Programming Language Features for Web Application Development

Ezra Cooper

Doctor of Philosophy

School of Informatics

University of Edinburgh

2009

Abstract

Web programming remains difficult, even with cutting-edge libraries, because the execution model of the web environment is essentially different from the classic models. Unlike a batch program which sits between input and output streams, a web program sits between user activity in the browser and server-side resources such as a database. Furthermore the use of URLs as durable entry points to an application makes the environment fundamentally concurrent and re-entrant, a challenge and opportunity for supporting web programmers.

This thesis makes four principal contributions to the technology for expressing web applications. First, it describes the features of Links, a new programming language with a unified model of the web environment, encompassing client and server. Among other things, the Links compiler can slice the program, generating JavaScript to run on the client and other code to run on the server, so that they interact transparently. To allow programmers to control the location of code, Links offers syntactic client and server annotations. The second contribution is a formal semantics of these client/server annotations, in the form of an "RPC calculus." Along with the calculus is provided a compilation technique that shows how these location annotations can be implemented in the web's asymmetrical client/server setting, where the server acts only in response to the client's requests. The third contribution is a description of a language feature, 'formlets,' which is an abstraction of HTML forms; as an abstraction, it allows reusing bundles of form elements, composing them hierarchically, and viewing their submitted data at an appropriate abstract type. And the fourth and final major contribution shows how to integrate relational database query expressions into a programming language while also extending those queries to allow nested data structures and functional abstraction.

Contents

Declaration

I declare that this thesis was composed by myself, that I was part of a research group and that I personally made a substantial contribution to the work, as indicated in each chapter, and that this work has not been submitted for any other degree or professional qualification.

Ezra Cooper

Acknowledgements

This thesis exists only because of the people who have had an unreasonable faith in me and so gave me the bedrock and the freedom to complete it. Among them are my teachers: George Wolfe, who introduced me to the joy of science and its method; Rick Hendricks, whose brave, insouciant approach armored me; Framji Minwalla, who taught me that "you never really understand something until you articulate your view and are told that you're wrong"; Terry Osborne, who taught me to write in the large; Bob Gray, my undergraduate thesis advisor, who had total confidence in me and so improved my technical writing; Prasad Jayanti, who taught computer science in the most memorable and infectious way and kindled my adult desire to study it formally. On the job, Krishanu Seal was a perfect Socratic partner and restored my faith in intellectual curiosity.

Jeremy Yallop, my partner in Links crime, with his humor and curiosity, helped me land safely in a foreign country and survive the new academic life. Sam Lindley was a great friend and sounding-board who also went the extra mile to make life in this new land rather enjoyable. Matija Pretnar energetically explored it with me and was a very patient friend, too. Ian Stark, as second supervisor, advised me kindly and enthusiastically. Don Sannella gave very wise counsel when I needed it most. Phil Wadler, my supervisor, started it all by taking a great chance on me, for which I am very grateful, and by watching him I have learned many valuable things about writing, presenting, and constructing technical work.

Most unreasonably faithful of all are my father, Robert Newton Cooper, and my mother, MaryDaniel Casey Cooper, who have put up with all the worst things I've ever done and trusted in me to do some good things as well. Most critically, they taught me that careful observation and persistent dedication, combined with an open, adventurous mind, would yield satisfying creative results.

Chapter 1

Introduction

Web programming, even with the best libraries, remains difficult. Not only do programmers need to contend with fast-changing and mutually-incompatible browsers, they need a menagerie of different languages to create a single application; and, as this thesis argues, the execution model of the web environment is essentially different from the classic models it is displacing.

Consider that typical applications demand software running on the web server as well as in the browser—thus we cannot easily apply the now-archaic *batch model* of programming, where a program runs on a single machine, transducing input into output. Instead, we have a heterogeneous distributed environment, where programs run on different platforms, on different machines, much of their activity is in communication, and the communication happens over a sometimes-awkward substrate, the HTTP protocol. In such an environment, it remains a significant challenge to establish communication between the principals and ensure that they are interacting as desired. The challenge here is more specialized than the general challenge of concurrent distributed computing, because the relationship among the principals (server and browser) is very constrained, and thus it may be more tractable.

