\ (\textsc{SetC}).$$

The proof of this case is similar to the case of SETS.

Case REQS. By the core semantics:

$$\frac{}{\mathbf{E}[(u\, w)] \to \mathbf{E}[u?w]}\ (\textsc{ReqS}).$$

By the hypothesis of typability of global configurations, $\Lambda, \rho \Vdash \mathbf{E}[(u\, w)] : T$ holds.
By Lemma 1, $\Lambda, \rho \Vdash (u\, w) : T'$ holds.
By the typing rules, T' is either R or not.
If T' is R:

— By the typing rules, we have $\Lambda, \rho \Vdash u : U$ and $\Lambda, \rho \Vdash w : T''$.
— By the typing rules, we can infer that $u \in \rho$.
— By the typing rules, we have $\Lambda, \rho \Vdash u?w : R$.

If T' is not R:

— By the typing rules, we have $\Lambda, \rho \Vdash u : E$.
— By the typing rules, we can infer that $u \notin \rho$.
— By the typing rules, we have $\Lambda, \rho \Vdash u?w : E$.

By the above analysis, $\Lambda, \rho \Vdash u?w : T'$ holds.
By Lemma 1, $\Lambda, \rho \Vdash \mathbf{E}[u?w] : T$ holds.
By the core semantics, $\Gamma' = ((S', \mu), C, \rho, \mathsf{Web}, J)$ where $S' = (S \backslash \{\mathbf{E}[(u\,w)]\}) \cup \{\mathbf{E}[u?w]\}$.
By the hypothesis of typability of global configurations, we have $\forall s \in S'.\ \Lambda, \rho \Vdash s : T'$ for some T'.
By the typing rule of global configuration, we have $\Lambda \Vdash \Gamma'$.

Case REQC. By the core semantics:

$$\frac{}{\langle \mathbf{E}[(u\,v)], \mu, W\rangle \rightarrow \langle \mathbf{E}[u?v], \mu, W\rangle}\ (\text{REQC}).$$

The proof of this case is similar to the case of REQS.

Case APPS. By the core semantics:

$$\frac{y \notin \mathsf{dom}(\mu)}{\mathbf{E}[((\mathsf{lambda}\,(x)\,s)w)] \parallel \mu \rightarrow \mathbf{E}[s\{y/x\}] \parallel \mu \cup \{y \mapsto w\}}\ (\text{APPS}).$$

By the hypothesis of typability of global configurations, $\Lambda, \rho \Vdash \mathbf{E}[((\mathsf{lambda}\,(x)\,s)w)] : T$ holds.
By Lemma 1, $\Lambda, \rho \Vdash ((\mathsf{lambda}\,(x)\,s)w) : T'$ holds.
By the core semantics, we have $y \notin \mathsf{dom}(\mu)$ and $y \notin \mathsf{dom}(\Lambda)$.
By the typing rules TAPP, there are two cases to consider:
If $\Lambda, \rho \Vdash (\mathsf{lambda}\,(x)\,s) : T_0 \rightarrow U$:

— By the typing rule, $T' = E$.
— By the typing rule, $\Lambda, \rho \Vdash w : T_0$.
— By the typing rule, $\Lambda \lhd x : T_0, \rho \Vdash s : U$.
— By Lemma 2, we have $\Lambda \lhd y : T_0, \rho \Vdash s\{y/x\} : U$.
— By the subtyping rule, we have $\Lambda \lhd y : T_0, \rho \Vdash s\{y/x\} : E$.
— By Lemma 1, we have $\Lambda \lhd y : T_0, \rho \Vdash \mathbf{E}[s\{y/x\}] : T$.

If $\Lambda, \rho \Vdash (\mathsf{lambda}\,(x)\,s) : T_0 \rightarrow T_1$

— By the typing rule, $T' = T_1$.
— By the typing rule, $\Lambda, \rho \Vdash w : T_0$.
— By the typing rule, $\Lambda \lhd x : T_0, \rho \Vdash s : T_1$.
— By Lemma 2, we have $\Lambda \lhd y : T_0, \rho \Vdash s\{y/x\} : T_1$.
— By Lemma 1, we have $\Lambda \lhd y : T_0, \rho \Vdash \mathbf{E}[s\{y/x\}] : T$.

By Definition 8, we have $\Lambda \lhd y : T_0, \rho \Vdash \mu'$, where $\mu' = \mu \cup \{y \mapsto w\}$.
By Lemma 4, and the typability of global configurations, other components in the configuration remains typable. Therefore we can conclude $\Lambda \lhd y : T_0 \Vdash \Gamma'$

Case APPC. By the core semantics:

$$\frac{y \notin \mathsf{dom}(\mu)}{\langle \mathbf{E}[((\mathsf{lambda}\,(x)\,c)v)], \mu, W\rangle \rightarrow \langle \mathbf{E}[c\{y/x\}], \mu \cup \{y \mapsto v\}, W\rangle}\ (\text{APPC}).$$

The proof of this case is similar to the case of APPC.

