Higher-Order Symb Comput (2010) 23:409–431
DOI 10.1007/s10990-010-9061-9

# A multi-tier semantics for Hop

**Manuel Serrano · Christian Queinnec**

Published online: 20 August 2010


**Abstract** Hop is a multi-tier programming language where the behavior of interacting servers and clients are expressed by a single program. Hop adheres to the standard web programming style where servers elaborate HTML pages containing JavaScript code. This JavaScript code responds locally to user's interactions but also (following the so-called Ajax style) requests services from remote servers. These services bring back new HTML fragments containing additional JavaScript code replacing or modifying the state of the client.

This paper presents a continuation-based denotational semantics for a sequential subset of Hop. Though restricted to a single server and a single client, this semantics takes into account the key feature of Hop namely that the server elaborates client code to be run in the client's browser. This new client-code dynamically requests services from the server which, again, elaborate new client code to be run in the client's browser.

This semantics details the programming model advocated by Hop and provides a sound basis for future studies such as security of web applications and web continuations.

**Keywords** Multi-tier language · Web programming · Semantics

## 1 Introduction

Hop is dedicated to programming interactive applications on the web [16]. It belongs to the new family of languages sometimes referred to as *multi-tier* languages [3, 12, 13]. Within a single formalism, these languages specify code residing on multiple locations or tiers, such as a client and a server. In Hop, the server is a full-fledged bootstrapped HTTP server [15] and clients are regular browsers able to execute JavaScript programs.

M. Serrano (✉)
INRIA/Sophia-Antipolis, 2004 route des lucioles, BP 93, 06902 Sophia Antipolis, France
e-mail: manuel.serrano@inria.fr

C. Queinnec
UPMC/LIP6, 4, place Jussieu, 75252 Paris, France
e-mail: christian.queinnec@lip6.fr

Hop rests on the model of classical Ajax applications programming where server-side code *elaborates* client-side code that will get evaluated by the clients, *i.e.*, the web browsers. Adopting this model allows Hop to be fully compatible with traditional web technologies and distinguishes it from other multi-tier languages.

Realistic Hop applications involve several servers and several clients. Each server runs a Hop runtime system that automatically deploys the application on the clients upon connections. The server-side parts are compiled to native code. The client-side parts are compiled to JavaScript on the fly [9]. The two sides of the application run in parallel[1] on different computers. Clients *pull* data from the server by calling special functions named services. Servers *push* data toward clients by means of signal broadcasts. This paper presents a formal semantics for a simplified version of Hop that describes *how* a program is deployed on a client and *how* the execution flows between one server and one client. It only focuses on these aspects that distinguish Hop from traditional programming languages. Other language features of Hop which are also common in other programming languages such as signal broadcasting, concurrency, multiple clients, and multiple servers are ignored. This study will provide a sound theoretical basis to address later these postponed topics. To our knowledge the semantics presented in this paper is the first ever published that copes with dynamic code generation of multi-tier programming languages.

Interactive web applications which demand server-side and client-side computations have specificities such as intensive HTML manipulations or asymmetric distributed execution that make general purpose programming languages inadequate or clumsy. These web applications actually demand new programming paradigms that are not well supported by traditional programming languages. Hop has been created to match this demand. Its design has been motivated by a suite of web applications that were envisioned. After a couple of years spent in developing Hop and web applications, we felt that it was time to polish the underlying model on which rests the language and to extract, from the implementation, a concise description of the specificities that distinguishes it from other languages. This is the first purpose of the semantics presented in the paper.

The second motivation is of a different nature. Being a multi-tier programming language, Hop exposes a single formalism used to program a whole web application. This eases the programming task because only one language has to be tamed. This also opens opportunities for computing programs global properties, in particular security properties.

One of the plague of web applications is a security violation based on *cross site scripting* (or XSS). It enables malicious attackers to inject client-side script into web pages viewed by other users.[2] The common approach to protect against XSS is to control client-side executions via browser [7] specific equipments. Provided with a coherent formalism for programming the client and the server other approaches can be envisioned. The Hop semantics explains how client-side programs are generated by servers. Thus, it can be used to reason about client-code generation, for instance, for tracking unexpected or malicious client-side codes generation.

The semantics given in this paper is formalized with a continuation-based denotational style [17]. Besides being compact and hiding useless implementation details, this choice offers several advantages: (i) it follows the spirit of the semantics used for the various revised reports on Scheme [8], (ii) it makes all continuations explicit thus providing a sound basis

[1] In the actual implementation, servers handle several clients concurrently. This server-side parallelism is not described in the simplified semantics presented in this paper.

[2] Sammy (http://en.wikipedia.org/wiki/Samy_(XSS)) is a famous example of a worm that propagate over MySpace in the end of 2007.

to study the introduction of web continuations [14], (iii) it gives us a first formal definition raising new interrogations concerning, for instance, possible alternate choices, (iv) finally, it represents a formal definition against which new semantics, currently under work, will be gauged.

## 2 The Hop programming language

The Hop programming language is designed *for* the web *according to* the programming model of the web. It assumes applications whose server sides generate HTML trees that represent the client sides. These HTML trees play three roles: (i) they declare the UI for the browsers, (ii) they declare client-side expressions, and (iii) they start the application on the clients. Contrary to most systems, Hop embraces the server-side *and* the client-side of web applications within a single language. Still, Hop is fully web compliant:

- server-side and client-side parts of the application communicate using HTTP connections,
- Hop client code is compiled into HTML and JavaScript,
- Hop can use foreign JavaScript APIs, REST APIs, or other common web technologies out of the box.

When designing Hop we have found it interesting to adhere to the traditional web programming model and to push it as far as possible. On the one hand we may consider this model as awkward in terms of programming language but on the other hand, we also think that we have to acknowledge that it has enabled the delivery of applications that were unconceivable only a couple of years ago. Using, studying, and exploring the traditional web programming model is the path we follow to understand if this model is the reason of the success of these web applications.

Hop is based on the Bigloo language,[3] a dialect of Scheme R5RS [8]. Hop extends Bigloo with four constructs to accommodate web programming:

- **service definitions**: services are server-side functions associated to URLs. These functions are defined by the `service` form whose syntax is close to the traditional `lambda` form. Although not presented in the paper, Hop syntactic sugars allows standard web REST APIs to be automatically wrapped into services. Hence, web services can be used in Hop programs using a single coherent syntax, whether they are implemented in the language or not.
- **service invocations**: servers and clients may invoke services with the `with-hop` form. When a client needs to fetch or send information from and to its origin server, it uses `with-hop`. When it needs to execute an action such as raising the sound volume of a multimedia application, it also uses `with-hop`. Symmetrically `with-hop` is also used to implement server-to-server communication.
- **client-side expressions**: a Hop program starts to run on the server. It generates a client-side program. On the server, this client-side program is a HTML document reified as an AST (Abstract Syntax Tree). On the client, this program displays a GUI to end-users. End-user actions, *e.g.*, button clicks, mouse moves, etc. trigger evaluations of Hop client-side expressions. The ∼ (tilde) form is a useful syntax to embed client-side expressions within ASTs.

[3] http://www.inria.fr/indes/fp/Bigloo.

$\pi$ = *Scheme*
| `(with-hop` $\pi$ $\pi$`)`
| $\sim\xi$
| `(service [:url` $\pi$`]` `(`$id_1$ $id_2$ `...)` $\pi$`)`

$\xi$ = *Scheme*
| `(with-hop` $\xi$ $\xi$`)`
| `$` $\pi$

**Fig. 1** Hop dual grammar, server-side on the left, client-side on the right

– **server-side expressions**: server-side values can be *injected* inside client-side expressions with the `$` (dollar) form. At first glance, `$`-expressions can be considered as symmetric to $\sim$-expressions but, in fact, they should be better considered as `unquote` forms within `quasiquote` forms.

Figure 1 shows the syntax of Hop. $\pi$ is a server-side expression while $\xi$ is a client-side expression. Both syntaxes extend the Scheme's syntax.

The rest of this section presents various examples that introduce Hop step by step. They illustrate the aspects of the language that are formalized by the presented semantics. The excerpts are all actual programs that can be run in Hop.

### 2.1 Creating a HTML document

Hop extends Scheme values with a new data type representing HTML ASTs that are created and manipulated by Hop library functions. By convention, the names of these functions are of the form <*upper-case-letter*$^+$>. The following function `hello1` creates such an AST that defines a HTML page with a simple message. Using an AST instead of a textual string-based representation ensures that HTML documents are well formed.

```
(define (hello1 x)
   (<HTML> (<BODY> "Hello " x '!)))
```

The expression `(hello1 "world")` produces a HTML page displaying `Hello world!`.

### 2.2 Creating a HTML document with a client-side action

Once installed on a client, *i.e.*, a web browser, a program reacts to end-user actions. These actions are Hop client-side expressions, they are introduced inside the HTML AST by the $\sim$-form, such as:

```
(define (hello2 x)
   (<HTML>
      (<BODY> :onclick ~(alert "Goodbye")
         "Hello " x '!)))
```

An expression such as `(hello2 "world")` produces an HTML document that, when displayed, looks similarly to `(hello1 "world")`. However, when the text is clicked, a pop-up window displays the message `Goodbye`.

