QWeS^{2}T for Type-Safe Web Programming*

Thierry Sans I liano Cervesato
Carnegie Mellon University, Qatar

Abstract

Web applications (webapps) are very popular because they are easy to prototype and they can invoke other external webapps, supplied by third parties, as building blocks. Yet, writing correct webapps is complex because developers are required to reason about distributed computation and to write code using heterogeneous languages, often not originally designed with distributed computing in mind. Testing is the common way to catch bugs as current technologies provide limited support. There are doubts this can scale up to meet the expectations of more sophisticated web applications. In this paper, we propose an abstraction that provides simple primitives to manage the two main forms of distributed computation found on the web: remote procedure calls (code executed on a server on behalf of a client) and mobile code (server code executed on a client). We embody this abstraction in a type-safe language with localized static typechecking that we call QWeS^{2}T and for which we have implemented a working prototype. We use it to express interaction patterns commonly found on the Web as well as more sophisticated forms that are beyond current web technologies.

1 Introduction

Web-based applications, also called *webapps*, are networked applications that use technologies that emerged from the Web. They range from simple browser centric web pages to rich Internet applications such as Google Docs to browserless server-to-server SOAP-based web services. They make use of two characteristic mechanisms: *remote execution* by which a client can invoke computation on a remote server, and *mobile code* by which server code is sent to a client and executed there. Furthermore, communication happens over the HTTP protocol. Webapps are very popular for two main reasons. From the user's perspective, they are easy to deploy on clients: there is no need to install a third party program on the end-user's platform since everything happens through the web browser. From the developer's perspective, it is very easy to build a rich graphical user interface by using HTML, JavaScript and other web-based technologies. Moreover, developers can use external third-party web services as building blocks for their webapps (obtaining what is called a mashup). So, webapps are very easy to prototype, yet web programmers know that development gets much more complex as the application grows. Indeed, it is very hard to ensure correctness (and security) when developing and maintaining large scale webapps.

Two factors contribute to this complexity. First, web application developers are required to reason about distributed computation, which is intrinsically hard: they must ensure adequate interaction between the code executed on the client and the code executed on the server — and it gets more complicated when additional hosts are involved. Second, typical webapp development orchestrates a multitude of heterogeneous languages: the client side code is often written in HTML and JavaScript which are the standards implemented by all browsers, and the server side can be written in any language, common choices being PHP, Java, ASP/.NET and Python. This disjointness of technologies ensures that implementations are platform independent and increases interoperability. However, now the developer must make sure that code written in different languages will be correctly interfaced and work well together. In particular, data must be used consistently across language boundaries: typechecking becomes heterogeneous and possibly distributed when making calls to third-party web applications.

Until recently, web developers had few options besides extensive testing, an expensive proposition that does not scale. Approaches to provide static assurance are nowadays emerging. The now standard way to develop code that will interface with a remote application is to program it against an API that lists the provided functionalities and their type. This is natively supported in Java once the API has been copied locally (but APIs can change unexpectedly on the Web). Language extensions that verify the correctness of service orchestration have also been proposed, for example ServiceJ [6] which augments the type system of Java to take remote functions into account at compile time. Additionally, standards have been proposed to ensure adequate interactions between different services, for example

*Partially supported by the Qatar Foundation under grant number 930107. The authors can be reached at `tsans@qatar.cmu.edu` and `iliano@cmu.edu`.

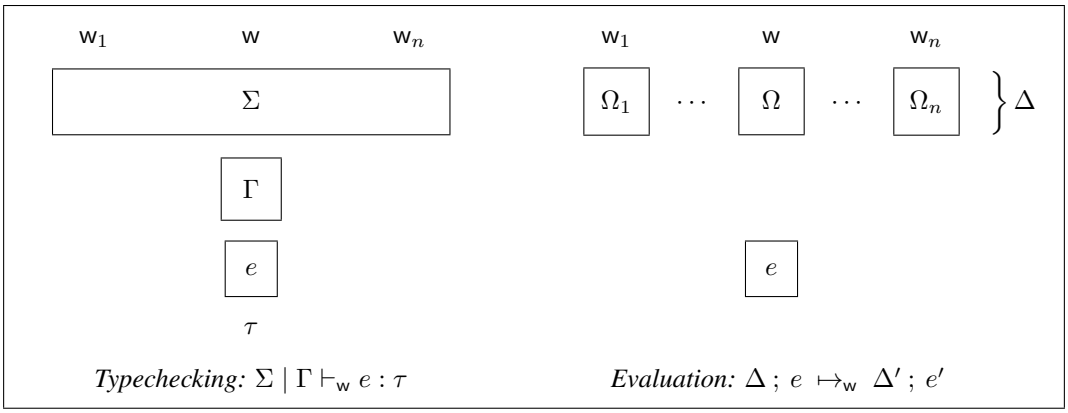


Figure 1: Views from World w

WSDL [19] declares the type interface of a service and BPEL [13] describes how services can be combined. However, these standards are not always implemented by web service platforms and web service programming frameworks. One common aspect of all these technologies is that they patch preexisting languages to permit web programming, which helps reasoning about a webapp as a distributed computation only up to a certain point. Instead, we propose to a language designed around the distributed nature of a web application. A complementary approach that specifically targets client-server applications is to develop them in a homogeneous language and then to compile them to the heterogeneous languages of the Web (in particular HTML and JavaScript on the client). In this way, type mismatches between client and server code are caught at compile time (rather than at execution time). One example is Google's Web Toolkit [8] where webapps are written entirely in Java. Another is Links [5]. One drawback with this approach is that the code is compiled statically into client and server roles, which makes it difficult to use to install services dynamically on a third-party host.

In this paper, we present an abstraction of web development that highlights its distinguishing features, remote code execution and mobile code. We realize this abstraction into a programming language skeleton designed with web applications in mind. This language, that we call QWeS^{2}T, supports remote execution through primitives that allow a server to publish a service and a client to call it as a remote procedure. It also embeds mobile code as a form of suspended computation that can be exchanged between nodes in the network. QWeS^{2}T is strongly typed and supports decentralized type-checking. We show that it is type-safe and we use it to implement, in a few lines, web interactions going from simple web page publishing to complex applications that create services dynamically and install them on third party servers. We have developed a prototype implementation of QWeS^{2}T.

The main contributions of this work are 1) the definition of a simple language that succinctly and naturally captures the principal constructs found in web programming; 2) the design of a type-safe language supporting a localized form of typechecking; and 3) an abstraction that allows specifying easily web interaction patterns that are difficult to achieve using current technologies. We do not see QWeS^{2}T as a replacement for existing languages for web programming, but as a simple experimental framework that facilitates exploring formally ideas about web programming. Indeed, we designed it as a stepping stone to study security mechanisms (specifically information flow) for web programming.

This paper is structured as follows: Section 2 lays out some operating assumptions for QWeS^{2}T, defines its syntax and semantics, and shows that it is type-safe. In Section 3, we use it to express some standard and some rather advanced web development efforts. Section 4 describes our prototype implementation while in Section 5 we review related work in the literature. We conclude in Section 6 with an outline of future developments.

2 A Language for Programming the Web

The goal of this paper is to propose a type-safe programming language for web development. This section describes this language, QWeS^{2}T, and establishes its properties. We begin by laying out the architecture of our intended system and the requirements that our design wants to address in Section 2.1. For the sake of clarity, we then introduce QWeS^{2}T

in stages: Section 2.2 illustrates the formal setup used in this paper on a handful of traditional constructs that we will use as our base language; Section 2.3 extends it with support for mobile code, which in turn Section 2.4 augments with constructs that enable remote code execution. We conclude in Section 2.5 with a metatheoretic investigation of the properties of QWeS^{2}T, which culminates in a distributed safety result.