Case SERVINS. By the core semantics:

$$\frac{u \notin \mathsf{dom}(\rho) \qquad \mathsf{Web}(u?w_0) = w}{\mathbf{E}[(\mathsf{with\text{-}hop}\, u?w_0\, w_1)] \parallel \rho \to \mathbf{E}[(w_1\, w)] \parallel \rho} \ (\textsc{ServInS}).$$

By the hypothesis of typability of global configurations,
$\Lambda, \rho \Vdash \mathbf{E}[(\mathsf{with\text{-}hop}\, u?w_0\, w_1)] : T$ holds.
By Lemma 1, $\Lambda, \rho \Vdash (\mathsf{with\text{-}hop}\, u?w_0\, w_1) : T'$.
By the typing rule, $\Lambda, \rho \Vdash w_1 : E \to T'$.
By the typing rule, $\Lambda, \rho \Vdash (w_1\, w) : T'$.
By Lemma 1, $\Lambda, \rho \Vdash \mathbf{E}[(w_1\, w)] : T$.
By the core semantics, $\Gamma' = ((S', \mu), C, \rho, \mathsf{Web}, J)$
where $S' = (S \backslash \{\mathbf{E}[(\mathsf{with\text{-}hop}\, u?w_0\, w_1)]\}) \cup \{\mathbf{E}[(w_1\, w)]\}$.
By the hypothesis of typability of global configurations, we have $\forall s \in S'.\ \Lambda, \rho \Vdash s : T'$ for some T'.
By the typing rule of global configuration, we have $\Lambda \Vdash \Gamma'$.

Case SERVDEFS. By the core semantics:

$$\frac{u \notin \mathsf{dom}(\rho)}{\mathbf{E}[(\mathsf{service}\,(x)\,s)] \parallel \rho \to \mathbf{E}[u] \parallel \rho \cup \{u \mapsto (\mathsf{lambda}\,(x)\,s)\}} \ (\textsc{ServDefS}).$$

By the hypothesis of typability of global configurations, $\Lambda, \rho \Vdash \mathbf{E}[(\mathsf{service}\,(x)\,s)] : T$ holds.
By Lemma 1, $\Lambda, \rho \Vdash (\mathsf{service}\,(x)\,s) : U$.
By the definition of typability of configuration, $\Lambda, \rho' \Vdash u : U$, where $\rho' = \rho \cup \{u \mapsto (\mathsf{lambda}\,(x)\,s)\}$.
By Lemma 1, $\Lambda, \rho' \Vdash \mathbf{E}[u] : T$.
By the core semantics, $\Gamma' = ((S', \mu), C, \rho', \mathsf{Web}, J)$
where $S' = (S \backslash \{\mathbf{E}[(\mathsf{with\text{-}hop}\, u?w_0\, w_1)]\}) \cup \{\mathbf{E}[(w_1\, w)]\}$.
By the hypothesis of typability of global configurations, we have $\forall s' \in S'.\ \Lambda, \rho \Vdash s' : T'$ for some T'.
By the typing rule, we have $\Lambda, \rho \Vdash s' : T'$.
Since $\rho'(u) = s'$, we have $\Lambda, \rho' \Vdash \rho(u) : T'$.
By the hypothesis of typability of global configurations, $\forall u \in \mathsf{dom}(\rho').\ \Lambda, \rho \Vdash \rho(u) : T$ for some T.

Case SERVINC. By the core semantics:

$$\frac{j \notin J \qquad \rho(u) = w \qquad W' = W \cup \{(v_1\, j)\} \qquad J' = J \cup \{j\}}{\langle \mathbf{E}[(\mathsf{with\text{-}hop}\, u?v_0\, v_1)], \mu, W\rangle \parallel \rho \parallel J \to (j(w\, v_0)) \parallel \langle \mathbf{E}[0], \mu, W'\rangle \parallel \rho \parallel J'} \ (\textsc{ServInC}).$$

By the hypothesis of typability of global configurations,
$\rho \Vdash_c \langle \mathbf{E}[(\mathsf{with\text{-}hop}\, u?v_0\, v_1)], \mu, W\rangle$ holds.
By the definition of typability of client configurations,
$\Lambda_c, \rho \Vdash_c \mathbf{E}[(\mathsf{with\text{-}hop}\, u?v_0\, v_1)] : T$.
By Lemma 1, we have $\Lambda_c, \rho \Vdash_c (\mathsf{with\text{-}hop}\, u?v_0\, v_1) : T'$.
By the typing rule, we have $\Lambda_c, \rho \Vdash_c u?v_0 : R$.
By the typing rule, we have $u \in \rho$.
By the typability of global configuration, we have $\Lambda, \rho \Vdash w : T''$.
By the typing rule, we have $\Lambda_c, \rho \Vdash_c (j\, v_1) : T'''$.
By the typability of configurations, we have $\Lambda \Vdash \Gamma'$.

