Type-Safe Web Programming in QWeS^{2}T

Thierry Sans* **Iliano Cervesato***

June 2010
CMU-CS-10-125
CMU-CS-QTR-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Carnegie Mellon University, Qatar campus.
The authors can be reached at tsans@qatar.cmu.edu and iliano@cmu.edu.

Partially supported by the Qatar Foundation under grant number 930107.

Keywords: Web programming, Mobile Code, Remote code, Type safety.

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.

Contents

List of Figures

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 [19] 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 WSDL [55] declares the type interface of a service and BPEL [36] 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 [24] where webapps are written entirely in Java. Another is Links [18]. 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. Section 5 discusses introducing parallelism within QWeS^{2}T. We review related work from the literature in Section 6. We conclude in Section 7 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.

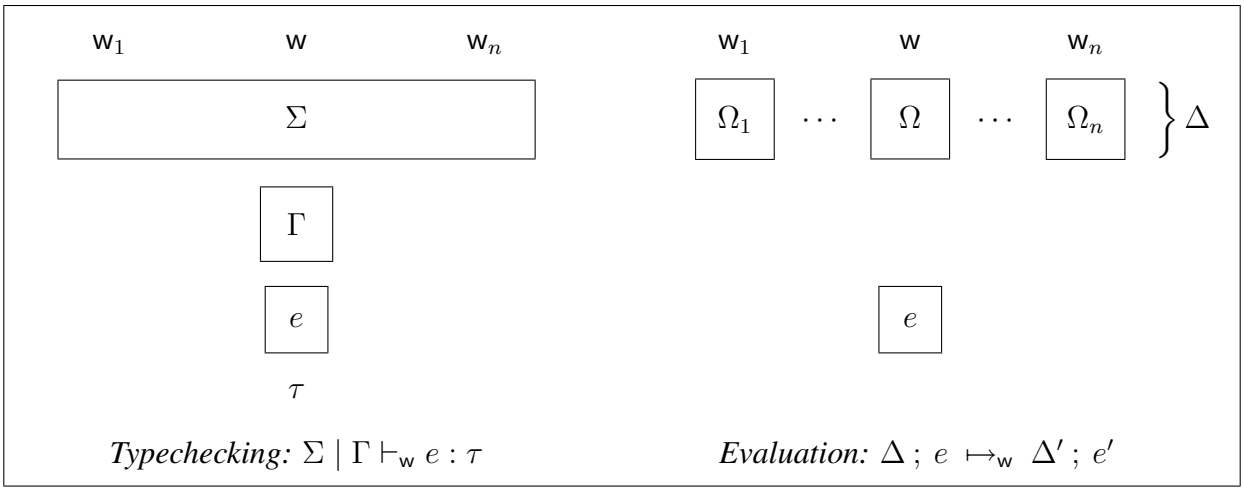


Figure 1: Views from World w

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 [30, 45]. 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 7).

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 — this is a special case of conformance testing in web services [3, 4, 11, 14]. "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

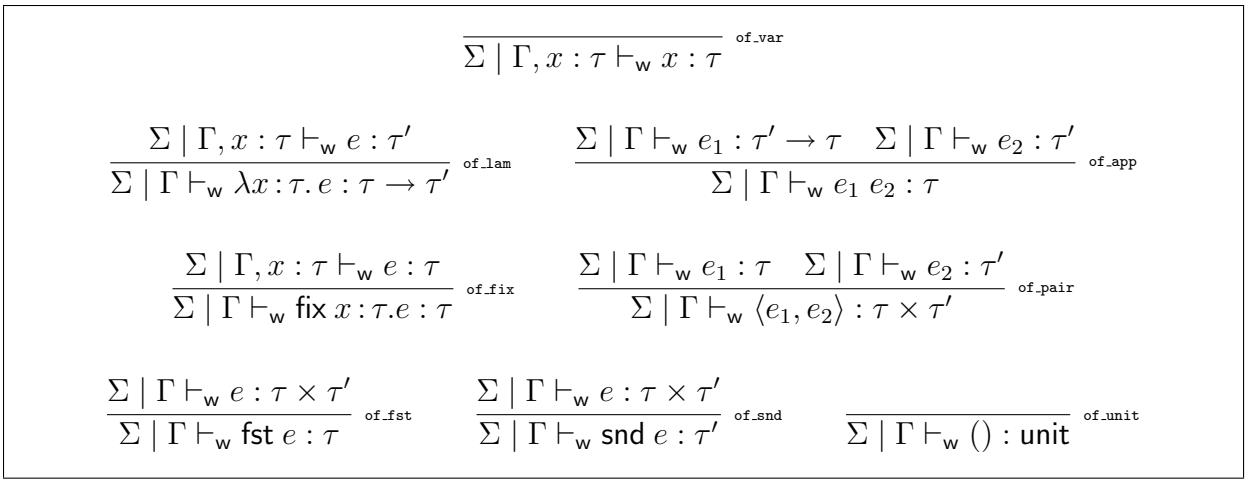


Figure 2: Typing Rules for $\mathcal{L}$

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* [34, 33], 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	$::= \quad \tau \to \tau' \mid \tau \times \tau' \mid \mathsf{unit}$
Expressions	e	$::= \quad x \mid \lambda x : \tau.\, e \mid e_1\ e_2 \mid \mathsf{fix}\, x : \tau.e$ $\mid \langle e_1, e_2 \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid ()$
Local typing context	Γ	$::= \quad \cdot \mid \Gamma, x : \tau$

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.

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

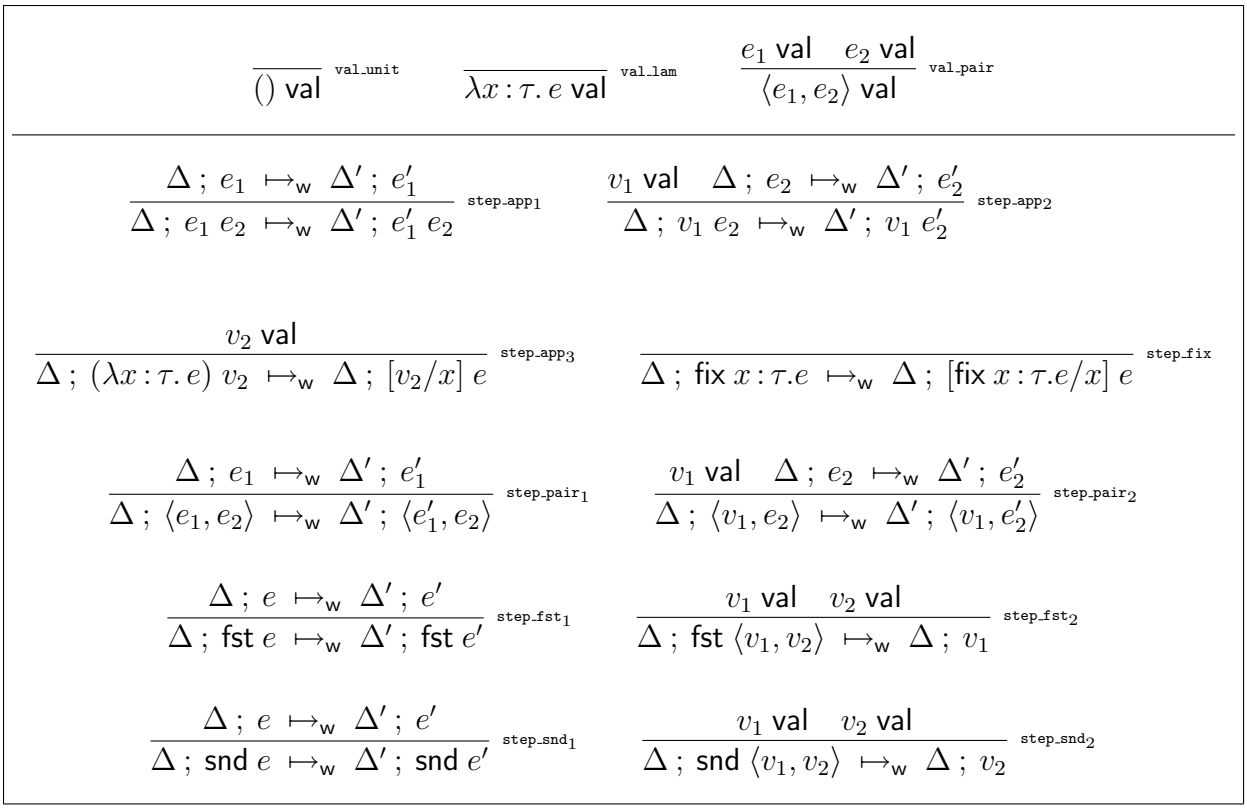


Figure 3: Evaluation Rules for $\mathcal{L}$

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

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

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 [18] and Google Web Toolkit [24]) permit writing client-side code and 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

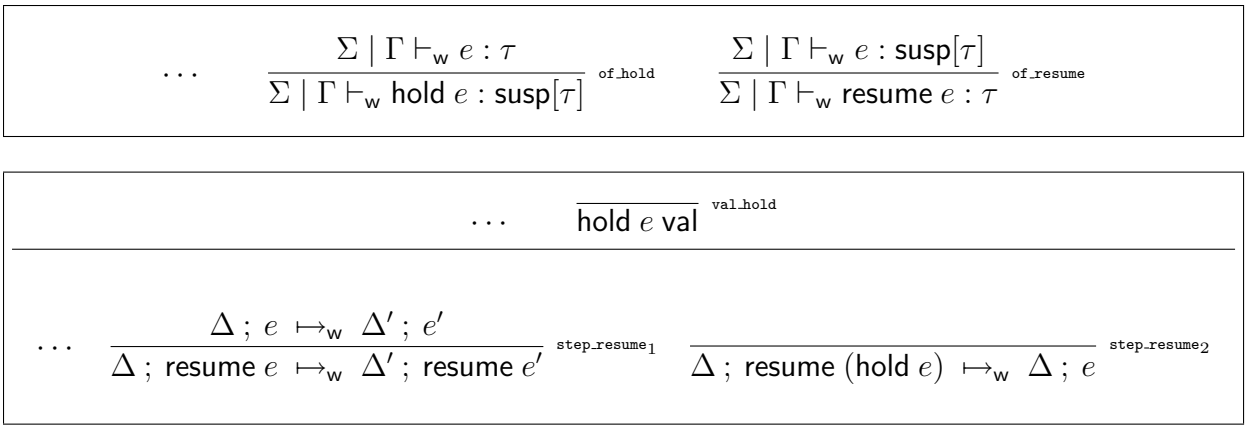


Figure 4: Additional Typing and Evaluation Rules for $\mathcal{L}^{\mathsf{m}}$

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 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

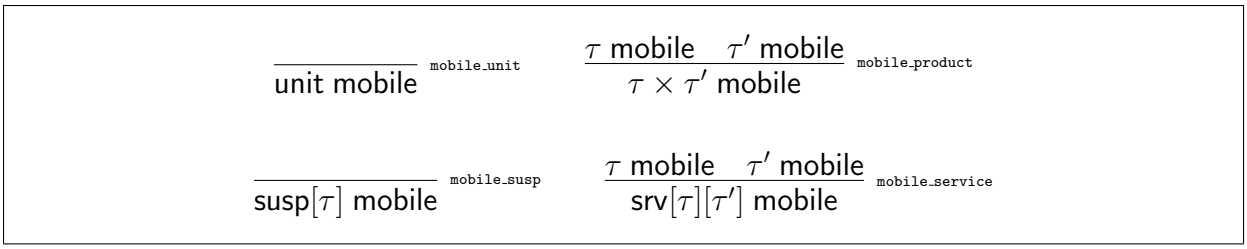


Figure 5: Mobile Types

constitute the core of the extension of $\mathcal{L}^{\mathsf{m}}$ to QWeS^{2}T are collected in the following grammar:

Types $\quad \tau \ ::= \ \cdots \ | \ \mathsf{srv}[\tau][\tau']$

Expressions $\quad e \ ::= \ \cdots \ | \ \mathsf{url}(\mathsf{w}, u) \ | \ \mathsf{publish}\ x : \tau.e \ | \ \mathsf{call}\ e_1\ \mathsf{with}\ e_2 \ | \ \mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w}$

The expression expect e from 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 [18] and GWT [24], 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:

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

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 point 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 return value. As done in Lambda 5 [34, 33], 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 [34, 33], 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:

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

Here, both Ω and Δ are multisets.

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

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

$$\frac{\mathsf{srv}[\tau][\tau'] \text{ mobile} \quad \Sigma \mid \Gamma, x : \tau \vdash_{\mathsf{w}} e : \tau'}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{publish}\ \tau : x.e : \mathsf{srv}[\tau][\tau']} \ \text{of\_publish}$$

$$\frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e_1 : \mathsf{srv}[\tau][\tau'] \qquad \Sigma \mid \Gamma \vdash_{\mathsf{w}} e_2 : \tau}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{call}\ e_1\ \mathsf{with}\ e_2 : \tau'} \ \text{of\_call}$$

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

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

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

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

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

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

$$\frac{\Delta\ ;\ e \ \mapsto_{\mathsf{w}'} \ \Delta'\ ;\ e'}{\Delta\ ;\ \mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w}' \ \mapsto_{\mathsf{w}} \ \Delta'\ ;\ \mathsf{expect}\ e'\ \mathsf{from}\ \mathsf{w}'} \ \text{step\_expect}_1$$

$$\frac{v\ \mathsf{val}}{\Delta\ ;\ \mathsf{expect}\ v\ \mathsf{from}\ \mathsf{w}' \ \mapsto_{\mathsf{w}} \ \Delta\ ;\ v} \ \text{step\_expect}_2$$

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

$\texttt{step\_expect}_1$ and $\texttt{step\_expect}_2$: 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 $\texttt{step\_expect}_1$), 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 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 $\texttt{st\_nonempty}$ 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).

