June 1, 1997

SRC Research Report 148

Service Combinators for Web Computing

Luca Cardelli
and
Rowan Davies

Service Combinators for Web Computing

Luca Cardelli and Rowan Davies

June 1, 1997

Author Affiliation

Rowan Davies is currently a Ph.D. student at Carnegie-Mellon University, School of Computer Science. He can be reached at `Rowan_Davies@cs.cmu.edu`.

Abstract

The World-Wide Web is rich in content and services, but access to these resources must be obtained mostly through manual browsers. We would like to be able to write programs that reproduce human browsing behavior, including reactions to slow transmission-rates and failures on many simultaneous links. We thus introduce a concurrent model that directly incorporates the notions of failure and rate of communication, and then describe programming constructs based on this model.

Contents

1 Introduction

The World-Wide Web [2] is a uniform, highly interconnected collection of computational resources, and as such it can be considered as forming a single global computer. But, what kind of computer is the Web, exactly? And what kind of languages are required for programming such a computer? Before approaching the second question, we must answer the first. In other words, what is the Web's model of computation?

1.1 Some Kind of Computer

We can quickly scan a checklist of possibilities. Is the Web a Von Neumann computer? Of course not: there is no stored program architecture, and no single instruction counter. Is the Web a collection of Von Neumann computers? Down below yes, but each computer is protected against outside access: its Von Neumann characteristics are not exploitable. Is the Web a file system? No, because there is no universally available "write" instruction (for obvious good reasons). Is the Web a distributed database? In many ways yes: it certainly contains a huge amount of information. But, on the one hand the Web lacks all the essential properties of distributed databases, such as precise data schemas, uniform query languages, distributed coherence, consistent replication, crash recovery, etc. On the other hand, the Web is more than a database, because answers to queries can be computed by non-trivial algorithms.

Is the Web a distributed object system? Now we are getting closer. Unfortunately the Web lacks some of the fundamental properties of traditional (in-memory, or local-area) object systems. The first problem that comes to mind is the lack of referential integrity: a pointer (URL[1]) on the Web does not always denote the same value as it did in a previous access. Even when a pointer denotes the same value, it does not always provide the same quality of access as it did in a previous access. Moreover, these pointers are subject to intermittent failures of various duration; while this is unpleasant, these failures are tolerated and do not negate the usefulness of the Web.

Most importantly, though, the Web does not work according to the Remote Procedure Call (RPC) semantics that is at the basis of distributed object systems. For example, if we could somehow replace HTTP[2] requests with RPC requests, we would drastically change the flavor of Web interactions. This is because the Web communication model relies on streaming data. A request results in a stream of data that is displayed interactively, as it is downloaded. A request does not block until it can produce an atomic result (in RPC style) that is displayed when the request is completed.

At a more abstract level, here are the main peculiarities of a Web computer, with respect to more familiar computational models. Three new classes of phenomena become observable:

1. Uniform Resource Locator [7, 8]

2. The HyperText Transfer Protocol is the Web's communication protocol [4, 7, 8].

- ***Wide-area distribution.*** Communication with distant locations involves a noticeable delay, and behavior may be location-dependent. This is much more dramatic than the distribution observable on a multiprocessor or a local-area network. It is not possible to build abstractions that hide this underlying reality, if only because the speed of light is a physical limit.
- ***Lack of referential integrity.*** A URL is a kind of network pointer, but it does not always point to the same entity, and occasionally it does not point at all. This is quite different from a pointer in a programming language.
- ***Quality of service.*** A URL is a "pointer with a bandwidth". The bandwidth of connections varies widely with time and route, and may influence algorithmic behavior.

A Web programmer will need to take these new observables into account. This calls for new programming models, and eventually new languages.