As HTML syntactic correctness is ensured by compiling an AST into HTML, client-side correctness is ensured by compiling ∼-forms into JavaScript. Being automatically generated, Hop client-side programs are guaranteed to be free syntactic errors that may plague programs generated by systems based on textual representations.

Server-side and client-side expressions use different global environments: they do not share variables. Hence, the next program creates a GUI displaying the sentence `Hello world!` that, when clicked, pops up a window displaying `Goodbye` because, while server-side `x` is bound to the string `"world"`, client-side (global) `x` is bound to the string `"Goodbye"`:

```
(define (hello3 x)
   (<HTML>
      (<BODY> :onclick ~(alert x)
         ~(define x "Goodbye")
         "Hello " x '!)))
(hello3 "world")
```

The server-side and client-side runtime libraries offer similar libraries. In particular, the HTML constructors and the DOM handling functions are available on both sides as demonstrated in the next function:

```
(define (hello4 x)
   (let ((sdiv (<DIV> "a server div")))
      (dom-append-child! sdiv (<SPAN> "a server span"))
      (<HTML>
         (<BODY> :onclick ~(dom-append-child!
                              document.body
                              (<DIV> "a client div"))
            sdiv))))
```

In this example, the library functions `dom-append-child!` and `<DIV>` are both used once on the server and once on the client. On the server, a HTML subtree with the message `"a server div"` is created and bound to the server local variable `sdiv`. Still on the server, a *span* is added to `sdiv`. When the message `"a server div"` is clicked on the client, a second *div* containing the message `"a client div"` is added to the document. The client-side document is pointed to by the global variable `document` which is initialized by the browser.

### 2.3 Defining a service

Functions defined by the server can only be used by server-side expressions. Similarly functions defined by the client can only be used by client-side expressions. *Services* overcome the limitation of regular functions. Services are special server functions that can be invoked by clients via URLs. They are defined by the special form `service`.

The following service `shello4` wraps the `hello4` function into a service:

```
(define shello4 (service :url "/hop/shello4" (x) (hello4 x)))
```

Defining a service *binds* a server-side function to a URL (here `"/hop/shello4"`) which can be used by any web browser to trigger a server-side computation, *i.e.*, to call a

server-side service. For instance, assuming a Hop server waiting connections on port `8080`, one may browse the URL `http://localhost:8080/hop/shello4?x=world` in which case the HTML document presented in Sect. 2.2 is returned to the web browser.

Just as Scheme provides a shorthand for binding functions to variables, Hop provides `define-service` to define and bind a service to a global variable while making this service reachable via a URL deduced from its name. With that syntax, the former definition of `shello4` is simplified as:

```
(define-service (shello4 x) (hello4 x))
```

### 2.4 Invoking a service

A service is a functional value that, when invoked, returns its associated URL. For instance, the expression `(shello4 "world")` produces the following URL:

`http://localhost:8080/hop/shello4?x=world`.

The `with-hop` special form triggers the evaluation of a service. `with-hop` forms may be used either by servers or clients as:

```
(with-hop (shello4 "Hop") (lambda (v) ...))
```

A `with-hop` form invokes *asynchronously* a service (here `shello4`) and returns instantly an unspecified value. When the service returns a result, a callback procedure (here, the anonymous `lambda`) is applied on that result. The purpose of a `with-hop` form is to spawn a remote service invocation and to register a callback that will get invoked asynchronously when the client has received the response to its request. The precise semantics of the `with-hop` form will be presented in Sect. 4.

### 2.5 Invoking a service from a client

Servers and clients use different name spaces but server values can be injected into client code constructed on the server side by means of `$`-expressions.

```
(define-service (shello5 x)
  (<HTML> (<BODY> :onclick ~(alert $x) "Hello!")))
```

This service returns a HTML page that, when clicked, pops up an alert displaying the argument passed to `shello5`. Here are three examples of invocations:

```
1. http://localhost:8080/hop/shello5?x=You%20clicked%20me!
2. (set! document.location (shello5 "You clicked me!"))
3. (set! document.location (shello5
   (<SPAN> "Yes you clicked me!")))
```

The global variable `document` which has been used in the example `hello4` to access the document currently displayed by the browser is now used to change the current location of the browser. Changing the value of `document.location` instructs the browser to download and to display a new URL.

Server-side values are injected within client-side expressions when the server builds the HTML AST so `$`-forms do not involve any communication between servers and clients.

`$`-forms are processed ahead of client-side evaluation. As a consequence, clients are unaware of any afterward modifications of injected values.

All data types but plain functions introduced by the keyword `lambda` can be injected into client-side code. Services can be injected into client-side code. This allows client-side expressions to reify services and hence, to invoke them, such as:

```
(define-service (shello6 x)
  (<HTML>
     (<BODY> :onclick ~(with-hop ($(service ()
                                      (format "Bonjour ~a" x)))
                           (lambda (v) (alert v)))
       "Hello!")))
```

When this document is clicked, the client invokes the anonymous service that returns a new string of characters. This string is sent back to the client and bound to the callback's parameter `v` that displays it in an alert dialog box.

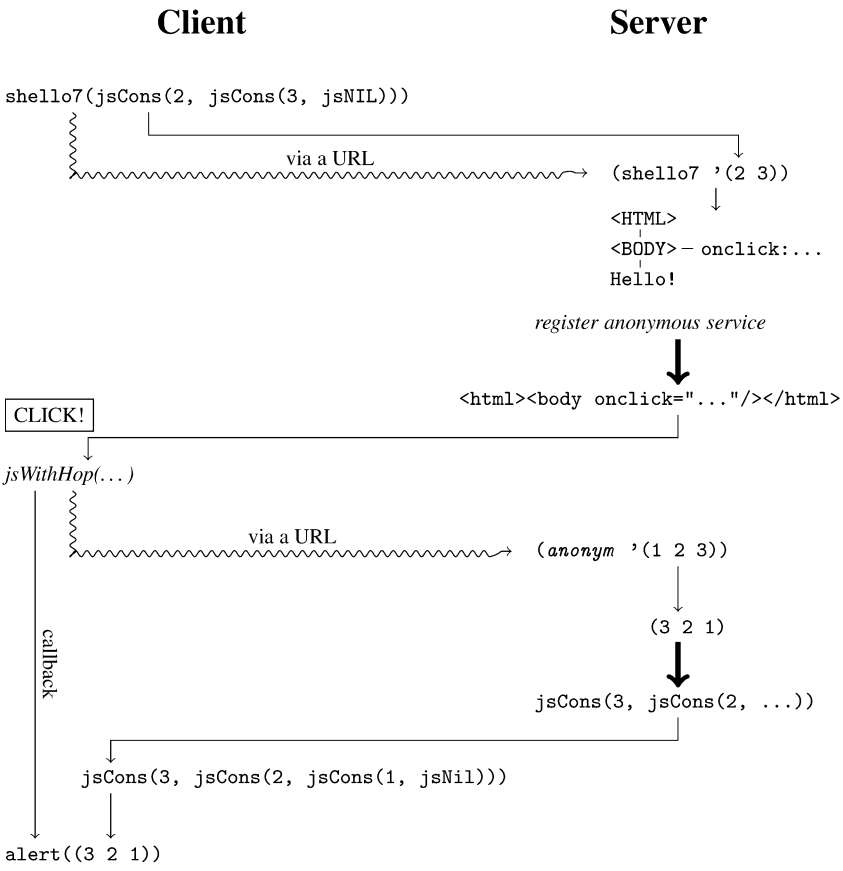


**Fig. 2** Sequence diagram for a service invocation. *Snake arrows* represent HTTP requests. *Solid arrows* represent evaluation and marshalling via HTTP. *Thick vertical arrows* mean compilation towards JavaScript. Names prefixed by *js* suggest they run in JavaScript

Note that the `:url` argument of services presented in Fig. 1 is optional. Unless specified, Hop automatically allocates fresh URLs. These are never reclaimed but the `service` form accepts two optional arguments that control the *life-time* and *time-to-live* of the services. Invoking a service that no longer exists raises a client exception.

Services accept parameters:

```
(define-service (shello7 x)
  (<HTML>
     (<BODY>
        :onclick ~(with-hop ($(service (l) (reverse l))
                            (cons 1 $x)) (lambda (v) (alert v)))
        "Hello!")))
```

Figure 2 displays the steps needed to evaluate the expression `(shello7 '(2 3))`. First, the `shello7` service is invoked. This produces a web page displaying the message `Hello!`. When clicked, it triggers a call to an anonymous service that reverses its argument. The result (the list `(3 2 1)`) is displayed, on the client, in an alert window.

## 3 Informal semantics

As seen in the previous section, Hop mixes two locations for evaluation (servers and clients) and two times of evaluation: initially on the server (when elaborating a HTTP response) then on the client. Moreover, when the user interacts with the GUI displayed by the browser, new evaluations may be requested on the server.

The main objective of a Hop server-side program is to generate new expressions to be evaluated on the client. That is, client-side programs (a mix of HTML and JavaScript) are first-class values generated by server-side computations. This is the standard model of most web applications, and in particular, the model of pure AJAX applications that are dominant in the landscape of the Web 2.0. The peculiarities of this model are reflected by an unusual staged semantics of the language first informally introduced in this section then formally presented in Sect. 4.