Another special feature of the web environment is the ubiquitous use of a separate database management system. This tradition arose both as a way to achieve data synchronization among the many servers providing a web application and also as a means of persisting data. Yet communication between the core program code and the database is normally through an interface dictated by the latter, and so fits uncomfortably with other activities of the core code. As such, web development would be made much easier by an integrated interface to the database.

Further, the web environment presents a new *user-interaction model*, distinct from the desktop GUI model and distinct from the batch- or line-oriented models of yore. Relevant elements of this model include the web's notion of *navigation* (including the forward and back buttons, bookmarking, and URLs as navigational atoms), and the *document object model* with its associated *event model*. Even HTML *forms* are a specialized piece of this interaction model. Working with these particular elements begs for good abstraction to wrangle them.

To ameliorate the difficulty of web programming, engineers are building libraries and frameworks that incorporate techniques and protocols for working in the web environment. There are, for example, many JavaScript libraries to ease browser programming; and there are many server-side web frameworks to ease that part of the job. Yet all of these packages are limited by the artificial partition of web environment into client and server. The programmer is left to stitch together the two worlds into one working application. This often involves carefully structuring a program to expose internal facilities as interfaces made available across the network. It can also require working around differences in semantics between languages, which give rise to subtle bugs.

This thesis aims to contribute across the full length of a web app, stretching from front end to back end. It describes (Chapter 2) Links, a programming language with a unified model of the web environment; it gives (Chapter 4) a composable abstraction for defining HTML forms; it defines (Chapter 3) a language feature for moving execution smoothly between client and server, even over the asymmetrical substrate of typical HTTP; and it shows (Chapter 5) how to express relational database queries with the increased flexibility of a general-purpose, higher-order programming language.

My colleague Jeremy Yallop often criticizes the so-called “design patterns” movement in software engineering. As he tells it, we wouldn’t need to write down our patterns if we could express them directly in code. This observation retrospectively forms a starting point for the present work: Considering that web programming is full of folklore techniques and “patterns,” why not build a language that supports defining these patterns *as code*? The Links language became an experiment in doing just that.

Contributions The key contributions of this thesis are:

- A description of the features and implementation of the Links programming language, with novel features for concurrent client-server computing, web application page flow, form composition, and language-integrated database queries,
- A formalization of a technique for executing location-aware programs on an asymmetrical client-server substrate, combining continuation-passing style, trampolined style, and defunctionalization,
- An application of the functional-programming interface called *idioms* to the problem of form composition for user interfaces,
- A type-and-effect system for an impure variant of the Nested Relational Calculus, which allows detecting the translatability of NRC expressions to SQL, along with such a translation.

Chapter 2

Links Overview

(This chapter is a greatly revised version of Cooper et al. [2006], joint work with Sam Lindley, Philip Wadler, and Jeremy Yallop.)

Here we overview Links, a programming language designed to ease web development.

2.1 Introduction

A typical web system is composed of many *tiers*, or sets of machines which play a certain role and have a certain architecture (see Figure 2.1). For example, there may be a user-interaction tier, comprising the web browser(s), a so-called application tier, which runs the core logic on many machines, and a back-end tier, comprising possibly many services, running on many machines; the back end typically includes persistent data in the form of a relational database (big systems also use caches and application-specific back-end services, although these

Browser
Server
Backend (DB)

Figure 2.1: Three-tier model of web programming

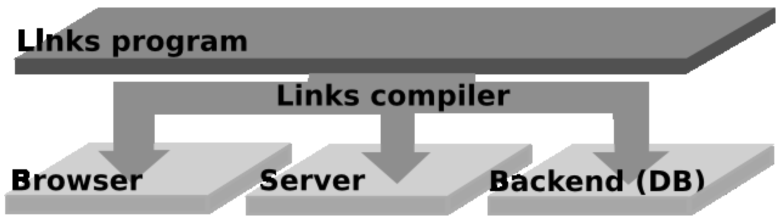


Figure 2.2: The *Links* model: Unified web programming

are beyond the scope of this thesis).