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

Figure 7: Typing Rules for Service Repositories

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, 39] to encode each of our proofs and to verify their correctness. All these proofs can be found in Appendix A. 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 [33, 34], another type-safe formalism for networked computation.

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 Appendix A as the type family `reloc`, but note that the meta-language of the proof checker did not support the quantification pattern in this proof. However, the Twelf specification of `reloc` 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 $\texttt{step\_app}_3$, `step_fix` and $\texttt{step\_call}_3$. It also uses the Relocation Lemma to handle rules $\texttt{step\_call}_3$ and $\texttt{step\_expect}_1$. It appears in Appendix A as the Twelf type family `tpres`. □

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. This lemma too comes for free in Twelf. □

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 $\Sigma \mid \cdot \vdash_{\mathsf{w}} e : \tau$. This proof is given by the Twelf code defining the type family `progress` and auxiliary definitions. □

By satisfying type preservation and progress, the previous results show that QWeS^{2}T is type-safe.

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. Finally, Section 3.4 explores applications of QWeS^{2}T beyond traditional web programming.

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.

www.server.com/$webpage$
(www.server.com/$scriptUrl$)

www.server.com

```
let
  scriptUrl = publish x : unit.(hold λx : html. x)
  fetcher = hold (resume (call scriptUrl with ()))
  page = ⟨fetcher, someHtml⟩
in
  publish x : unit.page
end
```

client

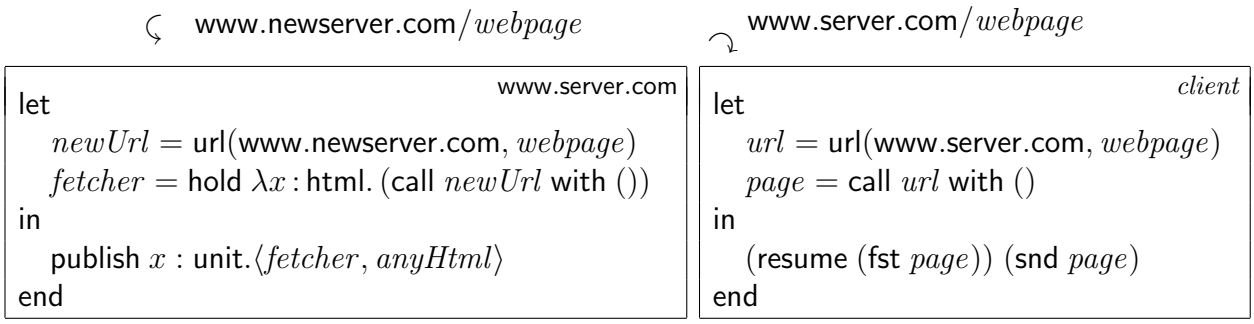
```
let
  url = url(www.server.com, webpage)
  page = call url with ()
in
  (resume (fst page)) (snd page)
end
```

3.1.4 Web Page Redirection

A variant of this mechanism forms also the basis for implementing web page redirection. Say that we moved a web page www.server.com/$webpage$ to www.newserver.com/$webpage$. Setting up redirection on www.server.com will allow clients to still be able to access this page using its former URL, www.server.com/$webpage$. We implement redirection in QWeS^{2}T by publishing a script on www.server.com that retrieves the new page (for simplicity, we assume it contains plain HTML as in Section 3.1.1, otherwise the call is adapted as in the retrieval in Section 3.1.2), as described below. The client accesses it like any other script (and indeed the client code has not

change). The server code is puts together a page containing a script, $fetcher$, but notice that the HTML part, rendered as $anyHTML$, is ignored: $fetcher$ does not use it.

Observe also that, had we not turned the call to www.newserver.com/$webpage$ into a script and suspended it, the code would simply cache www.newserver.com/$webpage$ on www.server.com as www.server.com/$webpage$.

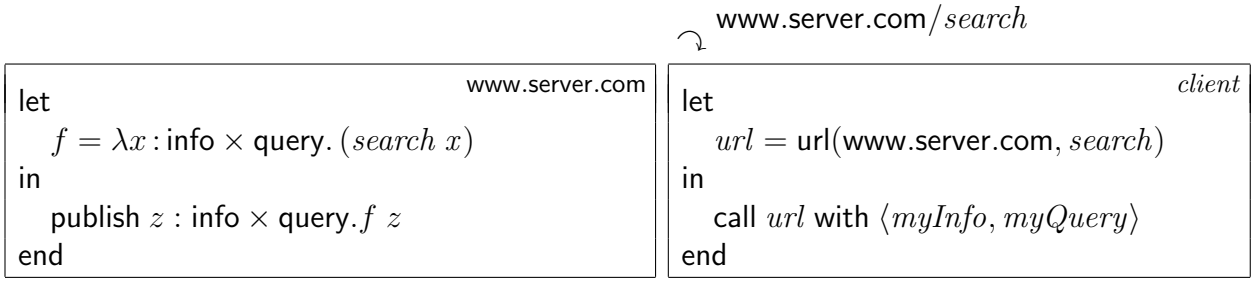


3.2 Web Services

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

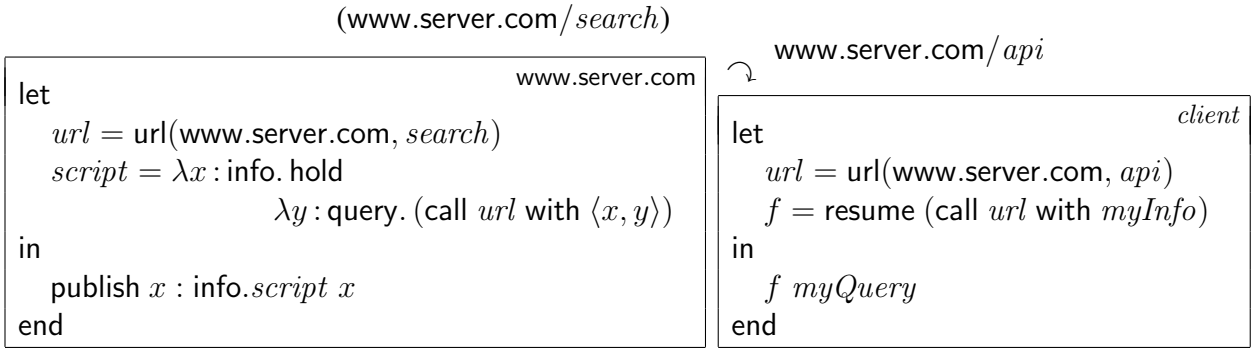
```
let
   url = url(www.server.com, api)
   f = resume (call api with ())
in
   f ⟨myInfo, myQuery⟩
end
```

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.

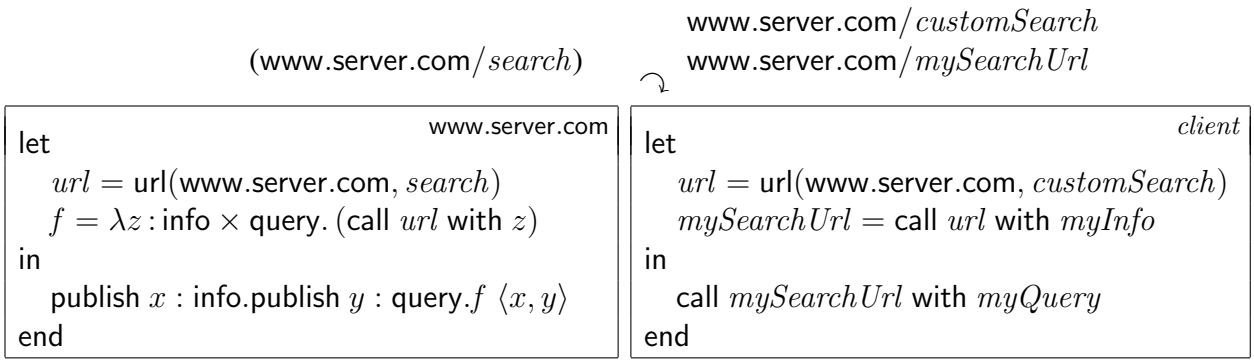
3.3.1 Customized API

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).



3.3.2 Customized Web Service

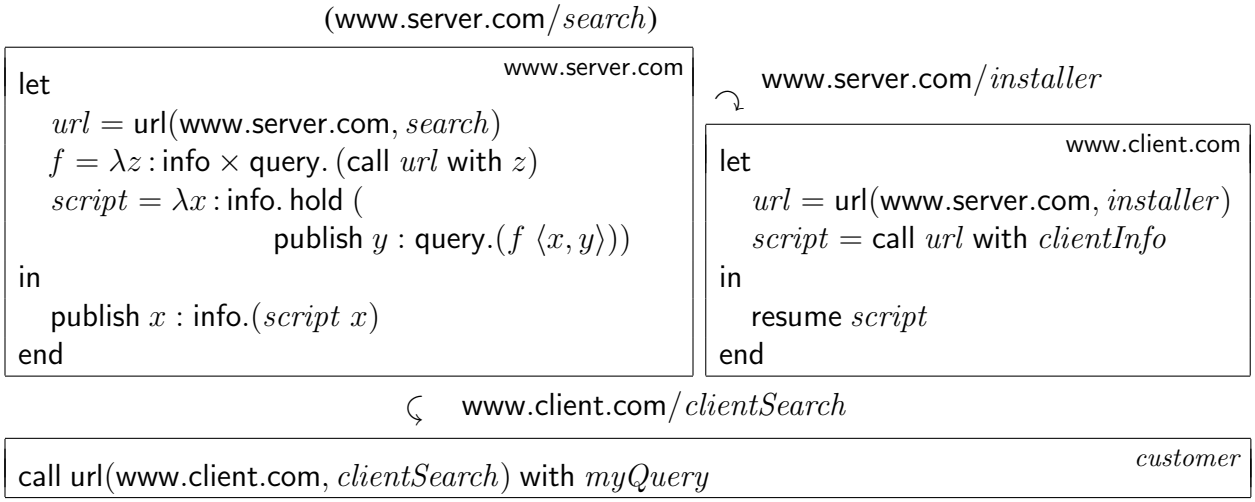
Going one step further, rather than returning a customized API, the server may want to provide a specialized service for each client. This client-specific service will be published automatically on the server side when the client needs it for the first time — Google Sites allows something like this. 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 a new URL that we call 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

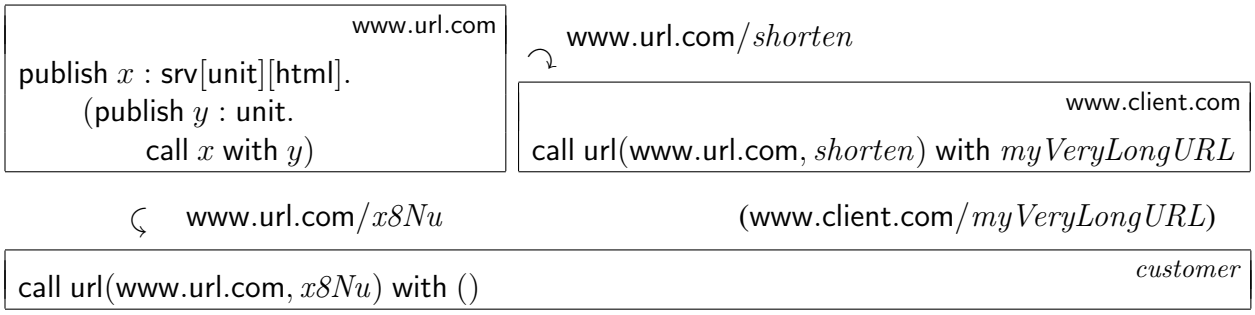
For our next example, assume the client has a web server of its own, www.client.com, and wants to provide its customers with a web service on www.client.com that makes use of functionalities supplied by www.server.com. When the client's customer, call it $customer$, needs the service, it can contact 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 given code snippet (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.