Therefore, there are no good names for describing the computational aspects of the Web. We might as well name such a computer a "Berners-Lee computer", after the inventor of HTTP. The model of computation of the Web is implicit in the HTTP protocol and in the Web's hardware and software infrastructure, but the implications of the interaction of the protocol and the infrastructure are not easy to grasp. The protocol is actually quite simple, but the infrastructure is likely to slow down, speed up, crash, stop, hang, and revive unpredictably. When the Web is seen as a computer, e.g., for the purpose of programming it, it is a very unusual computer.

1.2 Some Kind of Algorithms

What kind of activities can one carry out on such a strange computer? Here is an example of a typical behavior a user might exhibit.

Hal carries out a preliminary search for some document, and discovers that the document (say, a big postscript file) is available at four servers: in Japan, Australia, North America, and Europe. Hal does not want to start a parallel four-way download at first: it would be antisocial and, in any case, it might saturate his total incoming bandwidth. Hal first tries the North American server, but the server is overloaded and slow in downloading the data. So, he opens another browser window and contacts the European server. This server is much faster, initially, but suddenly the transfer rate drops to almost zero. Will the North American server catch up with it in the end? While he waits to find out, Hal remembers that it is night in Japan and Australia, so the servers should be unloaded and the intercontinental link should not be too congested. So he starts two more downloads. Japan immediately fails, but Australia starts crawling along. Now Hal notices that the European download has been totally idle for a few minutes so he kills it, and waits to see who wins out between Australia and North America.

What is described above is an instance of an "algorithmic" behavior that is used frequently for retrieving data. The decisions that determine the flow of the algorithm are based on the observable semantic properties of the Web: load, bandwidth, and even local time. The question is: what language could one use to comfortably program such an algorithm? An important criterion is that the language should be computationally complete with respect to the observable properties of the Web:

Every algorithmic behavior should be scriptable.

That is, if a user sitting in front of (say) a browser carries out a set of observations, decisions, and actions that are algorithmically describable, then it should be possible to write a program that emulates the same observation, decisions, and actions.

1.3 Some Kind of Run-Time System

The Web is one vast run-time system that, if we squint a bit, has many of the features of more conventional run-time systems.

There are atomic data structures (images, sounds, video), and compound data structures (HTML[3] documents, forms, tables, multipart data), as described by various Internet standards. There are pointers (URLs) into a universal address space. There are graph structures (MIME[4] multipart/related format) that can be used to transmit complex data. There is a standardized type system for data layout (MIME media types). There are subroutine calls and parameter passing conventions (HTTP and CGI[5]). There are plenty of available processors (Web servers) that can be seen as distributed objects that protect and dispense encapsulated data (e.g., local databases). Finally, there are some nice visual debuggers (Web browsers).

What programming language features could correspond to this run-time system? What could a “Web language” look like? Let's try to imagine it.

A “value” in a Web language would be a pair of a MIME media type and an appropriate content. (For example, the media type may be image/jpeg and the content would be a jpeg-encoded image). These values could be stored in variables, passed to procedures, etc. Note that the contents of such values may be in the process of being fetched, so values are naturally concurrently evaluated. Other kinds of values would include gateways and scripts (see below).

The syntax of a programming language usually begins with the description of the “literals”: the entities directly denoting a value, e.g., a numeral or a string. A “media literal” would be a pair of a media type and a URL indicating the corresponding content. Such a literal would be evaluated to a value by fetching the URL content (and verifying that it corresponds to the claimed media type).

A “gateway literal” would be a pair of a Gateway Type and a URL indicating a gateway (e.g., a CGI gateway). The gateway type indicates the parameter passing conventions expected by the gateway (e.g., GET, POST, or ISINDEX) and the media types

3. HyperText Markup Language [3, 7]
4. Multi-purpose Internet Mail Extension [1, 7]
5. Common Gateway Interface [7]

for the requests and replies. A gateway literal evaluates to a gateway value, which just sits there waiting to be activated.