### 3.1 Core Hop

We assume that a user's initial HTTP request forces the evaluation of a given Hop program on the server. We do not specify how this is done and focus on the semantics of this running Hop program. We define *Core Hop*, a simplified variant of Hop, which is an applied $\lambda$-calculus with a single grammatical extension: the $\sim$-form.

When a user interacts with the browser and this interaction requests a service from a server, we modeled this interaction as a Core Hop expression to be evaluated in the global environment of this server.

Computations in Core Hop may dynamically define new (anonymous) services (see `shello6` for instance) so we separate the global environment from the local lexical environment and we introduce a `(gdef` $\nu$ $\pi$`)` form to bind global variables from deeper lexical environments. The `(gdef` $\nu$ $\pi$`)` form binds (or rebinds) the global variable $\nu$ to the value of the expression $\pi$. The `gdef` form hides implementation details on how real services (bound to URLs in real Hop) are invoked and how URLs are managed. In Core

```
π = ν
  | (gdef ν π)
  | (lambda (ν) π)
  | (π1 π2)
  | (begin π1 π2)
  | ~ϖ

ν = variable
```

```
ϖ = ν
  | (gdef ν ϖ)
  | (lambda (ν) ϖ)
  | (ϖ1 ϖ2)
  | (begin ϖ1 ϖ2)
  | $ π
  | (with-hop (π ϖ1) ϖ2)
```

```
ξ = ν
  | (gdef ν ξ)
  | (lambda (ν) ξ)
  | (ξ1 ξ2)
  | (begin ξ1 ξ2)
  | (with-hop (π ξ) ξ1)
```

**Fig. 3** Grammar for Core Hop (*left*) and its Tilde Hop subset (*center*). Grammar for Core JavaScript is on the *right*

Hop, services are invoked as plain expressions evaluated in the global environment of the server.

Since the `gdef` form performs a side-effect, we also add the Scheme `begin` form to Core Hop.

An usual HTTP response is a HTML tree with nodes containing JavaScript expressions or references to JavaScript files. Since JavaScript expressions may dynamically construct HTML trees, we simplify the semantics by assuming that the values produced by a Core Hop program are JavaScript programs (values of the *Core JavaScript* language defined below) that get executed by the client.

### 3.2 Tilde Hop

The ∼-form allows Core Hop to create JavaScript programs. We name *Tilde Hop* this subset of Core Hop. The ∼-form might be compared to a `quasiquote` form where `unquote` is named `$`. A `quasiquote` form builds an S-expression whereas a ∼-form builds a Core JavaScript program.

A Tilde Hop expression looks like a Core JavaScript program with two additions (see center part of Fig. 3) — the `$`-form introduces a Core Hop value into the JavaScript program being built and — the `with-hop` form allows the client code, when run, to interact with the server.

Similarly to `quasiquote` forms, ∼-forms are not essential: JavaScript programs may also be built out of simple strings. However ∼-forms allow Hop programmers to easily build readable and always syntactically correct JavaScript programs.

### 3.3 Core JavaScript

Symmetrically, we simplify JavaScript into *Core JavaScript*, a quite similar applied λ-calculus (see Fig. 3 right) with a new extension: the `with-hop` form to interact with servers. The form `(with-hop (π ξ) ξ1)` (see Fig. 4) applies, in the global environment of the server, the value of $\pi$ (a Core Hop expression yielding a function) on the value of $\xi$ (a Core JavaScript expression). This is a simplification since the expression $\pi$ does not travel from the client to the server: in real Hop, $\pi$ stays on the server and is hidden behind a URL.

The result of this application is a new JavaScript program sent back to the client where it will run in the global environment. The value of this JavaScript program is then given as argument to the callback $\xi_1$ still in the global environment of the client.

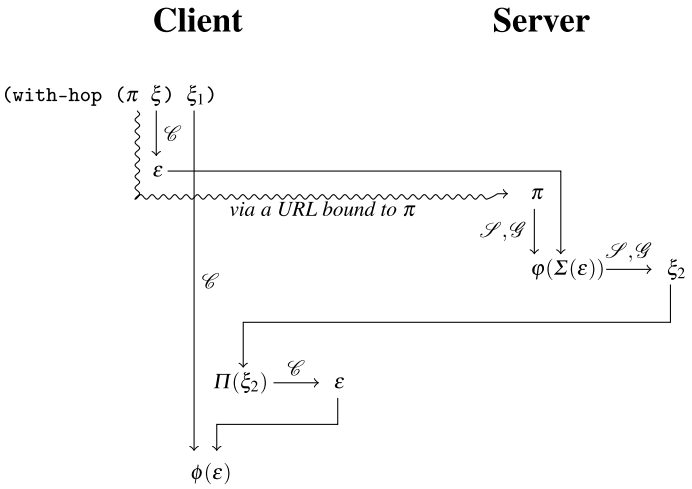


**Fig. 4** Evaluation sequence of a `with-hop` form in Core Hop and Core JavaScript. *Solid arrows* represent evaluation or marshalling. The *snake arrow* represents a service invocation

A HTTP response may contain HTML nodes with additional global JavaScript functions hence the necessity of `gdef` to take this into account within Core JavaScript (see `hello3` for instance).

The difference between the grammars of Core JavaScript ($\xi$) and Tilde Hop ($\varpi$) comes from `$`-forms that are only allowed in the latter. This reflects the different evaluation times involved in Hop: first, a program generates an AST ($\pi$ and $\varpi$), second, the AST is evaluated on the client ($\xi$).

### 3.4 User's interactions on the client

We model a user's interaction on the GUI displayed by the browser as the evaluation of a Core JavaScript expression. This expression is evaluated in the global environment of the client. It may involve some `with-hop` forms to fetch information from the server. That information will be bound to the formal parameter of the callback function. JavaScript is roughly sequential and so is Core JavaScript. All user's interactions are therefore queued and processed sequentially by an event loop. Conceptually, the sequential execution flows from the server to the client and vice-versa and further user's interactions are queued while the server or the client is busy.

### 3.5 Recapitulation

Core Hop and Tilde Hop run on the server while Core JavaScript runs on the client. The semantics of Core Hop twines a regular Scheme-like evaluation and a JavaScript program creation much like a quasiquotation mechanism.

The initial continuation of this server computation expects a Core JavaScript program that is sent to the client and run. The Core JavaScript evaluation looks like a regular Scheme evaluation except that `with-hop` transfers the evaluation onto the server which returns to the client a new Core JavaScript program to be queued for later evaluation. The evaluation on the server and the client are therefore also twined together.

The formal semantics of Core Hop, Tilde Hop, and Core JavaScript are given in the next section.

## 4 Semantics

This section displays a greekified evaluator for this simplified Hop. We nearly follow the conventions used for the formal semantics section in R5RS [8] that is:

| | |
|---|---|
| $\rho[\nu \rightarrow \epsilon]$ | substitution "$\rho$ with $\epsilon$ for $\nu$" |
| $\theta \S \theta^*$ | adjoining $\theta$ in sequence $\theta^*$ |
| $\eta\downarrow_\theta$ | tuple projection |
| $<>$ | empty sequence |
| $<\theta>$ | sequence of one term |

The semantics of Core Hop, Tilde Hop, and Core JavaScript appear respectively on Fig. 5, Fig. 6 and Fig. 7. The Appendix presents the Scheme source code of the executable interpreter for Core Hop from which the semantics has been automatically extracted.

### 4.1 Core Hop

Core Hop looks like a regular Scheme dialect but for the presence of `gdef` and $\sim$-forms. The domains are indexed with S for the server-side, with C for the client-side, or with G when generating Core JavaScript programs.

In Fig. 5, the first three arguments of $\mathcal{S}$ (as in Server) are the usual ones: lexical environment $\rho_S$, global environment $\gamma_S$ and continuation $\kappa_S$. Fourth and fifth arguments are explained below.