3.3.4 URL Transcriber

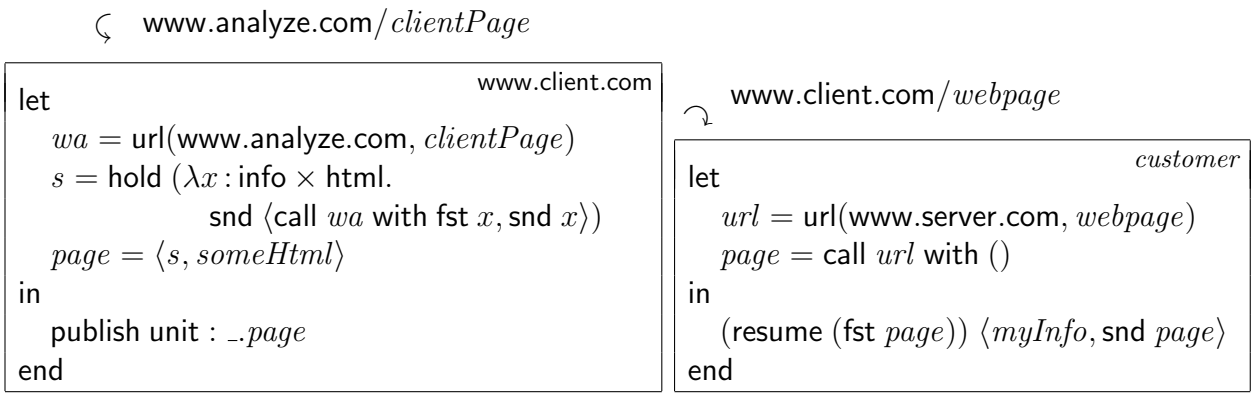
Consider next a service that rewrites a URL into a different-looking URL so that visiting the latter yields the contents of the former. A popular instance is www.tinyurl.com, which turns long unwieldy URLs into very short ones, which can be easier to communicate. The code fragment below is the skeleton of such a service. The site www.url.com provides the service *shorten* which expects an argument x of type srv[unit][html], that is a URL. It then publishes a new service that visits x when called. If the underlying implementation arranges for the services created in this way to have very short names, www.url.com/*shorten* effectively provides a functionality akin to www.tinyurl.com. The type of www.url.com/*shorten* is srv[srv[unit][html]][srv[unit][html]], i.e., it is a service that transforms URLs into URLs. The code also shows how a second site, www.client.com, can use this service to shorten some long URL for the benefit of a third site, called here *customer*.



This code snippet is the general blueprint of a whole family of services intended to transcribe URLs. Examples include web proxies and web anonymizers.

3.3.5 Web Analytics

Our last example considers a webpage augmented with tracking capabilities, as enabled for example by services like Google Analytics. The page in question will be www.client.com/$webpage$. When a customer accesses it, a tracking script embedded within the page will send information about the customer to a third-party service which will collect it. This third-party service will be www.analyze.com/$client$ on server www.analyze.com. The code snippets below describe what happens on the client and in her customer's browser. The latter simply loads a page containing a script. The only difference with respect to the code in Section 3.1.2 is that this script also processes information local to the customer, indicated as the argument $myInfo$. The code for the client shows what it does with it: the script extracts the customer's information (fst x) and sends it to the third-party tracker by invoking the customized service www.analyze.com/$client$. The code fragment snd x displays the page contents to the customer. The overall construction snd $\langle$call wa with fst x, snd $x\rangle$ is interesting: it logically reduces to just snd x, but the eager semantics of QWeS^{2}T causes both components of the pair to be executed, which has the effect of sending the customer's information to the analytics site.



We can easily compose this code (which already combines most of the techniques seen above) with the previous example. This would yield a web analytics auto-installer, by which the client could call a service on www.analyze.com which customizes the tracking agent and automatically installs it on the client's site, resulting in the code we just examined.

3.4 Beyond Traditional Web Programming

In this last section, we consider a group of examples that go beyond what is customarily done in traditional web programming, bordering aspects of cloud computing along the lines of *platform-as-a-service* (were we to cast our previous examples in the context of cloud computing, they would belong to what is known as *software-as-a-service*). We will indeed explore the possibility of outsourcing otherwise local execution to a remote server—this server could be vastly more powerful than the client, or it may maintain some useful libraries.

To make these examples more interesting, we will assume an extension of QWeS^{2}T that provides types for integers (int) and Boolean, together with some standard operations (all we will use is multiplication, subtraction, conditional, and equality). The current QWeS^{2}T prototype supports all these extensions.

3.4.1 Remote Libraries

In our first example, we will implement the factorial function in QWeS^{2}T. A single-host implementation of the factorial is totally standard, and can be glanced from Section 3.4.2. Instead, we will assume that server www.server.com provides a multiplication service, www.server.com/$mult$: then, rather than performing the multiplications locally during the recursive calls, we will offload them to this server. The resulting code is as follows:

www.server.com/$mult$ ↷

www.server.com

```
publish x : int × int.
    fst x ∗ snd x
```

client

```
let
  times = url(www.server.com, mult)
  fact = fix f : int → int.λx : int.
    if x = 0 then 1 else call times with ⟨x, f (n − 1)⟩
in
  fact 4
end
```

In this example, we have used www.server.com/$mult$ as a remote library. Just like any library, we do not need to know how it is implemented. Furthermore, the server can update its implementation at any time, transparently from the point of view of the client. Notice also that we used this library in the recursive call of the factorial function—when evaluating $fact\ n$, we are making n calls to www.server.com/$mult$.

3.4.2 Evaluation Service

Our next example takes this idea further: we will have the entire computation take place on the server. To this end, we have the server provide an evaluation service, www.server.com/$eval$ that accepts a function f from int to int and an argument n of type int, computes $f\ n$, and returns the result to the client. The type of this service is srv[susp[int → int] × int][int]: we need to suspend the function before shipping it to this server. The resulting client and server code is displayed below. It uses a client-side factorial function for demonstration.

www.server.com/$eval$ ↷

www.server.com

```
publish x : susp[int → int] × int.
    (resume (fst x)) (snd x)
```

client

```
let
  ev = url(www.server.com, eval)
  fact = fix f : int → int.λx : int.
    if x = 0 then 1 else x ∗ f (n − 1)
in
  call ev with ⟨hold fact, 4⟩
end
```

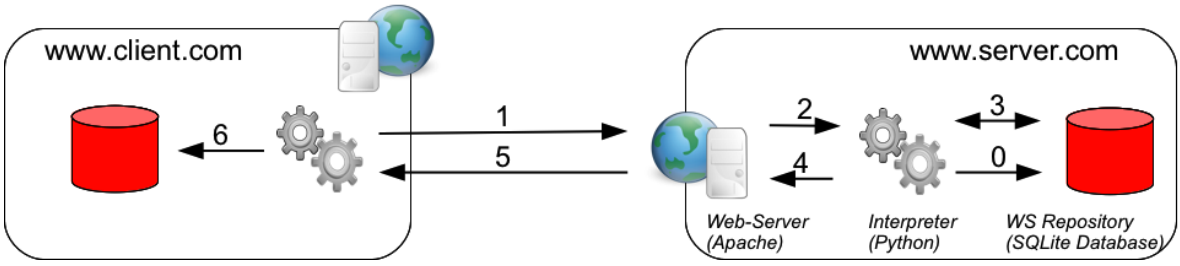


Figure 8: Prototype Implementation

In a sense, this last example operates in the opposite way with respect to the JavaScript-like examples we examined earlier: rather than having the server supply code that is executed on the client, it is the client that has code that will be executed remotely. Platform-as-a-service applications of cloud computing rely heavily on this idea: hire computing cycles from a well-provisioned provider to run computations that exceed the local computing infrastructure.

This example opens the doors to a number of other applications of cloud computing and distributed computing in general, for example grid computing. Furthermore, combining this example with that in Section 3.4.1 embodies some ideas of the *map-reduce* model [20], which could be realized in an extension of QWeS^{2}T with more expressive data structures such as lists and other collections. These examples also highlight the need for a language with primitive support for security.

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, as illustrated in Figure 8:

- **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 extended with Booleans, integers and strings and their most common operators. 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.

Users interact with the QWeS^{2}T prototype through the web. A node on the network can support several users. Each user has a *service page* that lists all the services he/she published, with each listing consisting of a URL for the service, its type, and a descriptive comment (see the bottom of Figure 9 for an example). Another programmer can make use of this information to call this service. The service page also allows the user that owns it to log in. Once logged in, he/she can delete services, modify their descriptions and open a code editor pane where he/she can write new services directly in QWeS^{2}T — the screenshot in Figure 9 shows the user in the act of adding code that, when evaluated using the "T" button, will publish a "hello world" service. When pressing "T", the code is sent to the web interface that evaluates it and displays the result back to the user. If this evaluation creates any new service, it is stored in the web service repository and the service page is updated.

We have implemented all the examples shown in section 3 and more. Here is a trace of what happens internally in our prototype during the execution of our most complex example, the web service auto-installer of Section 3.3.4. The steps of the computation can be traced on Figure 8 (steps 6 to 9 involve the $customer$ and are not shown).

0. As a preliminary step, the server publishes the web service in its local repository, and makes it available at www.server.com/$installer$.
1. The interpreter on the client sends an HTTP request through the web interface, thereby calling the service www.server.com/$installer$ with argument $clientInfo$.

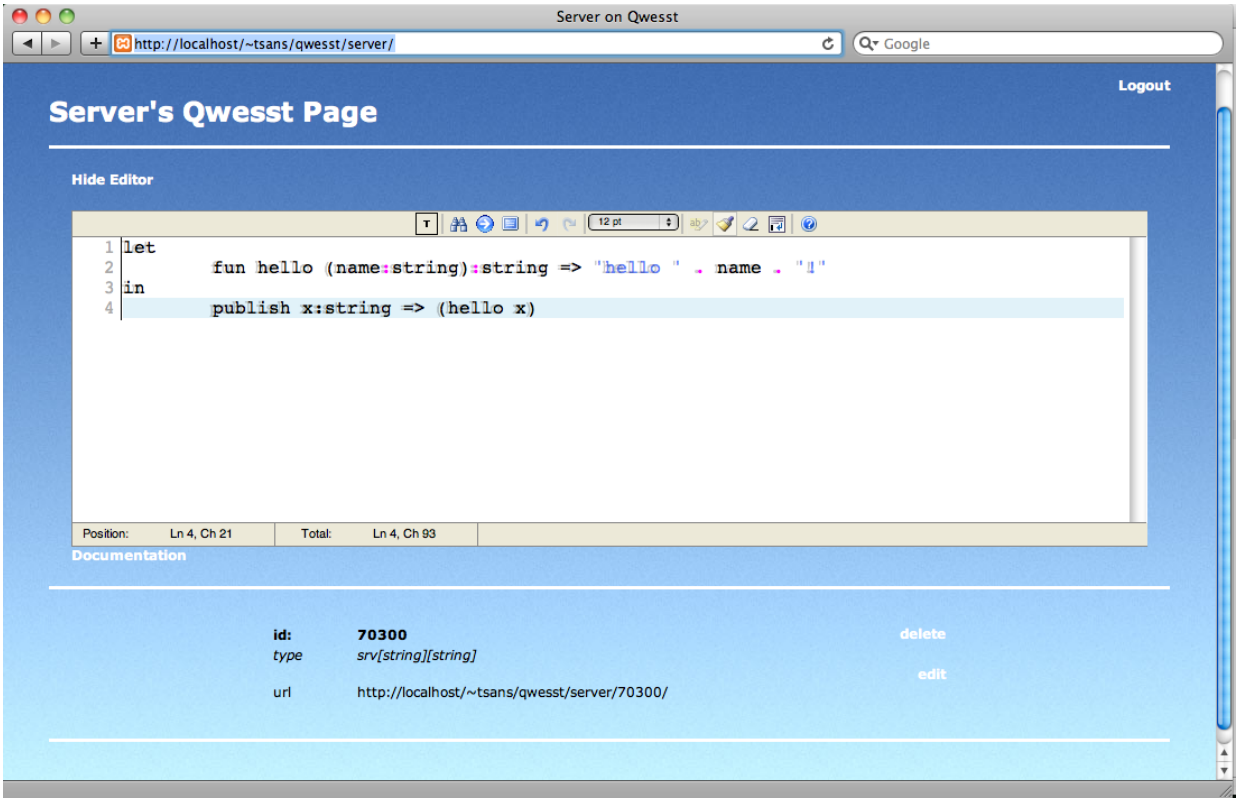


Figure 9: The Service Page as Seen by a User

2. www.server.com extracts the service and argument of this request, spawns an instance of its local interpreter, and passes these parameters to it.
3. The server's interpreter retrieves the service *installer* from its local web service repository and evaluates it on the client's argument.
4. The server's interpreter hands the resulting value (the personalized web search service) to its web interface.
5. The web interface packages it for transmission over HTTP and ships it back to the client.
6. The client completes the evaluation by publishing this new service in its local web service repository and generates a new identifier, say *clientSearch*, to make it available to the outside world through its own web server www.client.com.
7. The interpreter at the customer node invokes this service on data *myQuery* through its own web interface.
8. The client processes this query by parsing the request, spawning a new interpreter instance, retrieving the web service from its local repository, computing the result, and sending it back through its web interface.

9. Finally, the customer receive the result through its web interface and uses it for whatever purpose.

5 Parallelism

The dynamic semantics developed for QWeS^{2}T in Section 2 is distributed but sequential: computation happens at exactly one host at any time. Undeniably, the Web does a good job at fueling this illusion: we carry out most of our tasks on our local machine and occasionally send web requests, e.g., a search on Google, that get answered after a short wait. We easily forget that there are millions of other people on the Web, carrying out computations at the same time as we do, and that Google is servicing thousands of queries at any time.

In this section, we will explore parallel computation in the context of QWeS^{2}T. Specifically, we will develop three parallel semantics for this language, for three notions of parallelism. We will set the stage for a semantics that supports parallel transitions in Section 5.1. Our first parallel semantics, in Section 5.2, will allow QWeS^{2}T computations to happen simultaneously at multiple nodes in the network, thereby supporting a model of computation that is closer to the way the Web really works. Section 5.3 will extend QWeS^{2}T with a new construct that requests two subcomputations to be run in parallel, while the rest remains sequential. Section 5.4 will go back to our original QWeS^{2}T, but extract the fine-grained parallelism inherent in individual programs.