Case SERVINCERR. By the core semantics:

$$\frac{u \notin \mathsf{dom}(\rho)}{\langle \mathbf{E}[(\mathsf{with\text{-}hop}\, u?v_0\, v_1)], \mu, W\rangle \parallel \rho \rightarrow \mathsf{err} \parallel \langle \mathbf{E}[()], \mu, W\rangle \parallel \rho}\ (\textsc{ServInCErr}).$$

By the typability of global configurations $\Lambda, \rho \vdash_{\Lambda_s} \mathbf{E}[(\mathsf{with\text{-}hop}\, u?v_0\, v_1)] : T$.
By Lemma 1, $\Lambda, \rho \vdash_{\Lambda_s} (\mathsf{with\text{-}hop}\, u?v_0\, v_1) : T'$.
However, it implies that $u \in \mathsf{dom}(\rho)$ which is contradictory with the premise in the semantics rule. Therefore we can conclude that the transition cannot be of this case.

Case SERVRETS. By the core semantics, we have the following transition:

$$\frac{}{(j\, w) \parallel \langle c, \mu, W \cup \{(v\, j)\}\rangle \parallel J \rightarrow \langle c, \mu, W \cup \{(v\, (w))\}\rangle \parallel J - \{j\}}\ (\textsc{ServRetS}).$$

By the hypothesis of typability of global configurations, $\Lambda, \rho \Vdash w : T$, and $\rho \Vdash_c \langle c, \mu, W \cup \{(v\, j)\}\rangle$.
By Lemma 5, we have $\Lambda_c, \rho \Vdash_c w : T$ for some Λ_c.
By the typability of client configurations, we have $\rho \Vdash_c \langle c, \mu, W \cup \{(v\, (w))\}\rangle$.
By the typability of configurations, we have $\Lambda \Vdash \Gamma'$.

Case TILDES. By the core semantics, we have the following transition:

$$\frac{\Xi(\mu, t) = c}{\mathbf{E}[\sim t] \parallel \mu \rightarrow \mathbf{E}[\sim c] \parallel \mu}\ (\textsc{TildeS}).$$

By the hypothesis of typability of global configurations, we have $\Lambda, \rho \Vdash \mathbf{E}[\sim t] : T$;
By Lemma 1, $\Lambda, \rho \Vdash \sim t : E$;
By the typing rule, we have $\emptyset, \rho \Vdash_\Lambda t : T'$ for some T'.
By Lemma 6, we have $\emptyset, \rho \Vdash_c c : T'$.
By the typing rule, we have that $\Lambda, \rho \Vdash \sim c : E$.
By Lemma 4, we have $\Lambda, \rho \Vdash \sim c : E$.
By Lemma 1, $\Lambda, \rho \Vdash \mathbf{E}[\sim c] : T$.
By the typability of configurations, we have $\Lambda \Vdash \Gamma'$.

Case CALLBACKC. By the core semantics, we have the following transition:

$$\frac{}{\langle v, \mu, \{c\} \cup W\rangle \rightarrow \langle c, \mu, W\rangle}\ (\textsc{CallbackC}).$$

By the hypothesis of typability of global configurations, we have $\rho \Vdash_c \langle v, \mu, \{c\} \cup W\rangle$.
By the definition, we have $\Lambda_c \rho \Vdash_c c : T$. Therefore $\langle c, \mu, W\rangle$.
By the typability of global configurations, we have $\Lambda \Vdash \Gamma'$.

Case INITC. By the core semantics, we have the following transition:

$$\frac{j \notin J \qquad \rho(u) = w}{\langle v, \mu, \emptyset\rangle \parallel \rho \parallel J \rightarrow (j(w\, v)) \parallel \langle (), \emptyset, \{(\mathsf{setdoc}\, j)\}\rangle \parallel \rho \parallel J \cup \{j\}}\ (\textsc{InitC}).$$

By the hypothesis of typability of global configurations, we have $\Lambda, \rho \vdash w : E \rightarrow T$.
By the typing rules, we have $\Lambda, \rho \vdash v : E$.
By the typing rules, we have $\Lambda, \rho \vdash (j(w\, v)) : T$.
By the definition of typability of client configuration,
we have $\rho \Vdash_c \langle (), \emptyset, \{(\mathsf{setdoc}\, j)\}\rangle$.
By the typability of global configurations, we have $\Lambda \Vdash \Gamma'$. □

We use $\Gamma \rightarrow^n \Gamma_n$ to denote the standard notion that Γ_n can be reached from Γ by n steps.

LEMMA 9 RUNTIME TYPE SAFETY. *If* $\Lambda \Vdash \Gamma$, $\mathsf{err} \notin \Gamma$, *and* $\Gamma \rightarrow^n \Gamma_n$, *then* $\mathsf{err} \notin \Gamma_n$ *and there exist a* Λ_n *such that* $\Lambda_n \Vdash \Gamma_n$.

PROOF. The proof proceeds by induction on n.

Case $n = 0$. By definition of $\rightarrow^n$, $\Gamma_n = \Gamma$. Therefore $\mathsf{err} \notin \Gamma_n$. Let $\Lambda_n = \Lambda$, then $\Lambda_n \Vdash \Gamma_n$.