There is no mutable value hence the absence of a store in the semantics. However, global variables are mutable (with `gdef`) so the global environment is given back to continuations.

$$
\begin{array}{rl}
\pi & \in \mathbf{CoreHop} \\
\mathcal{S} & : \mathbf{CoreHop} \rightarrow \mathbf{LexEnv_S} \times \mathbf{GlobEnv_S} \times \mathbf{Cont_S} \times \mathbf{LexEnv_G} \times \mathbf{State_C} \rightarrow \mathbf{Ans} \\
\nu & \in \mathbf{Var} \\
\varepsilon & \in \mathbf{Val_S} \\
\varphi & \in (\mathbf{Val_S} \times \mathbf{GlobEnv_S} \times \mathbf{Cont_S} \times \mathbf{State_C} \rightarrow \mathbf{Ans}) \subset \mathbf{Val_S} \\
\rho_S & \in \mathbf{LexEnv_S} = \mathbf{Var} \rightarrow \mathbf{Val_S} \\
\gamma_S & \in \mathbf{GlobEnv_S} = \mathbf{Var} \rightarrow \mathbf{Val_S} \\
\kappa_S & \in \mathbf{Cont_S} = \mathbf{Val_S} \times \mathbf{GlobEnv_S} \times \mathbf{State_C} \rightarrow \mathbf{Ans} \\
\eta_S & \in \mathbf{State_S} = \mathbf{LexEnv_S} \times \mathbf{GlobEnv_S} \\
\downarrow_\rho & : \mathbf{State_S} \rightarrow \mathbf{LexEnv_S} \\
\downarrow_\gamma & : \mathbf{State_S} \rightarrow \mathbf{GlobEnv_S}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{S}[\![\nu]\!]\rho_S\,\gamma_S\,\kappa_S\,\rho_G\,\eta_C = \kappa_S(\mathit{lookup}_S(\nu,\rho_S,\gamma_S))\,\gamma_S\,\eta_C \\
\mathcal{S}[\![\texttt{(gdef } \nu\ \pi\texttt{)}]\!]\rho_S\,\gamma_S\,\kappa_S\,\rho_G\,\eta_C = \mathcal{S}[\![\pi]\!]\rho_S\,\gamma_S\,(\lambda\varepsilon\gamma_{S1}\eta_{C1}.\kappa_S\varepsilon\,\gamma_{S1}[\nu\rightarrow\varepsilon]\,\eta_{C1})\,\rho_G\,\eta_C \\
\mathcal{S}[\![\texttt{(lambda (}\nu\texttt{) }\pi\texttt{)}]\!]\rho_S\,\gamma_S\,\kappa_S\,\rho_G\,\eta_C = \kappa_S(\lambda\varepsilon\gamma_{S1}\kappa_{S1}\eta_{C1}.\mathcal{S}[\![\pi]\!]\rho_S[\nu\rightarrow\varepsilon]\,\gamma_{S1}\,\kappa_{S1}\,\rho_G\,\eta_{C1})\,\gamma_S\,\eta_C \\
\mathcal{S}[\![\texttt{(}\pi\ \pi_1\texttt{)}]\!]\rho_S\,\gamma_S\,\kappa_S\,\rho_G\,\eta_C = \mathcal{S}[\![\pi]\!]\rho_S\,\gamma_S\,(\lambda\varphi\gamma_{S1}\eta_{C1}.\mathcal{S}[\![\pi_1]\!]\rho_S\,\gamma_{S1}\,(\lambda\varepsilon\gamma_{S2}\eta_{C2}.\varphi\varepsilon\,\gamma_{S2}\,\kappa_S\,\eta_{C2})\,\rho_G\,\eta_{C1})\,\rho_G\,\eta_C \\
\mathcal{S}[\![\texttt{(begin }\pi\ \pi_1\texttt{)}]\!]\rho_S\,\gamma_S\,\kappa_S\,\rho_G\,\eta_C = \mathcal{S}[\![\pi]\!]\rho_S\,\gamma_S\,(\lambda\varepsilon\gamma_{S1}\eta_{C1}.\mathcal{S}[\![\pi_1]\!]\rho_S\,\gamma_{S1}\,\kappa_S\,\rho_G\,\eta_{C1})\,\rho_G\,\eta_C \\
\mathcal{S}[\![\tilde{}\varpi]\!]\rho_S\,\gamma_S\,\kappa_S\,\rho_G\,\eta_C = \mathcal{G}[\![\varpi]\!]\rho_G\,(\lambda\xi\eta_S\eta_{C1}.\kappa_S\xi\,\eta_S\downarrow_\gamma\,\eta_{C1})\,(\mathit{state}_S(\rho_S,\gamma_S))\,\eta_C
\end{array}
$$

**Fig. 5** Core Hop semantics

$$
\begin{array}{ll}
\varpi & \in \mathbf{TildeHop} \\
\psi & \in \mathbf{TildeHop} \qquad \textit{callback} \\
\mathcal{G} & : \mathbf{TildeHop} \to \mathbf{LexEnv_G} \times \mathbf{Cont_G} \times \mathbf{State_S} \times \mathbf{State_C} \to \mathbf{Ans} \\
\xi & \in \mathbf{CoreJS} \subset \mathbf{Val_S} \\
\rho_{\mathrm{G}} & \in \mathbf{LexEnv_G} = \mathbf{Var} \to \mathbf{Val_S} \\
\kappa_G & \in \mathbf{Cont_G} = \mathbf{CoreJS} \times \mathbf{State_S} \times \mathbf{State_C} \to \mathbf{Ans} \\
\Pi & : \mathbf{Val_S} \to \mathbf{CoreJS}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![\nu]\!]\rho_{\mathrm{G}}\,\kappa_{\mathrm{G}}\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}} = \kappa_{\mathrm{G}}(\mathit{lookup}_{\mathrm{G}}(\nu,\rho_{\mathrm{G}}))\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}} \\
\mathcal{G}[\![\texttt{(gdef}\ \nu\ \varpi\texttt{)}]\!]\rho_{\mathrm{G}}\,\kappa_{\mathrm{G}}\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}} = \mathcal{G}[\![\varpi]\!]\rho_{\mathrm{G}}\,(\lambda\xi\eta_{\mathrm{S}1}\eta_{\mathrm{C}1}.\kappa_{\mathrm{G}}\ulcorner\texttt{(gdef}\ \nu\ \xi\texttt{)}\urcorner\,\eta_{\mathrm{S}1}\,\eta_{\mathrm{C}1})\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}} \\
\mathcal{G}[\![\texttt{(lambda (}\nu\texttt{)}\ \varpi\texttt{)}]\!]\rho_{\mathrm{G}}\,\kappa_{\mathrm{G}}\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}} = \\
\quad \mathcal{G}[\![\varpi]\!]\rho_{\mathrm{G}}[\nu \to \ulcorner\nu\urcorner]\,(\lambda\xi\eta_{\mathrm{S}1}\eta_{\mathrm{C}1}.\kappa_{\mathrm{G}}\ulcorner\texttt{(lambda (}\nu\texttt{)}\ \xi\texttt{)}\urcorner\,\eta_{\mathrm{S}1}\,\eta_{\mathrm{C}1})\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![\texttt{(}\varpi\ \varpi_1\texttt{)}]\!]\rho_{\mathrm{G}}\,\kappa_{\mathrm{G}}\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}} = \\
\quad \mathcal{G}[\![\varpi]\!]\rho_{\mathrm{G}}\,(\lambda\xi\eta_{\mathrm{S}1}\eta_{\mathrm{C}1}.\mathcal{G}[\![\varpi_1]\!]\rho_{\mathrm{G}}\,(\lambda\xi_1\eta_{\mathrm{S}2}\eta_{\mathrm{C}2}.\kappa_{\mathrm{G}}\ulcorner\texttt{(}\xi\ \xi_1\texttt{)}\urcorner\,\eta_{\mathrm{S}2}\,\eta_{\mathrm{C}2})\,\eta_{\mathrm{S}1}\,\eta_{\mathrm{C}1})\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![\texttt{(begin}\ \varpi\ \varpi_1\texttt{)}]\!]\rho_{\mathrm{G}}\,\kappa_{\mathrm{G}}\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}} = \\
\quad \mathcal{G}[\![\varpi]\!]\rho_{\mathrm{G}}\,(\lambda\xi\eta_{\mathrm{S}1}\eta_{\mathrm{C}1}.\mathcal{G}[\![\varpi_1]\!]\rho_{\mathrm{G}}\,(\lambda\xi_1\eta_{\mathrm{S}2}\eta_{\mathrm{C}2}.\kappa_{\mathrm{G}}\ulcorner\texttt{(begin}\ \xi\ \xi_1\texttt{)}\urcorner\,\eta_{\mathrm{S}2}\,\eta_{\mathrm{C}2})\,\eta_{\mathrm{S}1}\,\eta_{\mathrm{C}1})\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{G}[\![\$\pi]\!]\rho_{\mathrm{G}}\,\kappa_{\mathrm{G}}\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}} = \mathcal{S}[\![\pi]\!]\eta_{\mathrm{S}}{\downarrow_\rho}\ \eta_{\mathrm{S}}{\downarrow_\gamma}\ (\lambda\xi\gamma_{\mathrm{S}}\eta_{\mathrm{C}1}.\kappa_{\mathrm{G}}\Pi(\xi)\,(\mathit{state}_{\mathrm{S}}(\eta_{\mathrm{S}}{\downarrow_\rho},\gamma_{\mathrm{S}}))\,\eta_{\mathrm{C}1})\,\rho_{\mathrm{G}}\,\eta_{\mathrm{C}} \\
\mathcal{G}[\![\texttt{(with-hop (}\pi\ \varpi\texttt{)}\ \psi\texttt{)}]\!]\rho_{\mathrm{G}}\,\kappa_{\mathrm{G}}\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}} = \\
\quad \mathcal{G}[\![\varpi]\!]\rho_{\mathrm{G}}\,(\lambda\xi\eta_{\mathrm{S}1}\eta_{\mathrm{C}1}.\mathcal{G}[\![\psi]\!]\rho_{\mathrm{G}}\,(\lambda\xi_1\eta_{\mathrm{S}2}\eta_{\mathrm{C}2}.\kappa_{\mathrm{G}}\ulcorner\texttt{(with-hop (}\pi\ \xi\texttt{)}\ \xi_1\texttt{)}\urcorner\,\eta_{\mathrm{S}2}\,\eta_{\mathrm{C}2})\,\eta_{\mathrm{S}1}\,\eta_{\mathrm{C}1})\,\eta_{\mathrm{S}}\,\eta_{\mathrm{C}}
\end{array}
$$

**Fig. 6** Tilde Hop semantics

The $\mathit{lookup}_{\mathrm{S}}$ function (not shown) looks for the value of a variable $\nu$, first in the lexical environment $\rho_{\mathrm{S}}$ then in the global environment $\gamma_{\mathrm{S}}$.