Even in the presence of parallelism, our models of computation will remain purely functional: evaluating the same expressions any number of times will always yield the same results. Evaluation is indeed effect-free — publishing new services is a benign effect, similar to fluid bindings, as there is no primitive in QWeS^{2}T for updating a service or testing for its existence. In particular, there is no mutable state and no concurrency. This means that we do not need to worry about race conditions or to provide mechanisms to resolve them. Similarly, computations will not need to compete for resources, avoiding to have to deal with deadlocks and livelocks. Of course, this is simplistic as web programming takes great advantage of state (most webapps use a database) and concurrency (e.g., a chat webapp). Nonetheless, we suspect this is a good basis for studying mechanisms to incorporate effects in our language in a way that is both useful and predictable.

We will not attempt to prove any meta-theoretic result about any of our parallel semantics in this report: this section is best seen as a preview of future work rather than an account of past achievements. We believe that these semantics are type safe and confluent in the sense that, although which threads are executed is highly non-deterministic, any choices can be reconciled in one additional step.

5.1 Parallel Transitions

The judgment modeling the transition semantics of QWeS^{2}T had the following form in Section 2:

$$\underbrace{\{\Omega_1\}_{\mathsf{w}_1}, \ldots, \{\Omega_n\}_{\mathsf{w}_n}}_{\Delta} \; ; \; e \; \mapsto_{\mathsf{w}} \; \underbrace{\{\Omega'_1\}_{\mathsf{w}_1}, \ldots, \{\Omega'_n\}_{\mathsf{w}_n}}_{\Delta'} \; ; \; e'$$

Host w (which is one among $\mathsf{w}_1, \ldots, \mathsf{w}_n$) was active, meaning that it was the one performing a step of computation on e, while the other hosts were idle (more precisely, w might also have been waiting for a remote computation to return a result — still, exactly one node is actively running). In this section, we will allow any node to be running its own computation while e is executing at w. This suggests a composite judgment of the following form:

$$\begin{array}{lll} \Omega_1; e_1 & \mapsto_{\mathsf{w}_1} & \Omega'_1; e'_1 \\ & \cdots & \\ \Omega_n; e_n & \mapsto_{\mathsf{w}_n} & \Omega'_n; e'_n \end{array}$$

where any number of nodes may execute a step of computation simultaneously.

In order to support remote procedure calls, we need to extend this idea in a couple of ways. First of all, when invoking a remote service, we want to execute it even if that node is already busy running something else. We will achieve this by letting a node execute not one expression e , but a multiset $\boldsymbol{\varepsilon}$ of computations at the same time: remember, Google performs thousands of searches simultaneously. Each of these computations will run on the same host but be otherwise independent. Note that this also captures the situation where a host has nothing to do: the multiset $\boldsymbol{\varepsilon}$ can simply be empty. The resulting composite judgment (we will call it a "macro-judgment") has the following form:

$$\begin{array}{lll} \Omega_1; \boldsymbol{\varepsilon}_1 & \mapsto_{\mathsf{w}_1} & \Omega'_1; \boldsymbol{\varepsilon}'_1 \\ & \cdots & \\ \Omega_n; \boldsymbol{\varepsilon}_n & \mapsto_{\mathsf{w}_n} & \Omega'_n; \boldsymbol{\varepsilon}'_n \end{array}$$

We will call, somewhat improperly, the multiset $\boldsymbol{\varepsilon}$ of computations executing at the same node w the *thread pool* of w, and each $\epsilon \in \boldsymbol{\varepsilon}$ a *thread* on w. Indeed, we have the following definition:

$$\text{Thread pools} \qquad \boldsymbol{\varepsilon} \;::=\; \cdot \;\mid\; \boldsymbol{\varepsilon}, \epsilon$$

where "$\cdot$" stands for the empty multiset. We will remain vague for the moment about what a thread ϵ is exactly. The precise definition will vary with each of the semantics in the sections to come.

We call the pair consisting of the services Ω and thread pool $\boldsymbol{\varepsilon}$ associated with a host w the *computational state* of w and the set of pairs $(\Omega_i, \boldsymbol{\varepsilon}_i)$ of all hosts on the network the (computational) state of the network, which we will denote as S. We will furthermore summarize a judgment such as the above as $S \mapsto\!\!\!\!\Rightarrow S'$. Specifically,

$$\underbrace{\boxed{\begin{array}{l} \Omega_1; \boldsymbol{\varepsilon}_1 \\ \cdots \\ \Omega_n; \boldsymbol{\varepsilon}_n \end{array}}}_{S} \quad \begin{array}{l} \mapsto_{\mathsf{w}_1} \\ \cdots \\ \mapsto_{\mathsf{w}_n} \end{array} \quad \underbrace{\boxed{\begin{array}{l} \Omega'_1; \boldsymbol{\varepsilon}'_1 \\ \cdots \\ \Omega'_n; \boldsymbol{\varepsilon}'_n \end{array}}}_{S'}$$

At this point, we have a notation for a network state transition, namely $S \mapsto\!\!\!\!\Rightarrow S'$, but we have not defined exactly what it means. Informally, we will have $S \mapsto\!\!\!\!\Rightarrow S'$ whenever each thread on each host performs at most one step of computation. The two extreme situations implied by this

definition are that each thread on each host performs one step in unison, and that no thread on no host performs a step (in which case $S' = S$). We allow intermediate forms of transitions (in which some threads make a step and others don't) to account for situations where a thread may be waiting for the result from a remote service invocation. We are also trying to avoid hardwiring a common clock on every host on the network.

Borrowing from computational multiset rewriting [5, 9, 16], we define $S \Mapsto S'$ by specifying the computation performed by each of these threads, or a small number of coordinated threads. We will use "micro-judgments" of the form

$$\Omega; \epsilon \mapsto_{\mathsf{w}} \Omega'; \epsilon'$$

to indicate that thread ϵ evolves to ϵ' in (exactly) one step, possibly extending the repository Ω into Ω'. When the repository does not change, we will further abbreviate it as $\epsilon \mapsto_{\mathsf{w}} \epsilon'$ for conciseness. Recall that all threads on the same host share the same service repository. Therefore, Ω in the above micro-judgment pertains to the host w rather than to the individual thread ϵ. This means that if two or more threads add services to the repository in the same step, the overall effect will be to add all those services to the repository [16].

5.2 Network Parallelism

With a framework to describe parallel transitions in place, we begin by modifying the dynamic semantics in Section 2 to support simultaneous computation at multiple nodes in the network. Because services can be invoked on a host that is already running code (either executing some expression of interest or responding to a service request), we further allow multiple independent threads to run at the same host, using the setup developed in the last section.

Predictably, supporting network parallelism does not require making any change to the external syntax of QWeS^{2}T, which the programmer uses to write her applications. Having now an independent notion of thread provides however an opportunity to factor out the expression form "expect e from w", which was used internally only. For clarity, we collect the (external) syntax of QWeS^{2}T (which was introduced piecemeal in Section 2) in the following grammar:

Types	τ	$::=$	$\tau \to \tau' \mid \tau \times \tau' \mid \mathsf{unit} \mid \mathsf{susp}[\tau] \mid \mathsf{srv}[\tau][\tau']$
Expressions	e	$::=$	$x \mid \lambda x:\tau.\, e \mid e_1\ e_2 \mid \mathsf{fix}\, x:\tau.e$
			$\mid \langle e_1, e_2\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid ()$
			$\mid \mathsf{hold}\ e \mid \mathsf{resume}\ e$
			$\mid \mathsf{url}(\mathsf{w}, u) \mid \mathsf{publish}\ x:\tau.e \mid \mathsf{call}\ e_1\ \mathsf{with}\ e_2$

Now, the way the transition semantics in Section 2 modeled the internal expression $\mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w}'$ was strange: the computation of e happened at the remote host w', yet the abstraction in Figure 6 showed it evolving locally as a sequence $\mathsf{expect}\ e'\ \mathsf{from}\ \mathsf{w}'$, $\mathsf{expect}\ e''\ \mathsf{from}\ \mathsf{w}'$, etc., until the embedded expression reduced to a value, which was handed to the local host. This is clearly not the way a remote procedure call work on the Web: the argument is sent to the remote host, executed there while the local host is waiting (assuming a synchronous behavior), and the result is sent back. We will now model this behavior more closely. To do so, we need to introduce *labels*,

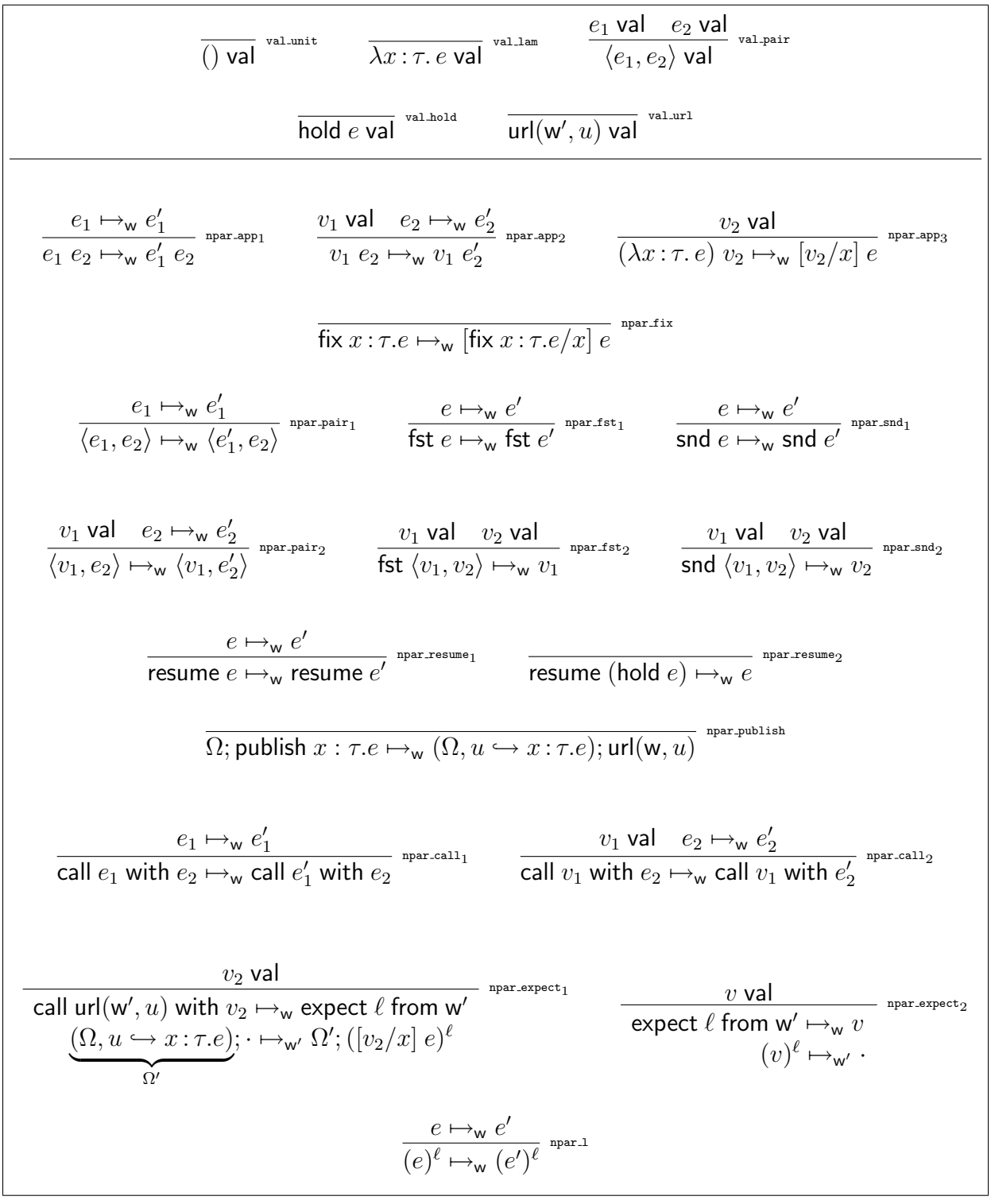


Figure 10: Parallel Network Evaluation Rules

also called *destinations*. When invoking a remote service, the local host will create a label ℓ and send it off together with the expression to be evaluated. It will then actively wait for the result from the remote host w' by means of the thread "expect ℓ from w'" — a modification of our former "expect e from w'" that replaces the expression e with the label ℓ. The remote host will execute e and once a value has been produced, it will send it back to the local host. In this section, the label acts therefore as a unique return address for a remote service request.

In this variant of the dynamic semantics of QWeS^{2}T, threads can be normal QWeS^{2}T expressions, artifacts of the form "expect ℓ from w", or labeled expressions of the form $(e)^\ell$. This is described by the following grammar.

$$\text{Threads} \qquad \epsilon \;::=\; e \;\mid\; \mathsf{expect}\ \ell\ \mathsf{from}\ \mathsf{w} \;\mid\; (e)^\ell$$

The resulting network semantics, which relies on the conventions introduced in the last section, is displayed in Figure 10. All but the last three rules correspond directly to the localized semantics of Section 2. Rule $\texttt{npar\_expect}_1$ reengineers rule $\texttt{step\_call}_3$: on the local host w, the expression call $\mathsf{url}(\mathsf{w}', u)$ with v_2 is reduced to the active waiting thread expect ℓ from w' for a new label ℓ; on the remote host w', the new thread $([v_2/x]\ e)^\ell$ is started where e is the body of the service u and ℓ is the very label w is waiting on. Rule $\texttt{npar\_expect}_2$ models the hand-off of the value computed by w' in response to this request and corresponds to rule $\texttt{step\_expect}_2$. Notice that neither rule forces the hosts w and w' to be different. The last rule bridges evaluation of labeled and unlabeled threads.

