Fundamenta Informaticae 130 (2014) 1–31
DOI 10.3233/FI-2012-0000
IOS Press

Substructural Meta-Theory of a Type-Safe Language for Web Programming

Iliano Cervesato[*]
Carnegie Mellon University
iliano@cmu.edu

Thierry Sans[†]
Carnegie Mellon University
tsans@qatar.cmu.edu

Abstract. This paper introduces an abstract web programming language, QWeSST, and a methodology for proving properties of formalisms, such are QWeSST, that are parallel, distributed and concurrent. At its core, QWeSST is a small functional programming language extended with primitives for mobile code and remote procedure calls, two distinguishing features of web programming. It supports a localized view of typechecking and of evaluation, which reflects the way we program web applications and web services. We have developed a prototype implementation for QWeSST and used it to elegantly write simple web applications that are however not easily expressed using current web technology. We give two semantics for QWeSST, one is standard and models a naive form of single-threaded evaluation, the other is maximally parallel and exploits a presentation of its typing and execution behaviors based on an extended form of substructural operational semantics. It augments standard inference rules with a construction that realizes parametric multiset comprehension, which makes it convenient to capture ensemble-level behaviors. We prove that both semantics are type safe, the former using traditional methods, the latter by developing a proof methodology that parallels the multiset-oriented presentation of the semantics.

Keywords: Web Programming, Type Safety, Parallelism, Mobile Code, Substructural Operational Semantics.

1. Introduction

Web-based applications (*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

[*]This work partially supported by the Qatar National Research Fund under NPRP grant 09-1107-1-168.
[†]This work partially supported by the Qatar National Research Fund under YSREP grant 1-033-1-006.
The statements made herein are solely the responsibility of the authors.

Docs to browserless server-to-server SOAP-based web services. They make use of some characteristic mechanisms: at a minimum, *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. More complex webapps are often stateful, e.g., through the use of databases or cookies, and have significant security requirements—two characteristic aspects this paper will not focus on. Communication happens over the HTTP protocol. Webapps are very popular for two main reasons. From the user's perspective, they are easy to deploy: there is no need to install a third party program 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[1] as building blocks for their webapps (obtaining what is called a mashup). However, as the application grows, web programmers know that development gets more complex and bugs more elusive. Indeed, it is very hard to ensure correctness (and security) when developing and maintaining large scale webapps.

Two factors contribute to this complexity: first, building a webapp requires reasoning about distributed computation, which is intrinsically hard. The second factor, which this paper is about, is that typical webapp development orchestrates a multitude of heterogeneous languages: the client-side code is often written in HTML and JavaScript which are implemented by all browsers, and the server side can be written in any language, common choices being PHP, Java, ASP/.NET, Ruby on Rails and Python. Getting communication patterns right becomes hard as nothing short of extensive testing will catch an incorrect service name or a mismatched argument. Types do not carry across languages, and JavaScript is untyped anyways. Current best practices to providing some static assurance against these issues include *i)* using APIs downloaded locally (but APIs may change unexpectedly on the web), *ii)* using language extensions such as ServiceJ [8] that verify the correctness of service orchestration, *iii)* using standards such as WSDL [33] for declaring the type interface of a service, and *iv)* writing webapps in a single strongly-typed language that gets compiled to the various idioms of the web as done in Links [6] or with the Google Web Toolkit (GWT) [13]. While helpful, none of these solutions covers the rich spectrum of modern webapps: for example the "distributed computing through centralized programming" model of GWT and Links does not encompass mashups and its compilation process does not support building web services dynamically—yet it has proved adequate for large client-server applications such as Google Docs. Furthermore, these technologies tend to patch preexisting languages to enable web programming, which helps reasoning about distributed computation in webapps only up to a certain point.

In this paper, we propose a language designed around the distributed nature of web applications and a methodology for proving properties about it. Specifically, we present an abstraction of web development that revolves around two of its distinguishing features, remote code execution and mobile code—we plan to extend it to stateful computation and security in future papers. We realize this abstraction into a programming language concept designed with web applications in mind. This language, that we call QWeSST [26, 27], 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. QWeSST is strongly typed and supports

[1]By service, or *web service*, we mean a function that resides on a remote server and that a client can invoke by sending a message carrying the arguments to be passed to this function; the function is executed on the server and only the result is returned to the client. This 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.

decentralized type-checking. We show that it is type-safe and we use it to implement, in just 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 our language [21]. QWeSST shares features with Lambda 5 [16, 17], an abstract programming language for distributed computing based on modal logic, of which it can be seen as simplification specialized to web programming.

We provide two operational semantics for QWeSST and two proofs of safety. The first is standard. It serves the sole purpose of describing the language and its characteristics but is otherwise unrealistic as exactly one node in the network is executing at any time. The second is novel. It directly expresses parallel execution and supports multiple threads at any node. It formalizes the semantics of parallel languages and proves some of their properties by leveraging parametric comprehension, as when writing a set E of expressions as $\{e_i\}$. Specifically, we allow multisets in judgments (e.g., $E \mapsto E'$ may express the fact that the expressions in the multiset E make one step of evaluation resulting in the multiset E') and describe them using parametric comprehension (here $\lbag e_i \rbag \mapsto \lbag e'_i \rbag$ where $\lbag \ldots \rbag$ delimits a multiset in the same way as $\{\ldots\}$ delimits a set). More importantly, we extend the traditional format of inference rules to support parametrically-defined multisets of judgments as their premises (that would be $\lbag e_i \mapsto e'_i \rbag$ in this example). These multiset-oriented rules extend the traditional induction principle over derivations by allowing generic proofs of the judgments in their premises to contribute to a proof of the judgment in their conclusion.

As exemplified in the case of QWeSST, the resulting specification of both the typing and execution semantics decomposes into a local phase that uses only traditional rules, and a global phase that makes use of our multiset-oriented rules to express parallelisms in a natural way. The local phase is closely related to a specification methodology known as substructural operational semantics (SSOS) [18, 4]. SSOS has been shown to scale effortlessly to complex programming features such as futures, laziness, concurrency and parallelism. The proof of type safety exploits this presentation as it decomposes the reasoning process into a parallel phase, which concentrates all statements and proofs about the parallel features of the language, and an atomic phase whose statements and proofs refer to elementary actions. Thanks to this modular structure, the safety proof for the parallel semantics is as concise as for the sequential semantics. In particular, it abstracts the need to serialize execution steps, to permute derivations, to manage non-deterministic interleavings, etc. This enables the proofs to focus on the object language (here QWeSST) rather than be diluted by the many bookkeeping lemmas. This substructural approach to the meta-theory scales with language complexity just as SSOS with the complexity of its specification.

The main contributions of this work are:

1. The design of a simple language that succinctly and naturally captures two major constructs found in web programming while being type-safe and supporting a localized form of typechecking. QWeSST is meant to be a clean abstraction to understand the most basic aspects of web programming, a baseline for exploring more complex issues such as state and security. In this regards, our prototype is both a proof of concept and a laboratory for future explorations.
2. The development of a form of substructural operational semantics specification and meta-theory that appears to capture quite naturally forms of parallelism as found in QWeSST. By leveraging our extended rule format, it enables the proofs to be as concise as in the sequential case.

This paper is structured as follows. We introduce QWeSST and a traditional single-threaded semantics for it in Section 2. In Section 3, we develop a parallel semantics in the style of SSOS and a matching

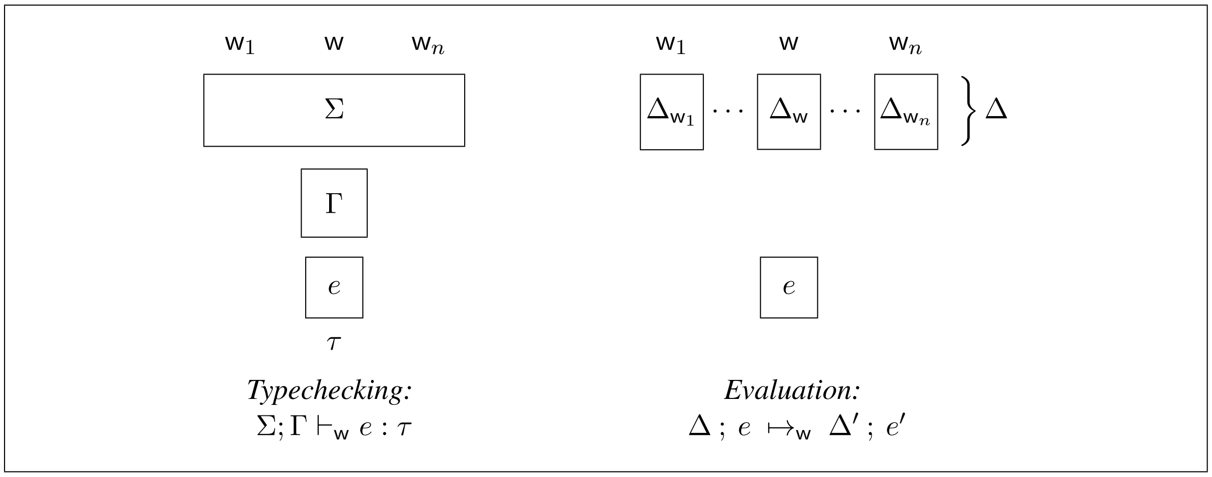
w_1
w
w_n
Σ
Γ
e
τ
Typechecking:
$\Sigma;\Gamma \vdash_{\mathsf{w}} e : \tau$
$\Delta_{\mathsf{w}_1} \cdots \Delta_{\mathsf{w}} \cdots \Delta_{\mathsf{w}_n}$
Δ
Evaluation:
$\Delta \,;\, e \mapsto_{\mathsf{w}} \Delta' \,;\, e'$

Figure 1. Views from World w

approach to proving type safety. Section 4 outlines our prototype implementation. We discuss related work from the literature in Section 5 and sketch our plans for future work in Section 6.

2. QWeSST

One of the goals of this paper is to propose a type-safe programming language for web development. This section describes this language, QWeSST [26, 27], uses it on some examples, gives a traditional semantics for it and establishes that this semantics is type safe, also in a traditional way. Section 3 will endow it with a more realistic semantics and develop a more interesting meta-theory.

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 (but we use these terms informally 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 rely on the machinery of traditional process algebras such as the π-calculus [25]. Our model is not concerned either with the topology of the network: 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.

The design of QWeSST 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 Δ_{w_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.

QWeSST is globally type-safe but supports 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—the web service community includes this property in the notion of conformance testing [1]. "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 type of the remote services it uses. This is illustrated on the left-hand side of Figure 1: the API of the services (of interest to w) on the Web is represented as Σ, while Γ and τ are the (local) typing context and type of the expression e. This localized static form of typechecking will be captured by the typing judgment $\Sigma; \Gamma \vdash_{\mathsf{w}} e : \tau$. The idea of localization, both for typing and evaluation, is inspired by *Lambda 5* [17, 16].

2.1. Syntax

QWeSST is a simple functional programming language extended with primitives to model two distinguishing characteristics of web programming: remote code (web services) and mobile code (e.g., JavaScript).

We model a web service as an expression of type $\tau \rightsquigarrow \tau'$: it represents a remote function that accepts arguments of type τ and returns results of type τ'. A server w' creates a service by evaluating the expression $\mathsf{publish}\ x : \tau.\, e$ which makes available the function e that takes an argument x of type τ and returns a result of type τ'. This produces a URL $\mathsf{url}(\mathsf{w}', u)$, where u is a unique identifier for this service. Once created, a client w can invoke this web service by calling its URL 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 v_2 of the argument e_2 to w', which applies e to v_2 and moves the result back to the client w.

Mobile code is code provided by one host but meant to be downloaded and executed at another host. We model 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'. QWeSST relies on the above call construct to do the moving of the suspended computations across the network.

The syntax of QWeSST is given by the following grammar. We include functions, fixed points and pairs to make the examples in Section 2.4 more interesting.

$$\begin{array}{llcl}
\text{Types} & \tau & ::= & \tau \to \tau' \mid \tau \times \tau' \mid \mathsf{unit} \\
 & & \mid & \mathsf{susp}[\tau] \mid \tau \rightsquigarrow \tau' \\
\text{Expressions} & e & ::= & x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid \mathsf{fix}\, x{:}\tau.e \mid \langle e_1, e_2\rangle \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e \mid () \\
 & & \mid & \mathsf{hold}\ e \mid \mathsf{resume}\ e \\
 & & \mid & \mathsf{url}(\mathsf{w}, u) \mid \mathsf{publish}\ x : \tau.\, e \mid \mathsf{call}\ e_1\ \mathsf{with}\ e_2 \mid \textit{expect}\ e\ \textit{from}\ \mathsf{w}
\end{array}$$

Here, x ranges over variables and u over service identifiers. As usual, we identify terms that differ only by the name of their bound variables and write $[e/x]e'$ for the capture-avoiding substitution of e for x in the expression e'. Service identifiers are never substituted.