2.1 Localized Computation

We consider a model of networked computation consisting of a fixed but arbitrary number of hosts, denoted w possibly subscripted (we also call them worlds or nodes). These hosts are capable of computation and are all equal, in the sense that we do not a priori classify them as clients or servers (instead, we use these terms based on their pattern of communication). They communicate exclusively through web services, which can be seen as a restricted form of message passing (each request is expected to result in a response). Since we are less interested in the communication resources (channels and messages) than on the computation performed at the various nodes, we do not need to rely on the machinery of traditional process algebras such as the π-calculus [16]. Our model is not concerned either with the topology of the network: indeed, just like we normally view the Web, we assume that every node can invoke services (including humble web pages) from every other node that publishes them. In this paper, we do not address any security or information flow issues (see Section 6).

The design of QWeS^{2}T espouses a node centric view of web programming, which to a large extent matches current development practices. Computation happens locally with occasional invocations of remote services. This intuition is depicted on the right-hand side of Figure 1: from node w's stance, it is executing a local program e and has access to a set of services Δ available on the Web, with each node w_i providing a subset Ω_i of these services. This will translate in an evaluation judgment $\Delta\,;\, e \mapsto_{\mathsf{w}} \Delta'\,;\, e'$ localized at node w, where each step of the computation of e with respect to Δ will yield a new expression and possibly extend Δ with a new service.

We want QWeS^{2}T to be globally type-safe and support localized type-checking. "Globally type-safe" means that if every service on the network is well-typed with respect to both its own declarations and any other web service it may invoke, then execution will never go wrong. "Local type-checking" implies that each node w will be able to statically verify any locally written program e by itself as long as it knows the correct types of the remote services it uses. This is illustrated on the left-hand side of Figure 1: the API of the services (of interest to w) on the Web is represented as Σ, while Γ and τ are the typing context and type of the expression e. In the following, this localized static form of typechecking will be captured by the typing judgment $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau$. The idea of localization, both for typing and evaluation, is inspired by *Lambda 5* [10, 11], a programming language for distributed computing.

2.2 Base Language

The interesting features of QWeS^{2}T are mobile and remote code, discussed in Sections 2.3 and 2.4, respectively. We will introduce them as extensions of a base language, which we call $\mathcal{L}$. The exact ingredients of this base language are unimportant for the overall discussion, as long as they interact nicely enough with mobility and remote code execution so that the overall language can be proved type safe. Therefore, for the sake of brevity, we choose $\mathcal{L}$ to be a very simple language featuring just functions, products and the unit type, with their usual constructors and destructors, as well as a fixed point operator. We do not foresee any difficulty in extending it with other common constructs. The syntax of $\mathcal{L}$ is summarized in the following grammar:

Types	τ	$::=$	$\tau \to \tau' \mid \tau \times \tau' \mid \mathsf{unit}$
Expressions	e	$::=$	$x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \mathsf{fix}\, x\!:\!\tau.e$
			$\mid \langle e_1, e_2\rangle \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e \mid ()$
Local typing context	Γ	$::=$	$\cdot \mid \Gamma, x : \tau$

Here, x ranges over variables. As usual, we identify terms that differ only in the name of their bound variables and write $[e/x]e'$ for the capture-avoiding substitution of e for x in the expression e'. Contexts (and similar collections that will be introduced shortly) are treated as multisets and we requires variables to be declared at most once in them — our rules will rely on implicit α-renaming to ensure this.

$$\frac{}{\Sigma \mid \Gamma, x : \tau \vdash_{\mathsf{w}} x : \tau}\ \text{of\_var}$$

$$\frac{\Sigma \mid \Gamma, x : \tau \vdash_{\mathsf{w}} e : \tau'}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \lambda x\!:\!\tau.\, e : \tau \to \tau'}\ \text{of\_lam} \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e_1 : \tau' \to \tau \quad \Sigma \mid \Gamma \vdash_{\mathsf{w}} e_2 : \tau'}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e_1\ e_2 : \tau}\ \text{of\_app}$$

$$\frac{\Sigma \mid \Gamma, x : \tau \vdash_{\mathsf{w}} e : \tau}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{fix}\, x\!:\!\tau.e : \tau}\ \text{of\_fix} \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e_1 : \tau \quad \Sigma \mid \Gamma \vdash_{\mathsf{w}} e_2 : \tau'}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \langle e_1, e_2 \rangle : \tau \times \tau'}\ \text{of\_pair}$$

$$\frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau \times \tau'}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{fst}\, e : \tau}\ \text{of\_fst} \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau \times \tau'}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{snd}\, e : \tau'}\ \text{of\_snd} \qquad \frac{}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} () : \mathsf{unit}}\ \text{of\_unit}$$

Figure 2: Typing Rules for $\mathcal{L}$

In preparation for our extension, we localize both the static and dynamic semantics of $\mathcal{L}$ at a world w, which represents the computing node where an expression will be typechecked or evaluated. In addition to a traditional context Γ for local variables, we equip the typing judgment with a *global service typing table*, written Σ, which we can safely assume to be empty for the time being. Therefore, the typing judgment assumes the following form:

$$\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau \qquad \textit{“}e \textit{ has type } \tau \textit{ in } \mathsf{w} \textit{ with respect to } \Sigma \textit{ and } \Gamma\textit{”}$$

The typing rules for this judgment are displayed in Figure 2. Notice in particular, that, apart from the presence of a world w and a service typing table Σ, these rules are completely standard. Notice also that neither w nor Σ changes in this figure — indeed they play no role in $\mathcal{L}$, besides setting the stage for the extensions.

For the same reason, we localize the dynamic semantics of $\mathcal{L}$ at a given world, w, and consider an evaluation configuration consisting of the expression e to be evaluated together with a global repository Δ that lists all the available services in the network. For the time being, we can safely assume that Δ too is empty. We describe evaluation in $\mathcal{L}$ using a standard small-step transition semantics with judgments

$$\begin{array}{ll} e\ \mathsf{val} & \textit{“}e \textit{ is a value”} \\ \Delta\ ;\ e\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e' & \textit{“}\Delta; e \textit{ transitions to } \Delta'; e' \textit{ in one step”} \end{array}$$

The rules for these judgments are displayed in the upper and lower parts of Figure 3, respectively. It is easy to see that, just as for the static semantics, the service repository Δ never changes. Aside from the repositories, these are textbook rules for the constructs in this language.

2.3 Mobile Code

Web developers use JavaScript code for two main reasons: 1) to call a remote service (through AJAX requests, for example) and process the result on the client side; 2) to interact with the user. This code is embedded in a web page provided by a web server, but it is executed by the web browser on the client's machine. This is what we intend by *mobile code*: a program that resides on a server, that will be transferred and executed on the client. There is a strict distinction between the code that provides a service on the server side (e.g., a database search) and the code that will call this service and process the result on the client side. From a developer's perspective, this distinction may take the form of having to switch between two programming languages (e.g., JavaScript for the client and PHP for the server) and insert this code appropriately in the page (for instance JavaScript code can be placed between HTML `<script>` tags and within specific HTML attributes). Dynamically generated web pages significantly complicate this picture as server code (e.g., PHP) and mobile code (JavaScript) as well various layout and styling markups (HTML and CCS) intended for the client are mixed in the same body of code.