Case $n > 0$. By definition of $\rightarrow^n$, $\Gamma \rightarrow^{n-1} \Gamma_{n-1}$ and $\Gamma_{n-1} \rightarrow \Gamma_n$. By inductive hypothesis, $\mathsf{err} \notin \Gamma_{n-1}$ and there exist a Λ_{n-1} such that $\Lambda_{n-1} \Vdash \Gamma_{n-1}$.

Now we examine $\Gamma_{n-1} \rightarrow \Gamma_n$. The only way to generate runtime error is by rule SERVINCERR.

$$\frac{u \notin \mathsf{dom}(\rho)}{\langle \mathbf{E}[(\text{with-hop}\, u?v_0\, v_1)], \mu, W\rangle \parallel \rho \rightarrow \mathsf{err} \parallel \langle \mathbf{E}[0], \mu, W\rangle \parallel \rho} \ (\textsc{ServInCErr}).$$

If err is generated, then by the typability of global configurations and Lemma 1, $\Lambda_{n-1}, \rho \vdash_{\Lambda_s} (\text{with-hop}\, u?v_0\, v_1) : T$. There must be that $\Lambda_{n-1}, \rho \vdash_{\Lambda_s} u?v_0 : R$ for $u \notin \mathsf{dom}(\rho)$. It is, however, not possible by typing rules. Thus we can conclude that $\mathsf{err} \notin \Gamma_n$. By Lemma 8, there also exist Λ_n such that $\Lambda_n \Vdash \Gamma_n$. □

Finally we state the theorem for type soundness to prove.

THEOREM 10 TYPE SOUNDNESS. *If* $\emptyset \vdash s : T$ *for some* T *then* s *is request-safe.*

PROOF. Let $\Gamma = ((\{s\}, \emptyset), \emptyset, \emptyset, \mathsf{Web}, \emptyset)$. By Lemma 7, we have $\emptyset \vdash \Gamma$. By Lemma 9, for any Γ' such that $\Gamma \rightarrow^* \Gamma'$, $\mathsf{err} \notin \Gamma'$. □

5.4. Same Origin Policy Safety

In this section we show that previous result of type soundness also holds for the core HOP semantics with the extension of the Same Origin Policy.

In order to model a safety property equivalent to request-safety, we add a semantics rule: calling a service within a different domain is also considered as a runtime error.

$$\frac{u \notin \mathsf{dom}(\rho) \quad \text{or} \quad \rho(u) = (d', s) \qquad d' \neq d}{\langle d, \mathbf{E}[(\text{with-hop}\, u?v_0\, v_1)], \mu, W\rangle \parallel \rho \rightarrow \mathsf{err} \parallel \langle d, \mathbf{E}[0], \mu, W\rangle \parallel \rho} \ (\textsc{ServInCErr}).$$

We denote the core HOP semantics by $\rightarrow_c$, and the core HOP semantics with the extension of the Same Origin Policy by $\rightarrow_s$.

The desired property is now defined as follows.

Definition 9 *SOP-Request-Safety.* A closed server-side expression s is *SOP-request-safe* if for any global configuration Γ' reachable from the initial configuration $\Gamma = ((\{d, s\}, \emptyset), \emptyset, \emptyset, \mathsf{Web}, \emptyset)$, that is $\Gamma \rightarrow_s^* \Gamma'$, we have $\mathsf{err} \notin \Gamma'$.

The following definition will be needed in the proof of extended type soundness. Basically we extend a normal configuration in the core HOP semantics to a corresponding configuration with a given domain d. That is, we assume all computations in the core HOP semantics happen in a single domain d.

Definition 10. Let Γ be a normal configuration in the core HOP semantics, we define Γ_d to be the corresponding configuration, denoting $\Gamma \sim \Gamma_d$, in the SOP extension that satisfy the following conditions:

(1) $\mu \in \Gamma \Leftrightarrow (d, \mu) \in \Gamma_d$;

(2) $\rho \in \Gamma$, $\rho' \in \Gamma_d$, $\forall u \in \mathsf{dom}(\rho) \cup \mathsf{dom}(\rho').\rho(u) = s \Leftrightarrow \rho'(u) = (d, s)$;
(3) $\mathsf{Web} \in \Gamma \Leftrightarrow \mathsf{Web} \in \Gamma_d$;
(4) $J \in \Gamma \Leftrightarrow J \in \Gamma_d$;
(5) $s \in \Gamma \Leftrightarrow (d, s) \in \Gamma_d$;
(6) $\langle c, \mu, W\rangle \in \Gamma \Leftrightarrow \langle d, c, \mu, W\rangle \in \Gamma_d$;
(7) $\mathsf{err} \in \Gamma \Leftrightarrow \mathsf{err} \in \Gamma_d$;

This lemma shows that every semantics step for core HOP simulates a semantic step from the SOP-augmented semantics.

LEMMA 11. *For any* Γ *and its corresponding* Γ_d *for any domain d, if* $\Gamma_d \rightarrow_s \Gamma'_d$, *we have* $\Gamma \rightarrow_c \Gamma'$ *and* $\Gamma' \sim \Gamma'_d$.