The expression *expect* e *from* w is used internally during evaluation and is not available to the programmer. It essentially models the client's waiting for the result of a web service call.

$$\frac{}{\Sigma; \Gamma, x : \tau \vdash_{\mathsf{w}} x : \tau}\ \texttt{of\_var} \qquad \frac{}{\Sigma, u : \tau \rightsquigarrow \tau' \mathbin{@} \mathsf{w}'; \Gamma \vdash_{\mathsf{w}} \mathsf{url}(\mathsf{w}', u) : \tau \rightsquigarrow \tau'}\ \texttt{of\_url}$$

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

$$\frac{\Sigma; \Gamma, x : \tau \vdash_{\mathsf{w}} e : \tau}{\Sigma; \Gamma \vdash_{\mathsf{w}} \mathsf{fix}\ x{:}\tau.e : \tau}\ \texttt{of\_fix} \qquad \frac{}{\Sigma; \Gamma \vdash_{\mathsf{w}} () : \mathsf{unit}}\ \texttt{of\_unit}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{w}} e_1 : \tau \quad \Sigma; \Gamma \vdash_{\mathsf{w}} e_2 : \tau'}{\Sigma; \Gamma \vdash_{\mathsf{w}} \langle e_1, e_2 \rangle : \tau \times \tau'}\ \texttt{of\_pair} \qquad \frac{\Sigma; \Gamma \vdash_{\mathsf{w}} e : \tau \times \tau'}{\Sigma; \Gamma \vdash_{\mathsf{w}} \mathsf{fst}\ e : \tau}\ \texttt{of\_fst} \qquad \frac{\Sigma; \Gamma \vdash_{\mathsf{w}} e : \tau \times \tau'}{\Sigma; \Gamma \vdash_{\mathsf{w}} \mathsf{snd}\ e : \tau'}\ \texttt{of\_snd}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{w}} e : \tau}{\Sigma; \Gamma \vdash_{\mathsf{w}} \mathsf{hold}\ e : \mathsf{susp}[\tau]}\ \texttt{of\_hold} \qquad \frac{\Sigma; \Gamma \vdash_{\mathsf{w}} e : \mathsf{susp}[\tau]}{\Sigma; \Gamma \vdash_{\mathsf{w}} \mathsf{resume}\ e : \tau}\ \texttt{of\_resume}$$

$$\frac{\Sigma; \Gamma, x : \tau \vdash_{\mathsf{w}} e : \tau'}{\Sigma; \Gamma \vdash_{\mathsf{w}} \mathsf{publish}\ x : \tau.\, e : \tau \rightsquigarrow \tau'}\ \texttt{of\_publish}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{w}} e_1 : \tau \rightsquigarrow \tau' \qquad \Sigma; \Gamma \vdash_{\mathsf{w}} e_2 : \tau}{\Sigma; \Gamma \vdash_{\mathsf{w}} \mathsf{call}\ e_1\ \mathsf{with}\ e_2 : \tau'}\ \texttt{of\_call} \qquad \frac{\Sigma; \Gamma \vdash_{\mathsf{w}'} e : \tau}{\Sigma; \Gamma \vdash_{\mathsf{w}} \mathit{expect}\ e\ \mathit{from}\ \mathsf{w}' : \tau}\ \texttt{of\_expect}$$

$$\frac{}{\cdot \vdash \cdot}\ \texttt{st\_\cdot} \qquad \frac{\Sigma \vdash \Delta \quad \Sigma; x : \tau \vdash_{\mathsf{w}} e : \tau'}{\Sigma, u : \tau \rightsquigarrow \tau' \mathbin{@} \mathsf{w} \vdash \Delta, u \mathbin{@} \mathsf{w} \hookrightarrow x{:}\tau.e}\ \texttt{st\_u}$$

Figure 2. Typing Rules for QWeSST

2.2. Typing

The typing semantics of QWeSST makes use of two contexts:

Local typing context	Γ	$::=$	$\cdot \mid \Gamma, x : \tau$
Service typing table	Σ	$::=$	$\cdot \mid \Sigma, u : \tau \rightsquigarrow \tau \mathbin{@} \mathsf{w}$

The (local) typing context Γ and the service typing table Σ keeps track of the type of free variables and service identifiers, respectively. Contexts are treated as multisets and we require variables and identifiers to be declared at most once in them—our rules will rely on implicit α-renaming to ensure this.

The typing semantics of QWeSST relies on the judgment:

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

As mentioned earlier, the typing judgment $\Sigma; \Gamma \vdash_{\mathsf{w}} e : \tau$ is located at each world w, which is where the expression e resides. Because e is local to w, so are the variables in Γ, meaning that they are implicitly typed at w. Instead, URLs used in e may refer to other hosts, which explains why entries in Σ mention worlds. The typing rules for this judgment are displayed in the top segment of Figure 2. The rules for functions, fixed point and pairs are standard, and the remaining ones are straightforward. Observe that `of_expect` is the only typing rule that involves a change of world. Rule `of_url` abstractly checks the type of a URL by looking it up in the global service table Σ, which by definition is not local. In practice,

$$\frac{}{()\ \mathsf{val}}\ \texttt{val\_unit} \qquad \frac{}{\lambda x : \tau.\, e\ \mathsf{val}}\ \texttt{val\_lam} \qquad \frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\langle e_1, e_2\rangle\ \mathsf{val}}\ \texttt{val\_pair}$$

$$\frac{}{\mathsf{hold}\ e\ \mathsf{val}}\ \texttt{val\_hold} \qquad \frac{}{\mathsf{url}(\mathsf{w}', u)\ \mathsf{val}}\ \texttt{val\_url}$$

$$\frac{\Delta\ ;\ e_1 \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_1'}{\Delta\ ;\ e_1\ e_2 \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_1'\ e_2}\ \texttt{ev\_app}_1 \qquad \frac{v_1\ \mathsf{val} \quad \Delta\ ;\ e_2 \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_2'}{\Delta\ ;\ v_1\ e_2 \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ v_1\ e_2'}\ \texttt{ev\_app}_2$$

$$\frac{v_2\ \mathsf{val}}{\Delta\ ;\ (\lambda x : \tau.\, e)\ v_2 \ \mapsto_{\mathsf{w}}\ \Delta\ ;\ [v_2/x]\ e}\ \texttt{ev\_app}_3 \qquad \frac{}{\Delta\ ;\ \mathsf{fix}\ x : \tau.e \ \mapsto_{\mathsf{w}}\ \Delta\ ;\ [\mathsf{fix}\ x : \tau.e/x]\ e}\ \texttt{ev\_fix}$$

$$\frac{\Delta\ ;\ e_1 \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_1'}{\Delta\ ;\ \langle e_1, e_2\rangle \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \langle e_1', e_2\rangle}\ \texttt{ev\_pair}_1 \qquad \frac{v_1\ \mathsf{val} \quad \Delta\ ;\ e_2 \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_2'}{\Delta\ ;\ \langle v_1, e_2\rangle \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \langle v_1, e_2'\rangle}\ \texttt{ev\_pair}_2$$

$$\frac{\Delta\ ;\ e \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e'}{\Delta\ ;\ \mathsf{fst}\ e \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{fst}\ e'}\ \texttt{ev\_fst}_1 \qquad \frac{v_1\ \mathsf{val} \quad v_2\ \mathsf{val}}{\Delta\ ;\ \mathsf{fst}\ \langle v_1, v_2\rangle \ \mapsto_{\mathsf{w}}\ \Delta\ ;\ v_1}\ \texttt{ev\_fst}_2$$

$$\frac{\Delta\ ;\ e \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e'}{\Delta\ ;\ \mathsf{snd}\ e \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{snd}\ e'}\ \texttt{ev\_snd}_1 \qquad \frac{v_1\ \mathsf{val} \quad v_2\ \mathsf{val}}{\Delta\ ;\ \mathsf{snd}\ \langle v_1, v_2\rangle \ \mapsto_{\mathsf{w}}\ \Delta\ ;\ v_2}\ \texttt{ev\_snd}_2$$

$$\frac{\Delta\ ;\ e \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e'}{\Delta\ ;\ \mathsf{resume}\ e \ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{resume}\ e'}\ \texttt{ev\_resume}_1 \qquad \frac{}{\Delta\ ;\ \mathsf{resume}\ (\mathsf{hold}\ e) \ \mapsto_{\mathsf{w}}\ \Delta\ ;\ e}\ \texttt{ev\_resume}_2$$

$$\frac{}{\Delta\ ;\ \mathsf{publish}\ x : \tau.\ e \ \mapsto_{\mathsf{w}}\ (\Delta, u \mathbin{@} \mathsf{w} \hookrightarrow x : \tau.e)\ ;\ \mathsf{url}(\mathsf{w}, u)}\ \texttt{ev\_publish}$$

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

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

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

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

Figure 3. Evaluation Rules for QWeSST

this can be realized in a number of ways: by downloading an API of a web service library at development time, thereby locally caching the typing specifications of the services of interest; by asking the remote server for the type of each service the local code uses at compile time (this is the approach implemented in our prototype and it is similar to typechecking a web service according to its WSDL file); by relying on a third-party service typing server.

2.3. Evaluation

In this section, we model evaluation in QWeSST using a small-step semantics that transforms states consisting of a service repository Δ and an expression e being evaluated at a world. Repositories are multisets defined as follows:

$$\text{Global service repository} \qquad \Delta \ ::= \ \cdot \ \mid \ \Delta, u \mathbin{@} \mathsf{w} \hookrightarrow x : \tau.e$$

Each item $u \mathbin{@} \mathsf{w} \hookrightarrow x : \tau.e$ is a service with URL $\mathsf{url}(u, \mathsf{w})$, formal argument x and body e. We will occasionally omit w when unimportant or easily reconstructible from the context. The repository Δ is global as it lists every service in the network. We write Δ_{w} for the submultiset of all services in Δ that are located at world w: Δ_{w} is the *local service repository* of world w and is meant to indicate all services that w makes available.

The evaluation semantics of QWeSST is expressed by the following judgments:

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

Notice again that the step judgment is localized at a world w. The rules for both judgments are given in Figure 3. Most are unsurprising. The evaluation of $\mathsf{publish}\ x : \tau.\, e$ immediately publishes its argument as a web service in its local repository, creating a new unique identifier for it and returning the corresponding URL. This URL can later be communicated to a client. 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 $\texttt{ev\_expect}_1$ and $\texttt{ev\_expect}_2$: 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.

It is important to observe that the rules in Figure 3 are single-threaded: exactly one computation at exactly one world is actively executing at any time. In particular, while a remote call is serviced, the current node is idle executing rule $\texttt{ev\_expect}_2$. Although the Web does a good job at feeding this illusion, this is not realistic: millions of computations are taking place in parallel on the Web. We will provide a more realistic semantics in Section 3, but for now, let us prove that QWeSST is type-safe with respect to the given semantics.

In order to do so, we need to define a judgment that describes what it means for a service repository Δ to be well-typed with respect to a typing table Σ.

$$\Sigma \vdash \Delta \qquad \text{“}\textit{the services in}\ \Delta\ \textit{are well-typed in}\ \Sigma\text{”}$$

The rules for this judgment are at the bottom of Figure 2.

2.4. Examples

Having presented QWeSST, we now give a few web programming examples written in it some simple, others non-trivial for current web technology. Many more examples can be found in the companion technical report [27]. For readability, we extend QWeSST with a let construct, where “$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$”

is understood as $(\lambda x : \tau.\, e_2)\ e_1$. Furthermore, we make the examples below more visually appealing by using types such as info, query, etc, which can be taken as synonyms for unit to stay within QWeSST. We also write host names and URLs as on the Web, e.g., www.server.com/u for some service u at node www.server.com. We display our example using figures such as the following:

www.server.com/$\mathit{service}$ $\curvearrowright$

$(\mathit{aux.\ URLs})$

www.server.com
publish x : . . . . *server code* . . .

client
let url = url(www.server.com, $\mathit{service}$)
. . . *client code* . . .
in call url with . . .

Each host is represented by a box, with the host name in the top-right corner and the rest of the box occupied by the QWeSST code executed by this host. The box on the left will typically stand for a server that publishes a service using publish that a client represented by the box on the right will call using call. The server code is executed first and results in the publication of a service. We assume that the name of this service is communicated to the client out-of-band, symbolized by the curly arrow labeled by the service name—for convenience, we pick mnemonic names. The server and sometimes the client may make use of auxiliary URLs shown in parentheses.

We begin with a basic 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, yielding 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 info $\times$ query $\rightsquigarrow$ result. A client can then use this service by calling this URL on arguments of interest, here $\langle \mathit{myInfo}, \mathit{myQuery} \rangle$:

$\curvearrowright$ www.server.com/search

www.server.com
let $f = \lambda x : \text{info} \times \text{query}.\ (\mathit{search}\ x)$
in publish z : info $\times$ query. $f\ z$

client
let url = url(www.server.com, search)
in call url with $\langle \mathit{myInfo}, \mathit{myQuery} \rangle$

To facilitate the usage of this service, we now 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 return the result. This script, published on the server at URL www.server.com/api, has type unit $\rightsquigarrow$ susp[query $\times$ info $\to$ 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 know the underlying search URL, www.server.com/search, anymore.