As we mentioned in the introduction, ensuring correctness between client-side code and server-side code is hard. Some programming languages (such as Links [5] and Google Web Toolkit [8]) permit writing client-side code and

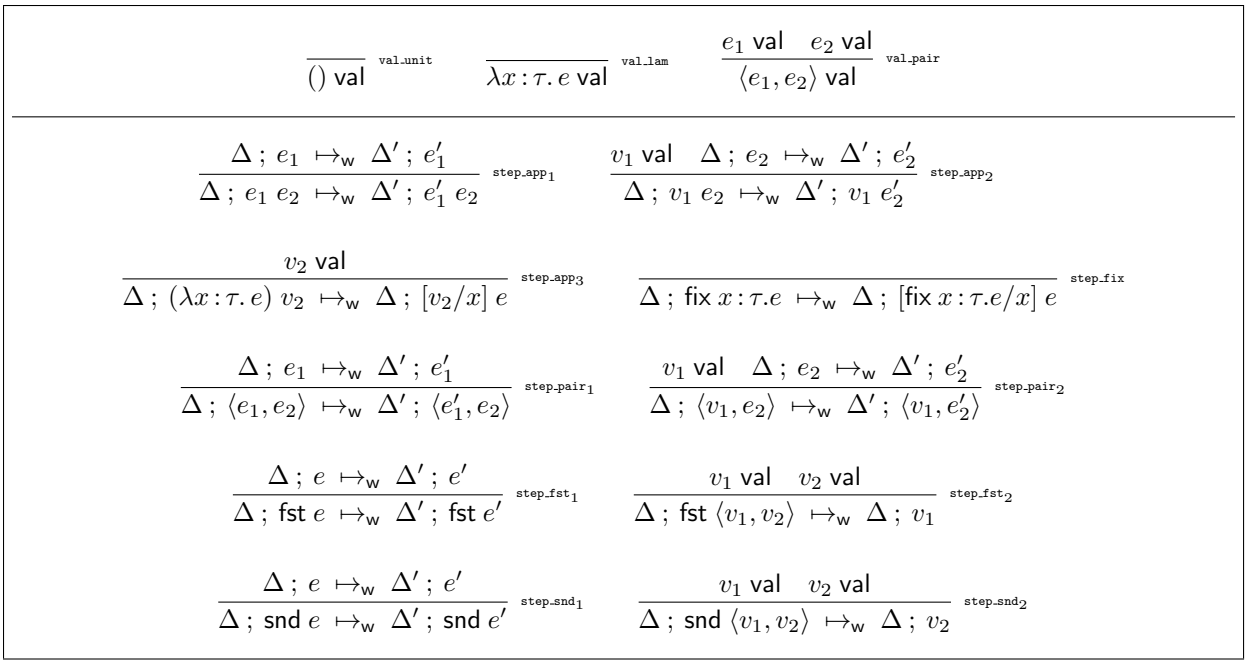


Figure 3: Evaluation Rules for $\mathcal{L}$

server-side code in the same formalism, allowing type mismatches to be caught at compile time. With Links for instance, the developers tag segments of code as "client" or "server" to specify where it is to be executed.

In this section, we extend our base language $\mathcal{L}$ to a language $\mathcal{L}^{\mathsf{m}}$ that is in a sense "mobile code ready". While remote code execution in Section 2.4 will provide the mechanism to actually move code from one world to another (among other things), $\mathcal{L}^{\mathsf{m}}$ embeds support for freezing expressions thereby preventing their evaluation, and for forcing execution in a controlled way. Once the transport mechanism is in place in Section 2.4, this will permit packaging mobile code on a server but evaluating it only when it reaches the client. For example, code that does machine-specific initialization on a web page (e.g., looking up the local time or a cookie) will reside in frozen form on the web server, and its execution will be triggered only when it reaches the client.

To realize this, we borrow from the concept of suspension in the theory of lazy programming languages. Specifically, we understand mobile code as expressions whose evaluation has been suspended and tag them with the type $\mathsf{susp}[\tau]$, where τ is the type of the original expression. The types τ and $\mathsf{susp}[\tau]$ are mediated by two basic constructs: $\mathsf{hold}\ e$ suspends the evaluation of e and $\mathsf{resume}\ e'$ resumes the computation of a previously suspended expression e'. Therefore, the language $\mathcal{L}^{\mathsf{m}}$ is obtained by extending the syntax of $\mathcal{L}$ as follows:

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \cdots \mid \mathsf{susp}[\tau] \\ \text{Expressions} & e & ::= & \cdots \mid \mathsf{hold}\ e \mid \mathsf{resume}\ e \end{array}$$

Although $\mathsf{hold}\ e$ suspends a computation, the standard definitions governing free and bound variables still apply. In particular, substitution traverses $\mathsf{hold}\ e$ just like any other expression.

The static and dynamic semantics of $\mathcal{L}^{\mathsf{m}}$ extend the corresponding rule sets for $\mathcal{L}$ as displayed in the top and bottom parts of Figure 4, respectively. We shall point out that, just as in $\mathcal{L}$, the world w and the service tables Σ, Δ and Δ' play no role in $\mathcal{L}^{\mathsf{m}}$. Indeed, were we to erase them, we would have a standard language with suspensions for single-node computing: everything is still happening locally.

2.4 Remote Code

The notion of *web service* is no more than a modern reincarnation of remote procedure calls (RPC) over the HTTP protocol. A web service can take several forms: it can be a simple web page (or AJAX) request that the client invokes

$$\cdots \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{hold}\ e : \mathsf{susp}[\tau]}\ {\scriptstyle \text{of\_hold}} \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \mathsf{susp}[\tau]}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{resume}\ e : \tau}\ {\scriptstyle \text{of\_resume}}$$

$$\cdots \qquad \frac{}{\mathsf{hold}\ e\ \mathsf{val}}\ {\scriptstyle \text{val\_hold}}$$

$$\cdots \qquad \frac{\Delta\ ;\ e\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e'}{\Delta\ ;\ \mathsf{resume}\ e\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{resume}\ e'}\ {\scriptstyle \text{step\_resume}_1} \qquad \frac{}{\Delta\ ;\ \mathsf{resume}\ (\mathsf{hold}\ e)\ \mapsto_{\mathsf{w}}\ \Delta\ ;\ e}\ {\scriptstyle \text{step\_resume}_2}$$

Figure 4: Additional Typing and Evaluation Rules for $\mathcal{L}^{\mathsf{m}}$

with POST/GET arguments, or it can be an RPC/SOAP request with a specific SOAP envelope format to pass the arguments. We abstract these various forms of service into what we call *remote code*, functions that resides on a remote server and that the client can invoke by sending a message carrying the arguments to be passed to this function. This function is executed on the server side and only the result is returned to the client.

We capture the notion of web service by extending $\mathcal{L}^{\mathsf{m}}$ into a language that we call QWeS^{2}T. To understand this extension, consider the constructs that must be available to a client and to a server. When a client interacts with a service, only two pieces of information are needed: 1) the *address* of this service (its *URL*), and 2) the type (or format) of the arguments that must be supplied together with the type of the result to be returned. Just as on the Web, we model a URL as a two-part locator consisting of the name of the server that provides it, say w', and of a unique identifier, u. We write it as $\mathsf{url}(\mathsf{w}', u)$. We introduce the type $\mathsf{srv}[\tau][\tau']$ to describe a remote service that expects arguments of type τ and will return a result of type τ'. A client w invokes a web service by calling the URL that identifies it with an argument of the appropriate type. This is achieved by means of the construct $\mathsf{call}\ e_1\ \mathsf{with}\ e_2$ which is akin to function application. It calls the URL e_1 by moving the value of the argument e_2 to where the remote code resides. There, the corresponding function is executed and the result is moved back to the client. Before a service can be called, a server w' must have created it. Our language models this by means of the operator $\mathsf{publish}\ x : \tau.e$ which publishes the function e that takes an argument x of type τ and returns a result of type τ'. The result is the symbolic URL $\mathsf{url}(\mathsf{w}', u)$ of type $\mathsf{srv}[\tau][\tau']$, for some new identifier u. These ingredients, which constitute the core of the extension of $\mathcal{L}^{\mathsf{m}}$ to QWeS^{2}T are collected in the following grammar:

$$\begin{array}{llllllllll} \text{Types} & \tau & ::= & \cdots & \mid & \mathsf{srv}[\tau][\tau'] & & & & \\ \text{Expressions} & e & ::= & \cdots & \mid & \mathsf{url}(\mathsf{w}, u) & \mid & \mathsf{publish}\ x : \tau.e & \mid & \mathsf{call}\ e_1\ \mathsf{with}\ e_2 \quad \mid \quad \mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w} \end{array}$$

The expression $\mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w}$ is used internally during evaluation and is not available to the programmer. It essentially models the client's awaiting for the result of a web service call. We will examine how it works in more detail below.