5.3 Explicit Parallelism

The technique just seen to model remote procedure calls and responses is amenable to describing other forms of parallelism. In this section, we use it to extend QWeS^{2}T with a simple parallel pair construct that the programmer can use to programmatically request that two expressions be evaluated as parallel threads. The main computation waits for both results before continuing. The syntactic extension to the external language is as follows:

$$\text{Expressions} \qquad e \;::=\; \ldots \;\mid\; \mathsf{letpar}\ x_1 \otimes x_2 = e_1 \otimes e_2\ \mathsf{in}\ e$$

For the sake of completeness, we give the obvious typing rule for parallel pairs:

$$\frac{\Sigma \mid \Gamma \vdash_{\mathsf{w}} e_1 : \tau_1 \quad \Sigma \mid \Gamma \vdash_{\mathsf{w}} e_2 : \tau_2 \quad \Sigma \mid \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash_{\mathsf{w}} e : \tau}{\Sigma \mid \Gamma \vdash_{\mathsf{w}} \mathsf{letpar}\ x_1 \otimes x_2 = e_1 \otimes e_2\ \mathsf{in}\ e : \tau}\ \texttt{of\_lp}$$

To model the execution of parallel pairs, we extend our definition of threads from Section 5.2 with the following production

$$\text{Threads} \qquad \epsilon \;::=\; \ldots \;\mid\; \mathsf{letpar}\ x_1 \otimes x_2 = \ell_1 \otimes \ell_2\ \mathsf{in}\ e$$

where the subexpressions e_1 and e_2 have been replaced with labels. The transition rules for this construct are very similar to $\texttt{npar\_expect}_1$ and $\texttt{npar\_expect}_2$ in Section 5.2:

$$\frac{}{\mathsf{letpar}\ x_1 \otimes x_2 = e_1 \otimes e_2\ \mathsf{in}\ e \mapsto_{\mathsf{w}} \mathsf{letpar}\ x_1 \otimes x_2 = \ell_1 \otimes \ell_2\ \mathsf{in}\ e, (e_1)^{\ell_1}, (e_2)^{\ell_2}}\ \texttt{epar\_lp}_1$$

$$\frac{v_1\ \mathsf{val} \quad v_2\ \mathsf{val}}{\mathsf{letpar}\ x_1 \otimes x_2 = \ell_1 \otimes \ell_2\ \mathsf{in}\ e, (v_1)^{\ell_1}, (v_2)^{\ell_2} \mapsto_{\mathsf{w}} [v_1/x_1]\,[v_2/x_2]\,e}\ {}^{\mathsf{epar\_lp}_2}$$

The first rule spawns new threads for e_1 and e_2 tagging them with new labels ℓ_1 and ℓ_2 respectively, and puts the current computation in an dormant state by replacing it with the thread "letpar $x_1 \otimes x_2 = \ell_1 \otimes \ell_2$ in e". The second rule kicks in once values v_1 and v_2 have been produced for e_1 and e_2, respectively. It simply substitutes them for the variables x_1 and x_2 in the body of the parallel pair construct. Notice how labels are used to remember where subcomputations should deliver their result to.

5.4 Implicit Parallelism

In this section, we devise a maximally parallel semantics for our original version of QWeS^{2}T (deprived of the parallel pair construct introduced in Section 5.4). We are going after the intrinsic parallelism present in the language. The technique we use to do so is known as *linear destination passing style* [38, 15] and can be seen as an extreme form of the labeling approach we sporadically used in the last two sections. Labels ℓ are known as *destinations* in this setting. Linear destination passing style can be understood as a systematic development of a standard stack semantics for a programming language, with which it shares many ingredients and properties, except the ability to be implemented using a stack.

We will rely on a more sophisticated notion of thread than what we needed so far. Threads can take one of three forms:

- They can be labeled expressions $(e)^{\ell}$ which will compute expression e to a value that will be delivered to destination ℓ.
- They can be labeled *frames*, which resemble expressions except that some of their subexpressions have been replaced with destinations. In the last two sections, "expect ℓ from w" and "letpar $x_1 \otimes x_2 = \ell_1 \otimes \ell_2$ in e" were (unlabeled) frames: their job was to wait for the computations tagged with their labels to yield values. We will make a systematic use of this technique.
- They can be labeled return values $(v)_{\ell}$ where v is a value in our traditional sense and ℓ is the destination that is waiting for it.

We call the first two forms *evaluation threads* (the destination appears as a superscript) and the last *return threads* (the destination is in a subscript position). Altogether, threads are defined by the following grammar:

$$\begin{array}{lrcl} \text{Threads} & \epsilon & ::= & (e)^{\ell} \\ & & \mid & (\ell_1\ \ell_2)^{\ell} \mid \langle \ell_1, \ell_2 \rangle^{\ell} \mid (\mathsf{fst}\ \ell')^{\ell} \mid (\mathsf{snd}\ \ell')^{\ell} \\ & & \mid & (\mathsf{resume}\ \ell')^{\ell} \mid (\mathsf{call}\ \ell_1\ \mathsf{with}\ \ell_2)^{\ell} \mid (\mathsf{expect}\ \ell'\ \mathsf{from}\ \mathsf{w})^{\ell} \\ & & \mid & ()_{\ell} \mid (\lambda x : \tau.\, e)_{\ell} \mid \langle v_1, v_2 \rangle_{\ell} \mid (\mathsf{hold}\ e)_{\ell} \mid (\mathsf{url}(u, \mathsf{w}))_{\ell} \end{array}$$

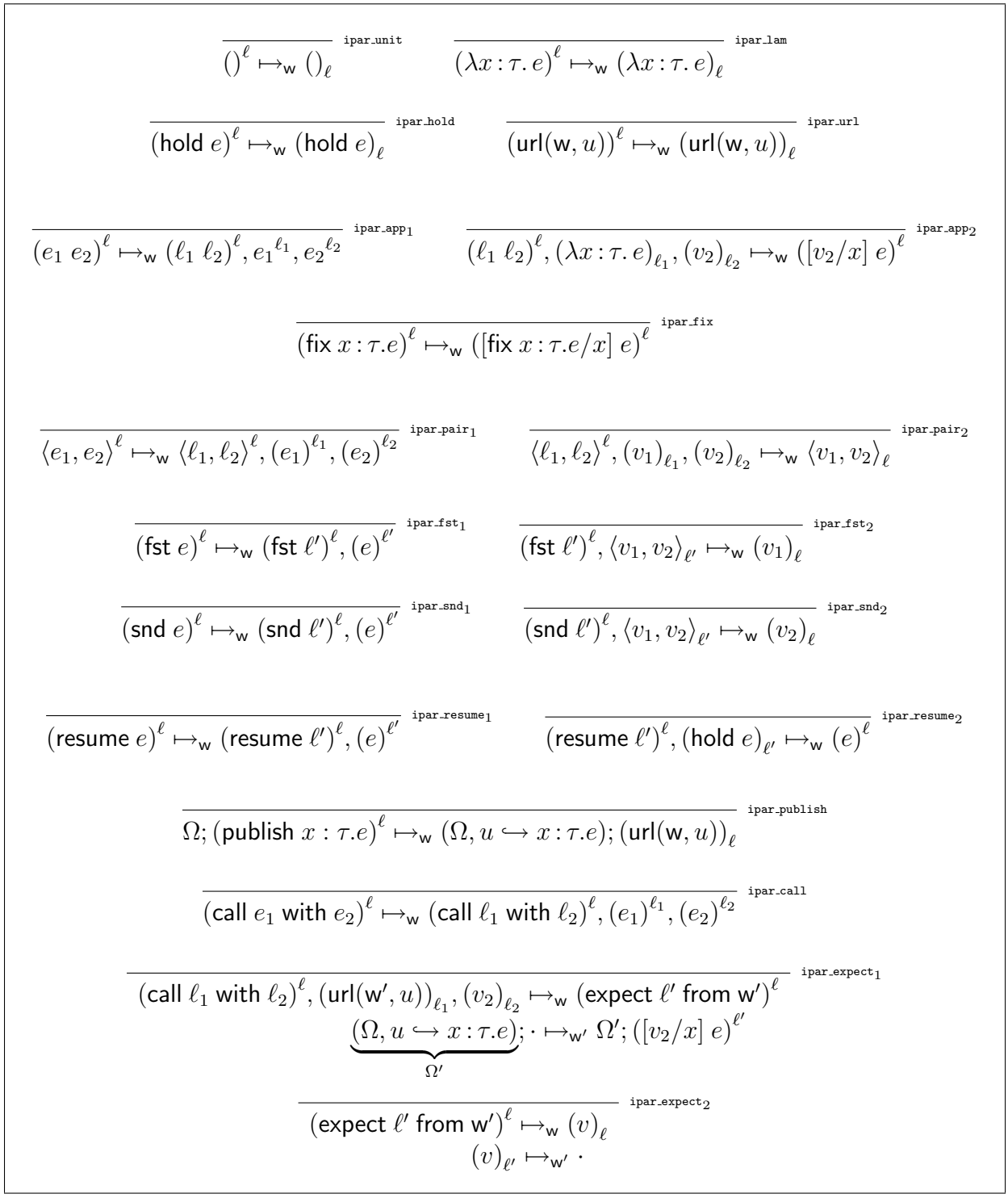


Figure 11: Linear Destination Passing Semantics for QWeS^{2}T

The linear destination passing semantics for QWeS^{2}T is displayed in Figure 11. A first thing to notice is that we have done away with the value judgment v val in favor of transition such as `ipar_unit` that rewrite an evaluation frame for a value to the corresponding return frame. Pairs deviate slightly from this pattern because their components need to be values themselves for the pair to be a value: this is captured by rule $\texttt{ipar\_pair}_2$.

The evaluation of a generic labeled expression $(e)^\ell$ proceeds by spawning threads for each of the subexpressions in e and replacing it with a frame that waits for their results. Once return threads are available for them, this frame is reduced as appropriate. For example, the evaluation of an application $(e_1\ e_2)^\ell$ in rule $\texttt{ipar\_app}_1$ spawns evaluation threads $(e_1)^{\ell_1}$ and $(e_2)^{\ell_2}$ for new labels ℓ_1 and ℓ_2 and places the application itself in a dormant state as the frame $(\ell_1\ \ell_2)^\ell$. Once values have been produced for e_1 and e_2, as witnessed by the return frames $(\lambda x : \tau.\, e)_{\ell_1}$ and $(v_2)_{\ell_2}$ respectively, rule $\texttt{ipar\_app}_2$ kicks in and reduces the application to $([v_2/x]\ e)^\ell$.

The form of linear destination passing style used in this section supports a very fine grained form of parallelism, essentially at the instruction level. While it is an interesting proof of concept, we do not advocate it as a practical model of computation, at least relative to current architectures. Indeed, the overhead involved in thread creation is likely to outweigh any performance benefits, even in a massively parallel computer. Yet, its realization at larger granularities for computation units has the potential of big performance gains relative to management overhead.

6 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) [24], 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: server code is not meant to be sent out and installed on another node — this is as if QWeS^{2}T disallowed a publish in the scope of a hold. 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 in the webapp arena. Links [18] 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 it has been generated, it cannot be customized and moved around as QWeS^{2}T could do in Section 3.3. Other efforts along similar lines include Swift [17] which also produces JavaScript code for the client and Java for the server but which additionally allows annotating a program with information flow properties to reason about secrecy and authentication, QHTML [21], a client-server module for Oz with support for declarative graphical user interface programming, and Hop [48], a dynamically typed language for web programming based on Scheme.

While research on web application programming has started in earnest only fairly recently, the

study of web services has a much longer history. By web service we mean a software agent that performs its functionality by engaging other agents (services) in a preestablished sequence of message exchanges. The allowable communication patterns, which internal and external choices can make quite complex, is specified through an interaction protocol (also called a contract). A major concern when composing web services is to ensure that they are interoperable, i.e., that their protocols are compatible — no service ever gets stuck waiting for a message that nobody will deliver. This is called the conformance problem. Conformance testing corresponds to cryptographic protocol verification deprived of the attacker's malice [51]. Recently, a number of authors have proposed abstract frameworks to study web services interactions and related concepts. Carpinetti et al. [14] formally define contracts in process algebra and cast performance testing as process simulation, while Bravetti and Zavattaro [11] study it as matching preorders of complementary input/output operations. Baldoni et al. [3] study the key issue of choice and achieve adaptability through surgical edit operations, while in previous work they approached conformance testing with the help of finite state automata [4]. Conformance has also been studied through session types [54], which describe communication channels in process algebras. Web applications, as considered here, can be seen as degenerate web services with nearly trivial interaction protocols: a host can at most invoke a remote function defined on another host on supplied arguments and wait for the result — this is no different from invoking a local library function except that we need to model where the computation is taking place. The conformance problem degenerates to type checking, which must be localized. It should be observed that this restricted form of interaction protocol is common in industrial web services and is essentially what is defined in the WSDL standard [55].