While on the server, the constant state of the client is held in $\eta_{\mathrm{C}}$. Symmetrically the state of the Core Hop evaluator on the server (its lexical and global environments) is captured in $\eta_{\mathrm{S}}$ (with the $\mathit{state}_{\mathrm{S}}$ tuple-maker) while processing $\sim$-forms.

Conversely the lexical environment where $\sim$-forms are processed is kept in $\rho_{\mathrm{G}}$ while evaluating Core Hop forms.

## 4.2 Tilde Hop

Basically, forms are compiled (copied) into Core JavaScript (see Fig. 6) but for `$`-forms whose value (that should be a Core JavaScript fragment) is grafted into the currently built Core JavaScript program after being vetted by the $\Pi$ filter (not further explained).

The generated Core JavaScript programs appear between $\ulcorner$ and $\urcorner$ marks.

The environment $\rho_{\mathrm{G}}$ lists the lexical variables present in the generated `lambda` forms of Core JavaScript. This environment is preserved while processing `$`-forms since they may contain recursively additional $\sim$-forms. Similarly, the state of the server is held in $\eta_{\mathrm{S}}$ when not processing `$`-forms.

While processing Tilde Hop (and Core Hop as well), the constant state of the client is held in $\eta_{\mathrm{C}}$. This argument may be safely ignored when not considering the role of the client.

Continuations in Tilde Hop are noted $\kappa_{\mathrm{G}}$. They have a different type from those of Core Hop. A conversion of continuations appears in the rule for `$`-forms.

$$
\begin{array}{rl}
\mathcal{C} & : \mathbf{CoreJS} \to \mathbf{LexEnv_C} \times \mathbf{GlobEnv_C} \times \mathbf{Cont_C} \times \mathbf{Susp}^* \times \mathbf{State_S} \to \mathbf{Ans} \\
\nu & \in \mathbf{Var} \\
\epsilon & \in \mathbf{Val_C} \\
\phi & \in (\mathbf{Val_C} \times \mathbf{GlobEnv_C} \times \mathbf{Cont_C} \times \mathbf{Susp}^* \times \mathbf{State_S} \to \mathbf{Ans}) \subset \mathbf{Val_C} \\
\rho_C & \in \mathbf{LexEnv_C} = \mathbf{Var} \to \mathbf{Val_C} \\
\gamma_C & \in \mathbf{GlobEnv_C} = \mathbf{Var} \to \mathbf{Val_C} \\
\kappa_C & \in \mathbf{Cont_C} = \mathbf{Val_C} \times \mathbf{GlobEnv_C} \times \mathbf{Susp}^* \times \mathbf{State_S} \to \mathbf{Ans} \\
\eta_C & \in \mathbf{State_C} = \mathbf{GlobEnv_C} \times \mathbf{Susp}^* \\
\downarrow_\gamma & : \mathbf{State_C} \to \mathbf{GlobEnv_C} \\
\downarrow_{\theta^*} & : \mathbf{State_C} \to \mathbf{Susp}^* \\
\Sigma & : \mathbf{Val_C} \to \mathbf{Val_S}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}[\![\nu]\!]\rho_C\ \gamma_C\ \kappa_C\ \theta^*\ \eta_S\ =\ \kappa_C(\mathit{lookup}_C(\nu, \rho_C, \gamma_C))\ \gamma_C\ \theta^*\ \eta_S \\
\mathcal{C}[\![\texttt{(gdef}\ \ \nu\ \xi\texttt{)}]\!]\rho_C\ \gamma_C\ \kappa_C\ \theta^*\ \eta_S\ =\ \mathcal{C}[\![\xi]\!]\rho_C\ \gamma_C\ (\lambda\epsilon\gamma_{C1}\theta^*{}_1\eta_{S1}.\kappa_C\epsilon\ \gamma_{C1}[\nu \to \epsilon]\ \theta^*{}_1\ \eta_{S1})\ \theta^*\ \eta_S \\
\mathcal{C}[\![\texttt{(lambda}\ \ \texttt{(}\nu\texttt{)}\ \ \xi\texttt{)}]\!]\rho_C\ \gamma_C\ \kappa_C\ \theta^*\ \eta_S\ =\ \kappa_C(\lambda\epsilon\gamma_{C1}\kappa_{C1}\theta^*{}_1\eta_{S1}.\mathcal{C}[\![\xi]\!]\rho_C[\nu \to \epsilon]\ \gamma_{C1}\ \kappa_{C1}\ \theta^*{}_1\ \eta_{S1})\ \gamma_C\ \theta^*\ \eta_S \\
\mathcal{C}[\![\texttt{(}\xi\ \xi_1\texttt{)}]\!]\rho_C\ \gamma_C\ \kappa_C\ \theta^*\ \eta_S\ = \\
\quad \mathcal{C}[\![\xi]\!]\rho_C\ \gamma_C\ (\lambda\phi\gamma_{C1}\theta^*{}_1\eta_{S1}.\mathcal{C}[\![\xi_1]\!]\rho_C\ \gamma_{C1}\ (\lambda\epsilon\gamma_{C2}\theta^*{}_2\eta_{S2}.\phi\epsilon\ \gamma_{C2}\ \kappa_C\ \theta^*{}_2\ \eta_{S2})\ \theta^*{}_1\ \eta_{S1})\ \theta^*\ \eta_S
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}[\![\texttt{(begin}\ \ \xi\ \xi_1\texttt{)}]\!]\rho_C\ \gamma_C\ \kappa_C\ \theta^*\ \eta_S\ =\ \mathcal{C}[\![\xi]\!]\rho_C\ \gamma_C\ (\lambda\epsilon\gamma_{C1}\theta^*{}_1\eta_{S1}.\mathcal{C}[\![\xi_1]\!]\rho_C\ \gamma_{C1}\ \kappa_C\ \theta^*{}_1\ \eta_{S1})\ \theta^*\ \eta_S \\
\mathcal{C}[\![\texttt{(with-hop}\ \ \texttt{(}\pi\ \xi\texttt{)}\ \ \xi_1\texttt{)}]\!]\rho_C\ \gamma_C\ \kappa_C\ \theta^*\ \eta_S\ = \\
\quad \textbf{let}\ \kappa_{C1} = (\lambda\epsilon\gamma_{C1}\theta^*{}_1\eta_{S1}. \\
\qquad \textbf{let}\ \kappa_{C2} = (\lambda\phi\gamma_{C2}\theta^*{}_2\eta_{S2}. \\
\qquad\quad \textbf{let}\ \mathrm{f} = (\lambda\gamma_{C3}\theta^*{}_3\eta_{S3}. \\
\qquad\qquad \textbf{let}\ \kappa_S = (\lambda\varphi\gamma_S\eta_C. \\
\qquad\qquad\quad \textbf{let}\ \kappa_{S1} = (\lambda\xi_2\gamma_{S1}\eta_{C1}. \\
\qquad\qquad\qquad \textbf{let}\ \theta^*{}_4 =< \mathit{Susp}(\phi, \Pi(\xi_2)) > \S\eta_{C1}\downarrow_{\theta^*} \\
\qquad\qquad\qquad \textbf{in}\ \mathrm{LOOP_C}(\eta_{C1}\downarrow_\gamma, \theta^*{}_4, (\mathit{state}_S(\rho_S^0, \gamma_{S1})))\ ) \\
\qquad\qquad\quad \textbf{in}\ \varphi\Sigma(\epsilon)\ \gamma_S\ \kappa_{S1}\ \eta_C\ ) \\
\qquad\qquad \textbf{in}\ \mathcal{S}[\![\pi]\!]\eta_{S3}\downarrow_\rho\ \eta_{S3}\downarrow_\gamma\ \kappa_S\ \gamma_C^0\ \mathit{state}_C(\gamma_{C3}, \theta^*{}_3)\ ) \\
\qquad\quad \textbf{in}\ \kappa_C\ \mathit{unspecified}\ \ \gamma_{C2}\ < \mathit{Susp}(\mathrm{f}) > \S\theta^*{}_2\ \eta_{S2}\ ) \\
\qquad \textbf{in}\ \mathcal{C}[\![\xi_1]\!]\rho_C\ \gamma_{C1}\ \kappa_{C2}\ \theta^*{}_1\ \eta_{S1}\ ) \\
\quad \textbf{in}\ \mathcal{C}[\![\xi]\!]\rho_C\ \gamma_C\ \kappa_{C1}\ \theta^*\ \eta_S
\end{array}
$$

$$
\begin{array}{l}
\kappa_S^0\xi\ \gamma_S\ \eta_C\ = \mathrm{LOOP_C}(\eta_C\downarrow_\gamma, \eta_C\downarrow_{\theta^*}\ \S < \mathit{Susp}(\mathrm{id_C}, \Pi(\xi)) >, (\mathit{state}_S(\rho_S^0, \gamma_S))) \\
\mathrm{id_C}\epsilon\ \gamma_C\ \kappa_C\ \theta^*\ \eta_S\ = \kappa_C\epsilon\ \gamma_C\ \theta^*\ \eta_S
\end{array}
$$

**Fig. 7** Core JavaScript semantics

The function $\mathit{lookup}_G$ (not shown) searches in the $\rho_G$ environment a variable $\nu$. This search returns a client-side expression (a Core JavaScript reference to a variable) of the form $\ulcorner\nu\urcorner$ (see the `lambda` rule).