(www.server.com/search) $\curvearrowright$ www.server.com/api

www.server.com
let url = url(www.server.com, search)
script = hold $\lambda z : \text{query} \times \text{info}.$ (call url with z)
in publish x : unit. script

client
let url = url(www.server.com, api)
f = resume (call url with ())
in $f\ \langle \mathit{myInfo}, \mathit{myQuery} \rangle$

If we assume that the client's information myInfo does not change from one query to another, then the server could customize the script for each client based on this information. In the code below, the updated script at www.server.com/api is called with just 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 info $\rightsquigarrow$ susp[query $\rightarrow$ result], which can be viewed as a curried version of our previous example (modulo the intervening suspension).

(www.server.com/*search*)

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

↷ www.server.com/*api*

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

Going one step further, rather than returning a customized API, the server may 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. The server will return the URL of this personalized service to the client.

(www.server.com/*search*)

```
                                             www.server.com
let url = url(www.server.com, search)
    f = λz : info × query. (call url with z)
in publish x : info. publish y : query. f ⟨x, y⟩
```

↷ www.server.com/*customSearch*
www.server.com/*xyz*

```
                                             client
let url = url(www.server.com, customSearch)
    mySearchUrl = call url with myInfo
in call mySearchUrl with myQuery
```

In this example, the search server www.server.com 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 some new URL (written here www.server.com/*xyz*) that the client accesses through the variable *mySearchURL*. The client can then use this personalized URL directly with its queries.

Next, assume the client has its own web server www.client.com, and wants to provide its customers with a web service on it that makes use of functionalities supplied by www.server.com. When the client's 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 would be 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 any of our previous one-click examples.

(www.server.com/*search*)

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

↷ www.server.com/*installer*

```
                                             www.client.com
let url = url(www.server.com, installer)
    script = call url with clientInfo
in resume script
```

↶ www.client.com/*clientSearch*

```
                                             customer
call url(www.client.com, clientSearch) with myQuery
```

QWeSST can do much better. In the code snippet shown, the server provides a script, www.server.com /*installer*, that, when invoked by the client with input *clientInfo*, will automatically create and publish the customized web service www.client.com/$clientSearch$ at www.client.com. It is this URL that $customer$ will use for its searches.

We next consider a group of examples that go beyond what is customarily done in traditional web programming, bordering aspects of cloud computing. They 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 QWeSST that provides types for integers (int) and Booleans, together with some standard operations—all are supported by the current QWeSST prototype.

Given a multiplication service www.server.com/$mult$ at www.server.com, our client code implements the factorial function in a distributed way: rather than performing the multiplications locally during the recursive calls, it offloads them to this server.

www.server.com/$mult$ $\curvearrowright$

```
                                      www.server.com
publish $x$ : int × int. fst $x * $ snd $x$
```

```
                                                              client
let $times$ = url(www.server.com, $mult$)
    $fact$ = fix $f$ : int → int.$\lambda x$ : int.
      if $x = 0$ then 1 else call $times$ with $\langle x, f\ (n-1)\rangle$
in $fact$ 4
```

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. In particular, the server may update its implementation at any time, transparently to the client.

Our final 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 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/$eval$ $\curvearrowright$

```
                                           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.$\lambda x$ : int.
      if $x = 0$ then 1 else $x * f\ (n-1)$
in call $ev$ with $\langle$hold $fact, 4\rangle$
```

In a sense, this code operates in a way opposite to the 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.

2.5. Type Safety

We conclude this section with a routine study of the meta-theory of QWeSST, 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 [31, 19] to encode each of our proofs and to verify their correctness. All these proofs can be found in the companion technical report [27]. All went through except for the proof of the relocation lemma 2.2, 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 URLs are never substituted in the semantics of QWeSST, and therefore do not require a separate substitution statement.

Lemma 2.1. (Substitution)
If $\Sigma; \Gamma \vdash_{\mathsf{w}} e : \tau$ and $\Sigma; \Gamma, x : \tau \vdash_{\mathsf{w}} e' : \tau'$, then $\Sigma; \Gamma \vdash_{\mathsf{w}} [e/x]\, e' : \tau'$.

Proof: By induction on the derivation of the second judgment. This result 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 the 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 [16, 17].

Lemma 2.2. (Relocation)
If $\Sigma; \Gamma \vdash_{\mathsf{w}} e : \tau$, then $\Sigma; \Gamma \vdash_{\mathsf{w}'} e : \tau$ for any world w'.

Proof: By induction on the derivation of the judgment $\Sigma; \Gamma \vdash_{\mathsf{w}} e : \tau$. A Twelf representation of this proof can be found in [27] 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 2.3. (Type preservation)
If $\Delta\ ;\ e \mapsto_{\mathsf{w}} \Delta'\ ;\ e'$ and $\Sigma; \cdot \vdash_{\mathsf{w}} e : \tau$ and $\Sigma \vdash \Delta$, then $\Sigma'; \cdot \vdash_{\mathsf{w}} e' : \tau$ and $\Sigma' \vdash \Delta'$.

Proof: The proof proceeds by induction on the derivation of $\Delta\ ;\ e \mapsto_{\mathsf{w}} \Delta\ ;\ e'$. It uses the substitution lemma 2.1 in the cases of rules $\mathtt{ev\_app}_3$, `ev_fix` and $\mathtt{ev\_call}_3$. It also uses the relocation lemma 2.2 to handle rules $\mathtt{ev\_call}_3$ and $\mathtt{ev\_expect}_1$. □

The proof of progress relies on the following canonical form lemma, whose statement and proof are standard.

Lemma 2.4. (Canonical Form)
If e val, then

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

Theorem 2.5. (Progress)
If $\Sigma; \cdot \vdash_{\mathsf{w}} e : \tau$ and $\Sigma \vdash \Delta$, then

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

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

3. Parallel Semantics

The dynamic semantics of QWeSST given in Section 2 is distributed but single-threaded: computation happens at exactly one host at any time. In this section, we will define a parallel semantics for this language that is closer to reality. We will actually go a step beyond and evaluate the subexpressions of binary constructs in parallel—although not strictly necessary (and possibly unwanted in practice), it will act as a stress test for the machinery we will be developing—see [27] for intermediate solutions.

As an a priori unknown number of threads may now be executing at the same time, it will be convenient to consider inference rules with a parametric number of premises. Specifically, threads and related entities will be modeled as multisets, and the rules that describe their behavior will have a corresponding multiset of premises. A similar idea, although not phrased in these terms, underlies the notion of substructural operational semantics (SSOS) [18, 4, 30, 20]. SSOS has traditionally targeted the dynamic semantics of a language: it views the state of the evaluation as a context in some substructural logic (linear logic in [18, 4] and ordered logic in [30, 20]) and describes state transitions as logical formulas that rewrite a portion of state. This idea has recently been extended to the static semantics of a language [30] and is related to the (also recent) use of rewriting logic for type inference [10, 11]. Although SSOS requires more infrastructure than a standard small-step semantics, it has been shown to scale better with the complexity of the object language.

We will use multiset-oriented rules to specify both the static and the dynamic parallel semantics of QWeSST and then leverage them to re-prove the type safety of our language. While the result itself is unsurprising, the associated proofs have about the same complexity as the single-threaded proofs in Section 2 and are much simpler than what would have resulted from a more standard specification of parallel evaluation. We call this style of proof *substructural meta-theory*.

3.1. Multiset-Oriented Inference Rules

In Section 2, we relied on various structures that we used as multisets, typically to reassociate and/or reorder elements as convenient: the typing context Γ, the service typing table Σ and the service repository Δ. Exploiting the monoidal structure of multisets when convenient is a common practice when studying the semantics of programming languages and other deductive systems.

In this section, we will make a much more central use of multisets. Specifically, we will leverage the practice of describing such collections using parametric comprehension—as when writing a context Γ as $\{x_i : \tau_i\}$ for example. We augment the traditional format of inference rules to take advantage of it and exemplify a proof methodology based on this extended format.

We write "$\cdot$" for the empty multiset. Given a multiset M and an element m, we write "M, m" for the multiset that extends M with m, and promote this notation to denote multiset union, so that "M_1, M_2" is the union of M_1 and M_2. In the following, we will be interested in multisets with certain properties inherited from properties of their parts or constituents. As a simple motivational example, consider a setup consisting of expressions, e, some of which are values, classified by the judgment e val. Now, given a multiset E of expressions, we want to define a judgment E vals that holds exactly when all the expressions in E are values. The standard way to do so is by means of inference rules such as the following:

$$\frac{}{\cdot\ \mathsf{vals}} \qquad\qquad \frac{E\ \mathsf{vals} \quad e\ \mathsf{val}}{E, e\ \mathsf{vals}}$$

Now, the property E vals is uniform over the multiset E: it is meant to hold if e val is derivable for each element e of E. We will express this idea more directly by relying on parametric comprehension to describe E and manipulate its elements.

Parametric comprehension allows us to describe a multiset M as $\wr m_i \int$ with $i \in I$ for some index set I: if $I = \{i_1, \ldots, i_n\}$ then $m_{i_1}, \ldots, m_{i_n}$ are exactly the elements of M (including any possible repetition since we are working with multisets). Intuitively, we will have two uses for this notation:

1. Given a multiset M, visualizing it as $\wr m_i \int$ with $i \in I$ for some index set I will allow us to express properties that hold uniformly over each element of M by referring to a generic element m_i.
2. Given elements generically described as m_i, we can form a multiset $\wr m_i \int$.

For us, the index set I will always be finite, although its size will generally be unknown. When unambiguous, we will omit it as well as the attribution "$i \in I$". We will keep indices abstract. Finally, just like we promoted "," to denote multiset union, we will occasionally write $M = \wr M_i \int$ to express that M is the union of multisets M_i.

Once we write E as $\wr e_i \int$ for $i \in I$ in our example, it is natural to want to define E vals as "e_i val is derivable for each $i \in I$". This suggests extending the traditional format of inference rules to include premises defined parametrically via comprehension, here:

$$\frac{\wr e_i\ \mathsf{val} \int}{\wr e_i \int\ \mathsf{vals}}\ \texttt{vals}\ \ (i{\in}I)$$

When an instance of this rule is used in an actual derivation, $\wr e_i \int$ will match a given multiset E of expressions: I will be a specific index set $\{i_1, \ldots, i_n\}$ and each element of E will correspond to a specific e_{i_j} for $j = 1..n$. This derivation will have n subderivations, one for each judgment instance e_{i_j} val for $j = 1..n$. Note that if $n = 0$, then $I = \varnothing$, in which case this particular instance has no premises. As a rule schema however, `vals` is parametric in the index set I. As such it has a parametric multiset of premises, each of which is uniformly described by the pattern e_i val.

We call inference rules such as `vals` above *multiset-oriented rules* since some of their premises consist of a parametrically-specified multiset of judgments. They form a conservative extension of traditional inference rules since a multiset-oriented rule that does not make use of parametric comprehension is just a normal rule.

Just like traditional inference rules determine an induction principle based on the structure of a derivation, our augmented rule format extends this induction principle to account for parametrically defined premises: to prove a property of a derivation of $\wr e_i \int$ vals, we can assume that a related property, here of e_i val, holds of a generic element e_i. We will examine such a proof later in this section.

As a second example, consider a variant of rules st_· and st_u that allows recursive service calls. The modified rules are:

$$\frac{}{\Sigma_0 \mid \cdot \vdash \cdot} \qquad \frac{\Sigma_0 \mid \Sigma \vdash \Delta \quad \Sigma_0; x:\tau \vdash_{\mathsf{w}} e:\tau'}{\Sigma_0 \mid \Sigma, u:\tau \rightsquigarrow \tau' \mathbin{@} \mathsf{w} \vdash \Delta, u \mathbin{@} \mathsf{w} \hookrightarrow x\!:\!\tau.e}$$

A derivation of $\Sigma_0 \mid \Sigma \vdash \Delta$ progressively removes matching elements from Σ and Δ until none are left. In the process, it forces an order on these multisets, even if they are themselves order-free. If we have a derivation $\mathcal{D}$ of $\Sigma_0 \mid \Sigma \vdash \Delta$ but need the multisets in a different order, then, technically, we cannot use $\mathcal{D}$. We do so anyway in informal arguments because there is a derivation of this judgment for every ordering of the multisets: we tacitly rely on this often implicit property to rearrange $\mathcal{D}$ as needed. A precise treatment, however, requires a multitude of bookkeeping lemmas to permute derivations and manage non-deterministic interleavings.

Our extended rule format captures the behavior of multiset-like entities more directly since the premises of a rule are parametrically-specified multisets of judgments. Indeed, it allows rewriting the above two rules as the following single extended rule:

$$\frac{\wr\, \Sigma_0; x:\tau_i \vdash_{\mathsf{w}_i} e_i : \tau_i' \,\int}{\Sigma_0 \mid \wr u_i:\tau_i \rightsquigarrow \tau_i' \mathbin{@} \mathsf{w}_i \int \vdash \wr u_i \mathbin{@} \mathsf{w}_i \hookrightarrow x\!:\!\tau_i.e_i \int} \ (i \in I)$$

The conclusion expresses Σ and Δ in terms of their generic constituents as $\wr u_i : \tau_i \rightsquigarrow \tau_i' \mathbin{@} \mathsf{w}_i \int$ and $\wr u_i \mathbin{@} \mathsf{w}_i \hookrightarrow x\!:\!\tau_i.e_i \int$ respectively. Moreover, the common index abstraction $i \in I$ forces every $u_i : \tau_i \rightsquigarrow \tau_i' \mathbin{@} \mathsf{w}_i$ in Σ to have a matching $u_i \mathbin{@} \mathsf{w}_i \hookrightarrow x\!:\!\tau_i.e_i$ in Δ, where entities with common indices must be equal. As in our first example, this rule has one premise for every index i, and therefore a parametric multiset of premises. This rule subsumes the axiom rule on the left in the particular case where Σ and Δ are empty.