PROOF. Let us prove by case analysis on the transition $\Gamma_d \rightarrow_s \Gamma'_d$. We show only important cases.

Case VARS. By the SOP extension, we have

$$\frac{\mu(x) = w}{(d, \mathbf{E}[x]) \parallel (d, \mu) \rightarrow (d, \mathbf{E}[w]) \parallel (d, \mu)} \text{ (VARS).}$$

By the core semantics, we have

$$\frac{\mu(x) = w}{\mathbf{E}[x] \parallel \mu \rightarrow \mathbf{E}[w] \parallel \mu} \text{ (VARS).}$$

Since $\mathbf{E}[w] \in \Gamma'$ and $(d, \mathbf{E}[w]) \in \Gamma'_d$, and other parts of the configurations remain unchanged, we have $\Gamma' \sim \Gamma'_d$ by Definition 10.

Case SERVINC. By the SOP extension, we have

$$\frac{\begin{array}{ll} c = \mathbf{E}[(\text{with-hop } u?v_0\ v_1)] & \rho_d(u) = (d, w) \\ j \notin J \quad W' = W \cup \{(v_1\ j)\} & J' = J \cup \{j\} \end{array}}{\langle d, c, \mu, W\rangle \parallel \rho_d \parallel J \rightarrow (d, (j(w\ v_0))) \parallel \langle d, \mathbf{E}[0], \mu, W'\rangle \parallel \rho_d \parallel J'} \text{ (SERVINC).}$$

By the core semantics, we have

$$\frac{j \notin J \quad \rho(u) = w \quad W' = W \cup \{(v_1\ j)\} \quad J' = J \cup \{j\}}{\langle \mathbf{E}[(\text{with-hop } u?v_0\ v_1)], \mu, W\rangle \parallel \rho \parallel J \rightarrow (j(w\ v_0)) \parallel \langle \mathbf{E}[0], \mu, W'\rangle \parallel \rho \parallel J'} \text{ (SERVINC).}$$

By Definition 10, we have $\rho(u) = s$ and $\rho_d(u) = (d, s)$ holds. Since $(j(w\ v_0)) \in \Gamma'$ and $\langle \mathbf{E}[0], \mu, W'\rangle \in \Gamma'$ and $(d, (j(w\ v_0))) \in \Gamma'_d$ and $\langle d, \mathbf{E}[0], \mu, W'\rangle \in \Gamma'_d$, and other parts of the configurations remain unchanged, we have $\Gamma' \sim \Gamma'_d$ by Definition 10.

Case SERVINCERR. By the SOP extension, we have

$$\frac{u \notin \mathsf{dom}(\rho_d) \quad \text{or} \quad \rho_d(u) = (d', s) \qquad d' \neq d}{\langle d, \mathbf{E}[(\text{with-hop } u?v_0\ v_1)], \mu, W\rangle \parallel \rho_d \rightarrow \mathsf{err} \parallel \langle d, \mathbf{E}[0], \mu, W\rangle \parallel \rho_d} \text{ (SERVINCERR).}$$

By the core semantics, we have

$$\frac{u \notin \mathsf{dom}(\rho)}{\langle \mathbf{E}[(\text{with-hop } u?v_0\ v_1)], \mu, W\rangle \parallel \rho \rightarrow \mathsf{err} \parallel \langle \mathbf{E}[0], \mu, W\rangle \parallel \rho} \text{ (SERVINCERR).}$$

By Definition 10, it is not possible that $\rho_d(u) = (d', s)$ and $d' \neq d$. Therefore $u \notin \mathsf{dom}(\rho)$. Since $\mathsf{err} \in \Gamma'$ and $\mathsf{err} \in \Gamma'_d$, and other parts of the configurations remain unchanged, we have $\Gamma' \sim \Gamma'_d$ by Definition 10.

Case SERVDEFS. By the SOP extension, we have

$$\frac{u \notin \mathsf{dom}(\rho_d)}{(d, \mathbf{E}[(\mathsf{service}\,(x)\,s)]) \parallel \rho_d \to (d, \mathbf{E}[u]) \parallel \rho_d \cup \{u \mapsto (d, (\mathsf{lambda}\,(x)\,s))\}} \text{ (SERVDEFS).}$$

By the core semantics, we have

$$\frac{u \notin \mathsf{dom}(\rho)}{\mathbf{E}[(\mathsf{service}\,(x)\,s)] \parallel \rho \to \mathbf{E}[u] \parallel \rho \cup \{u \mapsto (\mathsf{lambda}\,(x)\,s)\}} \text{ (SERVDEFS).}$$

Since ρ is updated with $\{u \mapsto (\mathsf{lambda}\,(x)\,s)\}$, and ρ_d is updated with $p\{u \mapsto (d, (\mathsf{lambda}\,(x)\,s))\}$, and other parts of the configurations remain unchanged, we have $\Gamma' \sim \Gamma'_d$ by Definition 10. □

LEMMA 12. *For any Γ and its corresponding Γ_d for any domain d, if $\Gamma_d \to_s^* \Gamma'_d$, we have $\Gamma \to_c^* \Gamma'$ and $\Gamma' \sim \Gamma'_d$.*

PROOF. The proof is straightforward by Lemma 11 and induction on the length of the transitions. □

The theorem in this section shows that the type system presented in the previous section is also sound with respect to the SOP-request-safe property.