With the definition of QWeS^{2}T, our goal is to obtain a type-safe language for web programming. Just as in Links [5] and GWT [8], we want to be able to statically typecheck expressions to be evaluated and moreover we want each node on the network to be able to do so locally, without relying on other nodes. We achieve this by taking full advantage of our localized typing judgment, $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau$. To do so, we abstractly define the service typing table Σ as a multiset listing the type and location of all web services available on the network:

$$\text{Service typing table} \qquad \Sigma \quad ::= \quad \cdot \quad \mid \quad \Sigma, u : \mathsf{srv}[\tau][\tau]\ @\ \mathsf{w}$$

Of course no such thing exists in the reality of web development. We view it as an abstraction that can be safely approximated in a number of ways, including what is done in current practice. We will come back to this later.

Before we examine the added typing rules of QWeS^{2}T, we need to take some precautions against unintended code execution. As mentioned above, current practice relies on different languages executing on the client (e.g., JavaScript or Flash) and the server (PHP, Perl, or pretty much anything). So, while invoking a web service with integers or other traditional types of data is fine, invoking it with a functional argument raises eyebrows, and similarly for the

$$\frac{}{\mathsf{unit\ mobile}}\ \texttt{mobile\_unit} \qquad \frac{\tau\ \mathsf{mobile} \quad \tau'\ \mathsf{mobile}}{\tau \times \tau'\ \mathsf{mobile}}\ \texttt{mobile\_product}$$

$$\frac{}{\mathsf{susp}[\tau]\ \mathsf{mobile}}\ \texttt{mobile\_susp} \qquad \frac{\tau\ \mathsf{mobile} \quad \tau'\ \mathsf{mobile}}{\mathsf{srv}[\tau][\tau']\ \mathsf{mobile}}\ \texttt{mobile\_service}$$

Figure 5: Mobile Types

return value. As done in Lambda 5 [10, 11], we will disallow remote procedures whose argument or return type is functional. Said this, functions are swapped all the time from servers to clients (and more rarely vice versa) in the form of JavaScript code provided by the server and to be executed by the client. We account for this by means of the mobile computation mechanism introduced in Section 2.3: we allow suspended computations to be part of a web service exchange, that is we force a function of type $\tau \to \tau'$ to be packaged as a suspension (yielding the type $\mathsf{susp}[\tau \to \tau']$) before being shipped around. This mechanism prevents arbitrary code from being moved and installed without the consent of the client or server. Just as in Lambda 5 [10, 11], we define the following judgment,

$$\tau\ \mathsf{mobile} \qquad \textit{“}\tau \textit{ is a mobile type”}$$

to describe the types whose values can be transmitted between nodes. The rules implementing it are displayed in Figure 5. They hereditarily disqualify any function type unless protected by a suspension. Notice again that a suspended computation is mobile, which matches the way the Web works. When a web page contains a script tag with a source file, this file is transferred over HTTP in the same way that a web page is transferred. So, transferring a script works similarly to calling a service that will return the script. In the same way, the result of a service can be a URL itself. This is also consistent with the fact that calling a web page can return a page with a link to another page.

The typing rules for QWeS^{2}T are displayed in the upper part of Figure 6. The rules for publishing and calling a web service are unsurprising. The other two rules refer to a world w' different from where the expression is located. As we said, expect e from w is an artifact of our evaluation semantics, and therefore plays a role only in our metatheoretic analysis: it cannot appear in a legal program. Rule `of_url` deserves a longer discussion. It checks the type of a URL occurring in a program by looking it up in the global service typing table Σ, which by definition is not local. As mentioned earlier, Σ is an abstraction that is or could be realized in a variety of ways in practice:

- At development time, a programmer often simply downloads the API of a web service library, thereby locally caching the typing specifications of the services of interest.
- At compile time, the client system asks the remote server for the type of each service the local code uses. This is the approach implemented in our prototype (see Section 4). This mechanism is similar to typechecking a web service according to its WSDL file.
- A third-party web service server acts as a repository of all web services of interest in the network and answers typing inquiries similar to the way a DNS works.

Notice that in all three cases, the client must trust the typing information provided or returned by the web server or the repository. We do not address issues of trust in this paper.

In order to describe the dynamic semantics of QWeS^{2}T, we need first to populate the multiset Δ we assumed empty up to now. Each node w_i in the network has a local service repository $\{\Omega\}_{\mathsf{w}_i}$ which contains the identifier and the code of all web services published by w_i. Then, the global service repository Δ just denotes the collection of all such local service repositories. They are defined by the following grammar:

$$\begin{array}{lllll} \text{Local service repository} & \Omega & ::= & \cdot \mid \Omega, u \hookrightarrow x : \tau.e \\ \text{Global service repository} & \Delta & ::= & \{\Omega\}_{\mathsf{w}} \mid \Delta, \{\Omega\}_{\mathsf{w}} \end{array}$$

Here, both Ω and Δ are multisets.

$$\cdots \qquad \frac{\mathsf{srv}[\tau][\tau'] \text{ mobile}}{\Sigma, u : \mathsf{srv}[\tau][\tau'] \ @ \ \mathsf{w}' \mid \Gamma \vdash_{\mathsf{w}} \mathsf{url}(\mathsf{w}', u) : \mathsf{srv}[\tau][\tau']}\ {\scriptstyle \texttt{of\_url}}$$

$$\frac{\mathsf{srv}[\tau][\tau'] \text{ mobile} \quad \Sigma \mid \Gamma, x : \tau \vdash_{\mathsf{w}} e : \tau'}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{publish}\ \tau : x.e : \mathsf{srv}[\tau][\tau']}\ {\scriptstyle \texttt{of\_publish}} \qquad \frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e_1 : \mathsf{srv}[\tau][\tau'] \qquad \Sigma \mid \Gamma \vdash_{\mathsf{w}} e_2 : \tau}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{call}\ e_1\ \mathsf{with}\ e_2 : \tau'}\ {\scriptstyle \texttt{of\_call}}$$

$$\frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}'} e : \tau}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w}' : \tau}\ {\scriptstyle \texttt{of\_expect}}$$

$$\cdots \qquad \frac{}{\mathsf{url}(\mathsf{w}', u)\ \mathsf{val}}\ {\scriptstyle \texttt{val\_url}}$$

$$\cdots \qquad \frac{}{\Delta, \{\Omega\}_{\mathsf{w}}\ ;\ \mathsf{publish}\ x : \tau.e\ \mapsto_{\mathsf{w}}\ \Delta, \{\Omega, u \hookrightarrow x : \tau.e\}_{\mathsf{w}}\ ;\ \mathsf{url}(\mathsf{w}, u)}\ {\scriptstyle \texttt{step\_publish}}$$

$$\frac{\Delta\ ;\ e_1\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_1'}{\Delta\ ;\ \mathsf{call}\ e_1\ \mathsf{with}\ e_2\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{call}\ e_1'\ \mathsf{with}\ e_2}\ {\scriptstyle \texttt{step\_call}_1}$$

$$\frac{v_1\ \mathsf{val} \quad \Delta\ ;\ e_2\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_2'}{\Delta\ ;\ \mathsf{call}\ v_1\ \mathsf{with}\ e_2\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{call}\ v_1\ \mathsf{with}\ e_2'}\ {\scriptstyle \texttt{step\_call}_2}$$

$$\frac{v_2\ \mathsf{val}}{\underbrace{\Delta, \{\Omega, u \hookrightarrow x : \tau.e\}_{\mathsf{w}'}}_{\Delta'}\ ;\ \mathsf{call}\ \mathsf{url}(\mathsf{w}', u)\ \mathsf{with}\ v_2\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{expect}\ [v_2/x]\ e\ \mathsf{from}\ \mathsf{w}'}\ {\scriptstyle \texttt{step\_call}_3}$$

$$\frac{\Delta\ ;\ e\ \mapsto_{\mathsf{w}'}\ \Delta'\ ;\ e'}{\Delta\ ;\ \mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w}'\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{expect}\ e'\ \mathsf{from}\ \mathsf{w}'}\ {\scriptstyle \texttt{step\_expect}_1} \qquad \frac{v\ \mathsf{val}}{\Delta\ ;\ \mathsf{expect}\ v\ \mathsf{from}\ \mathsf{w}'\ \mapsto_{\mathsf{w}}\ \Delta\ ;\ v}\ {\scriptstyle \texttt{step\_expect}_2}$$