As a final example, which will come handy later, consider oriented graphs where each node has at most one incoming edge but may have arbitrarily many outgoing edges. Observe that trees, and in general forests, are special cases of such graphs. Given a graph $G = (N, E)$ with these characteristics, it is convenient to describe its edges E as a set $\mathcal{L}$ of pairs (ℓ_i, L_i) where $\ell_i \in N$ is a node and $L_i \subseteq N$ is the set of nodes ℓ_j such that E contains the edge (ℓ_i, ℓ_j)—in particular L_i is empty if ℓ_i has no outgoing edges. Note that G is completely determined by $\mathcal{L}$, and indeed it will be convenient to write G as $G_{\mathcal{L}}$. Following our setup, we will represent these sets as multisets without duplicates. Then, using parametric comprehension, $\mathcal{L}$ assumes the form $\wr(\ell_i, L_i)\int$ with $i \in I$ for some I, and $L_i = \wr \ell_j \int$ for $j \in I_i$ for some I_i. Note furthermore that each node in N is the first component of a pair in $\mathcal{L}$—said differently, $N = \wr \ell_i \int$.

Given a graph $G_{\mathcal{L}}$ described in this way, we want to define a judgment $\mathsf{wt}\ L\ \mathcal{L}$ that holds if and only if $G_{\mathcal{L}}$ is a forest of trees with roots L. ("wt" abbreviates "well-threaded" as we will use this judgment to characterize properly built execution threads in Section 3.2.2.) The following multiset-oriented rule does the job assuming that the pairs in $\mathcal{L}$ have distinct nodes as their first component:

$$\frac{\wr\, \mathsf{wt}\ L_i\ \mathcal{L}_i \,\int}{\mathsf{wt}\ \wr \ell_i \int\ \wr(\ell_i, L_i), \mathcal{L}_i \int}\ \texttt{wt}\ \ (i \in I)$$

In defining $\mathsf{wt}\ L\ \mathcal{L}$, the conclusion of this rule expresses L as the multiset $\wr \ell_i \int$, it finds all the pairs (ℓ_i, L_i) in $\mathcal{L}$ that match these ℓ_i and partitions the rest of $\mathcal{L}$ into multisets $\mathcal{L}_i$. The premise requires that for each

i the judgment $\mathsf{wt}\ L_i\ \mathcal{L}_i$ hold recursively. Intuitively, it starts from the roots L of the forest and checks that the graphs starting at their children are themselves forests.

Let us prove that this intuition is correct: when derivable, $\mathsf{wt}\ L\ \mathcal{L}$ entails that $G_{\mathcal{L}}$ is a forest of trees with roots L.

Lemma 3.1. If there is a derivation $\mathcal{D}$ of $\mathsf{wt}\ L\ \mathcal{L}$ with $\mathcal{L} = \wr(\ell_i, L_i)\int$ where all ℓ_i are distinct, then $G_{\mathcal{L}}$ is a set of trees (a forest) with roots L.

Proof: The proof proceeds by induction on the structure of $\mathcal{D}$. Clearly, the only rule applicable is `wt`, so that $L = \wr\ell_i\int$ and $\mathcal{L} = \wr(\ell_i, L_i), \mathcal{L}_i\int$ and furthermore

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_i\\ \wr\ \mathsf{wt}\ L_i\ \mathcal{L}_i\ \int\end{array}}{\mathsf{wt}\ \wr\ell_i\int\ \wr(\ell_i, L_i), \mathcal{L}_i\int}\ \mathtt{wt}\ \ (i{\in}I)$$

Our induction hypotheses will be that the property can be assumed to hold on the immediate subderivations $\mathcal{D}_i$ of this rule, i.e., that for all i, $G_{\mathcal{L}_i}$ is a forest with roots L_i. Then, $\mathcal{L}_i$ together with (ℓ_i, L_i) is a tree with root ℓ_i. Now, taken all together as $\wr(\ell_i, L_i), \mathcal{L}_i\int$, they are a forest with roots $\wr\ell_i\int$. □

Notice that this proof does not have a distinguished base case. That it makes a correct use of induction stems from the fact that the given derivation $\mathcal{D}$ has finite height (by definition) and that every appeal to the induction hypotheses is on a smaller derivation $\mathcal{D}_i$. The induction stops when a rule conclusion assumes the form $\mathsf{wt}\ \cdot\ \cdot$, i.e., when the rule is invoked on an empty multiset of node and an empty multiset of pairs: only then will it have no premises.

Given a forest $G_{\mathcal{L}} = (N, E)$ with $\mathcal{L} = \wr(\ell_i, L_i)\int$, we have seen that $N = \wr\ell_i\int$. Since each L_i lists edge targets, $C = \wr L_i\int$ is the collection of all inner nodes in $G_{\mathcal{L}}$. Having just proved that if $\mathsf{wt}\ L\ \mathcal{L}$ then $G_{\mathcal{L}}$ is a forest with roots L, we have that $L = N \setminus C$ and $C \subseteq N$.

If $\mathsf{wt}\ L\ \mathcal{L}$ and ℓ is a node in $G_{\mathcal{L}}$, there is a unique subtree $G_{\mathcal{L}_{\mathit{thrd}(\ell)}}$ with root ℓ—i.e., such that $\mathcal{L}_{\mathit{thrd}(\ell)} \subseteq \mathcal{L}$ and $\mathsf{wt}\ \ell\ \mathcal{L}_{\mathit{thrd}(\ell)}$. We will write $\mathit{thrd}(\ell)$ for this tree either as the subgraph $G_{\mathcal{L}_{\mathit{thrd}(\ell)}}$ or the multiset representation $\mathcal{L}_{\mathit{thrd}(\ell)}$ depending on context. Given nodes $L = \wr\ell_i\int$, we write $\mathit{thrd}(L)$ for the forest $\wr\mathit{thrd}(\ell_i)\int$.

3.2. Substructural Operational Semantics

While the external syntax of QWeSST will remain unchanged, we will need to substantially alter the evaluation machinery to support parallel execution. The state of a computation will now be a pair (Δ, ε) consisting of a service repository Δ and a multiset ε of computations ϵ each running on some host w:

$$\text{Computations} \qquad \varepsilon\ ::=\ \cdot\ \mid\ \varepsilon, \epsilon\ @\ \mathsf{w}$$

Several computations may be running on the same node, possibly none. Rather than identifying "computations" ϵ with expressions, SSOS relies on *frames* [18, 4, 30, 20]. QWeSST uses the following frames:

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

The identifiers ℓ, possibly subscripted, appearing in this definition are called *destinations*: we can think of them as symbolic addresses where the result of an evaluation should be delivered. Frames come in three flavors. *Evaluation frames* have the form $(e)^{\ell}$: they indicate that the expression e has not yet been evaluated and that the resulting value should be returned to ℓ—here and in the rest of this section, e stands for an external expression, not the artifact $\textit{expect}\ e\ \textit{from}\ \mathsf{w}$ which we used in Section 2. The last row of the definition defines *return frames* of the form $(v)_{\ell}$ which indicate that the value v of a completed evaluation is to be delivered to destination ℓ—we will enforce the invariant that v is a value in the sense of Section 2. The two middle rows of the definition list the *continuation frames* of QWeSST: these are partially evaluated expressions which are waiting for the result of some subcomputation to be delivered to them before they can compute their own result. For example, $(\textit{fst}\ \ell')^{\ell}$ is expecting a pair value along ℓ' before it can return its first component along ℓ. Notice that $\textit{expect}$ is now a continuation frame.

This form of specification is clearly reminiscent of stack machines: the main differences are that it uses destinations in place of the traditional "hole" $\Box$ (this is convenient to support multiple simultaneous evaluations) and that the computational state is a multiset rather than a stack. This leads to a form of SSOS called *linear destination passing style*. It was pioneered in [18, 4].

The destination $dst(\epsilon)$ of a frame ϵ is the label ℓ appearing as an outer sub- or superscript in ϵ. For example, $dst((\textit{fst}\ \ell')^{\ell}) = \ell$. We write $dst(\varepsilon)$ for the multiset that collects $dst(\epsilon)$ for all ϵ in ε.

3.2.1. Parallel Evaluation

The parallel dynamic semantics of QWeSST makes use of the following four judgments, whose rules are given in Figure 4:

$\Delta; \varepsilon$ initial	"Δ, ε *is an initial state*"
$\Delta; \varepsilon$ final	"Δ, ε *is a final state*"
$\Delta; \varepsilon \mapsto \Delta'; \varepsilon'$	"Δ, ε *makes one atomic step to* Δ', ε'"
$\Delta; \varepsilon \Mapsto \Delta'; \varepsilon'$	"$\Delta; \varepsilon$ *makes one global step to* $\Delta'; \varepsilon'$"

The first two judgments describe an initial and a final state, which consist of only evaluation and return frames respectively.

The global step judgment $\Delta; \varepsilon \Mapsto \Delta'; \varepsilon'$ is meant to indicate that one or more computations in the state $(\Delta; \varepsilon)$ perform one step of evaluation resulting in the state $(\Delta'; \varepsilon')$. These computations may take place anywhere in the network, hence the adjective "global". It is specified by rule pev in Figure 4, which makes use of our extended rule format. Reading this rule clockwise from the bottom left, it partitions the current computations into a portion ε that will not change and in a number of fragments ε_i. Each of these fragments will make an atomic step of computation (defined below), leading to a partial state $(\Delta, \Delta_i; \varepsilon'_i)$ where Δ_i represents any service published during this step. The global state is reassembled as $(\Delta, \wr\Delta_i\int; \wr\varepsilon'_i\int, \varepsilon)$. The side condition $(I \neq \varnothing)$ requires that some thread takes a step, for the sole benefit of our progress theorem below.

The atomic step judgment $\Delta; \varepsilon \mapsto \Delta'; \varepsilon'$ describe the smallest units of computation in this semantics. It either happens locally at a single node, or it synchronizes the local computation of two nodes. We streamline the rules defining this judgment in Figure 4 by using two devices: first, we keep the service repository Δ implicit when a rule does not access or modify it. Second, when all frames in a rule refer to the same world w, we factor it out at the head of the judgment arrow: for example, the conclusion $(\mathsf{fst}\ e)^{\ell} \mapsto_{\mathsf{w}} (\textit{fst}\ \ell')^{\ell}, (e)^{\ell'}$ of rule $\texttt{pev\_fst}_1$ stands for $\Delta; (\mathsf{fst}\ e)^{\ell}\ @\ \mathsf{w} \mapsto \Delta; (\textit{fst}\ \ell')^{\ell}\ @\ \mathsf{w}, (e)^{\ell'}\ @\ \mathsf{w}$.

$$\frac{\lbag\, \Delta; \varepsilon_i \mapsto (\Delta, \Delta'_i); \varepsilon'_i \,\rbag \quad (i \in I \neq \varnothing)}{\Delta; \lbag \varepsilon_i \rbag, \varepsilon \Mapsto \Delta, \lbag \Delta'_i \rbag; \lbag \varepsilon'_i \rbag, \varepsilon}\;\texttt{pev} \qquad \frac{}{\Delta; \lbag (e_i)^{\ell_i} \rbag \ \mathsf{initial}}\;\texttt{init} \qquad \frac{}{\Delta; \lbag (v_i)_{\ell_i} \rbag \ \mathsf{final}}\;\texttt{fin}$$

. . . .