A gateway value can be activated by giving it its required parameters. The syntax for such an activation would look like a normal procedure call: $g(a_1, ..., a_n)$ where g is a literal, variable, or expression that produces a gateway value, and the arguments are (normally) media values. The effect of this call is to package the arguments according to the conventions of the gateway, ship them through the appropriate HTTP connection, get the result, and convert it back to a value. The final value may be rendered according to its type.

We now have primitive data structures (media literals) and primitive control structures (gateway calls). With this much we can already write “scripts”. These scripts could be stored on the Web as Internet Media, so that a script can refer to another one through a URL. The syntax for script calls would be the same as above, but might be a media literal that refers to a script, instead of a gateway. Scripts would have to be closed (i.e. no free variables, except for URLs), for security and network transparency.

This is arguably a Web language. The scripts are for the Web (not for a particular operating system or file system) and in the Web (not stored in a particular address space or file). Such a language uses the Web as its run-time system.

1.4 Other Issues

Two major issues remain to be addressed.

The first issue is output parsing. Because of the prominence of browsers and browser-ready content on the Web, the result of a query is almost always returned as an entity of type text/html (a page), even when its only purpose is to present, say, a single datum of type image/jpeg. The output has to be parsed to extract the information out of the HTML. Although the structure of HTML pages is relatively well defined, the parsing process is delicate, error-prone, and can be foiled by cosmetic changes in the pages. In order to make Web programming possible on a large scale, one would need some uniform way of describing the protocol of a gateway and, by extension, of a script. This problem is still unsolved and we will not discuss it here further.

The second issue is the design of control structures able to survive the flaky connectivity of the Web. This is the topic of the rest of the paper.

2 Service Algebra

Suppose we want to write a program that accesses and manipulates data on the Web. An obvious starting point is an HTTP library, embedded in some programming language, that gives us the ability to issue HTTP calls. Each HTTP call can fail with fairly high probability; therefore, error-handling code must be written using the error-handling primitives of the language. If we want to write code that reacts concurrently to network conditions and network failures in interesting ways, then the error-handling code ends up dominating the information-processing code. The error-handling and concurrency primitives of common languages are not very convenient when the exceptional code exceeds the normal code.

An alternative is to try to use high-level primitives that incorporate error handling and concurrency, and that are optimized for Web programming. In this section we introduce such primitives. A service is an HTTP information provider wrapped in error-detection and handling code. A service combinator is an operator for composing services, both in terms of their information output and of their error output, and possibly involving concurrency. The error recovery policy and concurrency are thus modularly embedded inside each service.

The idea of handling failures with combinators comes, in a sequential context, from LCF tactics [5].

2.1 Services

A Web server is an unreliable provider of data: any request for a service has a relatively high probability of failing or of being unacceptably slow. Different servers, though, may provide the same or similar services. Therefore it should be possible to combine unreliable services to obtain more reliable "virtual services".

A service, when invoked, may never initiate a response. If it initiates a response, it may never complete it. If it completes a response, it may respond "service denied", or produce a real answer in the form of a stream of data.

In the period of time between a request and the end of a response, the main datum of interest is the "transmission rate", counted as bytes per second averaged over an interval. It is interesting to notice that the basic communication protocol of the Internet does not provide direct data about the transmission rate: this must be estimated from the outside.

2.2 Service Combinators

We now describe the syntax and informal semantics of the service combinators in our language. The combinators were chosen to allow common manual Web-browsing techniques to be reproduced with simple programs.

The syntax for our language is given below in BNF-like notation. We use curly brackets { } for grouping, square brackets [ ] for zero or one occurrences, postfix * for zero or more occurrences, postfix + for one or more occurrences, infix | for disjunction, and simple juxtaposition for concatenation. We use '|' to indicate an occurrence of | in the language itself. For lexical items, [c_1-c_2] indicates a character in the range c_1-c_2.