Figure 6: Additional Typing and Evaluation Rules for QWeS^{2}T

The dynamic semantics of QWeS^{2}T is given in the lower part of Figure 6 as an extension of the rules for $\mathcal{L}^{\mathsf{m}}$. The only new form of values are URL's. The evaluation of publish $x : \tau.e$ immediately publishes its argument as a web service in the local repository $\{\Omega\}_{\mathsf{w}}$, creating a new unique identifier for it and returning the corresponding URL. To call a web service, we first reduce its first argument to a URL, its second argument to a value, and then carry out the remote invocation which is modeled using the internal construct expect $[v_2/x]e$ from w'. This implements the client's inactivity while awaiting for the server w' to evaluate $[v_2/x]e$ to a value. This is done in rules `step_expect`$_1$ and `step_expect`$_2$: the former performs one step of computation on the server w' while the client w is essentially waiting. Once this expression has been fully evaluated, the latter rule kicks in and delivers the result to the client.

The dynamic semantic of QWeS^{2}T supports a view of computation on the Web that is centered around the local host: indeed the conclusion of each rule refers to the same node, w, and it is only when calling a remote service that the computation migrates to a different node (w' in the premise of rule `step_expect`$_1$), while w is waiting. Furthermore, our presentation gives w the illusion that at any time computation happens in exactly one node on the network, either locally or at the remote server that is servicing a call — this is the same illusion that we normally have while surfing the Web: Google does nothing else than waiting for my searches. The rules in Figure 6 allow for slightly more complex patterns of execution, since a web service can call another one that can call another one and so on, and any of these could reside on the host that made the original call. For how compelling this illusion may be, this is not the way

$$\frac{}{\cdot \vdash \cdot}\ \texttt{st\_\cdot} \qquad \frac{\Sigma \vdash \Delta}{\Sigma \vdash \Delta, \{\cdot\}_{\mathsf{w}}}\ \texttt{st\_empty} \qquad \frac{\Sigma \vdash \Delta, \{\Omega\}_{\mathsf{w}} \quad \Sigma \mid x : \tau \vdash_{\mathsf{w}} e : \tau'}{\Sigma, u : \mathsf{srv}[\tau][\tau'] \mathbin{@} \mathsf{w} \vdash \Delta, \{\Omega, u \hookrightarrow x : \tau.e\}_{\mathsf{w}}}\ \texttt{st\_nonempty}$$

Figure 7: Typing Rules for Service Repositories

the Web works: all kinds of computations are happening in parallel on every node. It is relatively easy to extend the presentation in Figure 6 to model this behavior, but this is beyond the scope of this paper. With a presentation that supports parallel execution, it is then possible with moderate effort to extend QWeS^{2}T with constructs that support another feature of web services: asynchrony (this is the first "A" in "AJAX"). Again, this is beyond the scope of this work.

When investigating the metatheory of QWeS^{2}T in Section 2.5, it will be useful to know that the types listed for the services in the table Σ do indeed correspond to the actual types of the services present in the distributed repository Δ. We express this requirement by means of the following judgment,

$$\Sigma \vdash \Delta \qquad \textit{"the services in } \Delta \textit{ are well-type with respect to } \Sigma\textit{"}$$

whose rules are shown in Figure 7. These rules go systematically through all the declarations in Σ, find the corresponding implementation in Δ, and check that it is well-typed with respect to the rest of Σ. A few observations about these rules are in order.

First, note that once a web service u has been typechecked in rule `st_nonempty` the remaining services are checked in a typing table that does not mention u any more. This immediately entails that QWeS^{2}T does not support recursive web services (at this stage). Furthermore, since it is perfectly legal in QWeS^{2}T for a web service u to call another service u' (which in turn could call other services), a derivation of $\Sigma \vdash \Delta$ needs to typecheck u before u', thereby following a total ordering of web services that respects these dependencies. Rather than explicitly searching for a workable ordering, we exploit our having abstractly defined Σ, Δ and the constituent Ω's to be multisets. If $\Sigma \vdash \Delta$ is derivable, then we know that there is an appropriate ordering of the services therein. Note that this ordering may hop back and forth among the various nodes. Observe also that, by and large, this corresponds to the way we use the web: we need to know about a URL before we can create a web page or service that uses it (we do not model the possibility of modifying an already published service in this work).

2.5 Metatheory

We conclude this section with a study of the metatheory of QWeS^{2}T, which will show that this language admits localized versions of type preservation and progress, thereby making it a type safe language. The techniques used to prove these results are fairly traditional. We used the Twelf proof assistant [1, 14] to encode each of our proofs and to verify their correctness. All these proofs can be found at [17]. All went through except for the proof of the Relocation Lemma, because we could not express the form of its statement in the meta-language of the proof-checker. Instead, we carried out a detailed proof by hand.

The analysis begins with a very standard substitution lemma. Note that URL's are never substituted in the semantics for QWeS^{2}T, and therefore do not require a separate substitution statement.

Lemma 1 (Substitution)**.** *If* $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau$ *and* $\Sigma \mid \Gamma, x : \tau \vdash_{\mathsf{w}} e' : \tau'$*, then* $\Sigma \mid \Gamma \vdash_{\mathsf{w}} [e/x]\, e' : \tau'$.

Proof. By induction on the derivation of the judgment $\Sigma \mid \Gamma, x : \tau \vdash_{\mathsf{w}} e' : \tau'$. It comes "for free" in Twelf. □

The following *relocation lemma* says that if an expression is typable at a world w, then it is also typable at any other world w' (note that the statement implicitly relocates its local assumptions Γ to w'—we will however use this lemma in the special case where Γ is empty). A similar, but slightly more complicated result is used in Lambda 5 [11, 10].

Lemma 2 (Relocation)**.** *If* $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau$*, then* $\Sigma \mid \Gamma \vdash_{\mathsf{w}'} e : \tau$ *for any world* w'.

Proof. By induction on the derivation of the judgment $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau$. A Twelf representation of this proof can be found in [17], but note that the meta-language of the proof checker did not support the quantification pattern in this proof. However, the Twelf representation does encode our manual proof. □

At this point, we are able to state the type preservation theorem, whose form is fairly traditional except maybe for the need to account for the valid typing of the web service repository before and after the evaluation step.

Theorem 3 (Type preservation)**.** *If* $\Delta \,;\, e \;\mapsto_{\mathsf{w}}\; \Delta' \,;\, e'$ *and* $\Sigma \mid \cdot \vdash_{\mathsf{w}} e : \tau$ *and* $\Sigma \vdash \Delta$*, then* $\Sigma' \mid \cdot \vdash_{\mathsf{w}} e' : \tau$ *and* $\Sigma' \vdash \Delta'$.

Proof. The proof proceed by induction on the derivation of $\Delta \,;\, e \;\mapsto_{\mathsf{w}}\; \Delta \,;\, e'$. It uses the Substitution Lemma in the cases of rules $\texttt{step\_app}_3$, $\texttt{step\_fix}$ and $\texttt{step\_call}_3$. It also uses the Relocation Lemma to handle rules $\texttt{step\_call}_3$ and $\texttt{step\_expect}_1$. □

The proof of progress relies on the following Canonical Form Lemma, whose statement and proof are standard.