### 4.3 Core JavaScript

The semantics of Core JavaScript is again similar to Scheme with the addition of the `with-hop` form, see Fig. 7. The first three arguments of $\mathcal{C}$ (as in Client) are similar to those of $\mathcal{S}$: lexical environment $\rho_C$, global environment $\gamma_C$ and continuation $\kappa_C$. Two additional arguments will be explained below.

While evaluating Core JavaScript expressions (without `with-hop` forms), the constant state of the server is held in $\eta_S$ and may be safely ignored.

JavaScript is a sequential language. We model Core JavaScript as an engine that consumes a list of suspensions, see Fig. 8 for the $\mathrm{LOOP_C}$ definition. When the user interacts

$$
\begin{array}{rl}
\mathrm{LOOP_C} & : \mathbf{GlobEnv_C} \times \mathbf{Susp}^* \times \mathbf{State_S} \to \mathbf{Ans} \\
\theta & \in \mathbf{Susp} = \mathbf{Val_C} \times \mathbf{CoreJS} + (\mathbf{GlobEnv_C} \times \mathbf{Susp}^* \times \mathbf{State_S} \to \mathbf{Ans}) \\
\mathbf{Ans} & = \mathbf{CoreJS} \to \mathbf{Ans}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{LOOP_C}(\gamma_\mathrm{C}, <\phi, \xi> \S\theta^*, \eta_\mathrm{S}) = \\
\quad \mathbf{let}\ \kappa_\mathrm{C} = (\lambda\epsilon\gamma_{\mathrm{C}1}\theta^*{}_1\eta_{\mathrm{S}1}.\phi\epsilon\ \gamma_{\mathrm{C}1}\ (\lambda\epsilon_1\gamma_{\mathrm{C}2}\theta^*{}_2\eta_{\mathrm{S}2}.\mathrm{LOOP_C}(\gamma_{\mathrm{C}2}, \theta^*{}_2, \eta_{\mathrm{S}2}))\ \theta^*{}_1\ \eta_{\mathrm{S}1}) \\
\quad \mathbf{in}\ \mathcal{C}[\![\xi]\!]\rho_\mathrm{C}^0\ \gamma_\mathrm{C}\ \kappa_\mathrm{C}\ \theta^*\ \eta_\mathrm{S}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{LOOP_C}(\gamma_\mathrm{C}, <\mathrm{f}> \S\theta^*, \eta_\mathrm{S}) = \mathrm{f}\gamma_\mathrm{C}\ \theta^*\ \eta_\mathrm{S} \\
\mathrm{LOOP_C}(\gamma_\mathrm{C}, <>, \eta_\mathrm{S})\xi = \mathrm{LOOP_C}(\gamma_\mathrm{C}, <\mathit{Susp}(\mathrm{id_C}, \xi)>, \eta_\mathrm{S})
\end{array}
$$

**Fig. 8** Client-side event loop

with the browser, this interaction is converted into a suspension appended to the list of waiting suspensions. When a suspension is processed, its value is discarded and the next suspension is run. When the list of waiting suspensions is empty, the browser waits for new interactions.

A suspension is modeled as a pair made of a callback and a Core JavaScript program. When the program is evaluated (in the global environment of the client) its value is eventually given to the callback (a JavaScript closure). For a `with-hop`-form, the Core JavaScript program generated by the invoked service will be paired with the callback mentioned in the `with-hop`-form into a suspension.

Figure 4 details the evaluation of the `(with-hop (`$\pi$ $\xi$`)` $\xi_1$`)` form. First, the $\xi$ form is evaluated, whose value (a Core JavaScript value: $\epsilon$) is transferred to the server. Second, the $\xi_1$ form is evaluated whose value (a Core JavaScript function: $\phi$) is the callback. Third, the form $\pi$ (a Core Hop expression) is evaluated on the server. Its value should be a function that will be applied to $\Sigma(\epsilon)$ where $\Sigma$ transforms (and moves) a Core JavaScript value into a Core Hop value (not further explained). The result is a new Core JavaScript program ($\xi_2$) which is sent back to the client and packed into a suspension ($\mathbf{Susp}(\phi, \Pi(\xi_2))$) appended to the current list of suspensions $\eta_{\mathrm{C}1} \downarrow_{\theta^*}$. This suspension will be processed later, the $\xi_2$ expression will be evaluated yielding a value on which the callback $\phi$ will be applied.

In order to model the behavior of the browser, when the list of suspensions is empty, the final answer (of domain **Ans**) is a function that waits for a new user's interaction (a Core JavaScript expression) and restarts the event loop. Hence the recursive type for **Ans**.

The last topic of interest is the initial continuation $\kappa_\mathrm{S}^0$ (see Fig. 7) of the Core Hop program that packs its value (a Core JavaScript program) into a suspension on which the Core JavaScript event loop is started. The associated callback for this suspension is the Core JavaScript identity function named $id_\mathrm{C}$.

### 4.4 Wrap up

The form `with-hop` presented in Sect. 4 slightly differs from the one implemented in the Hop compiler. The syntax of the actual `with-hop` is `(with-hop` $\eta_1^s$ $\eta^s$`)` while the syntax of the `with-hop` used for the semantics is `(with-hop (`$\pi$ $\varpi_1$`)` $\varpi_2$`)`. In the actual Hop implementation the first argument $\eta_1^S$ is a service invocation. The $\eta_1^s$ expression should then yield a service value that is, a URL. For the sake of simplicity, the formal semantics elides services and represents them as global expressions. Then, its syntax slightly differs from the actual one. However, the two forms are comparable. Here follows a macro that implements the syntax used for the semantics in actual Hop syntax.

```
(define-macro (with-hop (pi varpi) varpik)
  `(with-hop ($(service (x) (,pi x)) ,varpi)
      ,varpik))
```

Mixing extensively GUI declarations and client-side expressions can lead to *spaghetti code* [11]. To avoid this bad programming style, Hop provides modules. A module is a compilation unit that may be either used for server code or client code. Modules *export* bindings and *import* other modules. A well-engineered Hop program consists of modules for the server, modules for the client, and HTML trees that merely contain one-line expressions that register callbacks or pop alert windows up. The goal of modules is to specify how values are bound to global identifiers. Since they play no role at run-time, modules are not visible from the dynamic semantics and then not addressed by this paper.

## 5 Related work

Several other languages address multi-tier programming. However, to our knowledge, Hop is the only one that relies on a model so closely related to traditional web programming. Invoking Hop services produce client-side expressions exactly as triggering XMLHttpRequests in Ajax applications produce new HTML expressions. Hop services respond full-fledged HTML trees (see the examples of Sect. 2) but for the sake of simplicity it has been assumed here that these HTML trees only contain a single `<SCRIPT>` node. In Hop, client-side programs are *values* of server-side computations. By contrast, all multi-tier programming languages we are aware of assume a traditional programming model where programs are entirely known at compile-time and deployed on a server and clients before they start.

Amongst these other systems, some address multi-tier web programming by extending the Java programming language.

**GWT** (the Google Web Toolkit) is a platform that allows Java-like programs to be executed on the web. GWT programs are partly compiled to JavaScript. GWT programs are static. They compile to classical desktop GUI programming such as advocated by the Java Swing library. GWT programs do not adopt the traditional web programming style. Invoking GWT methods does not produce new GWT expressions that should be evaluated by clients. To our knowledge no formal semantics has been given for GWT.

**Jif** [1, 2] extends the Java programming language with security type annotations. The Swift framework implements Jif. It uses the annotations to split Jif programs into a server part and a client part. Swift compilation enforces security which means that private information is unreachable by client-side programs. Swift generates GWT programs and thus Jif programs are as static as GWT programs. The formal studies describing Jif/Swift focus exclusively on security aspects: the type system and the security enforcement. However, they do not isolate the core language on which the system rests nor they present a formal dynamic semantics.

**Hilda** [19], as Jif, is a web environment that automatically partition multi-tier applications. It focuses on data-driven applications that run on top of a back-end data system. Hilda is a high-level declarative language closer to UML than to an algorithmic programming language. As such it is not directly related to Hop.

The RPC calculus [4] ($\lambda_{\text{RPC}}$) is an extension of the traditional $\lambda$-calculus used to model the **Links** programming language [3]. Several elements of $\lambda_{\text{RPC}}$ share similarities with Hop. In particular, its defunctionalization process transforms high-level expressions into forms

that only rely on functions similar to Hop's services. The request form introduced by the $\lambda_{RPC}$ Client-Server Machine is close to Hop's `with-hop` form. In spite of these similarities, the two studies cover different aspects of multi-tier programming. While $\lambda_{RPC}$ aims at modeling location-aware languages in the context of stateless servers, the Hop semantics aims at modeling dynamic client-side expressions production.