Services

S ::=

url(*String*) | S_1 ? S_2 | S_1 ‘|’ S_2 | *timeout*(*Real*, *S*) |

limit($Real_1$, $Real_2$, *S*) | *repeat*(*S*) | *stall* | *fail* |

index($String_1$, $String_2$) |

gateway G (*String*, {*Id*=*String*}*)

Gateway types

G ::= *get* | *post*

Lexical items

String ::= " *StringChar** "

StringChar ::=

any single legal character other than ' " \

or one of the pairs of characters \' \" \\

Id ::= {[A-Z] | [a-z] | [0-9]}*

Real ::= [~] *Digit*+ [. *Digit*+]

Digit ::= [0-9]

The basic model for the semantics of services is as follows: a service may be invoked at any time, and may be invoked multiple times. An invocation will either succeed and return a result after some time, or fail after some time, or continue forever. At each point in time it has a rate which is a real number indicating how fast it is progressing.

Basic Service

url(*String*)

The service *url*(*String*) fetches the resource associated with the URL indicated by the string. The result returned is the content fetched. The service fails if the fetch fails, and the rate of the service while it is running is the rate at which the data for the resource is being received, measured in kilobytes per second.

Gateways

index(*String*, $String_1$)

gateway get (*String*, Id_1=$String_1$ ... Id_n=$String_n$)

gateway post (*String*, Id_1=$String_1$ ... Id_n=$String_n$)

Each of these services is similar to the service *url*(*String*), except that the URL *String* should be associated with a CGI gateway having the corresponding type (*index*, *get* or *post*). The arguments are passed to the gateway according to the protocol for this gateway type.

Sequential Execution

S_1 ? S_2

The "?" combinator allows a secondary service to be consulted in the case that the primary service fails for some reason. Thus, the service S_1 ? S_2 acts like the service S_1, except that if S_1 fails then it acts like the service S_2.

Concurrent Execution

$S_1 \mid S_2$

The "|" combinator allows two services to be executed concurrently. The service $S_1 \mid S_2$ starts both services S_1 and S_2 at the same time, and returns the result of whichever succeeds first. If both S_1 and S_2 fail, then the combined service also fails. The rate of the combined service is always the maximum of the rates of S_1 and S_2.

Time Limit

timeout(t, S)

The *timeout* combinator allows a time limit to be placed on a service. The service *timeout*(t, S) acts like S except that it fails after t seconds if S has not completed within that time.

Rate Limit

limit(t, r, S)

This combinator provides a way to force a service to fail if the rate ever drops below a certain limit r. A start-up time of t seconds is allowed, since generally it takes some time before a service begins receiving any data.

In our original design, this start-up time was applied to the whole service S. We later realized that this design leads to an unfortunate interaction with some of the other combinators. This is demonstrated by the example: *limit*(t, r, (S_1 ? S_2)). The problem here is that if S_1 fails after the first t seconds, then S_2 is initiated but is not allowed any start-up time, so quite likely the whole service fails.

This motivates the following semantics. The service *limit*(t, r, S) acts like the service S, except that each physical connection is considered to have failed if the rate ever drops below r *Kbytes*/*sec* after the first t seconds of the connection. Physical connections are created by invocations of *url*, *index* and *gateway* combinators.

In general, a rate limit can be described as a function f from time to rate, and a combinator *limit*(f, S) could be used; the current combinator could then be defined via a step function. The more general combinator is supported by our semantics, but we decided to adopt the current, simpler, definition.

Repetition

repeat(S)

The *repeat* combinator provides a way to repeatedly invoke a service until it succeeds. The service *repeat*(S) acts like S, except that if S fails, *repeat*(S) starts again.

Unlike many traditional languages, the repeat combinator does not include a condition for terminating the loop. Instead, the loop can be terminated in other ways, e.g., *timeout*(*t*, *repeat*(*S*)).