To use all these tiers, the programmer must master a menagerie of languages: the core code would usually be written in a general-purpose language such as Java, Perl, PHP, or Python, the user interaction given in a mixture of HTML and JavaScript, and the relational database queried using SQL. There is no easy way to ensure that interfaces between the tiers match up—that an HTML form or an SQL query produces the sort of data that the core logic expects. And the problem is exacerbated because code for the browser or database is often partly generated at runtime, making web applications particularly difficult to debug. This difficulty is called the *impedance mismatch* in web systems.

The Links language reduces the impedance mismatch by providing a language whose execution model encompasses all three tiers (Figure 2.2). In the current version, Links translates into JavaScript to run on the browser and SQL to run on the database; core logic is either interpreted or, in an experimental version, compiled via OCaml. All this code is generated robustly by the Links compiler, rather than by ad-hoc techniques such as string interpolation, which are often used in the wild. The automatic code generation reduces the opportunity for programmer error, and supports type-checking the communication between the tiers, which is conventionally not automatically checked at all.

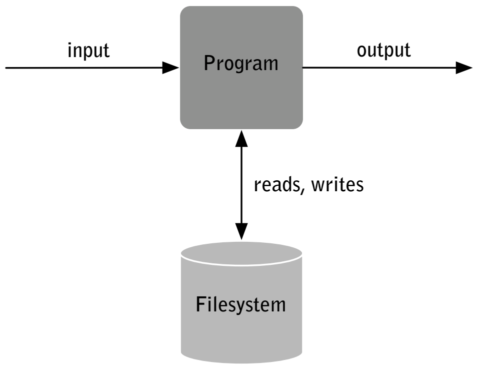


Figure 2.3: Conventional batch model of programming

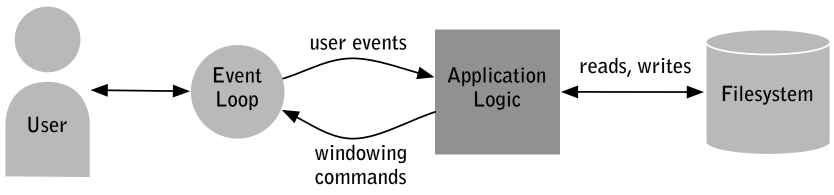


Figure 2.4: Event-loop model of programming

The tiered model of Figure 2.1 is markedly different from the very traditional *batch model* of programming (Figure 2.3), where a program simply consumes input and produces output, and from the *event-loop model* (Figure 2.4), where a program responds to events and uses windowing commands to display information to the user.)

Links does not completely free the programmer from thinking about the tier structure; it only makes it easier to manage the tiers' interaction. For example, the programmer may still need to consider whether a given bit of code should run in the client or the server—especially when security is an issue—and it might even be necessary to tweak code for different locations to achieve optimal performance. But in Links it is a simple matter of *annotation* to retarget code between the tiers when necessary. In terms of the *cognitive dimensions of* *notations* [Green, 1989], this feature offers "low viscosity."

There are aspects of Links syntax that are clearly adapted for client, server, or database, and they have their distinct flavors, but this does not mean that we have fallen back to three unrelated languages. On the contrary, Links still unifies the programming process—in particular, it type-checks the entire program at once, and it offers common syntax for basic operations (arithmetic, string processing, and so on). Thus the semantic impedance mismatch is reduced, even though there is a superficial heterogeneity of syntax. And this heterogeneity has a benefit: each syntactic area has some similarity with an existing language for that domain; for example, (X)HTML, which is a familiar language for specifying web documents. In brief, Links is a *unified, heteroglot language* for web development.

All the features of Links work on top of existing web infrastructure. The Links compiler generates code for existing client-side and backend technologies (JavaScript for the browser and SQL for the database), so for example it is not necessary to download a browser plugin to use a Links application. For the middle-tier server logic, Links currently runs under the CGI interface—thus it fits easily with existing HTTP servers. The interpreter could be adapted to use other server interfaces, such as FastCGI, NSAPI, or to run as an Apache module, without disturbing its essential architecture.