Lemma 4 (Canonical Form)**.** *If* e val*, then*

- *if* $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \mathsf{unit}$, *then* $e = ()$;
- *if* $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau \to \tau'$, *then there exists* e' *such that* $e = \lambda\tau : x.\, e'$;
- *if* $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \tau \times \tau'$, *then there exist* e_1 *and* e_2 *such that* $e = \langle e_1, e_2 \rangle$, e_1 val *and* e_2 val;
- *if* $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \mathsf{susp}[\tau]$, *then there exists* e' *such that* $e = \mathsf{hold}\ e'$;
- *if* $\Sigma \mid \Gamma \vdash_{\mathsf{w}} e : \mathsf{srv}[\tau][\tau']$, *then there exist* w' *and* u *such that* $e = \mathsf{url}(\mathsf{w}', u)$.

Proof. By induction on the given derivation of e val and inversion on the appropriate typing rules. □

The progress theorem is again fairly standard, with a proviso for the web service repositories of QWeS^{2}T.

Theorem 5 (Progress)**.** *If* $\Sigma \mid \cdot \vdash_{\mathsf{w}} e : \tau$ *and* $\Sigma \vdash \Delta$*, then*

- *either* e val,
- *or there exist* e' *and* Δ' *such that* $\Delta \,;\, e \;\mapsto_{\mathsf{w}}\; \Delta' \,;\, e'$.

Proof. By induction on the given derivation of $\Sigma \mid \cdot \vdash_{\mathsf{w}} e : \tau$. □

3 Examples

We will now show how some common and not-so-common web development efforts can be modeled using QWeS^{2}T. We begin in Section 3.1 with expressing the creation and use of standard web pages, possibly containing JavaScript-style mobile code. In Section 3.2, we look at parameterized web pages and services where the server must process client input to dynamically produce the result. Up to that point, all examples will be pretty standard and can be implemented more or less straightforwardly with current tools. In Section 3.3, we show some more advanced web service interactions that are currently much harder to implement using existing technologies.

For readability, we extend QWeS^{2}T with an ML-like let construct, where "let $x = e_1$ in e_2 end" is understood as syntactic sugar for $(\lambda x : \tau.\, e_2)\ e_1$. Furthermore, to make the examples below more visually appealing, we will use the types html, info, query and result. To stay within the confines of QWeS^{2}T as described in Section 2, they can all be taken as synonyms for unit. Rather than denoting our hosts as w possibly subscripted, we assume we have a node www.server.com that will be playing a server role in our example, and a client called $client$. We will introduce more complex setups as needed. Finally, in commentary, we will often write a URL $\mathsf{url}(\mathsf{www.server.com}, u)$ as $\mathsf{www.server.com}/u$ for service identifier u.

3.1 Web Pages in QWeS^{2}T

When a browser requests a web page, it sends an HTTP request to the web server that provides it, which returns it as HTML code. Therefore, a web page can be understood as a web service that returns a value of type html. Since no interesting input needs to be processed, it is simply invoked with the unit element, (), of type unit. This is the typical way static web pages are retrieved.

3.1.1 Web Page without JavaScript Code

We start with a very simple example where server www.server.com publishes a web page with contents $someHTML$. The server's execution results in the creation of an externally visible identifier for it, which we call $webpage$. From there on, this page can be referenced by the URL www.server.com/$webpage$, which is a service of type srv[unit][html] located at www.server.com. The client will need to acquire this URL somehow (this is not modeled within the example). To retrieve the contents of the web page, the QWeS^{2}T client code simply call this URL with value $()$. The code for both the server and the client is given next. The curvy arrow between the two blocks of code indicates that the URL is communicated out of band.

```
                                                        www.server.com
let
  page = someHtml
in
  publish x : unit.page
end
```

www.server.com/$webpage$

```
                                                                client
call url(www.server.com, webpage) with ()
```

A simple variant of this code can model dynamically generated web pages that use server-side include (SSI), a technique by which the server puts together the page from HTML snippets held in various files. Say for example that our page is assembled from parts $header$, $body$ and $footer$. Then, we would define $page$ as $f\ header\ body\ footer$ for some concatenation function f.

3.1.2 Web Pages with Embedded JavaScript Code

A web page can embed JavaScript code that will be executed by the browser on the client side. Since JavaScript code will have an effect on the page rendered to the user, it can be seen as a function that takes an HTML document and returns an HTML document.

www.server.com/$webpage$

```
                                                        www.server.com
let
  script = hold λx : html. x
  page = ⟨script, someHtml⟩
in
  publish x : unit.page
end
```

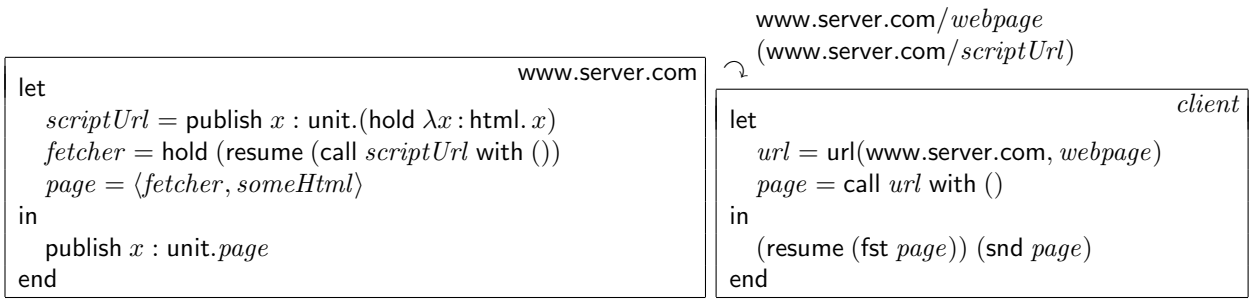
```
                                                                client
let
  url = url(www.server.com, webpage)
  page = call url with ()
in
  (resume (fst page)) (snd page)
end
```

Therefore, a web page is now a pair consisting of a (suspended) script of type susp[html → html] and a source HTML document of type html. Execution on the server will again result in a URL www.server.com/$webpage$, this time of type srv[unit][(susp[html → html]) × html]. Upon retrieving this page, the client's browser will extract the script and the source HTML markups, and apply the former to the latter, thereby rendering the processed page to the user. The client and server code above illustrates this idea using the identity function as the script.

3.1.3 Web Pages with External JavaScript Code

A web page can directly embed JavaScript code between `<script>` tags, or it can contain a URL to an external JavaScript file using the `src` attribute of this tag. In this case, the web browser first retrieves the page, then the JavaScript file, and finally applies this script to the contents of the web page. We model this mechanism in QWeS^{2}T as follows: the browser publishes the script file at some URL $scriptUrl$ (which is public, but that the client does not need to know), it installs an auxiliary function in the page, $fetcher$, whose job is to fetch $scriptUrl$ for the client, and finally it publishes a pair consisting of the fetcher and the source HTML document at URL www.server.com/$webpage$.

The client code does not change with respect to the previous example and is reproduced below only for clarity. This matches our everyday experience on the Web: how JavaScript is embedded in a web page is irrelevant to us. At execution time, however, the behavior on the client changes significantly: it retrieves the page just as before, it calls the code portion (now the fetcher) on the HTML data, but this times this has the effect of downloading the script file at $scriptUrl$ which is unpacked into the function that is applied to the HTML data.