THEOREM 13. *If $\emptyset \vdash s : T$, then (d, s) is SOP-request-safe for any domain d.*

PROOF. By Theorem 10, for $\Gamma = ((\{s\}, \emptyset), \emptyset, \emptyset, \mathsf{Web}, \emptyset)$ such that $\Gamma \to_c^* \Gamma'$, we have $\mathsf{err} \notin \Gamma'$. By Lemma 12, for $\Gamma_d = ((\{(d, s)\}, \emptyset), \emptyset, \emptyset, \mathsf{Web}, \emptyset)$, we have $\Gamma_b \to_s^* \Gamma'_b$. By Definition 10 $\mathsf{err} \notin \Gamma'_b$. □

COROLLARY 14. *Let $s_1 \ldots s_n$ be n SOP-request-safe expressions, and $d_1 \ldots d_n$ be n different domains. Let $\Gamma = ((\{(d_1, s_1) \ldots (d_n, s_n)\}, \emptyset), \emptyset, \emptyset, \mathsf{Web}, \emptyset)$. For any Γ' such that $\Gamma \to_s^* \Gamma'$, $\mathsf{err} \notin \Gamma'$.*

SKETCH OF PROOF. Let us assume that $\mathsf{err} \in \Gamma'$. Since transition among different domains are orthogonal. There must exist a $\Gamma_i = ((\{(d_i, s_i)\}, \emptyset), \emptyset, \emptyset, \mathsf{Web}, \emptyset)$, and $\Gamma_i \to_s^* \Gamma'_i$, such that Γ'_i is a projection of Γ' with respect to domain d_i and $\mathsf{err} \in \Gamma'_i$. It is, however, contradictory to the assumption that s_i is SOP-request-safe, which implies $\mathsf{err} \notin \Gamma'_i$. □

6. CONCLUSION

We present a formal specification of Web applications via a small-step operational semantics for the HOP programming language. The specification models HTTP requests and responses, AJAX requests in the browser, DOM trees, program behavior on client side, and on server side. The operational semantics faithfully models the behavior of

HOP programs [Serrano et al. 2006] that include the features considered in this work. We do not model certain features of the language such as inclusion of CSS specifications and cookies. Neither do we give a complete specification of the DOM, although we believe that the given specification is enough to easily build further DOM elements and functions [Gardner et al. 2008a] in future extensions. We also model the same origin policy (SOP) and code inlining. The formalization has allowed us to benefit from language-based techniques for analysis and verification. We have specified a static type system that enforces a request safety policy with respect to SOP. We have formally proved the soundness of the type system using the small step operational semantics. Our next goal is to use the operational semantics to study and propose language-based mechanims for security concerns and develop a certified trustworthy HOP compiler.