Semantics for multi-tier λ-calculus has been first studied in $\lambda_{MT}$ [13] which seems to have inspired both Jif and Links. It consists in a λ-calculus augmented with location annotations. A large amount of this work consists in introducing and presenting the implementation of the $\lambda_{MT}$ which is split by an automatic transformation. Hop relies on a different model: it is up to the programmer to choose where expressions are evaluated.

**ML5** [12] is a variant of ML dedicated to spatially distributed computing. It allows an entire multi-tier application to be developed and reasoned about as a unified program. ML5 is not theoretically restricted to two-sites applications although the current implementation is specialized for the web and it only supports applications distributed over a web server and a web browser. ML5's static type system is based on modal logic which permits simultaneous reasoning from multiple perspectives. The special form (`from/get`) allows any well-typed ML5 expression to be executed on a different host. Contrary to Hop, network traversals are thus not restricted to function calls. ML5 generates HTML trees using plain character strings. Because these trees contain ML5 expressions, the language provides the library function `say` that dynamically transforms a client-side ML5 expression into a string. This technique does not permit the semantics to accommodate the multi-stage programming needed for generating dynamically new client GUIs. This is another main difference with Hop.

**Flapjax** [10] is an extension of JavaScript that fosters functional reactive programming. No formal presentation of Flapjax has been given yet but FrTime, the spiritual father of Flapjax, has been described in a previous publication [5]. Event streams are ubiquitous in Flapjax. They are used to intercept GUI events (mouse click or keystroke) but also remote server calls. Merging and composing web services become easier than with traditional callback-based systems. Hence, Flapjax is convenient for programming web applications such as mashup. Although it provides some facilities to implement persistent data storage, Flapjax is client centric. It is not a multi-tier programming language because it does not address the programming of the server. This is a deep difference with Hop.

Hop is a sort of a multi-stage programming language [18] but, contrary to usual multi-stage languages such as **MacroML** [6] or **MetaOCaml**, a Hop program does not generate another Hop program. As presented in the semantics, Hop relies on different languages: core Hop and core JavaScript. The latter being generated by the former. Furthermore core JavaScript programs are first class values of core Hop which can then inspect and modify them dynamically. This strongly differentiates Hop from other multi-stage languages.

## 6 Conclusion and future work

This paper presents a denotational semantics for a simplified version of Hop, a Lisp dialect designed for programming the web. The formal semantics focuses on the main specificity of Hop: its *elaboration* of programs where client-side expressions are first-class values generated by server-side computations. To our knowledge, this is the first theoretical study that copes with this aspect of the multi-tier programming advocated by the web.

```
(define (S e)
  (if (pair? e)
      (case (car e)
        ((gdef) (Sgdef (cadr e) (caddr e)))
        ((lambda) (Slambda (cadr e) (caddr e)))
        ((begin) (Sbegin (cadr e) (caddr e)))
        ((tilde) (Stilde (cadr e)))
        (else (Scombination (car e) (cadr e))) )
      (if (symbol? e)
          (Svariable e)
          (Sconstant e) ) ) )
(define (Sconstant cst)
  (lambda (s.r s.g s.k g.r c.state)
    (s.k cst s.g c.state) ) )
(define (Svariable var)
  (lambda (s.r s.g s.k g.r c.state)
    (s.k (s.lookup var s.r s.g) s.g c.state) ) )
(define (Sgdef var e)
  (lambda (s.r s.g s.k g.r c.state)
    ((S e) s.r
           s.g
           (lambda (s.val s.g1 c.state1)
             (s.k s.val (extend s.g1 var s.val) c.state1) )
           g.r
           c.state ) ) )
(define (Slambda var e)
  (lambda (s.r s.g s.k g.r c.state)
    (s.k (lambda (s.val s.g1 s.k1 c.state1)
           ((S e) (extend s.r var s.val) s.g1 s.k1 g.r c.state1) )
         s.g
         c.state ) ) )
(define (Scombination e1 e2)
  (lambda (s.r s.g s.k g.r c.state)
    ((S e1) s.r
            s.g
            (lambda (s.phi s.g1 c.state1)
              ((S e2) s.r
                      s.g1
                      (lambda (s.val s.g2 c.state2)
                        (s.phi s.val s.g2 s.k c.state2) )
                      g.r
                      c.state1 ) )
            g.r
            c.state ) ) )
(define (Sbegin e1 e2)
  (lambda (s.r s.g s.k g.r c.state)
    ((S e1) s.r
            s.g
            (lambda (s.val s.g1 c.state1)
              ((S e2) s.r s.g1 s.k g.r c.state1) )
            g.r
            c.state ) ) )
(define (Stilde j)
  (lambda (s.r s.g s.k g.r c.state)
    ((G j) g.r
           (lambda (code s.state c.state)
             (s.k code (Sstate.g s.state) c.state) )
           (mkSstate s.r s.g)
           c.state ) ) )
```

**Fig. 9** The S evaluation function

```
(define (C c)
  (case (car c)
    ((Cconstant)    (Cconstant (cadr c)))
    ((Cvariable)    (Cvariable (cadr c)))
    ((Cgdef)        (Cgdef (cadr c) (caddr c)))
    ((Clambda)      (Clambda (cadr c) (caddr c)))
    ((Ccombination) (Ccombination (cadr c) (caddr c)))
    ((Cbegin)       (Cbegin (cadr c) (caddr c)))
    ((Cwith-hop)    (Cwith-hop (car (cadr c)) (cadr (cadr c)) (caddr c)))
    (else (error)) ) )
(define (Cconstant val)
  (lambda (c.r c.g c.k c.susp* s.state)
    (c.k val c.g c.susp* s.state) ) )
(define (Cvariable var)
  (lambda (c.r c.g c.k c.susp* s.state)
    (c.k (c.lookup var c.r c.g) c.g c.susp* s.state) ) )
(define (Cgdef var c)
  (lambda (c.r c.g c.k c.susp* s.state)
    ((C c) c.r
           c.g
           (lambda (c.val c.g1 c.susp1* s.state1)
             (c.k c.val (extend c.g1 var c.val) c.susp1* s.state1) )
           c.susp*
           s.state ) ) )
(define (Clambda var c)
  (lambda (c.r c.g c.k c.susp* s.state)
    (c.k (lambda (c.val c.g1 c.k1 c.susp1* s.state1)
           ((C c) (extend c.r var c.val)
                  c.g1
                  c.k1
                  c.susp1*
                  s.state1 ) )
         c.g
         c.susp*
         s.state ) ) )
(define (Ccombination c1 c2)
  (lambda (c.r c.g c.k c.susp* s.state)
    ((C c1) c.r
            c.g
            (lambda (c.phi c.g1 c.susp1* s.state1)
              ((C c2) c.r
                      c.g1
                      (lambda (c.val c.g2 c.susp2* s.state2)
                        (c.phi c.val c.g2 c.k c.susp2* s.state2) )
                      c.susp1*
                      s.state1 ) )
            c.susp*
            s.state ) ) )
(define (Cbegin c1 c2)
  (lambda (c.r c.g c.k c.susp* s.state)
    ((C c1) c.r
            c.g
            (lambda (c.val1 c.g1 c.susp1* s.state1)
              ((C c2) c.r
                      c.g1
                      c.k
                      c.susp1*
                      s.state1 ) )
            c.susp*
            s.state ) ) )
```

**Fig. 10** The C evaluation function

```
(define (G j)
  (if (pair? j)
      (case (car j)
        ((gdef) (Ggdef (cadr j) (caddr j)))
        ((lambda) (Glambda (cadr j) (caddr j)))
        ((begin) (Gbegin (cadr j) (caddr j)))
        ((dollar) (Gdollar (cadr j)))
        ((with-hop) (Gwith-hop (car (cadr j)) (cadr (cadr j)) (caddr j)))
        (else (Gcombination (car j) (cadr j))) )
      (if (symbol? j) (Gvariable j) (Gconstant j)) ) )
(define (Gconstant val)
  (lambda (g.r g.k s.state c.state)
    (g.k `(Cconstant ,val) s.state c.state) ) )
(define (Gvariable var)
  (lambda (g.r g.k s.state c.state)
    (g.k (g.lookup var g.r) s.state c.state) ) )
(define (Ggdef var j)
  (lambda (g.r g.k s.state c.state)
    ((G j) g.r
           (lambda (g.code s.state1 c.state1)
             (g.k `(Cgdef ,var ,g.code) s.state1 c.state1) )
           s.state
           c.state ) ) )
(define (Glambda var j)
  (lambda (g.r g.k s.state c.state)
    ((G j) (extend g.r var `(Cvariable ,var))
           (lambda (g.code s.state1 c.state1)
             (g.k `(Clambda ,var ,g.code) s.state1 c.state1) )
           s.state
           c.state ) ) )
(define (Gcombination j1 j2)
  (lambda (g.r g.k s.state c.state)
    ((G j1) g.r
            (lambda (g.code1 s.state1 c.state1)
              ((G j2) g.r
                      (lambda (g.code2 s.state2 c.state2)
                        (g.k `(Ccombination ,g.code1 ,g.code2)
                             s.state2
                             c.state2 ) )
                      s.state1
                      c.state1 ) )
            s.state
            c.state ) ) )
(define (Gbegin j1 j2)
  (lambda (g.r g.k s.state c.state)
    ((G j1) g.r
            (lambda (g.code1 s.state1 c.state1)
              ((G j2) g.r
                      (lambda (g.code2 s.state2 c.state2)
                        (g.k `(Cbegin ,g.code1 ,g.code2)
                             s.state2
                             c.state2 ) )
                      s.state1
                      c.state1 ) )
            s.state
            c.state ) ) )
```