$$\frac{}{()^{\ell} \mapsto_{\mathsf{w}} ()_{\ell}}\;\texttt{pev\_unit} \qquad \frac{}{(\lambda x : \tau.\, e)^{\ell} \mapsto_{\mathsf{w}} (\lambda x : \tau.\, e)_{\ell}}\;\texttt{pev\_lam}$$

$$\frac{}{(\mathsf{hold}\ e)^{\ell} \mapsto_{\mathsf{w}} (\mathsf{hold}\ e)_{\ell}}\;\texttt{pev\_hold} \qquad \frac{}{(\mathsf{url}(\mathsf{w}, u))^{\ell} \mapsto_{\mathsf{w}} (\mathsf{url}(\mathsf{w}, u))_{\ell}}\;\texttt{pev\_url}$$

$$\frac{}{(e_1\ e_2)^{\ell} \mapsto_{\mathsf{w}} (\ell_1\ \ell_2)^{\ell}, (e_1)^{\ell_1}, (e_2)^{\ell_2}}\;\texttt{pev\_app}_1 \qquad \frac{}{(\ell_1\ \ell_2)^{\ell}, (\lambda x : \tau.\, e)_{\ell_1}, (v_2)_{\ell_2} \mapsto_{\mathsf{w}} ([v_2/x]\ e)^{\ell}}\;\texttt{pev\_app}_2$$

$$\frac{}{(\mathsf{fix}\ x : \tau.e)^{\ell} \mapsto_{\mathsf{w}} ([\mathsf{fix}\ x : \tau.e/x]\ e)^{\ell}}\;\texttt{pev\_fix}$$

$$\frac{}{\langle e_1, e_2\rangle^{\ell} \mapsto_{\mathsf{w}} \langle \ell_1, \ell_2\rangle^{\ell}, (e_1)^{\ell_1}, (e_2)^{\ell_2}}\;\texttt{pev\_pair}_1 \qquad \frac{}{\langle \ell_1, \ell_2\rangle^{\ell}, (v_1)_{\ell_1}, (v_2)_{\ell_2} \mapsto_{\mathsf{w}} \langle v_1, v_2\rangle_{\ell}}\;\texttt{pev\_pair}_2$$

$$\frac{}{(\mathsf{fst}\ e)^{\ell} \mapsto_{\mathsf{w}} (\mathit{fst}\ \ell')^{\ell}, (e)^{\ell'}}\;\texttt{pev\_fst}_1 \qquad \frac{}{(\mathit{fst}\ \ell')^{\ell}, \langle v_1, v_2\rangle_{\ell'} \mapsto_{\mathsf{w}} (v_1)_{\ell}}\;\texttt{pev\_fst}_2$$

$$\frac{}{(\mathsf{snd}\ e)^{\ell} \mapsto_{\mathsf{w}} (\mathit{snd}\ \ell')^{\ell}, (e)^{\ell'}}\;\texttt{pev\_snd}_1 \qquad \frac{}{(\mathit{snd}\ \ell')^{\ell}, \langle v_1, v_2\rangle_{\ell'} \mapsto_{\mathsf{w}} (v_2)_{\ell}}\;\texttt{pev\_snd}_2$$

$$\frac{}{(\mathsf{resume}\ e)^{\ell} \mapsto_{\mathsf{w}} (\mathit{resume}\ \ell')^{\ell}, (e)^{\ell'}}\;\texttt{pev\_resume}_1 \qquad \frac{}{(\mathit{resume}\ \ell')^{\ell}, (\mathsf{hold}\ e)_{\ell'} \mapsto_{\mathsf{w}} (e)^{\ell}}\;\texttt{pev\_resume}_2$$

$$\frac{}{\Delta_{\mathsf{w}}; (\mathsf{publish}\ x : \tau.\, e)^{\ell} \mapsto_{\mathsf{w}} (\Delta_{\mathsf{w}}, u \mathbin{@} \mathsf{w} \hookrightarrow x : \tau.e); (\mathsf{url}(\mathsf{w}, u))_{\ell}}\;\texttt{pev\_publish}$$

$$\frac{}{(\mathsf{call}\ e_1\ \mathsf{with}\ e_2)^{\ell} \mapsto_{\mathsf{w}} (\mathit{call}\ \ell_1\ \mathit{with}\ \ell_2)^{\ell}, (e_1)^{\ell_1}, (e_2)^{\ell_2}}\;\texttt{pev\_call}$$

$$\frac{}{\begin{array}{l}(\mathit{call}\ \ell_1\ \mathit{with}\ \ell_2)^{\ell}, (\mathsf{url}(\mathsf{w}', u))_{\ell_1}, (v_2)_{\ell_2} \mapsto_{\mathsf{w}} (\mathit{expect}\ \ell'\ \mathit{from}\ \mathsf{w}')^{\ell} \\ \qquad\qquad \underbrace{(\Delta^{*}_{\mathsf{w}'}, u \mathbin{@} \mathsf{w}' \hookrightarrow x : \tau.e)}_{\Delta_{\mathsf{w}'}}; \cdot \mapsto_{\mathsf{w}'} \Delta_{\mathsf{w}'}; ([v_2/x]\ e)^{\ell'}\end{array}}\;\texttt{pev\_expect}_1$$

$$\frac{}{\begin{array}{c}(\mathit{expect}\ \ell'\ \mathit{from}\ \mathsf{w}')^{\ell} \mapsto_{\mathsf{w}} (v)_{\ell} \\ \qquad (v)_{\ell'} \mapsto_{\mathsf{w}'} \cdot\end{array}}\;\texttt{pev\_expect}_2$$

Whenever the conclusion of an atomic step judgment mentions destinations on the right-hand side that do not occur in the left-hand side, these destinations are assumed to be fresh, i.e., not to appear anywhere in the left-hand side.

Figure 4. Multiset-Based Evaluation Semantics for QWeSST

This highlights the fact that these rules are local to a world. We use this notation also for the two rules where two hosts are involved, pev_expect$_1$ and pev_expect$_2$, which highlights the communication taking place between the two nodes. Fully spelled out, the conclusion of pev_expect$_2$ would be $\Delta; (\mathit{expect}\ \ell'\ \mathit{from}\ \mathsf{w}')^{\ell} \mathbin{@} \mathsf{w}, (v)_{\ell'} \mathbin{@} \mathsf{w}' \mapsto \Delta; (v)_{\ell} \mathbin{@} \mathsf{w}$.

The rules for the atomic step judgment are displayed in the lower part of Figure 4. A first thing to notice is that we have done away with the value judgment v val in favor of transitions such as pev_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 $\texttt{pev\_pair}_2$.

The evaluation of a generic evaluation frame $(e)^{\ell}$ proceeds by spawning threads for each of the subexpressions in e and replacing it with a continuation frame that waits for their result. Once return frames are available for them, it is reduced as appropriate. Each spawned thread is labeled with a new unique destination. For example, the evaluation of an application $(e_1\ e_2)^{\ell}$ in rule $\texttt{pev\_app}_1$ spawns evaluation frames $(e_1)^{\ell_1}$ 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 $\texttt{pev\_app}_2$ kicks in and reduces the application to $([v_2/x]\ e)^{\ell}$.

Rule $\texttt{pev\_expect}_1$ reengineers rule $\texttt{ev\_call}_3$ from Section 2: on the local host w, the expression $\mathsf{call\ url}(\mathsf{w}', u)\ \mathsf{with}\ v_2$ is reduced to the continuation frame $\textit{expect}\ \ell'\ \textit{from}\ \mathsf{w}'$ 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 $\texttt{pev\_expect}_2$ models the hand-off of the value computed by w' in response to this request and corresponds to rule $\texttt{ev\_expect}_2$.

The rules defining the atomic step judgment are closely related to the linear destination passing rules in [18, 4]—they essentially use a different syntax. The global step judgment and its defining rule $\texttt{pev}$ are new however. This rule makes explicit how the atomic step rules relate to the global state, which is implicit in [18, 4] and in the SSOS line of work. It also supports simultaneous atomic transitions where [18, 4] understood parallelism as true concurrency.

3.2.2. Parallel Typing

Before we can prove that QWeSST's parallel evaluation semantics is type safe, we need to extend our existing static semantics to account for computational states and frames. This will have two parts: make sure that the destinations are properly linked together, and provide typing rules for frames and states.

The destinations in a computation are threading frames together to describe the state of evaluation of an expression. Intuitively, these threads should have a structure akin to the sketch in Figure 5: trees with continuations frames as their inner nodes and evaluation or return frames at their leaves. We formalize this intuition in the well-threading judgment

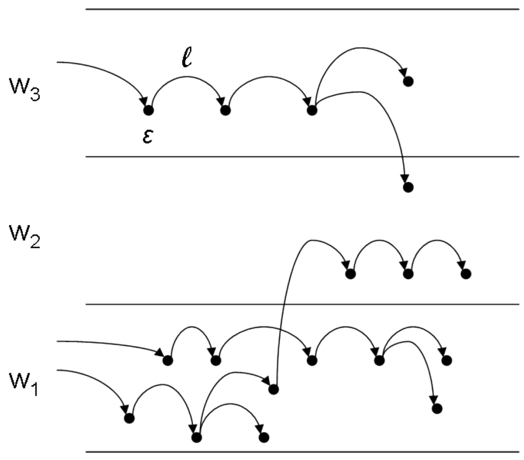

Figure 5. Threads in QWeSST

$\mathsf{wt}\ L\ \mathcal{L}$ *"$\mathcal{L}$ is well-threaded starting at L"*

introduced in Section 3.1, which operates uniquely on destinations. The rule for this judgment is repeated at the top right of Figure 6.

Recall that $dst(\epsilon)$ denotes the destination appearing as an outer super- or subscript in any frame ϵ. We will be interested in computations ε such that the destination $dst(\epsilon)$ of every ϵ in ε is unique, which allows writing $frm(\ell)$ for the unique frame with label ℓ. Then the labels in a computation ε determine a graph G_{ε} whose nodes are simply $dst(\varepsilon)$ and such that there is an edge from ℓ to ℓ' if ℓ' appears inside

$\mathit{frm}(\ell)$ (i.e., not as the frame sub- or superscript). Notice that only continuation frames contribute edges. Under these circumstances, the graph G_ε is a forest and, using the notation of Section 3.1, we view it as a multiset $\mathcal{L}_\varepsilon$ of the form $\wr(\ell_i, L_i)\int$ for $L_i = \wr\ell_{i_j}\int$. We call a computation with these characteristics *well-threaded*.

Each atomic step rule operates on a single tree and, because the rules in Figure 4 assign a fresh destination to every newly created frame, our semantics maintains this well-threading. Indeed, it also preserves the root of this tree, a property maintained at the level of global computations:

Lemma 3.2.

- If $\Delta; \varepsilon \mapsto \Delta'; \varepsilon'$, then $\mathcal{L}_\varepsilon$ has a single root ℓ_ε.
- If $\Delta; \varepsilon \mapsto \Delta'; \varepsilon'$ and $\mathsf{wt}\ \ell_\varepsilon\ \mathcal{L}_\varepsilon$, then $\mathsf{wt}\ \ell_\varepsilon\ \mathcal{L}_{\varepsilon'}$.
- If $\Delta; \varepsilon \Mapsto \Delta'; \varepsilon'$ and $\mathsf{wt}\ L\ \mathcal{L}_\varepsilon$, then $\mathsf{wt}\ L\ \mathcal{L}_{\varepsilon'}$.

Proof: The first two statements are proved by inspection of the atomic step rules in Figure 4. The third is obtained by inversion on rules `ev` and `wt`, parametric appeal to the second statement, and a final application of rule `wt`. □

Given ε and $L \subseteq \mathit{dst}(\varepsilon)$, we write $\mathit{frm}(L)$ for the computation $\wr\mathit{frm}(\ell_i) @ \mathsf{w}_i\int$ for $\ell_i \in L$ for $\epsilon_i @ \mathsf{w}_i \in \varepsilon$—it is the subcomputation of ε whose destinations are in L. Given $\ell \in \mathit{dst}(\varepsilon)$, we will be particularly interested in the subcomputation, or *thread*, identified by the subtree of $\mathcal{L}_\varepsilon$ rooted at ℓ. We write $\mathit{frm}(\mathit{thrd}(\ell))$ for it—recall from Section 3.1 that whenever $\mathsf{wt}\ L\ \mathcal{L}$ and $\ell \in L$, $\mathit{thrd}(\ell)$ denotes the subtree rooted at ℓ in $\mathcal{L}$.

In preparation for typing frames, we define an additional context Λ that assigns a type to a multiset of destinations:

$$\text{Destination typing context:} \qquad \Lambda \ ::= \ \cdot \mid \Lambda, \ell : \tau$$

We treat it as a multiset. We need two additional typing judgments for our parallel evaluation semantics:

$$\begin{array}{ll} \Sigma \vdash_{\mathsf{w}} \ell : \tau \lhd \epsilon \lhd \Lambda & \text{“}\epsilon \textit{ maps } \Lambda \textit{ to } \ell : \tau \textit{ in } \Sigma \textit{ at } \mathsf{w}\text{”} \\ \Sigma; \Lambda \vdash \Delta; \varepsilon & \text{“}\Delta, \varepsilon \textit{ is well-typed in } \Sigma, \Lambda\text{”} \end{array}$$

The first determines the typing pattern of each individual frame: ℓ is the frame's destination, i.e., $\mathit{dst}(\epsilon)$ while Λ lists the type of any other destinations appearing inside it (for continuation frames—it is empty for evaluation and return frames). It is localized at the frame's world. The definition of this judgment is given in the lower part of Figure 6. Notice that the rules for evaluation and return frames refers to the expression typing judgment of Section 2, while the rules for continuation frames have no premises.