APPENDIX

A. DIRECT IMPLEMENTATION OF HOP SMALL-STEP SEMANTICS

We have implemented a tool that allows to display all the steps in the semantics execution for a given HOP program. We have used the tool to generate the examples given in the body of the article. The tool can be found in the following URL: `http://www-sop.inria.fr/members/Zhengqin.Luo/trwa-long/`. Its code is listed as follows.

```
;   The module
(module hopsos
   (main main))

;   verbose ...
(define trace '())

;   Senv ...
(define Senv (make-env))

;   W ...
(define W '())

;   main ...
(define (main args)
   (let ((in #f))
      (args-parse (cdr args)
   (("-g?x")
    (if (string=? x "")
        (bigloo-debug-set! (+fx 1 (bigloo-debug)))
        (bigloo-debug-set! (string->integer x))))
   (else
    (set! in else)))

      (let ((exp (if (string? in)
         (call-with-input-file in read)
         (read))))
   (set! trace '())
   (set! Senv '())
   (set! W (list (cons 'C `(init ,exp))))
   (let loop ()
```

```
32        (when (pair? W)
33          (with-trace 1 "loop"
34        (trace-item "W=" W)
35        (let ((w (car W)))
36          (set! W (cdr W))
37          (print (evaluate (cdr w) (make-env) (car w)))
38          (loop)))))
39     (print "trace: " (reverse trace)))))
40
41  ;    make-env ...
42  (define (make-env)
43     `((setdoc . (lambda (x) ()))))
44
45  ;    env-lookup ...
46  (define (env-lookup id env)
47     (with-trace 4 "env-lookup"
48        (trace-item "id=" id " env=" env)
49        (let ((c (assoc id env)))
50     (if (pair? c)
51         (cdr c)
52         (error "env-lookup" "unbound variable" id)))))
53
54  ;    env-extend ...
55  (define (env-extend id val env)
56     (cons (cons id val) env))
57
58  ;    env-update! ...
59  (define (env-update! id val env)
60     (let ((c (assq id env)))
61        (if (pair? c)
62      (set-cdr! c val)
63      (error "env-update!" "unbound variable" id))))
64
65  ;    trace-step! ...
66  (define (trace-step! rule::symbol tier)
67     (set! trace (cons (symbol-append rule tier) trace)))
68
69  ;    make-url ...
70  (define (make-url u v)
71     (list '%url u v))
72
73  ;    url? ...
74  (define (url? x)
75     (and (pair? x) (eq? (car x) '%url)))
76
77  ;    url-u ...
78  (define (url-u url)
79     (cadr url))
80
81  ;    url-v ...
82  (define (url-v u)
83     (caddr u))
84
85  ;    service? ...
86  (define (service? v)
87     (match-case v
```

```
88      ((%service (?-) ?-) #t)
89      (else #f)))
90
91  ;   make-service ...
92  (define (make-service exp)
93    (match-case exp
94      ((service (?x) ?t) `(%service (,x) ,t))))
95
96  ;   Web ...
97  (define (Web u)
98    (match-case u
99      (((service (?x) ?s) ?v ?env)
100      (evaluate s (env-extend x v env) 'S))))
101
102 ;   rho ...
103 (define (rho u)
104   (env-lookup u Senv))
105
106 ;   E ...
107 (define (E env x)
108   (with-trace 3 "E"
109     (trace-item "x=" x)
110     (let ((e (match-case x
111     (() x)
112     ((? symbol?) x)
113     ((? service?) x)
114     ((? url?) x)
115     ((lambda (?x) ?t) `(lambda (x) ,(E env t)))
116     ((set! ?x ?t) `(set! ,x ,(E env t)))
117     ((dollar ?x) (env-lookup x env))
118     ((with-hop ?t0 ?t1) `(with-hop ,(E env t0) ,(E env t1)))
119     ((?t0 ?t1) `(,(E env t0) ,(E env t1)))
120     (else (error "E" "Illegal expression" x)))))
121   (trace-item e)
122   e)))
123
124 ;   lambda-arg ...
125 (define (lambda-arg val)
126   (match-case val
127     ((lambda (?x) ?-)
128      x)))
129
130 ;   lambda-body ...
131 (define (lambda-body val)
132   (match-case val
133     ((lambda (?-) ?t)
134      t)))
135
136 ;   evaluate ...
137 (define (evaluate exp::obj env::pair-nil tier::symbol)
138   (with-trace 1 "evaluate"
139     (trace-item "exp=" exp)
140     (match-case exp
141   ((? symbol?)
142    ;; Var
143    (with-trace 2 (string-append "Var" (symbol->string tier))
```

```
144        (trace-step! 'Var tier)
145        (env-lookup exp env)))
146     (()
147      ;; a literal
148      '())
149     ((or (? url?) ((or %service lambda) (?x) ?e))
150      ;; a value
151      (with-trace 2 (string-append "Val" (symbol->string tier))
152        exp))
153     ((set! ?x ?w)
154      ;; Set
155      (with-trace 2 (string-append "Set" (symbol->string tier))
156        (trace-step! 'Set tier)
157        (env-update! x w env)))
158     ((service (?x) ?s)
159      ;; ServDef
160      (with-trace 2 (string-append "ServDef" (symbol->string tier))
161        (if (eq? tier 'C)
162         (error "evaluate" "service cannot be created on clients" exp)
163         (let ((u (make-service exp)))
164           (trace-step! 'ServDef 'S)
165           (set! Senv (env-extend u exp Senv))
166           u))))
167     ((with-hop ?e1 ?e2)
168      ;; ServIn
169      (with-trace 2 (string-append "ServIn" (symbol->string tier))
170        (if (eq? tier 'S)
171         (let* ((u?w0 (evaluate e1 env tier))
172          (w (Web u?w0))
173          (w1 (evaluate e2 env tier)))
174           (trace-step! 'ServIn 'S)
175           (evaluate `(,w1 ,w) env tier))
176         (let* ((u?v0 (evaluate e1 env tier))
177          (u (url-u u?v0))
178          (v0 (url-v u?v0))
179          (v1 (evaluate e2 env tier))
180          (w (rho u)))
181           (trace-step! 'ServIn 'C)
182           (let ((j (evaluate `(,w ,v0) env 'S)))
183             (set! W (append W (list (cons tier `(return (,v1 ,j))))))))))
184     ((tilde ?t)
185      ;; Tilde
186      (with-trace 2 "Tilde"
187        (trace-step! 'Tilde 'S)
188        (E env t)))
189     ((init ?t)
190      ;; Init
191      (with-trace 2 "Init"
192        (trace-step! 'Init 'C)
193        (let ((j (evaluate t env 'S)))
194      (trace-item "j=" j)
195      (set! W (append W (list (cons 'C `(return (setdoc ,j))))))))
196     ((return ?t)
197      ;; ServRet
198      (with-trace 2 "ServRet"
199        (trace-step! 'ServRet tier)
```

```
200         (evaluate t env tier)))
201       ((S ?t)
202        (evaluate t env 'S))
203       ((C ?t)
204        (evaluate t env 'C))
205       ((let ((?x ?e1)) ?e2)
206        (evaluate `((lambda (,x) ,e2) ,e1) env tier))
207       ((?e1 ?e2)
208        ;; Req/App
209        (with-trace 2 (string-append "Req/App" (symbol->string tier))
210          (let* ((u (evaluate e1 env tier))
211           (w (evaluate e2 env tier)))
212        (if (service? u)
213            (begin
214              (trace-step! 'Req tier)
215              (make-url u w))
216            (let ((s (lambda-body u))
217            (x (lambda-arg u)))
218              (trace-step! 'App tier)
219              (evaluate s (env-extend x w env) tier))))))
220       (else
221        (error "evaluate" "Illegal expression" exp)))))
```