As well as reducing the impedance mismatch between tiers, Links offers special abstractions for web-specific programming tasks, such as specifying page relationships, and constructing and processing forms.

Because scalability is essential for web programs, Links is designed to scale. Many web frameworks that incorporate powerful features, such as *web continuations* (see Sec. 2.8), do so at the cost of using unbounded, long-lived, server-side storage. Web engineers are highly conscious of this issue, and will not deploy systems whose persistent-resource consumption depends directly on the volume of individual user actions. Such a mistake can bring down a large site, or prevent a small one from reaching "web scale." Links proves that it is possible to implement advanced web-language features scalably, using no server-side resources to track computations that are executing on the clients.

Chapter Road Map

Links is introduced in Section 2.2 through a brief list of distinguishing features and further in Section 2.3 through three fully-worked examples. Section 2.4 describes the syntax and special features of Links in detail. Section 2.5 gives background on the environment in which web programs run. The next six sections each cover a special aspect of Links. Section 2.6 describes Links' unusual execution model, including the client-server relationship, threading model, and reentrancy (part of this model is formalized in Chapter 3). Section 2.7 describes the implementation of concurrency. Section 2.8 shows how Links supports defining an application's page relationships, as experienced by the user, including some useful control abstractions. Section 2.9 explains the behavior and implementation of Links' location-aware distribution annotations. Section 2.10 notes the features for composable forms (formalized in Chapter 4). Section 2.11 overviews the language-integrated query features (formalized in Chapter 5). Section 2.12 describes related work, and Section 2.13 concludes the overview.

2.2 Features

Web execution model Unlike the batch and event-loop execution models described ealier, the web model is fundamentally re-entrant: A user can multiply the number of windows she has open on a site, explore her history freely, and save a URL in a durable medium.

Each of these branching paths contains references into the code (URLs) which can be triggered at any time. This fact gives rise to a common form of bug, documented for example by Graunke et al. [2003], where clicking to purchase an item displayed in one window can have the effect of purchasing an item displayed in a different window. Many websites still shout to users "Do not use the back button," thereby asking their users to cover for problems in the application. Avoiding this problem using traditional languages and frameworks requires explicitly managing the data in the multiple exploration paths that the back button gives rise to.

What's more, the program has to re-establish context upon each entrance, usually by manual coding. If a user enters a shipping address on one page and a billing address on the next, the coder needs to manually store the shipping address somewhere and fetch it again later. Recent research has applied the notion of a *continuation* from functional programming to ease this, allowing the programmer to code such a page-flow in direct style, as if the whole interaction were a sequential program, without losing the control and data context. A number of researchers have made use of the concept, including Queinnec [2000], Graham [2001a] (in a commercial system used by Yahoo for building web stores), Graunke et al. [2001a,b], Matthews et al. [2004], and Thiemann [2002]. Links, too, uses continuations to ease working in the web execution model, as described in Section 2.8.

Client-server calls Links is *distributed* and *location-aware*, meaning that a given expression can potentially take place at any of the locations (client or server) in the low-level execution model, but the programmer can explicitly locate an expression if necessary—due to reasons of efficiency or security, for example. For the implementation, this requires performing *remote-procedure calls* (RPC) symmetrically (i.e. in either direction) over the asymmetrical substrate of HTTP.

DOM interface Links presents information to the user through the Document Object Model (DOM) [World Wide Web Consortium, 2004] and web pages expressed in (X)HTML. Besides defining the page structure itself, the DOM interface gives a way to update documents and to listen for user-interaction events, such as mouse and keyboard actions [World Wide Web Consortium, 2003b]. HTML web pages act as a convenient notation to specify DOM structures and associated event listeners, so its syntax is embedded in Links.

Concurrency Links supports concurrent programming under a message-passing model: The only facility for thread interaction is message-passing. There is no shared memory between threads, nor indeed any directly mutable cells to make shared memory meaningful for concurrency purposes. This style of programming was pioneered in the languages Erlang [Armstrong et al., 1993] and Mozart [van Roy, 2006].