The judgment $\Sigma; \Lambda \vdash \Delta; \varepsilon$ typechecks an entire state. It is defined in the top-left corner of Figure 6 using our extended rule format. The multiset of premises to the left synthesizes types for every ϵ_i in ε using the frame typing judgment: the destination structure is read off from the frames themselves and their typing rules propagate typing information backwards along each thread. This layout is convenient because it provides immediate access to all frames, especially to the frames at the end of a thread, which is where atomic evaluation steps takes place. This direct access is particularly useful in the type preservation proof below. The next premise explicitly checks that the destinations in ε are well-threaded with respect to Λ—abusing notation, we write $\mathit{dst}(\Lambda)$ for the destinations in Λ, stripped of their type. Because its derivation is isomorphic to the threads themselves, this is useful when inducting on the

$$\dfrac{\{\!\{\, \Sigma \vdash_{\mathsf{w}_i} \ell_i : \tau_i \lhd \epsilon_i \lhd \Lambda_i \,\}\!\} \quad \mathsf{wt}\ dst(\Lambda) \quad \{\!\{ (\ell_i, dst(\Lambda_i)) \}\!\} \quad \Sigma \vdash \Delta}{\Sigma; \Lambda \vdash \Delta; \{\!\{ \epsilon_i \mathbin{@} \mathsf{w}_i \}\!\}}\ \texttt{sof} \qquad \dfrac{\{\!\{\, \mathsf{wt}\ L_i\ \mathcal{L}_i \,\}\!\}}{\mathsf{wt}\ \{\!\{\ell_i\}\!\}\ \{\!\{ (\ell_i, L_i), \mathcal{L}_i \}\!\}}\ \texttt{wt}$$

$$\dfrac{\Sigma; \cdot \vdash_{\mathsf{w}} e : \tau}{\Sigma \vdash_{\mathsf{w}} \ell : \tau \lhd (e)^{\ell} \lhd \cdot}\ \texttt{tof\_exp} \qquad \dfrac{\Sigma; \cdot \vdash_{\mathsf{w}} v : \tau}{\Sigma \vdash_{\mathsf{w}} \ell : \tau \lhd (v)_{\ell} \lhd \cdot}\ \texttt{tof\_val}$$

$$\dfrac{}{\Sigma \vdash_{\mathsf{w}} \ell : \tau \lhd (\ell_1\ \ell_2)^{\ell} \lhd \ell_1 : \tau' \to \tau, \ell_2 : \tau'}\ \texttt{tof\_app}$$

$$\dfrac{}{\Sigma \vdash_{\mathsf{w}} \ell : \tau_1 \times \tau_2 \lhd \langle \ell_1, \ell_2 \rangle^{\ell} \lhd \ell_1 : \tau_1, \ell_2 : \tau_2}\ \texttt{tof\_pair}$$

$$\dfrac{}{\Sigma \vdash_{\mathsf{w}} \ell : \tau_1 \lhd (\mathit{fst}\ \ell')^{\ell} \lhd \ell' : \tau_1 \times \tau_2}\ \texttt{tof\_fst} \qquad \dfrac{}{\Sigma \vdash_{\mathsf{w}} \ell : \tau_2 \lhd (\mathit{snd}\ \ell')^{\ell} \lhd \ell' : \tau_1 \times \tau_2}\ \texttt{tof\_snd}$$

$$\dfrac{}{\Sigma \vdash_{\mathsf{w}} \ell : \tau \lhd (\mathit{resume}\ \ell')^{\ell} \lhd \ell' : \mathsf{susp}[\tau]}\ \texttt{tof\_resume}$$

$$\dfrac{}{\Sigma \vdash_{\mathsf{w}} \ell : \tau \lhd (\mathit{call}\ \ell_1\ \mathit{with}\ \ell_2)^{\ell} \lhd \ell_1 : \tau' \rightsquigarrow \tau, \ell_2 : \tau'}\ \texttt{tof\_call}$$

$$\dfrac{}{\Sigma \vdash_{\mathsf{w}} \ell : \tau \lhd (\mathit{expect}\ \ell'\ \mathit{from}\ \mathsf{w})^{\ell} \lhd \ell' : \tau}\ \texttt{tof\_expect}$$

Figure 6. Multiset-Based Typing Semantics for QWeSST

structure of a thread, as in the proof of progress below. The last premise makes sure that the services are well-typed using the rules defined in Section 2.

The semantics we have just given for our extended typing infrastructure has the same characteristics as our dynamic semantics in Figure 4. Using SSOS to specify typing is very recent [30] although techniques based on rewriting have been known for a couple of years [10, 11]. Rule `sof` is again new: it explicitly describes how local frame typing is used for typing global states and it does so in parallel for each thread.

3.3. Substructural Meta-Theory

We now have all the ingredients to prove the safety of QWeSST's parallel semantics. Interestingly, our layering of the evaluation and typing judgment into atomic and global will be reflected in the structure of our proofs. Indeed, we will have a type preservation result for atomic steps which directly matches the atomic step rules in Figure 4 and a global type preservation theorem which will lift it to the global state of computation. The same will happen with progress. This gives the meta-theory of this presentation of the semantics of a language a strong substructural flavor that was not observed before, as far as we know.

The glue between our atomic and global results is given by the following lemma, which asserts that if a state of computation is well-typed, then any (sub)thread in it is also well-typed.

Lemma 3.3. If $\Sigma; \Lambda \vdash \Delta; \varepsilon$, then for any $\epsilon \in \varepsilon$ with $\ell = dst(\epsilon)$ there is a derivation of $\Sigma; \ell : \tau \vdash \Delta; \mathit{frm}(\mathit{thrd}(\ell))$ for some type τ.

Proof: By inversion on `sof`, there are derivations $\wr \mathcal{D}_i \int$ of $\wr\ \Sigma \vdash_{\mathsf{w}_i} \ell_i : \tau_i \lhd \epsilon_i \lhd \Lambda_i\ \int$ for $\varepsilon = \wr \epsilon_i \int$ and $\ell_i = dst(\epsilon_i)$, a derivation $\mathcal{F}$ of wt $dst(\Lambda)\ \ \wr(\ell_i, dst(\Lambda_i))\int$ and a derivation $\mathcal{S}$ of $\Sigma \vdash \Delta$. At this point, we proceed by induction on $\mathcal{F}$ to build the desired derivation of $\Sigma; \ell : \tau \vdash \Delta; \mathit{frm}(\mathit{thrd}(\ell))$. This auxiliary proof essentially searches for ℓ in $\mathcal{F}$, keeping only needed components.

If ℓ is among $dst(\Lambda)$, we extract on the one hand a derivation $\mathcal{F}_\ell$ of wt ℓ $\mathit{thrd}(\ell)$ from $\mathcal{F}$ and on the other hand all the derivations $\mathcal{D}_j$ among $\mathcal{D}_i$ with ℓ_j in $\mathit{thrd}(\ell)$—altogether they are exactly the constituents of $\mathit{frm}(\mathit{thrd}(\ell))$. Rule `sof` on $\wr \mathcal{D}_j \int$, $\mathcal{F}_\ell$ and $\mathcal{S}$ then yields the desired derivation.

If instead ℓ is not among $dst(\Lambda)$, it is an inner node of one of the subtrees starting at $dst(\Lambda)$, meaning that it is mentioned by one of the subderivations $\mathcal{F}'$ of $\mathcal{F}$. The nodes in $\mathcal{F}'$ identify a submultiset $\wr \mathcal{D}_j \int$ of $\wr \mathcal{D}_i \int$, and the result follows by an appeal to the induction hypothesis on $\wr \mathcal{D}_j \int$, $\mathcal{F}'$ and $\mathcal{S}$. □

This proof closely tracks the well-threading subderivation in it as it searches for ℓ. If ℓ is a label in Λ, it builds the desired typing derivation by keeping only frames of the subtree starting at ℓ, i.e., $\mathit{frm}(\mathit{thrd}(\ell))$. Otherwise, ℓ appears in the thread of some label ℓ' in Λ: it then inducts on the typing derivation obtained by considering the children of ℓ', pruning the other trees.

Type preservation for atomic steps is essentially obtained by proving that each step rule preserves types.

Lemma 3.4. (Thread-level type preservation)
If $\Delta; \varepsilon_i \mapsto \Delta'; \varepsilon'_i$ and $\Sigma; \ell : \tau \vdash \Delta; \varepsilon_i$, then there is a derivation of $\Sigma'; \ell : \tau \vdash \Delta'; \varepsilon'_i$ for some Σ'.

Proof: This proof is not inductive. Instead, it proceeds by cases on the derivation rule witnessing $\Delta; \varepsilon_i \mapsto \Delta'; \varepsilon'_i$ and inversion on the typing derivation. The cases for rules `pev_unit` through `pev_url` in Figure 4 are immediate. The other cases match directly a corresponding case in the proof of type preservation for the single-threaded semantics of QWeSST (Theorem 2.3) in Section 2. In particular, it makes use of the substitution lemma 2.1 in the cases of rules `pev_app`$_2$, `pev_fix`, `pev_expect`$_1$ which use substitution, and of the relocation lemma 2.2 in the cases of rules `pev_expect`$_1$ and `pev_expect`$_2$ where relocation happens.

For example, consider the case where the derivation of $\Delta; \varepsilon_i \mapsto \Delta'; \varepsilon'_i$ ends with rule `pev_expect`$_1$. Then, $\varepsilon_i = (\textit{call}\ \ell_1\ \textit{with}\ \ell_2)^{\ell}, (\mathsf{url}(\mathsf{w}', u))_{\ell_1}, (v_2)_{\ell_2}$ and Δ contains the service u @ $\mathsf{w}' \hookrightarrow x{:}\tau.e$. By inversion on rules `sof`, `tof_val` (twice), `of_url` and `st_u`, there are typing derivations $\mathcal{T}_1$ of $\Sigma; \cdot \vdash_{\mathsf{w}} v_2 : \tau'$ and $\mathcal{T}_2$ of $\Sigma; x : \tau' \vdash_{\mathsf{w}'} e : \tau$. By the relocation lemma, there is a derivation $\mathcal{T}'_1$ of $\Sigma; \cdot \vdash_{\mathsf{w}'} v_2 : \tau'$ and by the substitution lemma, a derivation $\mathcal{T}'$ of $\Sigma; \cdot \vdash_{\mathsf{w}'} [v_2/x]e : \tau$. At this point, the desired derivation is obtained by applying rules `tof_exp`, `tof_expect` and `sof`. The required derivation of wt $\wr\ \ell\ \int\ \wr(\ell, \wr\ell'\int), (\ell', \cdot)\int$ is immediate. □

The global type preservation theorem stitches together the instances of the above result for all atomic steps in an application of rule `pev`. The proof itself is inherently parallel.

Theorem 3.5. (Global type preservation)
If $\Delta; \varepsilon \Mapsto \Delta'; \varepsilon'$ and $\Sigma; \Lambda \vdash \Delta; \varepsilon$, then $\Sigma'; \Lambda \vdash \Delta'; \varepsilon'$ for some Σ'.

Proof: By inversion on rule `ev`, there are derivations $\mathcal{E}_i$ of $\Delta; \varepsilon_i \mapsto (\Delta, \Delta'_i); \varepsilon'_i$ for $i \in I$ in an appropriate index set I. By lemma 3.2, each subcomputation ε_i has exactly one root ℓ_i which is also the root of ε'_i. For each of them, Lemma 3.3 allows synthesizing a derivation $\mathcal{T}_i$ of $\Sigma; \ell_i : \tau_i \vdash \Delta; \varepsilon_i$, from which the thread-level preservation lemma 3.4 extracts a derivation of $(\Sigma, \Sigma'_i); \ell_i : \tau_i \vdash (\Delta, \Delta'_i); \varepsilon'_i$. Finally, we use rule `ev` to assemble them into the desired derivation. □

It should be noted that the proof of Theorem 3.5 is largely independent of the programming constructs present in our language. Indeed, were we to extend QWeSST with new features, we would need to re-prove the thread-level type preservation lemma 3.4 but not its global counterpart (as long as the structure of the judgments remains the same).

The gist of the small-scale version of the progress theorem is that either we have a return frame or we can make one atomic step of evaluation. We express this idea in terms of threads: given a well-typed thread starting at a destination ℓ, either it is a return frame, or an atomic step can take place at any of its leaves. Because a thread may have more than one leaf, any combination of these atomic steps can happen.

Lemma 3.6. (Thread-level progress)
If $\Sigma; \ell : \tau \vdash \Delta; \varepsilon$, then

- either $\varepsilon = (v)_\ell$ and v val,
- or there exist Δ' and ε' such that $\Delta; \varepsilon \mapsto\!\!\!> \Delta'; \varepsilon'$.

Proof: The proof proceeds by induction on $\mathit{thrd}(\ell)$ and discriminates subcases on $\mathit{frm}(\ell)$. The base case, where $\mathit{thrd}(\ell)$ consists of the single root node ℓ, entails that either $\varepsilon = (v)_\ell$ or $\varepsilon = (e)^\ell$. If $\varepsilon = (v)_\ell$, then a simple induction shows that v must be a value, i.e., v val is derivable. If $\varepsilon = (e)^\ell$, then the rules in Figure 4 always permit making a step, whatever e is.

The inductive cases, in which $\mathit{thrd}(\ell)$ has other nodes besides ℓ, require that $\mathit{frm}(\ell)$ be a continuation frame. We will only illustrate the case where $\mathit{frm}(\ell) = \langle \ell_1, \ell_2 \rangle^\ell$. By inversion on the given derivation of $\Sigma; \ell : \tau \vdash \Delta; \varepsilon$ and of the embedded well-threading subderivation, it must be the case that $\varepsilon = \langle \ell_1, \ell_2 \rangle^\ell, \varepsilon_1, \varepsilon_2$ with $\varepsilon_i = \mathit{frm}(\mathit{thrd}(\ell_i))$ for $i = 1, 2$, which entails that there are typing derivations of $\Sigma; \ell_i : \tau_i \vdash \Delta; \varepsilon_i$. We can then apply the induction hypothesis, obtaining that either $\varepsilon_i = (v_i)_{\ell_i}$ and v_i val, or $\Delta; \varepsilon_i \mapsto\!\!\!> \Delta_i'; \varepsilon_i'$ for some Δ_i' and ε_i'. If a value is returned in both cases, then we can make a step by applying rule $\mathtt{pev\_pair_2}$ followed by $\mathtt{pev}$. If at least one of them makes a step, then simply use rule $\mathtt{pev}$ directly. □

The progress theorem for global states applies the previous lemma to all maximal threads in this state, again in parallel.