The design of QWeS^{2}T has been influenced by Lambda 5 [33, 34]. Lambda 5 is an abstract programming language for distributed computing that uses a modal type system to capture the notion of localized computation. It deploys three modalities for this purpose:

- The type $\Box\tau$, with expression constructors $\mathsf{box}\,\omega.\,M$ and $\mathsf{unbox}\,M$, corresponds rather directly to our $\mathsf{susp}[\tau]$ (the world variable ω in $\mathsf{box}\,\omega.\,M$ allows making expressions parametric over a node, which is not supported in our language as host names are not first-class objects—a similar choice was made in [27]).

- The type $\Diamond\tau$ specialized to functional types, its expression forms $\mathsf{here}\,M$ and $\mathsf{letd}\,\omega, x = M\,\mathsf{in}\,N$ together with the world shift expression $\mathsf{get}[\mathsf{w}]M$, conjure our type $\mathsf{srv}[\tau][\tau']$ and the publish/call remote procedure call mechanism. Specifically, $\mathsf{publish}\ x : \tau.e$ of type $\mathsf{srv}[\tau][\tau']$ would be written as the Lambda 5 expression $\mathsf{here}\,(\lambda x\!:\!\tau.\,e)$ of type $\Diamond(\tau \to \tau')$. Instead, $\mathsf{call}\ \mathsf{url}(u, \mathsf{w}')\ \mathsf{with}\ e'$ at world w where $u : \mathsf{srv}[\tau][\tau']$ is the URL $u \hookrightarrow x\!:\!\tau.e$ on node w', corresponds to the Lambda 5 expression $\mathsf{letd}\,\omega, x = e\,\mathsf{in}\,\mathsf{get}[\mathsf{w}](x\ \mathsf{get}[\mathsf{w}']e')$ where e would have type $\Diamond(\tau \to \tau')$.

- The additional "shamrock types" with constructors $\mathsf{hold}\,M$ (which is different from our hold) and $\mathsf{leta}\ x = M\ \mathsf{in}\ N$ do not have counterparts in our language.

Altogether, QWeS^{2}T can be seen as a simplification of Lambda 5 specialized to web programming. As such, it streamlines this language and provides constructs that abstractly capture the way we use the web directly. Lambda 5's prototype, ML5 [34], although targeting web programming, is

limited to basic browser-server interactions and cannot express all examples in Section 3.3. On the other hand, it implements a large fragment of Standard ML [31] as well as a basic document object model (DOM). Furthermore, its capabilities have been demonstrated on several non-trivial case studies [33].

Other efforts besides Lambda 5 have looked at modal logic to express located computation [10, 32, 37]. Modal logic is appealing because it naturally supports expressing phenomena, here computation, from a variety of view points, network nodes here. Among them, Jia and Walker [27] find a foundation of distributed computing in an intuitionist modal logic and distill a programming language from it using the Curry-Howard isomorphism. Additionally, Cardelli and Gordon studied a modal semantics for the ambient calculus [13] (more below).

Just like Lambda 5, QWeS^{2}T also fits the long-standing thread of research endeavors aimed at extending functional programming languages with support for distributed computation. Distributed ML [28] looks at concurrency and fault tolerance, while Facile [52] fuses functional computation à la ML [31] and process algebra à la CCS [30] by supporting typed channels in the style of another effort, Concurrent ML [41]. JoCaml [29] brings O'Caml [40] and the join calculus [23] together by defining hierarchical locations similar to mobile ambients [12]. Another offspring of O'Caml is Acute [49], which explores typing mechanisms for data exchange in programs whose running environment may not be fully known in advance. All these efforts tend to focus on low level aspects of distributed programming, which contrasts with QWeS^{2}T's direct mechanisms to model web programming as done in practice. A particularly interesting effort is Alice [43, 42], a language for "open programming", a paradigm which supports a distributed view of programming by which a program routinely integrates calls to remote libraries and other remote code rather than compiling local copies. This fits well a "social programming" model for which QWeS^{2}T would be particularly appropriate.

Several authors have proposed extending process calculi [45] to support location-based distributed programming. Nomadic Pict [53] tags every process with the host it is running on and defines communication primitives among them. By contrast, SafeDpi [26] corals processes into locations and provides operators to move processes between them. Mobile ambients [12, 13, 6] push this idea further by segregating processes into possibly nested locations (ambients) and providing operators to move ambients around and dissolve their boundaries to permit them to communicate. Closer to us, Ferrara [22] expresses web services in a process algebra, thereby enabling the use of verification techniques based on temporal logic and process equivalence to gain correctness assurances, without losing the ability to compile them to executable code. Efforts born out of process algebra, including all of the above, bring to the foreground the communication aspect of distributed systems and web programming. By emphasizing channels, messages and processes, they differ from QWeS^{2}T and some of the languages mentioned earlier, which tend to endorse a localized view of computation which weaves remote calls and mobile code within the fabric of the bulk of the execution taking place at a host. Indeed, these languages typically rely on fairly simple communication modalities, a form of message passing in which each request results in a response in the case of QWeS^{2}T, which contrasts with the sophisticate communication patterns of process algebra [45].

7 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, 8]. Therefore, we believe that we can braid a policy language for static information flow security [35, 44, 7] 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 [50, 56].

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. We are also thankful to the anonymous colleagues who reviewed an early version of this paper [47] for the LAM'10 workshop.

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] Matteo Baldoni, Cristina Baroglio, Amit K. Chopra, Nirmit Desai, Viviana Patti, and Munindar P. Singh. Choice, interoperability, and conformance in interaction protocols and service choreographies. In K. Decker, J. Sichman, C. Sierra, and C. Castelfranchi, editors, *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'09)*, pages 843–850, Budapest, Hungary, 2009.

[4] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. A priori conformance verification for guaranteeing interoperability in open environments. In A. Dan and W. Lamersdorf, editors, *Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC'06)*, pages 339–351, Chicago, IL, 2006. Springer-Verlag LNCS 4294.

[5] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993.

[6] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivano Salvo, and Vladimiro Sassone. A type inference algorithm for secure ambients. In *Theory of Concurrency, Higher Order and Types, TOSCA Workshop 2001*, 2001.

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

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

[9] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[10] Tijn Borghuis and Loe Feijs. A constructive logic for services and information flow in computer networks. *The Computer Journal*, 43(4):274–289, 2000.

[11] Mario Bravetti and Gianluigi Zavattaro. Contract based multi-party service composition. In *Proceedings of the IPM International Symposium on Fundamentals of Software Engineering (FSEN'07)*, pages 207–222, Tehran, Iran, 2007. Springer-Verlag LNCS 4767.

[12] Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 79–92. ACM Press, 1999.

[13] Luca Cardelli and Andrew D. Gordon. Anytime, anywhere. modal logics for mobile ambients. In *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL'00)*, pages 365–377, San Antonio, TX, 2000. ACM Press.

[14] Samuele Carpineti, Giuseppe Castagna, Cosimo Laneve, and Luca Padovani. A formal account of contracts for web services. In *In Proceedings of the 3rd Workshop on Web Services and Formal Methods (FM-WS'06*, pages 148–162. Springer-Verlag LNCS 4184, 2006.

[15] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A Concurrent Logical Framework II: Examples and Applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 2002, revised May 2003.

[16] Iliano Cervesato and Andre Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information & Computation*, 207(10):1044–1077, 2009.

[17] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Stevenson, WA, 2007.

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

[19] 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.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation — OSDI'04*, pages 137–150, San Francisco, CA, 2004.

[21] Sameh El-Ansary, Donatien Grolaux, Peter Van Roy, and Mahmoud Rafea. Overcoming the multiplicity of languages and technologies for web-based development using a multi-paradigm approach. In *Procceedings of the 2nd International Conference on Multiparadigm Programming in Mozart/Oz (MOZ'04)*, pages 113–124, 2004.

[22] 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.

[23] Cédric Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, Palaiseau, 1998. INRIA TU-0556. Also available from `http://research.microsoft.com/~fournet`.

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

[25] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[26] Matthew Hennessy, Julian Rathke, and Nobuko Yoshida. SafeDPi: A language for controlling mobile code. Technical report, Department of Computer Science, University of Sussex, 2003.

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

[28] Clifford Krumvieda. *Distributed ML: abstractions for efficient and fault-tolerant programming*. PhD thesis, Department of Computer Science, Cornell University, 1993.

[29] Louis Mandel and Luc Maranget. Programming in JoCaml. Technical Report RR-6261, MOSCOVA – INRIA Rocquencourt, 2007.

[30] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[31] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[32] Jonathan Moody. Logical mobility and locality types. In Sandro Etalle, editor, *Proceedings of the 14th Symposium on Logic Based Program Synthesis and Transformation (LOPSTR'04)*, pages 69–84, Verona, Italy, 2004. Springer-Verlag LNCS 3573.

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

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

[35] 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.

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

[37] Sungwoo Park. A modal language for the safety of mobile values. In Naoki Kobayashi, editor, *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS'06)*, pages 217–233, Sydney, Australia, 2006. Springer-Verlag LNCS 4279.

[38] Frank Pfenning. Substructural operational semantics and linear destination-passing style. Slides of an invited talk to the Second Asian Symposium on Programming Languages and Semantics (APLAS'04). Available at `http://www.cs.cmu.edu/~fp/talks/aplas04-talk.ps`., 2004.

[39] 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.

[40] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Objects Systems*, 4(1):27–50, 1998.

[41] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[42] Andreas Rossberg. *Typed Open Programming - A higher-order, typed approach to dynamic modularity and distribution*. PhD thesis, Universität des Saarlandes, 2007.

[43] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice through the looking glass. In *Proceedings of the 4th Symposium on Trends in Functional Programming (TFP'04)*, Munich, Germany, 2004. Intellect Books.

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

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

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

[47] Thierry Sans and Iliano Cervesato. QWeS^{2}T for Type-Safe Web Programming. In Berndt Farwer, editor, *Third International Workshop on Logics, Agents, and Mobility — LAM'10*, Edinburgh, Scotland, UK, 2010.

[48] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a Language for Programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium (DLS'06)*, pages 975–985, Portland, OR, 2006.

[49] Peter Sewell, James Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. *Journal of Functional Programming*, 17(4–5):547–612, 2007.

[50] 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.

[51] Paul F. Syverson and Iliano Cervesato. The Logic of Authentication Protocols. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*. Springer-Verlag LNCS 2171, 2001.

[52] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)*, pages 278–298, London, UK, 1996. Springer-Verlag LNCS 1119.

[53] Asis Unypoth and Peter Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, London, UK, 2001.

[54] Vasco T. Vasconcelos. *9th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Web Services (SFM'09)*, chapter Fundamentals of Session Types, pages 158–186. Springer-Verlag LNCS 5569, 2009.

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

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

A Twelf Specification

This appendix contains the Twelf specification for the QWeS^{2}T language, its semantics and its proof of safety. This specification can also be found in [46] as a single file. A description of Twelf goes beyond the scope of this report: a wealth of information can be found at [1], a brief system description at [39], and its underpinning in the LF type theory at [25].

A.1 Syntax

The Twelf specification below represents every construct exactly as it was introduced in Section 2, with one exception: $\mathsf{url}(\mathsf{w}, u)$. The evaluation semantics of this construct requires that the variable u be dynamically generated. Doing so is cumbersome in Twelf — a direct implementation would unnecessarily complicate the specification and the proofs.

Our work-around is to treat URLs as if they were inlined versions of the remote service they point to. Therefore, we represent the value $\mathsf{url}(\mathsf{w}, u)$ with matching service $u \hookrightarrow x : \tau.e$ at host w as a *value* "published $x : \tau.\, e$ at w", which we encode in Twelf as `url w T ([x:exp]E x)`. Because it is a value, it is never modified. When in Section 2 call uses $\mathsf{url}(\mathsf{w}, u)$ to trigger $u \hookrightarrow x : \tau.e$ at w, the specification below achieves the same effect by using `url w T ([x:exp]E x)`. Of course, doing so defeats the practical purpose of making a *remote* procedure call, but it does not change the meta-theoretic properties of our language.