Message-passing concurrency is the only general-purpose form of *state* in Links, a mostly pure language. State arises from concurrency because each process has a state, including a program counter, local variables and a message queue. In fact, through message-passing we can write a process that simulates a mutable cell.

Concurrency is the only *general* form of state, but there are other stateful aspects of the system, notably the DOM. And since the DOM and the user interface it represents are fundamentally stateful, wrangling these is greatly aided by having some usable form of state in the language. We might imagine using the DOM itself as a mechanism, but this is unwieldy, and we prefer to have a language-oriented solution to the need for state; concurrency is the Links solution.

Besides that, message-passing concurrency enables a fairly natural, functional style for writing model-view-controller user-interfaces: in this style each UI component is managed by a "model" process which receives model-update messages and modifies the DOM accordingly; examples are given in Section 2.3.

Language-integrated query Database queries can be written completely in Links' native comprehension syntax and translated by the system into SQL. A syntactic annotation allows the programmer to specify that a given expression must be SQL-translatable (if it is not, an error is produced). This entails determining whether the expression could take any side-effects at runtime, or call non-SQL-friendly primitives. However, the permissible query expressions are more flexible than SQL, and may even be higher-order, that is, they can use first-class functions. This allows query "fragments" to be given as code elsewhere in the program and passed into a query expression, thus assembling a query dynamically at runtime, and with the same type safety as any Links code.

2.3 Links by example

This section introduces Links by a series of examples. The examples and their source code are available at

`http://homepages.inf.ed.ac.uk/s0567141/examples/`

The first example (Links Dictionary) shows the language basics, and the next two demonstrate special features.

For reference, Figure 2.5 lists the most common Links expressions. Figure 2.6 lists some common operators and their meaning. The code examples use the color and style conventions shown in Figure 2.7.

Links Dictionary

The Links Dictionary application allows looking up and modifying English dictionary entries. At startup, it presents the user with a search box. As the user begins to type in this box, the application continuously displays a list of ten dictionary words that could complete the entry at that point (see Figure 2.8). Many popular applications, famously Google Suggest [Google Inc., 2004], behave similarly. Links Dictionary is based on an ASP.NET version, available online [Narra, 2004], using the same database of 99,320 entries. The Links version extends the original by allowing the definitions to be added, updated and deleted.

To add a new definition, the user fills in the form in the “Add a definition” panel at the bottom of the page and clicks ‘Add.’ To update an existing definition, the user clicks on one of the suggestions, which then expands into a form (see Figure 2.9). To modify the entry, the user fills in a new word or definition and clicks ‘Update.’ To delete it, the user clicks ‘Delete.’

This application demonstrates the interactive capabilities of Links and the way Links programs can communicate between client and server, since every

General expressions

x	variable
[exp_1, exp_2, ... exp_n]	list construction
(l_1=exp_1, l_2=exp_2, ... l_n=exp_n)	record construction
fun (x_1, x_2, ... x_n) { exp }	anonymous function
Foo(exp_1)	variant tagging
query { exp_1; exp_2; ... exp_n }	query-translation assertion
exp.label	record projection
exp_1 op exp_2	infix operator application
if ($condExp$) $trueExp$ else $falseExp$	conditional
switch (exp) { case pat_1 -> exp_1; exp_2; ... exp_n ... }	pattern matching
for (qs) ⟨where (exp_1)⟩ ⟨orderby (exp_2)⟩ exp_n	list comprehension
escape x in exp	continuation capture
formlet $body$ yields exp	formlet expression
table $name$ with $type$ ⟨where $constraints$⟩. from $dbexp$	table handle
database $name$	database handle

XML expressions

Concurrency expressions

receive { case pat_1 -> exp_1; exp_2; ... exp_n ... }	pattern-matching message reception
spawn { exp_1; exp_2; ... exp_n }	spawn a thread
spawnWait { exp_1; exp_2; ... exp_n }	spawn thread, wait for its return value

Database actions

insert $table$ values exp

delete (x <-- src) ⟨where ($cond$)⟩

update (x <-- src) ⟨where ($cond$)⟩ set (l1 = exp_1, l2 = exp_2, ... ln = exp_n)

Figure 2.5: Links syntax reference.