Theorem 3.7. (Global progress)
If $\Sigma; \Lambda \vdash \Delta; \varepsilon$, then

- either $\Delta; \varepsilon$ final,
- or there exist Δ' and ε' such that $\Delta; \varepsilon \mapsto\!\!\!> \Delta'; \varepsilon'$.

Proof: By lemma 3.3, for every $\ell_i : \tau_i$ in Λ, there is a derivation of $\Sigma; \ell_i : \tau_i \vdash \Delta; \mathit{frm}(\mathit{thrd}(\ell_i))$. By the thread-level progress lemma 3.6, either $\Delta; \mathit{frm}(\mathit{thrd}(\ell_i))$ final or there are Δ_i' and ε_i' such that $\Delta; \mathit{frm}(\mathit{thrd}(\ell_i)) \mapsto\!\!\!> \Delta'; \varepsilon_i'$. If the first option applies of all ℓ_i, then we have that $\Delta; \varepsilon$ final. Otherwise, we pick one or more of the ℓ_i that take a step and package them into a global step by using rule $\mathtt{ev}$. □

Similarly to global type preservation, this result is only concerned with parallelism and is independent of the the actual constructs present in QWeSST. Adding features does not require a new proof.

In Section 2, we encoded the syntax, semantics and safety proof of QWeSST in Twelf and let this system check our proof. We could not do so for the semantics and proof in this section: not only is Twelf inadequate for this task—it does not provide support for manipulating multisets—but we do not have any

$$\frac{\lbag \varepsilon_i \overset{\ell_i}{\looparrowright} e_i \mathbin{@} \mathsf{w}_i \rbag}{\lbag \varepsilon_i \rbag \overset{\lbag \ell_i \rbag}{\looparrowright} \lbag e_i \mathbin{@} \mathsf{w}_i \rbag}\;\texttt{uw}$$

$$\frac{}{(e)^{\ell} \mathbin{@} \mathsf{w} \overset{\ell}{\looparrowright} e \mathbin{@} \mathsf{w}}\;\texttt{uw\_e} \qquad \frac{}{(v)_{\ell} \mathbin{@} \mathsf{w} \overset{\ell}{\looparrowright} v \mathbin{@} \mathsf{w}}\;\texttt{uw\_v}$$

$$\frac{\varepsilon_1 \overset{\ell_1}{\looparrowright} e_1 \mathbin{@} \mathsf{w} \quad \varepsilon_2 \overset{\ell_2}{\looparrowright} e_2 \mathbin{@} \mathsf{w}}{\varepsilon_1, \varepsilon_2, (\ell_1\ \ell_2)^{\ell} \mathbin{@} \mathsf{w} \overset{\ell}{\looparrowright} (e_1\ e_2) \mathbin{@} \mathsf{w}}\;\texttt{uw\_app} \qquad \frac{\varepsilon_1 \overset{\ell_1}{\looparrowright} e_1 \mathbin{@} \mathsf{w} \quad \varepsilon_2 \overset{\ell_2}{\looparrowright} e_2 \mathbin{@} \mathsf{w}}{\varepsilon_1, \varepsilon_2, \langle \ell_1, \ell_2 \rangle^{\ell} \mathbin{@} \mathsf{w} \overset{\ell}{\looparrowright} \langle e_1, e_2 \rangle \mathbin{@} \mathsf{w}}\;\texttt{uw\_pair}$$

$$\frac{\varepsilon \overset{\ell'}{\looparrowright} e \mathbin{@} \mathsf{w}}{\varepsilon, (\mathit{fst}\ \ell')^{\ell} \mathbin{@} \mathsf{w} \overset{\ell}{\looparrowright} (\mathsf{fst}\ e) \mathbin{@} \mathsf{w}}\;\texttt{uw\_fst} \qquad \frac{\varepsilon \overset{\ell'}{\looparrowright} e \mathbin{@} \mathsf{w}}{\varepsilon, (\mathit{snd}\ \ell')^{\ell} \mathbin{@} \mathsf{w} \overset{\ell}{\looparrowright} (\mathsf{snd}\ e) \mathbin{@} \mathsf{w}}\;\texttt{uw\_snd}$$

$$\frac{\varepsilon \overset{\ell'}{\looparrowright} e \mathbin{@} \mathsf{w}}{\varepsilon, (\mathit{resume}\ \ell')^{\ell} \mathbin{@} \mathsf{w} \overset{\ell}{\looparrowright} (\mathsf{resume}\ e) \mathbin{@} \mathsf{w}}\;\texttt{uw\_resume}$$

$$\frac{\varepsilon_1 \overset{\ell_1}{\looparrowright} e_1 \mathbin{@} \mathsf{w} \quad \varepsilon_2 \overset{\ell_2}{\looparrowright} e_2 \mathbin{@} \mathsf{w}}{\varepsilon_1, \varepsilon_2, (\mathit{call}\ \ell_1\ \mathit{with}\ \ell_2)^{\ell} \mathbin{@} \mathsf{w} \overset{\ell}{\looparrowright} (\mathsf{call}\ e_1\ \mathsf{with}\ e_2) \mathbin{@} \mathsf{w}}\;\texttt{uw\_call}$$

$$\frac{\varepsilon \overset{\ell'}{\looparrowright} e \mathbin{@} \mathsf{w}'}{\varepsilon, (\mathit{expect}\ \ell'\ \mathit{from}\ \mathsf{w}')^{\ell} \mathbin{@} \mathsf{w} \overset{\ell}{\looparrowright} (\mathit{expect}\ e\ \mathit{from}\ \mathsf{w}') \mathbin{@} \mathsf{w}}\;\texttt{uw\_expect}$$

Figure 7. Unraveling Computations

tool that could do the job while taking advantage of the representation methodology espoused in Twelf. More precisely, we could use the concurrent logical framework CLF [4] and its Celf implementation [28] to encode all our rules (except those that manipulate multisets, which map directly to the semantics of CLF). However, this system provides no way to describe the statements of our lemmas. Had we a way to express these statements, we believe that their proofs could easily be encoded using the same substructural methodology already used for the inference rules.

Relationship between the Presentations

Having given two different presentations for the evaluation semantics of QWeSST, it is natural to inquire about how they are related. As we will see, the sequential presentation is easily embedded in the parallel presentation using standard techniques. The reverse direction is more subtle for two reasons: first, parallel execution features multiple threads in general while the sequential presentation has only one; second, multiple subexpressions of a thread in the former can be reduced simultaneously while the latter follows a strict left-to-right reduction policy. The first issue will be mediated by the very substructural meta-theoretic approach advocated in this paper, while the second will force us to concentrate on final values.

Intuitively, a well-formed computation stands for zero or more expressions, some of whose subexpressions have been pulled out yet remain connected in an orderly way through the destination infrastructure. We formalize this intuition by means of the following two *unraveling judgments*, one targeting a single expression, the other a multiset of expressions:

$\varepsilon \overset{\ell}{\looparrowright} e \,@\, \mathsf{w}$	*"Computation ε unravels to expression e at* w *starting from ℓ"*
$\varepsilon \overset{L}{\looparrowright} \wr e_i \,@\, \mathsf{w}_i \int$	*"Computation ε unravels to expressions e_i at* w_i *starting from L"*

The rules for these judgments are given in Figure 7. The first is a simple generalization of a similar device routinely used to prove the correctness of stack machines.

A first noteworthy observation is that the unraveling judgments subsume well-threading, as expressed by the following lemma. Recall that $\mathcal{L}_\varepsilon$ is the destination graph underling computation ε.

Lemma 3.8.

1. If $\varepsilon \overset{\ell}{\looparrowright} e \,@\, \mathsf{w}$, then $\mathsf{wt}\, \ell\, \mathcal{L}_\varepsilon$.
2. If $\varepsilon \overset{L}{\looparrowright} \wr e_i \,@\, \mathsf{w}_i \int$, then $\mathsf{wt}\, L\, \mathcal{L}_\varepsilon$.

Proof: The first statement is proved by induction on the given derivation. The second by multiset comprehension on the basis of the first. □

A simple induction shows that, whenever $\varepsilon \overset{\ell}{\looparrowright} e \,@\, \mathsf{w}$ is derivable, the computation ε determines uniquely the destination ℓ, the expression e and the world w. On the other hand, there can be multiple computations ε, in general, for given ℓ, e and w. Multiset comprehension lifts this observation to the second unraveling judgment: if $\varepsilon \overset{\wr \ell_i \int}{\looparrowright} \wr e_i \,@\, \mathsf{w}_i \int$ for $i \in I$, then ε determines the abstract index set I as well as ℓ_i, e_i and w_i for each $i \in I$.

Unraveling maps well-typed computations to well-typed expressions and vice versa. The forward direction is captured by the following lemma. Notice that it indirectly states that, in the presence of reputable typing information, well-threading determines unraveling.

Lemma 3.9. (Soundness of Parallel Typing)

1. If $\Sigma; \ell : \tau \vdash \Delta; \varepsilon$, then $\varepsilon \overset{\ell}{\looparrowright} e \,@\, \mathsf{w}$ and $\Sigma; \cdot \vdash_{\mathsf{w}} e : \tau$ and $\Sigma \vdash \Delta$.
2. If $\Sigma; \wr \ell_i : \tau_i \int \vdash \Delta; \varepsilon$ for $i \in I$, then $\varepsilon \overset{\wr \ell_i \int}{\looparrowright} \wr e_i \,@\, \mathsf{w}_i \int$ for $i \in I$ and $\Sigma; \cdot \vdash_{\mathsf{w}_i} e_i : \tau_i$ for every $i \in I$ and $\Sigma \vdash \Delta$.

Proof: The proof of the first statement proceeds by induction on the well-threading derivation in the premise of rule `sof`. The second statement is obtained from the first by multiset comprehension. □

The reverse direction states that any computation that unravels to a given typable expression is itself well-typed.

Lemma 3.10. (Completeness of Parallel Typing)
If $\Sigma; \cdot \vdash_{\mathsf{w}} e : \tau$ and $\Sigma \vdash \Delta$, then for every ε such that $\varepsilon \overset{\ell}{\looparrowright} e \,@\, \mathsf{w}$ we have that $\Sigma; \ell : \tau \vdash \Delta; \varepsilon$.

Proof: The proof proceeds by induction on a given unraveling derivation and inversion on the mentioned typing derivation. $\square$

To complete the comparison between our sequential and parallel semantics for QWeSST, we show how their respective evaluations are related. Mapping the former to the latter is relatively straightforward and is captured by the following lemma:

Lemma 3.11. (Completeness of Parallel Evaluation)
If $\Delta\ ;\ e\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e'$, then there exist $\boldsymbol{\varepsilon}$ and $\boldsymbol{\varepsilon}'$ such that $\boldsymbol{\varepsilon} \overset{\ell}{\looparrowright} e \mathbin{@} \mathsf{w}$ and $\boldsymbol{\varepsilon}' \overset{\ell}{\looparrowright} e' \mathbin{@} \mathsf{w}$ and $\Delta; \boldsymbol{\varepsilon} \Mapsto \Delta'; \boldsymbol{\varepsilon}'$.

Proof: By induction on the given sequential step derivation. $\square$

The reverse direction is complicated by the fact that parallel evaluation can reduce multiple redexes in an expression while its sequential counterpart proceeds left-to-right, shifting the focus of reduction forward only when all subexpressions behind it have been reduced to values. As customary when showing that the big-step semantics of a language is sound with respect to its small-step semantics, we will concentrate on completed evaluations, those resulting in a value.

The following auxiliary lemma states that parallel reduction is sound for values and that it is closed under weak-head expansion. We write $\Mapsto^*$ for parallel multi-step relation, i.e., the reflexive and transitive closure of the parallel step judgment $\Mapsto$.

Lemma 3.12.

- If v val, then for all w, ℓ and Δ we have that $\Delta; (v)^{\ell} \mathbin{@} \mathsf{w} \Mapsto^* \Delta; (v)_{\ell} \mathbin{@} \mathsf{w}$.
- If $\Delta; e \mapsto_{\mathsf{w}} \Delta'; e'$ and $\boldsymbol{\varepsilon}' \overset{\ell}{\looparrowright} e' \mathbin{@} \mathsf{w}$ and $\Delta'; \boldsymbol{\varepsilon}' \Mapsto^* \Delta''; (v)_{\ell} \mathbin{@} \mathsf{w}$, then there is $\boldsymbol{\varepsilon}$ such that $\boldsymbol{\varepsilon} \overset{\ell}{\looparrowright} e \mathbin{@} \mathsf{w}$ and $\Delta; \boldsymbol{\varepsilon} \Mapsto^* \Delta''; (v)_{\ell} \mathbin{@} \mathsf{w}$.