Non-termination

stall

The *stall* combinator never completes or fails and always has a rate of zero. The following examples show how this can be useful.

timeout(10, *stall*) ? *S*

This program waits 10 seconds before starting *S*.

repeat(*url*("http://www.cs.cmu.edu/~rowan") ? *timeout*(10, *stall*))

This program repeatedly tries to fetch the URL, but waits 10 seconds between attempts.

Failure

fail

The *fail* combinator fails immediately. It is hard to construct examples in our small language where this is useful, though we include it anyway for completeness, and because we expect it to be useful when the language is extended to include conditionals and other more traditional programming language constructs.

2.3 Examples

We now show some simple examples to illustrate the expressiveness of the service combinators. It is our intention that our service combinators be included as a fragment of a larger language, so for these examples (and in our implementation) we include some extensions. We use "*let*" to make top-level bindings, and we use "*fun*(x) body" and "function(argument)" for function abstraction and application. It is not completely clear how to define the semantics for these extensions in terms of the service model used above. If we are going to very Web-oriented, then perhaps functions should be implemented as gateways, and bound variables should actually refer to dynamically allocated URLs. Regardless, for the simple examples which follow, the meaning should be clear.

Example 1

url("http://www.cs.cmu.edu/")

This program simply attempts to fetch the named URL.

Example 2

gateway get("http://www.altavista.digital.com/cgi-bin/query",
pg="q" what="web" q="java")

This program looks up the word "java" on the AltaVista search engine.

Example 3

```
url("http://www.cs.umd.edu/~pugh/popl97/") |
url("http://www.diku.dk/popl97/")
```

This program attempts to fetch the POPL'97 conference page from one of two alternate sites. Both sites are attempted concurrently, and the result is that from whichever site successfully completes first.

Example 4

```
repeat(limit(1, 1, url("http://www7.conf.au/"))) |
(timeout(20, stall) ? url("http://www.cs.cmu.edu/~rowan/failed.txt")
```

This program attempts to fetch the WWW7 conference page from Australia. If the fetch fails or the rate ever drops below 1 *Kbytes*/*sec*, then it starts again. If the page is not successfully fetched within 20 seconds, then a site known to be easily reachable is used to retrieve a failure message.

Example 5

```
let av = fun(x)
    gateway get("http://www.altavista.digital.com/cgi-bin/query",
        pg="q" what="web" q=x)
let hb = fun(x)
    gateway get("http://www.HotBot.com/search.html",
        ... MT=x ... )            (large number of other parameters omitted)
let avhb = fun(x) av(x) | hb(x)
avhb("java")
```

This program defines two functions for looking up search strings on AltaVista and HotBot, and a single function which tries both concurrently, returning whichever succeeds first. It then uses this function to lookup the word “java”, to see which engine performs this task the fastest.

Example 6

```
let dbc = fun(ticker)
    gateway post("http://www.dbc.com/cgi-bin/htx.exe/squote",
        source="dbcc" TICKER=ticker format="decimals" tables="table")
let grayfire = fun(ticker)
    index("http://www.grayfire.com/cgi-bin/get-price", ticker)
let getquote = fun(ticker) repeat(grayfire(ticker) ? dbc(ticker))
getquote("DEC")
```

This program defines two functions for looking up stock quotes based on two different gateways. It then defines a very reliable function which makes repeated attempts in the case of failure, alternating between the gateways. It then uses this function to lookup the quote for Digital Equipment Corporation.

3 Formal Semantics

We now give a formal semantics for the service combinators.

3.1 The Meaning Function

The basic idea of the semantics is to define the status of a service at a particular time u, given the starting time t. Possible values for this status are $\langle rate, r\rangle$, $\langle done, c\rangle$, and $\langle fail\rangle$, where r is the rate of a service in progress, and c is the content returned by a successful service.

The *limit* combinator does not immediately fit into this framework. We handle it by introducing an additional parameter in the semantics that is a function that indicates the minimum rate that satisfies all applicable rate limits, in terms of the duration since a connection was started.