Within a Twelf representation, this has the additional advantage of dispensing with the need of giving an explicit representation to the local service repositories Ω_i (because all services are inlined), and consequently of the global repository Δ. While these collections can easily be encoded in Twelf, their presence significantly complicates the encoding of the semantics of a language, and especially its meta-theory.

```
% Hosts
host: type.              %name host W.

% Types
tp: type.                %name tp T.
unit  : tp.
prod  : tp -> tp -> tp.
arrow : tp -> tp -> tp.
susp  : tp -> tp.
srv   : tp -> tp -> tp.

% Expressions
exp: type.               %name exp E.
triv    : exp.
pair    : exp -> exp -> exp.
fst     : exp -> exp.
```

```
snd     : exp -> exp.
lam     : tp -> (exp -> exp) -> exp.
app     : exp -> exp -> exp.
fix     : tp -> (exp -> exp) -> exp.
hold    : exp -> exp.
resume  : exp -> exp.
url     : host -> tp -> (exp -> exp) -> exp.
publish : tp -> (exp -> exp) -> exp.
call    : exp -> exp -> exp.
expect  : exp -> host -> exp.
```

A.2 Static Semantics

A.2.1 Mobility

```
mobile: tp -> type.     %name mobile M.
%mode mobile +T.

mob_unit : mobile unit.
mob_prod : mobile (prod T1 T2)
            <- mobile T1
            <- mobile T2.
mob_susp : mobile (susp T).
mob_srv  : mobile (srv T1 T2)
            <- mobile T1
            <- mobile T2.
```

A.2.2 Typing

To leverage Twelf's representation of object variables as meta-variables, we store typing assumptions in Twelf's context as `of E W T` rather than `of E T`, which would be a direct encoding of the typing judgment of QWeS^{2}T in Section 2. A direct representation would significantly complicate the specification and the proofs.

```
of: host -> exp -> tp -> type.   %name of TP.
%mode of +W +E -T.

of_triv    : of W triv unit.
of_pair    : of W (pair E1 E2) (prod T1 T2)
               <- of W E2 T2
               <- of W E1 T1.
of_fst     : of W (fst E) T1
               <- of W E (prod T1 T2).
of_snd     : of W (snd E) T2
               <- of W E (prod T1 T2).
of_lam     : of W (lam T [x]E x) (arrow T T')
               <- {x:exp} of W x T -> of W (E x) T'.
of_app     : of W (app E1 E2) T'
               <- of W E2 T
               <- of W E1 (arrow T T').
```

```
of_fix     : of W (fix T [x]E x) T
               <- {x:exp} of W x T -> of W (E x) T.
of_hold    : of W (hold E) (susp T)
               <- of W E T.
of_resume  : of W (resume E) T
               <- of W E (susp T).
of_url     : of W (url W' T [x]E x) (srv T T')
               <- ({x: exp} of W' x T -> of W' (E x) T')
               <- mobile (srv T T').
of_publish : of W (publish T [x]E x) (srv T T')
               <- ({x: exp} of W x T -> of W (E x) T')
               <- mobile (srv T T').
of_call    : of W (call E1 E2) T'
               <- of W E2 T
               <- of W E1 (srv T T').
of_expect  : of W (expect E W') T
               <- of W' E T.
```

A.3 Dynamic Semantics

A.3.1 Values

```
val: exp -> type.   %name val VV.
%mode val +E.

val_unit : val triv.
val_pair : val (pair V1 V2)
             <- val V2
             <- val V1.
val_lam  : val (lam T E).
val_hold : val (hold E).
val_url  : val (url W T E).
```

A.3.2 Transition Rules

```
ev: host -> exp -> exp -> type.   %name ev EV.
%mode ev +H +E1 -E2.

ev_pair1   : ev W (pair E1 E2) (pair E1' E2)
               <- ev W E1 E1'.
ev_pair2   : ev W (pair V1 E2) (pair V1 E2')
               <- ev W E2 E2'
               <- val V1.
ev_fst1    : ev W (fst E) (fst E')
               <- ev W E E'.
ev_fst2    : ev W (fst (pair V1 V2)) V1
               <- val V2
               <- val V1.
ev_snd1    : ev W (snd E) (snd E')
               <- ev W E E'.
```

```
ev_snd2    : ev W (snd (pair V1 V2)) V2
               <- val V2
               <- val V1.
ev_app1    : ev W (app E1 E2) (app E1' E2)
               <- ev W E1 E1'.
ev_app2    : ev W (app V1 E2) (app V1 E2')
               <- ev W E2 E2'
               <- val V1.
ev_app3    : ev W (app (lam _ [x]E x) V2) (E V2)
               <- val V2.
ev_fix     : ev W (fix T [x]E x) (E (fix T [x]E x)).
ev_resume1 : ev W (resume E) (resume E')
               <- ev W E E'.
ev_resume2 : ev W (resume (hold E)) E.
ev_publish : ev W (publish T [x]E x) (url W T [x]E x).
ev_call1   : ev W (call E1 E2) (call E1' E2)
               <- ev W E1 E1'.
ev_call2   : ev W (call V1 E2) (call V1 E2')
               <- ev W E2 E2'
               <- val V1.
ev_call3   : ev W (call (url W' T [x]E x) V2) (expect (E V2) W')
               <- val V2.
ev_expect1 : ev W (expect E W') (expect E' W')
               <- ev W' E E'.
ev_expect2 : ev W (expect V _) V
               <- val V.
```

A.4 Type Preservation

QWeS^{2}T's type safety is the combination of the type preservation theorem and the progress theorems. Meta-theoretic properties are represented in Twelf in exactly the same way as other aspects of an object language. In particular, the various clauses for a type family correspond to the cases of a manual proof. Twelf provides mechanized support to ensure that a set of clauses actually implement a proof. For proofs of the form "for all ... exists ...", this amounts to showing that these clauses implement a total functions. This is achieved using two directives: `%mode` identifies which argument positions that are input and which are output (and checks that they are used as such), and `%total` which verifies that all cases are covered and that they all terminate (i.e., that it actually is a function and it is total).

In Section 2.5, type preservation relied on two auxiliary properties: the substitution and the relocation lemma. The substitution lemma comes for free in Twelf because of the methodology we used, which encodes QWeS^{2}T variables as Twelf variables. This means that a substitution $[v/x]e$ in QWeS^{2}T is represented as an application `E V` in Twelf, where `E` and `V` are the representations of e and v, respectively. Specifically the substitution lemma for Twelf [25, 39] subsumes the substitution lemma for any object language as long as it follows this methodology.

A.4.1 Relocation Lemma

The Twelf code below is a complete specification of our manual proof for the relocation lemma in Section 2.5. However, Twelf could not prove its totality. After consulting with the Twelf developers, we determined that this is apparently due to an incompleteness in Twelf's coverage checker (a part of what `%total` checks). We tried to use alternative encodings that would yield a machine-checkable proof, but were unable to do so short of significantly increasing the complexity of the specification. Below, we rely on the directive `%trustme` to tell Twelf that it should assume that the totality proof goes through, even if it is not able to do so itself.

```
reloc: of W E T -> {W':host} of W' E T -> type.
%mode reloc +OF +W' -OF'.

rel_triv:
  reloc (of_triv                  : of W  triv unit)
     W' (of_triv                  : of W' triv unit).
rel_pair:
  reloc (of_pair TP1  TP2         : of W  (pair E1 E2) (prod T1 T2))
     W' (of_pair TP1' TP2'        : of W' (pair E1 E2) (prod T1 T2))
  <- reloc (TP1                   : of W  E1 T1)
        W' (TP1'                  : of W' E1 T1)
  <- reloc (TP2                   : of W  E2 T2)
        W' (TP2'                  : of W' E2 T2).
rel_fst:
  reloc (of_fst TP                : of W  (fst E) T1)
     W' (of_fst TP'               : of W' (fst E) T1)
  <- reloc (TP                    : of W  E (prod T1 T2))
        W' (TP'                   : of W' E (prod T1 T2)).
rel_snd:
  reloc (of_snd TP                : of W  (snd E) T2)
     W' (of_snd TP'               : of W' (snd E) T2)
  <- reloc (TP                    : of W  E (prod T1 T2))
        W' (TP'                   : of W' E (prod T1 T2)).
rel_lam:
  reloc ((of_lam [x:exp][dx:of W  x T]TP  x dx)
                                  : of W  (lam T [x]E x) (arrow T T'))
     W' ((of_lam [x:exp][dx:of W' x T]TP' x dx)
                                  : of W' (lam T [x]E x) (arrow T T'))
  <- {x:exp}
     {dx :of W  x T}
     {dx':of W' x T}
        reloc dx W' dx'
          -> reloc (TP  x dx      : of W  (E x) T')
                W' (TP' x dx'     : of W' (E x) T').
rel_app:
  reloc (of_app TP1  TP2          : of W  (app E1 E2) T')
     W' (of_app TP1' TP2'         : of W' (app E1 E2) T')
  <- reloc (TP1                   : of W  E1 (arrow T T'))
        W' (TP1'                  : of W' E1 (arrow T T'))
  <- reloc (TP2                   : of W  E2 T)
```

```
        W' (TP2'                    : of W' E2 T).
rel_fix:
  reloc ((of_fix [x:exp][dx:of W  x T]TP  x dx)
                                    : of W  (fix T [x]E x) T)
     W' ((of_fix [x:exp][dx:of W' x T]TP' x dx)
                                    : of W' (fix T [x]E x) T)
  <- {x:exp}
     {dx :of W  x T}
     {dx':of W' x T}
        reloc dx W' dx'
          -> reloc (TP  x dx        : of W  (E x) T)
                W' (TP' x dx'       : of W' (E x) T).
rel_hold:
  reloc (of_hold TP                 : of W  (hold E) (susp T))
     W' (of_hold TP'                : of W' (hold E) (susp T))
  <- reloc (TP                      : of W  E T)
        W' (TP'                     : of W' E T).
rel_resume:
  reloc (of_resume TP               : of W  (resume E) T)
     W' (of_resume TP'              : of W' (resume E) T)
  <- reloc (TP                      : of W  E (susp T))
        W' (TP'                     : of W' E (susp T)).
rel_url:
  reloc (of_url M TP                : of W  (url W'' T [x]E x) (srv T T'))
     W' (of_url M TP                : of W' (url W'' T [x]E x) (srv T T')).
rel_publish:
  reloc ((of_publish M [x:exp][dx:of W  x T]TP x dx)
                                    : of W  (publish T [x]E x) (srv T T'))
     W' ((of_publish M [x:exp][dx:of W' x T]TP' x dx)
                                    : of W' (publish T [x]E x) (srv T T'))
  <- {x:exp}
     {dx :of W  x T}
     {dx':of W' x T}
        reloc dx W' dx'
          -> reloc (TP  x dx        : of W  (E x) T')
                W' (TP' x dx'       : of W' (E x) T').
rel_call:
  reloc (of_call TP1  TP2           : of W  (call E1 E2) T')
     W' (of_call TP1' TP2'          : of W' (call E1 E2) T')
  <- reloc (TP1                     : of W  E1 (srv T T'))
        W' (TP1'                    : of W' E1 (srv T T'))
  <- reloc (TP2                     : of W  E2 T)
        W' (TP2'                    : of W' E2 T).
rel_expect:
  reloc (of_expect TP               : of W  (expect E W'') T)
     W' (of_expect TP               : of W' (expect E W'') T).

%block reloc_ctx: some {W:host}{W':host}{T:tp}
                  block {x:exp}{dx:of W x T}{dx':of W' x T}{_:reloc dx W' dx'}.
%worlds (reloc_ctx) (reloc _ _ _).
%trustme
```

```
%total (TP) (reloc TP _ _).


relocate: of W E T -> {W':host} of W' E T -> type.
%mode relocate +OF +W' -OF'.

rr : relocate OF W' OF' <- reloc OF W' OF'.

%worlds () (relocate _ _ _).
%total (TP) (relocate TP _ _).
```