**Fig. 11** The G evaluation function

```
(define (Gdollar e)
  (lambda (g.r g.k s.state c.state)
    ((S e) (Sstate.r s.state)
           (Sstate.g s.state)
           (lambda (code s.g1 c.state1)
             (g.k (cs2js code)
                  (mkSstate (Sstate.r s.state) s.g1)
                  c.state1 ) )
           g.r
           c.state ) ) )
(define (Gwith-hop e jarg jcb)
  (lambda (g.r g.k s.state c.state)
    ((G jarg) g.r
              (lambda (g.code1 s.state1 c.state1)
                ((G jcb) g.r
                         (lambda (g.code2 s.state2 c.state2)
                           (g.k `(Cwith-hop (,e ,g.code1) ,g.code2)
                                s.state2
                                c.state2 ) )
                         s.state1
                         c.state1 ) )
              s.state
              c.state ) ) )
```

**Fig. 12** The G evaluation function for `$` and `with-hop`

```
(define (Cwith-hop e carg ccb)
  (lambda (c.r c.g c.k c.susp* s.state)
    (let ((c.k1 (lambda (c.val c.g1 c.susp1* s.state1)
                  (let ((c.k2 (lambda (c.cbval c.g2 c.susp2* s.state2)
                                (let ((cf (lambda (c.g3 c.susp3* s.state3)
    ;; reindented!
      (let ((s.k3 (lambda (s.phi s.g3 c.state3)
                    (let ((s.k4 (lambda (prg s.g4 c.state4)
                                  (let ((susp* (cons (mkSusp c.cbval (cs2js prg))
                                                     (Cstate.susp* c.state4) )))
                                    (loop_c (Cstate.g c.state4)
                                            susp*
                                            (mkSstate s.r0 s.g4) ) ) )))
                      (s.phi (js2cs c.val) s.g3 s.k4 c.state3) ) )))
        ((S e) (Sstate.r s.state3)
               (Sstate.g s.state3)
               s.k3
               g.g0
               (mkCstate c.g3 c.susp3*) ) ) )))
    ;; end of reindentation!
                                  (c.k (unspecified)
                                       c.g2
                                       (cons (mkSusp cf) c.susp2*)
                                       s.state2 ) ))))
                    ((C ccb) c.r
                             c.g1
                             c.k2
                             c.susp1*
                             s.state1 ) ) )))
      ((C carg) c.r
                c.g
                c.k1
                c.susp*
                s.state ) ) ) )
```

**Fig. 13** Client-side `with-hop`

```
(define (s.lookup var r g)
  (r var g) )
(define (g.lookup var g.r)
  (g.r var g.g0) )
(define (c.lookup var c.r c.g)
  (c.r var c.g) )
(define (s.r0 var s.g)
  (s.g var s.g0) )
(define (s.g0 var s.g)
  (wrong "No such variable" var) )
(define (extend r pt img)
  (lambda (var g)
    (if (eq? var pt)
        img
        (r var g) ) ) )
(define (k0_s prg s.g c.state)
  (loop_c (Cstate.g c.state)
          (append (Cstate.susp* c.state)
                  (list (mkSusp id_c (cs2js prg))) )
          (mkSstate s.r0 s.g) ) )
(define (id_c c.val c.g c.k c.susp* s.state)
  (c.k c.val c.g c.susp* s.state) )
```

**Fig. 14** Initialization

```
(define (loop_c c.g c.susp* s.state)
  (if (pair? c.susp*)
      (if (= 1 (length (car c.susp*)))
          (loop_c_fun (susp->function (car c.susp*))
                      c.g (cdr c.susp*) s.state )
          (loop_c_susp (susp->callback (car c.susp*))
                       (susp->program (car c.susp*))
                       c.g (cdr c.susp*) s.state ) )
      (loop_c_empty c.g s.state) ) )
(define (loop_c_empty c.g s.state)
  (lambda (c)
    (loop_c c.g
            (list (mkSusp id_c c))
            s.state ) ) )
(define (loop_c_susp c.cbval cprg c.g c.susp* s.state)
  (let ((c.k (lambda (c.val1 c.g1 c.susp1* s.state1)
               (c.cbval
                c.val1
                c.g1
                (lambda (c.val2 c.g2 c.susp2* s.state2)
                  (loop_c c.g2 c.susp2* s.state2) )
                c.susp1*
                s.state1 ) )))
    ((C cprg)
     c.r0
     c.g
     c.k
     c.susp*
     s.state ) ) )
(define (loop_c_fun cf c.g c.susp* s.state)
  (cf c.g
      c.susp*
      s.state ) )
```

**Fig. 15** Client-side event loop

The presented semantics assumes a simplified version of Hop. It assumes a sequential evaluation that flows from servers to clients and vice-versa and it assumes that communications between clients and servers can only be initiated by clients. The actual Hop server-side programming supports concurrency features and it allows servers to *push* data towards clients. These two aspects have not been addressed here.

In the present study the marshaling and unmarshaling operations needed for invoking services has been considered as opaque and implemented by two external functions ($\Sigma$ and $\Pi$). Since HTML trees contain identifiers referenced to by the rest of the program, we think that the semantics of these operations should also be more deeply understood.

Formalizing a complete semantics for Hop will be the subject of future studies.

**Acknowledgements** Thanks for the reviewers whose help greatly improved this paper.

## Appendix

The semantics has been automatically extracted from an executable Scheme interpreter written in a denotational style. This interpreter is given here (Figs. 9–15).

## References

1. Chong, S., Liu, J., Myers, A.C., Qi, X., Vikram, K., Zheng, L., Zheng, X.: Building secure web applications with automatic partitioning. Commun. ACM **52**(2), 79–87 (2009). doi:10.1145/1461928.1461949
2. Chong, S., Vikram, K., Myers, A.C.: Sif: Enforcing confidentiality and integrity in web applications. In: Proc. 16th USENIX Security (2007)
3. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: 5th International Symposium on Formal Methods for Components and Objects (2006)
4. Cooper, E., Wadler, P.: The RPC calculus. In: International Conference on Principles and Practice of Declarative Programming, Coimbra, Portugal (2009)
5. Cooper, G., Krishnamurthi, S.: Embedding dynamic dataflow in a call-by-value language. In: Proceedings of the European Symposium on Programming (ESOP'06), pp. 194–308 (2006)
6. Ganz, S.E., Sabry, A., Taha, W.: Macros as multi-stage computations: Type-safe, generative, binding macros in macroml. In: International Conference on Functional Programming (ICFP'01), pp. 74–85. ACM Press (2001)
7. Jim, T., Swammy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: 16th International World Wide Web Conference (WWW 2007) (2007)
8. Kelsey, R., Clinger, W., Rees, J.: The revised(5) report on the algorithmic language Scheme. Higher-Order and Symbolic Computation **11**(1) (1998)
9. Loitsch, F., Serrano, M.: Hop client-side compilation. In: Morazán, M.T. (ed.) Trends in Functional Programming, vol. 8, pp. 141–158. Intellect, Bristol (2008)
10. Meyerovich, L., Guha, A., Baskin, J., Cooper, G., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: A programming language for Ajax applications. In: Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA'09). Orlando, FL, USA (2009)
11. Mikkonen, T., Taivalsaari, A.: Web applications—spaghetti code for the 21st century. Tech. Rep. SMLI TR-2007-166, Sun Microsystems (2007)
12. Murphy, T., Crary, K., Harper, R.: Type-safe distributed programming with ML5. In: Trustworthy Global Computing (2007). http://tom7.org/papers
13. Neubauer, M., Thiemann, P.: From sequential programs to multi-tier applications by program transformation. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05), pp. 221–232. ACM, New York (2005)
14. Queinnec, C.: Continuations and web servers. High.-Order Symb. Comput. **17**(4), 277–295 (2004)
15. Serrano, M.: HOP, a fast server for the diffuse web. In: Proceedings of the 11th International Conference on Coordination Models and Languages (COORDINATION'09, Lisbon, Portugal). LNCS, vol. 5521. Springer, Berlin (2009)

16. Serrano, M., Gallesio, E., Loitsch, F.: HOP, a language for programming the Web 2.0. In: Proceedings of the First Dynamic Languages Symposium. Portland, Oregon, USA (2006)
17. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge (1977)
18. Taha, W.: A gentle introduction to multi-stage programming. In: Domain-Specific Program Generation. LNCS, vol. 3016. Springer, Berlin (2004). doi:10.1007/b98156
19. Yang, F., et al.: A unified platform for data driven web applications with automatic client-server partitioning. In: 16th International World Wide Web Conference (WWW'07), pp. 341–350. Alberta, Canada (2007)