Thus our semantics is based on a meaning function M with four arguments: a service, a start time, a status time, and a rate limit function.

The meaning function implicitly depends on the state of the Web at any time. Instead of building a mathematical model of the whole Web, we assume that a *url* query returns an arbitrary but fixed result that, in reality, depends on the state of the Web at the time of the query.

A complication arises from the fact that Web queries started at the same time with the same parameters may not return the same value. For example, two identical *url* queries could reach a server at different times and fetch different versions of a page; moreover, two identical gateway queries may return pages that contain different hit counters. For simplicity, to make M deterministic, we assume the existence of an "instantaneous caching proxy" that caches, for an instant, the result of any query initiated at that instant. That is, we assume that $url(String) \mid url(String) = url(String)$, while we do not assume that $timeout(t, stall) \mathbin{?} url(String) = url(String)$ for any $t > 0$.

The meaning function is defined compositionally on the first argument as follows:

$$M(stall, t, u, f) = \langle rate, 0\rangle$$

$$M(fail, t, u, f) = \langle fail\rangle$$

$$\begin{aligned}
&M(S_1 ? S_2, t, u, f) = \\
&\qquad M(S_2, v_1, u, f) \qquad \text{if } M(S_1, t, u, f) = \langle fail\rangle \\
&\qquad M(S_1, t, u, f) \qquad \text{otherwise} \\
&\qquad \text{where } v_1 = inf\,\{v \mid M(S_1, t, v, f) = \langle fail\rangle\} \quad \text{(i.e. the time at which } S_1 \text{ fails)}
\end{aligned}$$

$M(S_1 \mid S_2, t, u, f) =$

$$\begin{array}{ll}
\langle rate, max(r_1, r_2)\rangle & \text{if } s_1 = \langle rate, r_1\rangle \text{ and } s_2 = \langle rate, r_2\rangle \\
\langle rate, r_1\rangle & \text{if } s_1 = \langle rate, r_1\rangle \text{ and } s_2 = \langle fail\rangle \\
\langle rate, r_2\rangle & \text{if } s_2 = \langle rate, r_2\rangle \text{ and } s_1 = \langle fail\rangle \\
\langle done, c_1\rangle & \text{if } s_1 = \langle done, c_1\rangle \text{ and } (s_2 = \langle rate, r_2\rangle \text{ or } s_2 = \langle fail\rangle \text{ or } v_1 \le v_2) \\
\langle done, c_2\rangle & \text{if } s_2 = \langle done, c_2\rangle \text{ and } (s_1 = \langle rate, r_1\rangle \text{ or } s_1 = \langle fail\rangle \text{ or } v_2 < v_1) \\
\langle fail\rangle & \text{if } s_1 = \langle fail\rangle \text{ and } s_2 = \langle fail\rangle
\end{array}$$

where $s_1 = M(S_1, t, u, f)$
and $s_2 = M(S_2, t, u, f)$
and $v_1 = inf\{v \mid M(S_1, t, v, f) = \langle done, c_1\rangle\}$
and $v_2 = inf\{v \mid M(S_2, t, v, f) = \langle done, c_2\rangle\}$

$M(timeout(v, S), t, u, f) =$

$$\begin{array}{ll}
M(S, t, u, f) & \text{if } u - t < v \\
\langle fail\rangle & \text{otherwise}
\end{array}$$

$M(limit(v, r, S), t, u, f) = M(S, t, u, g)$
where $g(v') = max(f(v'), h(v'))$
and $h(v') =$

$$\begin{array}{ll}
0 & \text{if } v' < v \\
r & \text{if } v' \ge v
\end{array}$$

$M(repeat(S), t, u, f) =$

$$\begin{array}{ll}
\langle rate, 0\rangle & \text{if } u \ge v_n \text{ for all } n \ge 0 \\
M(S, v_n, u, f) & \text{if } v_n \le u < v_{n+1}
\end{array}$$

where (with $inf\{\} =$ infinity)
$v_0 = t$
$v_{m+1} = inf\{v \mid v \ge v_m \text{ and } M(S, v_m, v, f) = \langle fail\rangle\}$