A.4.2 Type Preservation Theorem

```
tpres: of W E T -> ev W E E' -> of W E' T -> type.
%mode tpres +OF +EV -OF'.

tpres_pair1:
  tpres (of_pair TP1 TP2            : of W (pair E1 E2) (prod T1 T2))
        (ev_pair1 EV1               : ev W (pair E1 E2) (pair E1' E2))
        (of_pair TP1' TP2           : of W (pair E1' E2) (prod T1 T2))
  <- tpres (TP1                     : of W E1 T1)
           (EV1                     : ev W E1 E1')
           (TP1'                    : of W E1' T1).
tpres_pair2:
  tpres (of_pair TP1 TP2            : of W (pair V1 E2) (prod T1 T2))
        (ev_pair2 VV1 EV2           : ev W (pair V1 E2) (pair V1 E2'))
        (of_pair TP1 TP2'           : of W (pair V1 E2') (prod T1 T2))
  <- tpres (TP2                     : of W E2 T2)
           (EV2                     : ev W E2 E2')
           (TP2'                    : of W E2' T2).
tpres_fst1:
  tpres (of_fst TP                  : of W (fst E) T1)
        (ev_fst1 EV                 : ev W (fst E) (fst E'))
        (of_fst TP'                 : of W (fst E') T1)
  <- tpres (TP                      : of W E (prod T1 T2))
           (EV                      : ev W E E')
           (TP'                     : of W E' (prod T1 T2)).
tpres_fst2:
  tpres (of_fst (of_pair TP1 TP2)   : of W (fst (pair V1 V2)) T1)
        (ev_fst2 VV1 VV2            : ev W (fst (pair V1 V2)) V1)
        (TP1                        : of W V1 T1).
tpres_snd1:
  tpres (of_snd TP                  : of W (snd E) T2)
        (ev_snd1 EV                 : ev W (snd E) (snd E'))
        (of_snd TP'                 : of W (snd E') T2)
  <- tpres (TP                      : of W E (prod T1 T2))
           (EV                      : ev W E E')
           (TP'                     : of W E' (prod T1 T2)).
tpres_snd2:
  tpres (of_snd (of_pair TP1 TP2)   : of W (snd (pair V1 V2)) T2)
        (ev_snd2 VV1 VV2            : ev W (snd (pair V1 V2)) V2)
```

```
        (TP2                          : of W V2 T2).
tpres_app1:
  tpres (of_app TP1 TP2               : of W (app E1 E2) T')
        (ev_app1 EV1                  : ev W (app E1 E2) (app E1' E2))
        (of_app TP1' TP2              : of W (app E1' E2) T')
  <- tpres (TP1                       : of W E1 (arrow T T'))
           (EV1                       : ev W E1 E1')
           (TP1'                      : of W E1' (arrow T T')).
tpres_app2:
  tpres (of_app TP1 TP2               : of W (app V1 E2) T')
        (ev_app2 VV1 EV2              : ev W (app V1 E2) (app V1 E2'))
        (of_app TP1 TP2'              : of W (app V1 E2') T')
  <- tpres (TP2                       : of W E2 T)
           (EV2                       : ev W E2 E2')
           (TP2'                      : of W E2' T).
tpres_app3:
  tpres (of_app (of_lam [x][dx]TP1 x dx) TP2
                                      : of W (app (lam T [x]E x) V2) T')
        (ev_app3 VV2                  : ev W (app (lam T [x]E x) V2) (E V2))
        (TP1 V2 TP2                   : of W (E V2) T').
tpres_fix:
  tpres ((of_fix [x:exp][dx:of W x T]TP x dx)
                                      : of W (fix T [x]E x) T)
        (ev_fix                       : ev W (fix T [x]E x) (E (fix T [x]E x)))
        (TP (fix T [x]E x) (of_fix [x][dx]TP x dx)
                                      : of W (E (fix T [x]E x)) T).
tpres_resume1:
  tpres (of_resume TP                 : of W (resume E) T)
        (ev_resume1 EV                : ev W (resume E) (resume E'))
        (of_resume TP'                : of W (resume E') T)
  <- tpres (TP                        : of W E (susp T))
           (EV                        : ev W E E')
           (TP'                       : of W E' (susp T)).
tpres_resume2:
  tpres (of_resume (of_hold TP)       : of W (resume (hold E)) T)
        (ev_resume2                   : ev W (resume (hold E)) E)
        (TP                           : of W E T).
tpres_publish:
  tpres ((of_publish M [x:exp][dx:of W x T]TP x dx)
                                      : of W (publish T [x]E x) (srv T T'))
        (ev_publish                   : ev W (publish T [x]E x)
                                             (url W T [x]E x))
        ((of_url M [x:exp][dx:of W x T]TP x dx)
                                      : of W (url W T [x]E x) (srv T T')).
tpres_call1:
  tpres (of_call TP1 TP2              : of W (call E1 E2) T')
        (ev_call1 EV1                 : ev W (call E1 E2) (call E1' E2))
        (of_call TP1' TP2             : of W (call E1' E2) T')
  <- tpres (TP1                       : of W E1 (srv T T'))
           (EV1                       : ev W E1 E1')
           (TP1'                      : of W E1' (srv T T')).
```

```
tpres_call2:
  tpres (of_call TP1 TP2            : of W (call V1 E2) T')
        (ev_call2 VV1 EV2           : ev W (call V1 E2) (call V1 E2'))
        (of_call TP1 TP2'           : of W (call V1 E2') T')
  <- tpres (TP2                     : of W E2 T)
           (EV2                     : ev W E2 E2')
           (TP2'                    : of W E2' T).
tpres_call3:
  tpres (of_call (of_url M [x:exp][dx:of W' x T]TP1 x dx) TP2
                                    : of W (call (url W' T [x]E x) V2) T')
        (ev_call3 VV2               : ev W (call (url W' T [x]E x) V2)
                                           (expect (E V2) W'))
        (of_expect (TP1 V2 TP2')    : of W (expect (E V2) W') T')
  <- relocate TP2 W' TP2'.
```

A.5 Progress

The progress theorem in Section 2.5 depends uniquely on the canonical forms lemma. For a language as simple as QWeS^{2}T, this lemma too comes "for free" in Twelf: it is emulated through inversion and the coverage checker ensures us that no cases have been missed.

A.5.1 Not Stuck

As often when encoding a progress proof in Twelf [1], this proof defines an auxiliary property, `notStuck`, which states that its argument expression is either a value or can perform a step of computation. The proof of progress proper maps a well typed expression e to a derivation that shows that e is `notStuck`. This is a technicality to capture the disjunctive nature of a progress theorem.

```
notStuck: exp -> type.   %name notStuck NS.

ns-val: notStuck V
         <- val V.
ns-ev: notStuck E
         <- ev W E E'.
```

A.5.2 Progress Lemmas

Another technicality, this time to help Twelf machine-check our proof, is to factor constructs that correspond to more than one evaluation rule into their own mini progress lemmas. These are displayed next.

Progress Lemma for `pair`

```
pg-pair : notStuck E1
       -> notStuck E2
       -> notStuck (pair E1 E2)
```

```
      -> type.
%mode pg-pair +NS1 +NS2 -NS.

pg-pair1:
  pg-pair (ns-ev (EV1                : ev W E1 E1'))
          (NS2                       : notStuck E2)
          (ns-ev (ev_pair1 EV1       : ev W (pair E1 E2) (pair E1' E2))).
pg-pair2:
  pg-pair (ns-val (VV1               : val V1))
          (ns-ev (EV2                : ev W E2 E2'))
          (ns-ev (ev_pair2 VV1 EV2   : ev W (pair V1 E2) (pair V1 E2'))).
pg-pair3:
  pg-pair (ns-val (VV1               : val V1))
          (ns-val (VV2               : val V2))
          (ns-val (val_pair VV1 VV2  : val (pair V1 V2))).

%worlds () (pg-pair _ _ _).
%total  {} (pg-pair _ _ _).
```

Progress Lemma for `fst`

```
pg-fst : notStuck E
      -> {W:host} notStuck (fst E)
      -> type.
%mode pg-fst +NS +TP -NS'.

pg-fst1:
  pg-fst (ns-ev (EV                  : ev W E E'))
       W (ns-ev (ev_fst1 EV          : ev W (fst E) (fst E'))).
pg-fst2:
  pg-fst (ns-val (val_pair VV1 VV2   : val (pair V1 V2)))
      W  (ns-ev (ev_fst2 VV1 VV2     : ev W (fst (pair V1 V2)) V1)).

%worlds () (pg-fst _ _ _).
%total  {} (pg-fst _ _ _).
```

Progress Lemma for `snd`

```
pg-snd : notStuck E
      -> {W:host} notStuck (snd E)
      -> type.
%mode pg-snd +NS +TP -NS'.

pg-snd1:
  pg-snd (ns-ev (EV                  : ev W E E'))
       W (ns-ev (ev_snd1 EV          : ev W (snd E) (snd E'))).
pg-snd2:
  pg-snd (ns-val (val_pair VV1 VV2   : val (pair V1 V2)))
       W (ns-ev (ev_snd2 VV1 VV2     : ev W (snd (pair V1 V2)) V2)).

%worlds () (pg-snd _ _ _).
```

```
%total  {} (pg-snd _ _ _).
```

Progress Lemma for `app`

```
pg-app : notStuck E1
      -> notStuck E2
      -> {W:host} notStuck (app E1 E2)
      -> type.
%mode pg-app +NS1 +NS2 +TP -NS.

pg-app1:
  pg-app (ns-ev (EV1                : ev W E1 E1'))
         (NS2                       : notStuck E2)
       W (ns-ev (ev_app1 EV1        : ev W (app E1 E2) (app E1' E2))).
pg-app2:
  pg-app (ns-val (VV1               : val V1))
         (ns-ev (EV2                : ev W E2 E2'))
       W (ns-ev (ev_app2 VV1 EV2    : ev W (app V1 E2) (app V1 E2'))).
pg-app3:
  pg-app (ns-val (val_lam           : val (lam T [x]E x)))
         (ns-val (VV2               : val V2))
       W (ns-ev (ev_app3 VV2        : ev W (app (lam T [x]E x) V2) (E V2))).

%worlds () (pg-app _ _ _ _).
%total  {} (pg-app _ _ _ _).
```

Progress Lemma for `resume`

```
pg-resume : notStuck E
         -> {W:host} notStuck (resume E)
         -> type.
%mode pg-resume +NS +TP -NS'.

pg-resume1:
  pg-resume (ns-ev (EV              : ev W E E'))
          W (ns-ev (ev_resume1 EV   : ev W (resume E) (resume E'))).
pg-resume2:
  pg-resume (ns-val (val_hold       : val (hold E)))
          W (ns-ev (ev_resume2      : ev W (resume (hold E)) E)).

%worlds () (pg-resume _ _ _).
%total  {} (pg-resume _ _ _).
```

Progress Lemma for `call`

```
pg-call : notStuck E1
       -> notStuck E2
       -> {W:host} notStuck (call E1 E2)
       -> type.
%mode pg-call +NS1 +NS2 +TP -NS.
```

```
pg-call1:
  pg-call (ns-ev (EV1                : ev W E1 E1')
          (NS2                       : notStuck E2)
        W (ns-ev (ev_call1 EV1       : ev W (call E1 E2) (call E1' E2))).
pg-call2:
  pg-call (ns-val (VV1               : val V1))
          (ns-ev (EV2                : ev W E2 E2'))
        W (ns-ev (ev_call2 VV1 EV2   : ev W (call V1 E2) (call V1 E2'))).
pg-call3:
  pg-call (ns-val (val_url           : val (url W' T [x]E x)))
          (ns-val (VV2               : val V2))
        W (ns-ev (ev_call3 VV2       : ev W (call (url W' T [x]E x) V2)
                                            (expect (E V2) W'))).

%worlds () (pg-call _ _ _ _).
%total  {} (pg-call _ _ _ _).
```

Progress Lemma for `expect`

```
pg-expect : notStuck E
         -> {W:host}{W':host}notStuck (expect E W')
         -> type.
%mode pg-expect +NS +W +W' -NS'.

pg-expect1:
  pg-expect (ns-ev (EV               : ev W' E E'))
       W W' (ns-ev (ev_expect1 EV    : ev W (expect E W') (expect E' W'))).
pg-expect2:
  pg-expect (ns-val (VV              : val V))
       W W' (ns-ev (ev_expect2 VV    : ev W (expect V W') V)).

%worlds () (pg-expect _ _ _ _).
%total  {} (pg-expect _ _ _ _).
```

A.6 Progress Theorem

```
progress: of W E T -> notStuck E -> type.
%mode progress +TP -NS.

prg_triv:
  progress (of_triv                  : of W triv unit)
           (ns-val (val_unit         : val triv)).
prg_pair:
  progress (of_pair TP1 TP2          : of W (pair E1 E2) (prod T1 T2))
           (NS                       : notStuck (pair E1 E2))
  <- progress (TP1                   : of W E1 T1)
              (NS1                   : notStuck E1)
  <- progress (TP2                   : of W E2 T2)
              (NS2                   : notStuck E2)
  <- pg-pair NS1 NS2 NS.
```

```
prg_fst:
  progress (of_fst TP                : of W (fst E) T1)
           (NS'                      : notStuck (fst E))
  <- progress (TP                    : of W E (prod T1 T2))
              (NS                    : notStuck E)
  <- pg-fst NS W NS'.
prg_snd:
  progress (of_snd TP                : of W (snd E) T2)
           (NS'                      : notStuck (snd E))
  <- progress (TP                    : of W E (prod T1 T2))
              (NS                    : notStuck E)
  <- pg-snd NS W NS'.
prg_lam:
  progress ((of_lam [x][dx]TP x dx)  : of W (lam T [x]E x) (arrow T T'))
           (ns-val (val_lam          : val (lam T [x]E x))).
prg_app:
  progress (of_app TP1 TP2           : of W (app E1 E2) T')
           (NS                       : notStuck (app E1 E2))
  <- progress (TP1                   : of W E1 (arrow T T'))
              (NS1                   : notStuck E1)
  <- progress (TP2                   : of W E2 T)
              (NS2                   : notStuck E2)
  <- pg-app NS1 NS2 W NS.
prg_fix:
  progress ((of_fix [x][dx]TP x dx)  : of W (fix T [x]E x) T)
           (ns-ev (ev_fix            : ev W (fix T [x]E x) (E (fix T [x]E x)))).
prg_hold:
  progress (of_hold TP               : of W (hold E) (susp T))
           (ns-val (val_hold         : val (hold E))).
prg_resume:
  progress (of_resume TP             : of W (resume E) T)
           (NS'                      : notStuck (resume E))
  <- progress (TP                    : of W E (susp T))
              (NS                    : notStuck E)
  <- pg-resume NS W NS'.
prg_url:
  progress ((of_url M [x][dx]TP x dx)
                                     : of W (url W' T [x]E x) (srv T T'))
           (ns-val (val_url          : val (url W' T E))).
prg_publish:
  progress ((of_publish M [x][dx]TP x dx)
                                     : of W (publish T [x]E x) (srv T T'))
           (ns-ev (ev_publish        : ev W (publish T [x]E x)
                                            (url W T [x]E x))).
prg_call:
  progress (of_call TP1 TP2          : of W (call E1 E2) T')
           (NS                       : notStuck (call E1 E2))
  <- progress (TP1                   : of W E1 (srv T T'))
              (NS1                   : notStuck E1)
  <- progress (TP2                   : of W E2 T)
              (NS2                   : notStuck E2)
```

```
  <- pg-call NS1 NS2 W NS.
prg_expect:
  progress (of_expect TP            : of W (expect E W') T)
           (NS'                     : notStuck (expect E W'))
  <- progress (TP                   : of W' E T)
              (NS                   : notStuck E)
  <- pg-expect NS W W' NS'.

%worlds () (progress _ _).
%total (TP) (progress TP _).
```