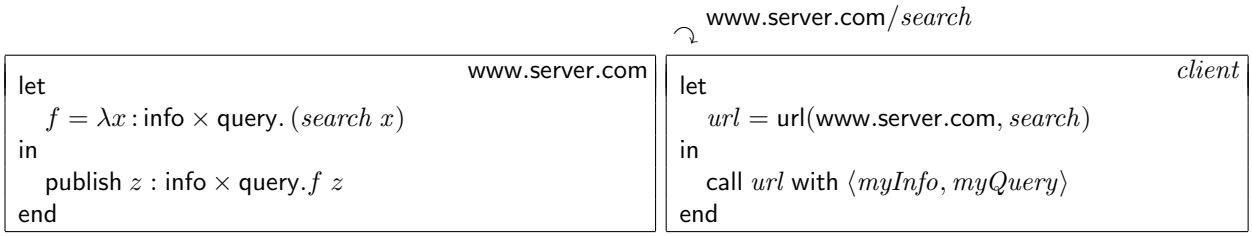


3.2 Web Services

In this section, we allow the client to pass parameters to the remote code on the server. In this way, we can use QWeS^{2}T to express a variety of common web-based interactions, in particular: dynamically generated web pages requested using HTTP requests with POST/GET arguments (e.g., when doing web searches), AJAX calls embedded within web pages (e.g., when panning outside the current view in Google Maps), and SOAP-based web services (often found in server-to-server exchanges).

3.2.1 Web Service Definition

In our next example, we want to write a web service that performs a search based on meta-information about who is doing the search (of type info) and on a search query (of type query) submitted by the client. We assume that this service returns a result of type result. The server simply publishes a service (at URL www.server.com/$search$) that takes these two arguments, calls an internal search function, and returns the result to the client. This service has type srv[info × query][result]. A client can then use this service by calling this URL on arguments of interest, modeled below as the pair $\langle myInfo, myQuery\rangle$.



3.2.2 Web Service API

To facilitate the usage of this service, we want to provide the client with an API that will perform the call. This API will

be published as a script that, once executed on the client side, will call the remote web service and returns the result. This script is published on the server at the URL www.server.com/api. The type of this URL is srv[unit][susp[query × info → result]]. The client can then download the script, install it and use the embedded function just as any local function. Note that the client does not need to be aware of the underlying search URL, www.server.com/$search$, even if it is public.

(www.server.com/$search$) ↷ www.server.com/api

```
                                                        www.server.com
let
   url = url(www.server.com, search)
   script = hold λz : query × info. (call url with z)
in
   publish x : unit.script
end
```

```
                                                                client
let
   url = url(www.server.com, api)
   f = resume (call api with ())
in
   f ⟨myInfo, myQuery⟩
end
```

3.3 Advanced Web Service Interactions

In this section, we show how QWeS^{2}T can easily express more complex forms of web interaction. Although useful, these types of services are uncommon on today's Web. We speculate that this may be due to the fact that expressing such complex interactions is difficult with current web technologies—remember that client and server code are typically written in different languages, with few recent exceptions. Yet, it takes just a couple of lines for QWeS^{2}T to express each of these interactions.

3.3.1 Customized API

If we assume that the client's information $myInfo$ does not change from one query to another in the last example, then the server could customize the script for each client based on its information. In the code below, the server publishes a script www.server.com/api that is expected to be called with the client's information. The server then returns a specialized version of this script that the client can repeatedly call by just providing queries. This time, the published script has type srv[info][susp[query → result]], which can be viewed as a curried version of what we had in Section 3.2.2 (modulo the intervening suspension).

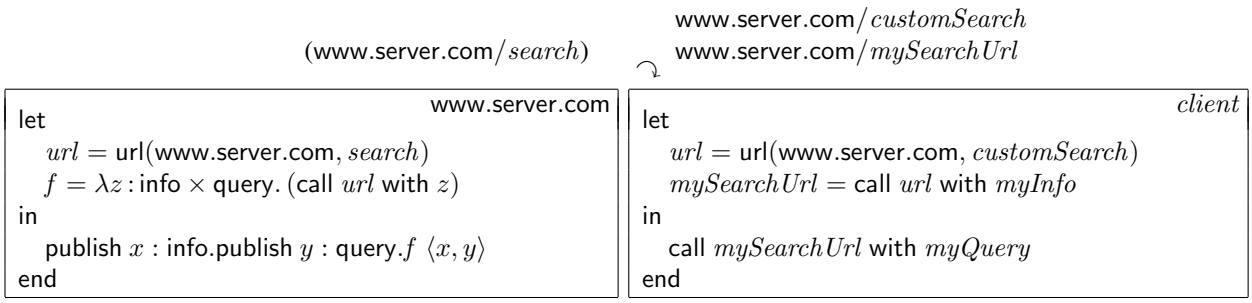
(www.server.com/$search$) ↷ www.server.com/api

```
                                                        www.server.com
let
   url = url(www.server.com, search)
   script = λx : info. hold λy : query. (call url with ⟨x, y⟩)
in
   publish x : info.script x
end
```

```
                                                                client
let
   url = url(www.server.com, api)
   f = resume (call url with myInfo)
in
   f myQuery
end
```

3.3.2 Customized Web Service

Going one step further, rather than returning a customized API, the server may want to provide a specialized service for each client. This client-specific service will be published automatically on the server side when the client needs it for the first time — Google Sites allows something like this. The server will return the URL of this personalized service to the client. The code below illustrates this idea with a custom search functionality.

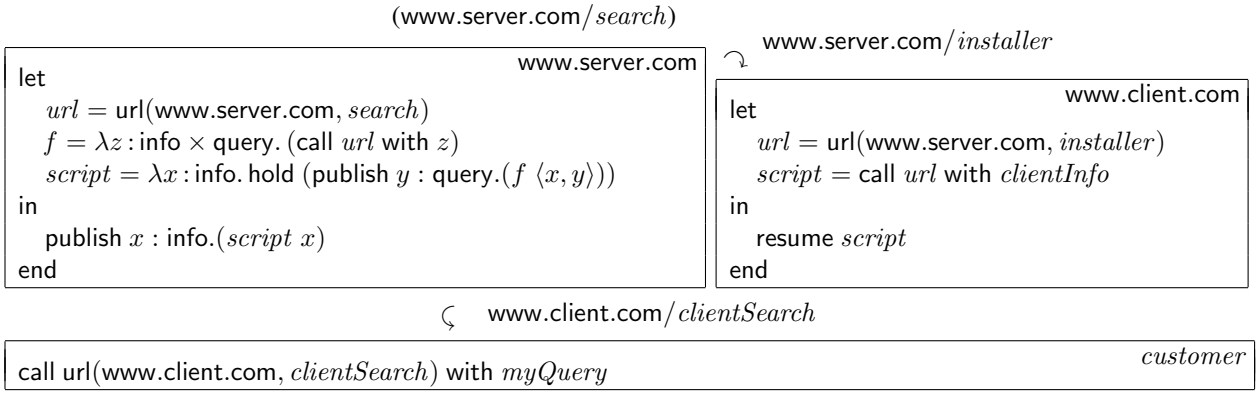


Here, the server initially publishes a URL www.server.com/$customSearch$ available to any host. When a specific client invokes it with its own information, $myInfo$, the server publishes a search service customized to this client and makes it available on the spot as the URL www.server.com/$mySearchUrl$. The client can then use this personalized URL directly to carry out its queries. The type of www.server.com/$customSearch$ is srv[info][srv[query][result]] and the type of www.server.com/$mySearchUrl$ is srv[query][result].

3.3.3 Web Service Auto-Installer

For our last example, assume the client has a web server of its own, www.client.com, and wants to provide its customers with a web service on www.client.com that makes use of functionalities supplied by www.server.com. When the client's customer, call it $customer$, needs the service, it can contact the client directly without any need to know that the bulk of the work is done by the server. The standard way to do all this is for the client to manually create a service on www.client.com and configure it to call the server. This requires more sophistication of the client than in all the previous examples, which involved doing no more than clicking on a link.