$M(url(String), t, u, f) =$

$\langle done, c\rangle$	if a connection fetching URL *String* at time t succeeds before time u with content c.
$\langle fail\rangle$	if there exists u' s.t. $u' \ge t$ and $u' \le u$ and a connection fetching URL *String* at time t fails at time u' or has rate r' at time u', with $r' < f(u'-t)$
$\langle rate, r\rangle$	otherwise, if a connection fetching URL *String* at time t has rate r at time u

The semantics for *gateway* is essentially the same as for *url*.

A basic property of this semantics, which can be proven by structural induction, is that if $M(S, t, u, f) = R$, with $R = \langle fail\rangle$ or $R = \langle done, c\rangle$ for some c, then for all $u' \ge u$, $M(S, t, u', f) = R$.

3.2 Algebraic Properties

Our semantics can be used to prove algebraic properties of the combinators. In turn, these properties could be used to transform and optimize Web queries.

We define:

$$S = S' \quad \text{iff} \quad \forall t, u{\geq}t, f.\ M(S, t, u, f) = M(S', t, u, f)$$

Simple properties can be easily derived, for example:

$fail \ ?\ S = S \ ?\ fail = S$
$fail \mid S = S \mid fail = S$
$stall \ ?\ S = stall$
$S \ ?\ stall = S \mid stall$

An interesting observation is that our semantics equates the services *repeat*(*fail*) and *stall*. However, it is still useful to include *stall* in the language, since the obvious implementation will be inefficient for the service *repeat*(*fail*). Conversely, we could consider eliminating *fail* in favor of *timeout*(0, *stall*).

$stall = repeat(fail)$
$fail = timeout(0, S)$

A range of other equations can be derived:

$(S \mid S) = S$
$(S_1 \mid S_2) \mid S_3 = S_1 \mid (S_2 \mid S_3)$
$(S_1 \ ?\ S_2) \ ?\ S_3 = S_1 \ ?\ (S_2 \ ?\ S_3)$

$repeat(S) = S \ ?\ repeat(S) = repeat(S \ ?\ S) = repeat(S \ ?\ ... \ ?\ S)$
$repeat(S) = repeat(S) \ ?\ S'$

$timeout(t, limit(u, r, S)) = timeout(t, S)$ if $t \leq u$

$limit(t, r, S \mid S') = limit(t, r, S) \mid limit(t, r, S')$
$limit(t, r, S \ ?\ S') = limit(t, r, S) \ ?\ limit(t, r, S')$
$limit(t, r, timeout(u, S)) = timeout(u, limit(t, r, S))$
$limit(t, r, repeat(S)) = repeat(limit(t, r, S))$
$limit(t, r, stall) = stall$
$limit(t, r, fail) = fail$

Other "intuitive" properties can be checked against the semantics, and sometimes we may discover they are not satisfied. For example, $S \mid S' \neq S' \mid S$, because the $\mid$ operators asymmetrically pick one result if two results are obtained at exactly the same time.

4 Implementation

We have implemented an interpreter for the language of service combinators, including top-level definitions and functions as used in the examples. This implementation is written in Java [6].

The implementation also provides an easy interface for programming with service combinators directly from Java. Services are defined by the abstract class Service, declared as follows:

```
public abstract class Service {
    public Content getContent(FuncTime tf);
    public float getRate();
    public void stop();   }
```

To invoke a service, the getContent method is passed a function of time which determines the minimum rate for physical connections, exactly as in the formal semantics. At the top-level, the function ZeroThreshold is normally used, indicating no minimum rate. This method returns the content when the service completes, or returns null when the service fails. During the invocation of the service, the current rate of the service can be found using a concurrent call to the getRate method. Also, the current invocation can be aborted by calling the stop method.