Proof: The first statement is proved by induction on the structure of the value v. The second proceeds by simultaneous induction on the given step and unraveling derivations. $\square$

Then, the soundness of the parallel semantics is a simple corollary of this lemma. In the following statement, we also lift it to computations representing multiple expressions.

Corollary 3.13. (Soundness of Parallel Evaluation)

1. If $\Delta; \boldsymbol{\varepsilon} \Mapsto^* \Delta'; (v)_{\ell} \mathbin{@} \mathsf{w}$, then there is e such that $\boldsymbol{\varepsilon} \overset{\ell}{\looparrowright} e \mathbin{@} \mathsf{w}$ and $\Delta; e \mapsto^*_{\mathsf{w}} \Delta'; v$ with v val.
2. If $\Delta; \boldsymbol{\varepsilon} \Mapsto^* \Delta'; \boldsymbol{\varepsilon}'$ with $\boldsymbol{\varepsilon}'$ final, then there are L, I, $\lbag e_i \rbag$, $\lbag \mathsf{w}_i \rbag$ and $\lbag v_i \rbag$ for $i \in I$ such that $\boldsymbol{\varepsilon} \overset{L}{\looparrowright} \lbag e_i \mathbin{@} \mathsf{w}_i \rbag$ and for all $i \in I$ we have $\Delta; e_i \mapsto^*_{\mathsf{w}_i} \Delta'; v_i$ with v_i val,

Proof: The proof of the first statement is obtained by induction on the length of the given derivation, where the two parts of Lemma 3.12 are used in the base and inductive cases respectively. The second statement is proved using multiset comprehension on the first. $\square$

4. Prototype Implementation

A first prototype of QWeSST was described in [26, 27]. Since then, we have developed a second interpreter for our language, which is available at [21]. This new version is entirely written in JavaScript, which allows QWeSST programs to be executed in a platform-independent way: they run on any modern web browser and, using *NodeJS*, they can also operate in server-mode. The interpreter 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 the examples just discussed and many more [27]. Within each node, our prototype consists of three components.

The web service repository holds the services published by the node as well as information about their type. In this way, it implements both the local service repository Δ_{w} and the portion of the service typing table Σ that pertains to this host. The former is used at run-time, the latter during typechecking. The web service repository is implemented as a NoSQL database using *MongoDB*.

The interpreter, the core of our prototype, is a faithful implementation of the language in Section 2 extended with Booleans, integers, strings and their most common operators. It typechecks all the code that originates at the local node and answers any typing requests from other nodes. It typechecks URLs found in local code by asking the host where the corresponding service resides for the correct type, thereby implementing rule $\texttt{of\_url}$ in a distributed fashion. The interpreter also 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, as in rules $\texttt{pev\_expect}_i$. The interpreter is written in JavaScript 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 *NodeJS* web application. It translates service requests to/from the interpreter into HTTP messages.

5. Related Work

Google's Web Toolkit (GWT) [13] is a web development environment that allows writing webapps entirely in Java. Because Java is strongly typed, typing mismatches between client code and server code are caught at compile time. Then, client-side code is compiled to JavaScript and server-side code to Java bytecode. Links [6] is a web oriented programming language that tags functions as client or server to indicate where they shall be executed and also compiles client-side code into JavaScript. Such code can be optimized as not to require the server to maintain any session state [7]. While GWT and Links are well suited for traditional client-server applications, neither supports dynamic services such as our web service auto-installer example. Other efforts along similar lines include Swift [5] 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 [9], a client-server module for Oz with support for declarative graphical user interface programming, and Hop [29], a dynamically typed language for web programming based on Scheme.

The web service community has actively studied conformance mechanisms [1] to ensure that interacting services have compatible communication patterns. Web applications, as considered here, can be seen as degenerate web services with nearly trivial interaction protocols: a host invokes a remote function defined on another host with 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 localized type checking. Interestingly, this restricted form of interaction protocol is common in industrial web services and is essentially what is defined in the WSDL standard [33].

The design of QWeSST has been influenced by Lambda 5 [16, 17], an abstract programming language for distributed computing that uses a modal type system to capture the notion of localized computation. QWeSST can be seen as a simplification of Lambda 5 specialized to web programming by providing exactly the constructs that abstractly capture the way we use the web. Its type $\Box\tau$ corresponds rather directly to our $\mathsf{susp}[\tau]$ and its type $\Diamond\tau$ specialized to functions is exactly our $\tau \rightsquigarrow \tau'$. A more detailed comparison can be found in [27]. On the modal theme, Jia and Walker [14] find a foundation of distributed computing in an intuitionistic modal logic and distill a programming language from it via the Curry-Howard isomorphism. Cardelli and Gordon studied a modal semantics for the ambient calculus [3].

Many languages have been proposed that extend functional programming with support for distributed computation, for example Concurrent ML [22], JoCaml [15] and Erlang [2]. They tend to focus on low level aspects of distributed programming, which contrasts with QWeSST's direct mechanisms to model web programming as done in practice. These are also fully-fledged languages with extensive features while QWeSST is primarily meant as a clean abstraction of some key aspects of web programming. A particularly interesting idea is the "open programming" paradigm of Alice [24, 23], which supports a distributed view of programming by which a program integrates calls to remote libraries and other remote code rather than compiling local copies.

Several authors have proposed extending process calculi [25] to support location-based distributed programming, for example Nomadic Pict [32] and mobile ambients [3]. Ferrara [12] 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. Efforts born out of process algebra bring to the foreground the communication aspect of distributed systems and web programming. However, QWeSST and some of the languages mentioned earlier rely on fairly simple communication modalities, often just a form of message passing in which each request results in a response.

Substructural operational semantics (SSOS) has been proposed fairly recently as a succinct, modular and scalable methodology for specifying programming languages. Initial efforts [18, 4] made use of multisets and destinations in the same way we did on Section 3 to capture the dynamic semantics of various extensions of ML. More recently, Simmons and Pfenning have proposed using ordered structures to do away with destinations in many specifications [30, 20] and extended the scope of SSOS to the static semantics [30]. In a related vein, Ellison uses rewriting logic to describe type systems [10, 11]. This body of work links indirectly the SSOS specification to the global state, often via the underlying logic, while we make it explicit. This allows us to draw attention to the substructural nature of the resulting meta-theoretic proofs.

6. Conclusion and Future Work

In this paper, we have presented QWeSST, a locally typed programming language that isolates two key features of web programming as done in practice, namely remote execution and mobile code. We have illustrated its use by means of a few examples, all of which run on our prototype implementation. We have also proved that it is globally type safe according to two different semantics: a naive single-threaded evaluation semantics and a more realistic parallel semantics. For the latter we extended the SSOS style of specification with multiset-oriented inference rules, which express parallelism well and which support an elegant style for proving results about them.

In future work, we plan to extend QWeSST to capture more aspects of practical web programming, namely concurrency in the presence of mutable state, and critically security, in particular access control and information flow. We also intend to develop our prototype to include richer types and a document object model (DOM) library to allow us to run more realistic web programming experiments. We also plan to apply our style of SSOS specification to other settings, and to explore logical frameworks that could natively express the resulting reasoning patterns, as observed in our second safety proof.

Acknowledgments

We are grateful to Rob Simmons, Frank Pfenning, Bob Harper, Dan Licata and Jorge Luis Sacchini for the fruitful discussions during the design of QWeSST, its SSOS specification and for their help using Twelf. We are also thankful to the anonymous colleagues who reviewed a very early version of this paper for the LAM'10 workshop.

References

[1] Baldoni, M., Baroglio, C., Martelli, A., Patti, V.: A priori conformance verification for guaranteeing interoperability in open environments, *Proc. 4th International Conference on Service Oriented Computing (ICSOC'06)* (A. Dan, W. Lamersdorf, Eds.), Springer-Verlag LNCS 4294, Chicago, IL, 2006.

[2] Brown, L., Sahlin, D.: Extending Erlang for Safe Mobile Code Execution, *Proceedings of the 2nd International conference on Information and Communication Security*, Sydney, Australia, 1999.

[3] Cardelli, L., Gordon, A. D.: Anytime, anywhere. Modal logics for mobile ambients, *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL'00)*, ACM Press, San Antonio, TX, 2000.

[4] Cervesato, I., Pfenning, F., Walker, D., Watkins, K.: *A Concurrent Logical Framework II: Examples and Applications*, Technical Report CMU-CS-02-102, Carnegie Mellon University, Pittsburgh, PA, 2003.

[5] Chong, S., Liu, J., Myers, A. C., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Secure web applications via automatic partitioning, *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Stevenson, WA, 2007.

[6] Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web Programming Without Tiers, *Formal Methods for Components and Objects*, 2007, 266–296.

[7] Cooper, E., Wadler, P.: The RPC Calculus, *Proceedings of the 11th Symposium on Principles and Practice of Declarative Programming — PPDP'09* (A. Porto, F. López-Fraguas, Eds.), ACM, Coimbra, Portugal, 2009.

[8] De Labey, S., van Dooren, M., Steegmans, E.: ServiceJ A Java Extension for Programming Web Services Interactions, *Web Services, 2007. ICWS 2007. IEEE International Conference on*, 2007.

[9] El-Ansary, S., Grolaux, D., Roy, P. V., Rafea, M.: Overcoming the multiplicity of languages and technologies for web-based development using a multi-paradigm approach, *Procceedings of the 2nd International Conference on Multiparadigm Programming in Mozart/Oz (MOZ'04)*, 2004.

[10] Ellison III, C. M.: *A Rewriting Logic Approach to Defining Type Systems*, Master Thesis, University of Illinois at Urbana-Champaign, 2008.

[11] Ellison III, C. M., Şerbănuţă, T. F., Roşu, G.: *A Rewriting Approach to Type Inference*, Technical Report UIUCDCS-R-2008-2934, University of Illinois at Urbana-Champaign, 2008.

[12] Ferrara, A.: Web services: a process algebra approach, *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, ACM, New York, NY, USA, 2004.

[13] Google Inc.: Google Web Toolkit, `http://code.google.com/webtoolkit/`.

[14] Jia, L., Walker, D.: Modal Proofs as Distributed Programs, *Programming Languages and Systems*, 2004, 219–233.

[15] Mandel, L., Maranget, L.: *Programming in JoCaml*, Technical Report RR-6261, MOSCOVA – INRIA Rocquencourt, 2007.

[16] Murphy VII, T.: *Modal Types for Mobile Code*, Ph.D. Thesis, Carnegie Mellon University, January 2008, Available as technical report CMU-CS-08-126.

[17] Murphy VII, T., Crary, K., Harper, R.: Type-Safe Distributed Programming with ML5, *Trustworthy Global Computing*, 2008, 108–123.

[18] Pfenning, F.: Substructural Operational Semantics and Linear Destination-Passing Style, 2004, Invited talk to the Second Asian Symposium on Programming Languages and Semantics (APLAS'04). Slides at `http://www.cs.cmu.edu/~fp/talks/aplas04-talk.ps`.

[19] Pfenning, F., Schürmann, C.: System Description: Twelf — A Meta-Logical Framework for Deductive Systems, *Proc. CADE-16* (H. Ganzinger, Ed.), Springer-Verlag LNAI 1632, Trento, Italy, 1999.

[20] Pfenning, F., Simmons, R. J.: Substructural Operational Semantics as Ordered Logic Programming, *Proc. 24th IEEE Symposium on Logic in Computer Science—LICS'09*, Los Angeles, CA, 2009.

[21] Qwel — programming language for the web, `http://qwel.qatar.cmu.edu`.

[22] Reppy, J.: *Concurrent Programming in ML*, Cambridge University Press, 1999.

[23] Rossberg, A.: *Typed Open Programming - A higher-order, typed approach to dynamic modularity and distribution*, Ph.D. Thesis, Universität des Saarlandes, 2007.

[24] Rossberg, A., Botlan, D. L., Tack, G., Brunklaus, T., Smolka, G.: Alice through the looking glass, *Proceedings of the 4th Symposium on Trends in Functional Programming (TFP'04)*, Intellect Books, Munich, Germany, 2004.

[25] Sangiorgi, D., Walker, D.: *The π-Calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.

[26] Sans, T., Cervesato, I.: QWeSST for Type-Safe Web Programming, *Third International Workshop on Logics, Agents, and Mobility — LAM'10* (B. Farwer, Ed.), Edinburgh, Scotland, UK, 2010.

[27] Sans, T., Cervesato, I.: *Type-Safe Web Programming in QWeSST*, Technical Report CMU-CS-10-125, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 2010.

[28] Schack-Nielsen, A., Schürmann, C.: Celf—a logical framework for deductive and concurrent systems (system description), *Proceeding of IJCAR'08*, Springer-Verlag LNCS 5195, 2008.

[29] Serrano, M., Gallesio, E., Loitsch, F.: Hop, a Language for Programming the Web 2.0, *Proceedings of the First Dynamic Languages Symposium (DLS'06)*, Portland, OR, 2006.

[30] Simmons, R. J.: *Type safety for substructural specifications: preliminary results*, Technical Report CMU-CS-10-134, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 2010.

[31] The Twelf project wiki, `http://twelf.org`.

[32] Unypoth, A., Sewell, P.: Nomadic Pict: Correct communication infrastructure for mobile computation, *Proc. 28th ACM Symposium on Principles of Programming Languages (POPL'01)*, London, UK, 2001.

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