In QWeS^{2}T, we can do much better. In the code snippet below (which builds on our search example), the server provides a script, www.server.com/$installer$ (of type srv[info][susp[srv[query][result]]]), that, when invoked by the client with input $clientInfo$, will automatically create and publish the customized web service www.client.com/$clientSearch$ (of type srv[query][result]) at www.client.com. It is this URL that $customer$ will use for its searches. Then we are back to the situation where the client needs to do no more than clicking on a link.



4 Prototype Implementation

As a proof of concept, we have developed a fully-functional prototype implementation of QWeS^{2}T. It uses the HTTP protocol to enable any host on the Internet on which our prototype has been installed to interact with any similarly equipped node. We have installed it on various machines in our lab and used this setup to run all the examples discussed in Section 3, and a few more. Within each node, our prototype consists of three components:

- **The web service repository** holds the services published by the node as well as information about their type. In this way, it implements both the local service repository that we called Ω in Section 2 and the portion of the

service typing table Σ that pertains to this host (we chose to implement Σ in a distributed fashion). The former is used at run-time, the latter during typechecking. The web service repository is implemented as an SQLite database.

- **The interpreter**, the core of our prototype, is a relatively faithful implementation of the rules in Section 2. It performs two main duties:

  - It typechecks all the code that originates at the local node and answers any typing requests from other nodes. It typechecks URL's found in local code by asking the host where the corresponding service resides for the correct type. Therefore, our prototype implements rule `of_url` in Section 2 in a distributed fashion.
  - It executes both local code and any mobile code that is received while interacting with other hosts. It publishes local services by inserting them in the local web service repository and fetches them from there when receiving an execution request from a remote host. It also calls services elsewhere through the web interface (see next). In our prototype, remote code is executed remotely: the internal expression expect e from w is an artifact of the semantics and does not appear in our implementation—this is the only departure from the rules in Section 2.

  The interpreter is written in Python and interfaces with the other two components of the local copy of the system. A new instance of the interpreter is started every time a service on the local node is invoked.

- **The web interface** is the gateway to all remote hosts. Half of its job is to receive remote requests, extract the service name and arguments, spawn a new interpreter and send the result back. The other half is to package remote service invocations, send them on the network, and deliver the result to the local interpreter. It is implemented as a PHP script running on the local web server (Apache in our setup). It translates service requests to/from the interpreter into HTTP messages. It uses POST/GET arguments to transmit the service identifier and the web service parameters ("e_2" in call e_1 with e_2) using the JSON library to encode the latter into ASCII.

5 Related Work

The popularity of web applications has fueled a market for development environments and programming support, mainly from industry. The closest to our proposal is Google's Web Toolkit (GWT) [8], which allows writing webapps entirely in Java. Because Java is strongly typed, typing mismatches between client code and server code are caught at compile time. Client-side code is written as an extension of the RemoteService class and is compiled to JavaScript. Server side code extends the RemoteServiceServlet class and is compiled to Java bytecode. While GWT is well suited for traditional client-server applications, it was not designed for dynamic services such as our web service auto-installer example. Other software companies have development tools for proprietary web applications solution, e.g., Adobe's Flash and Microsoft's Silverlight.

Academic research has been following industry. Links [5] is a web oriented programming language that also compiles client-side code into JavaScript. With Links, functions are tagged as client or server to indicate where they shall be executed. Server code is again static and once code has been generated, it cannot be customized and moved around as QWeS^{2}T could do in Section 3.3.

The design of QWeS^{2}T is heavily inspired to Lambda 5 [10, 11]. Lambda 5 is an abstract programming language for distributed computing that uses a modal type system to capture the notion of localized computation. QWeS^{2}T can be seen as a simplification of Lambda 5 specialized to web programming. As such, it streamlines this languages and provides constructs that abstract the way we use the web directly. Lambda 5's prototype, ML5 [11], although targeting web programming, is limited to basic browser-server interactions and cannot express all examples in Section 3.3.

A number of authors have proposed abstract frameworks to study web programming and related concepts. By expressing web services in a process algebra, Ferrara [7] is able to use verification techniques based on temporal logic and process equivalence to gain correctness assurances, without loosing the ability to compile them to executable code. By contrast, Jia and Walker [9] find a foundation of distributed computing in an intuitionist modal logic and distill a programming language from it using the Curry-Howard isomorphism.

6 Conclusions and Future Work

In this paper, we proposed a type safe language, QWeS^{2}T, that provides an abstraction for two characteristic forms of distributed computing found in web programming: mobile code and remote procedure calls. By specifying client-server and server-server interactions in a single formalism, this model makes it easy to dynamically generate and disseminate scripts and services. We proved that QWeS^{2}T is type safe and we implemented a prototype.

In future work, we intend to develop QWeS^{2}T in two directions. One is an extension of this language with additional constructs, in particular support for parallel execution and asynchronous communication. We also plan to extend our prototype with richer types, and eventually with a Document Object Model (DOM) Library to cope with web standards, thereby allowing us to run more realistic web programming experiments with QWeS^{2}T.

The other direction involves using QWeS^{2}T as a basis for studying language-level mechanisms for secure web programming. Specifically, we are interested in approaches to control access to services (e.g., a service provider will want to restrict the use of a service to certain nodes) and to control the dissemination of data supplied to those services (a client may refuse to supply sensitive data to a service if it were to forward this data to an untrusted node). Preliminary work indicates that it is possible to extend the static semantics of QWeS^{2}T to identify the nodes that will be involved in the computation of a given expression and how their different services will be combined. This is similar to the history-based security models proposed in [2, 3]. Therefore, we believe that we can braid a policy language for static information flow security [12, 15, 4] into QWeS^{2}T. Specifically, we anticipate attaching a security policy to published services and the data they are invoked with. A similar idea was explored in [18, 20].

Acknowledgments

We are grateful to Bob Harper, Rob Simmons and Dan Licata for the fruitful discussions during the design of QWeS^{2}T and for their help using Twelf.

References

[1] The Twelf project wiki. Available at `http://twelf.plparty.org/wiki`.

[2] Martín Abadi and Cédric Fournet. Access control based on execution history. In The Internet Society, editor, *Network and Distributed System Security Symposium, NDSS*, San Diego, CA, 2003.

[3] Massimo Bartolett, Pierpaolo Degano, and Gian Luigi Ferrari. History-based access control with local policies. *Foundations of Software Science and Computational Structures*, pages 316–332, 2005.

[4] Massimo Bartoletti, Pierpaolo Degano, Gian Ferrari, and Roberto Zunino. Model checking usage policies. *Trustworthy Global Computing*, pages 19–35, 2009.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. *Formal Methods for Components and Objects*, pages 266–296, 2007.

[6] S. De Labey, M. van Dooren, and E. Steegmans. ServiceJ A Java Extension for Programming Web Services Interactions. In *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pages 505 –512, july 2007.

[7] Andrea Ferrara. Web services: a process algebra approach. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 242–251, New York, NY, USA, 2004. ACM.

[8] Google Inc. Google Web Toolkit. Available at `http://code.google.com/webtoolkit/`.

[9] Limin Jia and David Walker. Modal proofs as distributed programs. *Programming Languages and Systems*, pages 219–233, 2004.

[10] T. Murphy VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, January 2008. Available as technical report CMU-CS-08-126.

[11] Tom Murphy VII, Karl Crary, and Robert Harper. Type-Safe Distributed Programming with ML5. *Trustworthy Global Computing*, pages 108–123, 2008.

[12] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM.

[13] OASIS. *Business Process Execution Language (WS-BPEL)*. Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[14] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[15] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan 2003.

[16] Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[17] Thierry Sans and Iliano Cervesato. The QWeS^{2}T Project. Available at `http://www.qatar.cmu.edu/~tsans/qwest/`.

[18] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. *Security and Privacy, IEEE Symposium on*, 0:369–383, 2008.

[19] W3C. *Web Services Description Language (WSDL)*. World Wide Web Consortium (W3C), 2007.

[20] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2):67–84, 2007.