REFERENCES

BERRY, G. AND BOUDOL, G. 1990. The chemical abstract machine. In *Proceedings of the ACM International Conference on Principle of Programming Languages*. ACM Press, New York, 81–94.

BOHANNON, A. AND PIERCE, B. C. 2010. Featherweight Firefox: Formalizing the core of a web browser. In *Proceedings of the Usenix Conference on Web Application Development*. USENIX Association, Berkeley, CA.

BOUDOL, G., LUO, Z., REZK, T., AND SERRANO, M. 2010. Towards reasoning for web applications: An operational semantics for hop. In *Proceedings of the Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications (APLWACA'10)*. ACM, New York, NY, 3–14.

CHLIPALA, A. 2010. Ur: Statically-typed metaprogramming with type-level record computation. In *Proceedings of the International Conference on Programming Languages and Implementation*. ACM.

CHONG, S., LIU, J., MYERS, A., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. 2009. Building secure web applications with automatic partitioning. *Comm. ACM 52*, 2, 79–87.

COOPER, E., LINDLEY, S., WADLER, P., AND YALLOP, J. 2006. Links: Web programming without tiers. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects*. Springer.

COOPER, E. E. AND WADLER, P. 2009. The rpc calculus. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and Practice of Declarative Programming (PPDP'09)*. ACM, New York, NY, 231–242.

FELLEISEN, M. AND HIEB, R. 1992. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci. 103*, 2, 235–271.

GARDNER, P., SMITH, G., WHEELHOUSE, M., AND ZARFATY, U. 2008a. DOM: Towards a formal specification. In *Proceedings of the ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X)*. ACM Press.

GARDNER, P., SMITH, G., WHEELHOUSE, M., AND ZARFATY, U. 2008b. Local Hoare reasoning about DOM. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM Press, 261–270.

GRAUNKE, P., FINDLER, R. B., KRISHNAMURTHI, S., AND FELLEISEN, M. 2003. Modeling web interactions. In *Proceedings of the European Symposium on Programming*. Springer.

GUHA, A., SAFTOIU, C., AND KRISHNAMURTHY, S. 2010. The essence of JavaScript. In *Proceedings of the 24th European Conference on Object-Oriented Programming*. Springer.

HORS, A. L., HEGARET, P. L., NICOL, G., ROBIE, J., CHAMPION, M., AND BYRNE, S. 2000. Document Object Model (DOM) level 2 core specification. Tech. rep., W3C.

KELSEY, R., CLINGER, W. D., AND REES, J. 1998. Revised report on the algorithmic language scheme. *SIGPLAN Not. 33*, 9, 26–76.

LOITSCH, F. AND SERRANO, M. 2008. LOITSCH, F. AND SERRANO, M. 2008. HOP Client-side compilation. In *Trends in Functional Programming*. Vol. 8, M. T. Morazán Ed., Intellect Bristol, UK, 141–158.

MAFFEIS, S., MITCHELL, J., AND TALY, A. 2008. An operational semantics for JavaScript. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Springer.

MATTHEWS, J., FINDLER, R. B., GRAUNKE, P. T., KRISHNAMURTHI, S., AND FELLEISEN, M. 2004. Automatically restructuring programs for the web. *Autom. Softw. Engin. 11*, 4, 337–364.

QUEINNEC, C. 2000. The inuence of browsers on evaluators. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. ACM, 23–33.

SERRANO, M. 2009. HOP, a fast server for the di_use web. In *Proceedings of the 11th International Conference on Coordination Models and Languages*. Lecture Notes in Computer Science, vol. 5521, Springer 1–26.

SERRANO, M., GALLESIO, E., AND LOITSCH, F. 2006. HOP, a language for programming the web 2.0. In *Proceedings of the 1st Dynamic Languages Symposium*. ACM, New York, NY.

SERRANO, M. AND QUEINNEC, C. 2010. A multi-tier semantics for HOP. In *Higher Order and Symbolic Computation*.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Inf. Comput. 115*, 1, 38–94.

Received December 2010; revised August 2011; accepted March 2012