The various service combinators are provided as Java classes, whose constructors take sub-services as arguments. For example, the following Java code corresponds to example 3:

```
new Par(
    new Media("http://www.cs.umd.edu/~pugh/popl97/"),
    new Media ("http://www.diku.dk/popl97/"))
```

This technique could also be used to define interfaces for other domain specific languages within general purpose languages. Essentially the technique is to provide an interface to the abstract syntax representation and the interpreter. If the interpreter is object-oriented, then it will be actually built into the abstract-syntax classes as methods. This is more efficient than providing an interface to the whole interpreter using strings, and it avoids parsing and lexing errors at run-time.

In some sense the implementation is only an approximation to the formal semantics because the semantics ignores interpretation overhead. However, it is quite a close approximation, since interpretation overhead is very small for most programs compared to the time for data to be transmitted.

The rate of a basic service is defined to be the average over the previous two seconds, as calculated by samples done five times a second. This appears to give good results in practice, though admittedly it is somewhat ad hoc.

The implementation uses the Sun Java classes to fetch URLs. A small modification was made to them so that failures are correctly detected, since they normally catch failures and instead return an error page.

5 Conclusions and Future Directions

We have shown that a simple language allows easy expression of common strategies for handling failure and slow communication when fetching content on the Web. We have defined such a language based on service combinators, implemented it, and given a formal semantics for it.

Our intention is that our language will be extended to a more powerful Web-scripting language. Such a language would include some common language features such as functions and conditionals. It should also include additional features for Web-programming beyond our service combinators, for example special constructs for manipulating HTML content, possibly linked to a browser. Another direction is to allow scripts themselves to be stored on the Web, and in our implementation we have experimented with this. It should also be possible to write scripts which provide content on the Web, and perhaps even export a function as a CGI gateway. A full Web-scripting language might even allow a thread of execution to migrate via the Web.

A language with all these features would certainly be very powerful and useful. In this paper we have concentrated only on one particular aspect which is unique to the Web, namely its unreliable nature. By first considering the fundamental properties of the Web we have built a small language whose computation model is tailored for Web programming. We hope that this language and model will serve as a firm foundation for larger Web scripting languages. Elements of our language design and formal semantics should also be useful to designers of other domain specific languages in domains which include real-time concerns or where failures are common.

Acknowledgments

Luca Cardelli would like to thank David Jefferson and a group of people at Digital Palo Alto who met to discuss Web-related issues. Those discussions provided the initial stimulus for this work.

Rowan Davies would like to thank Digital SRC, including all its staff, for providing the opportunity to work in a exceptionally stimulating and pleasant environment during the summer of 1996.

Martín Abadi and Paul McJones reviewed early drafts.

References

[1] Borenstein, N., and N. Freed, **MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies**, RFC 1521, Bellcore, Innosoft, September 1993.

[2] Berners-Lee, T., R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret: **The World-Wide Web**. CACM 37(8): 76-82, 1994.

[3] Berners-Lee, T., and D. Connolly, **Hypertext Markup Language - 2.0**, RFC 1866, MIT/W3C, November 1995.

[4] Berners-Lee, T., R. Fielding, and H. Frystyk, **Hypertext Transfer Protocol - HTTP/1.0**, RFC 1945, MIT/UC Irvine, May 1996.

[5] Gordon, M., R. Milner, and C. Wadsworth, **Edinburgh LCF**. Lecture Notes in Computer Science 78. Springer-Verlag. 1979.

[6] Gosling, J., B. Joy, and G. Steele, **The Java Language Specification**. Addison-Wesley. 1996.

[7] Internet Engineering Task Force, **Internet Standards**. The Internet Society, <http://www.isoc.org>. 1997.

[8] World Wide Web Consortium, **HTTP - Hypertext Transfer Protocol**, <http://www.w3.org/pub/WWW/Protocols/>. 1997.