`++`	List (string) concatenation
`==`, `<>`	Value-(in)equality test
`~`	Regular-expression matching
`+.`, `-.`, `*.`, `/.`, `^.`	Floating-point arithmetic
`+`, `-`, `*`, `/`, `^`, `mod`	Integer arithmetic
`<`, `>`, `<=`, `>=`	Numeric comparison
`!`	Send a message to a process

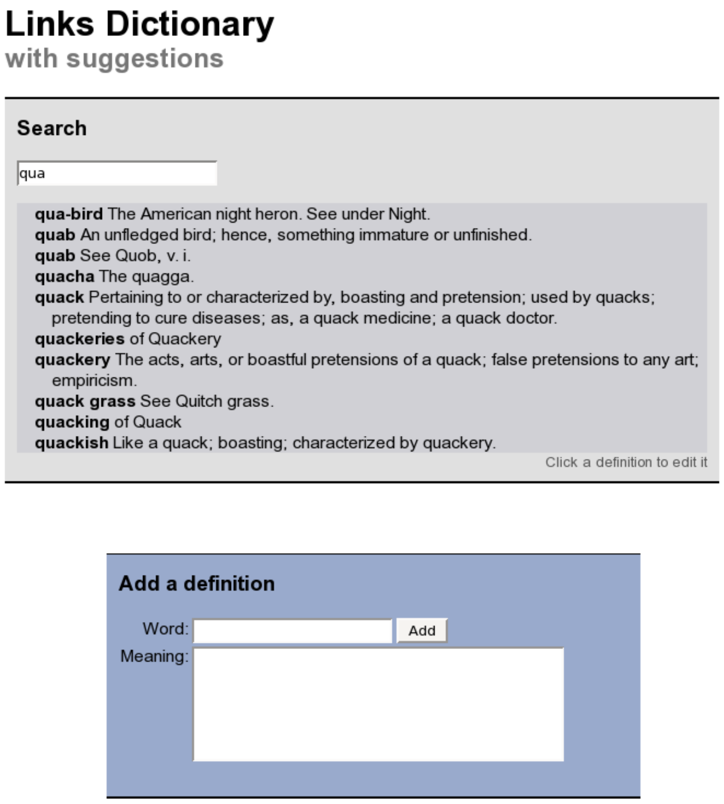
Figure 2.6: Table of basic Links operators.

`black`	Links code
`boldface`	keywords
`blue italic`	identifiers
`blue upright`	field/variant labels, type/data constructors
`gray`	XML tags
`green`	XML text

Figure 2.7: Color/style coding in listings.

keystroke at the client requires it to perform a database lookup (at the back end) and update the display (at the front end).

To examine performance, I compared the two versions (Links vs. ASP.NET) by measuring the time taken to handle a keystroke and produce the list of ten suggested completions. For the Links version, measurements were made by instrumenting the code to output timing information; the ASP.NET version was measured with a handheld stopwatch, controlling for reaction time. Over the course of 36 trial lookups with various prefixes, response time (from keypress to page update) for the Links version was 649ms on average, with a standard deviation of 199ms. If we subtract the time spent performing the database query in each trial, the average time taken by Links itself was 297ms with a standard deviation of 54ms. Given that no effort has been spent trying to optimize per-



Links Dictionary
with suggestions

Search

qua

qua-bird The American night heron. See under Night.
quab An unfledged bird; hence, something immature or unfinished.
quab See Quob, v. i.
quacha The quagga.
quack Pertaining to or characterized by, boasting and pretension; used by quacks; pretending to cure diseases; as, a quack medicine; a quack doctor.
quackeries of Quackery
quackery The acts, arts, or boastful pretensions of a quack; false pretensions to any art; empiricism.
quack grass See Quitch grass.
quacking of Quack
quackish Like a quack; boasting; characterized by quackery.

Click a definition to edit it

Add a definition

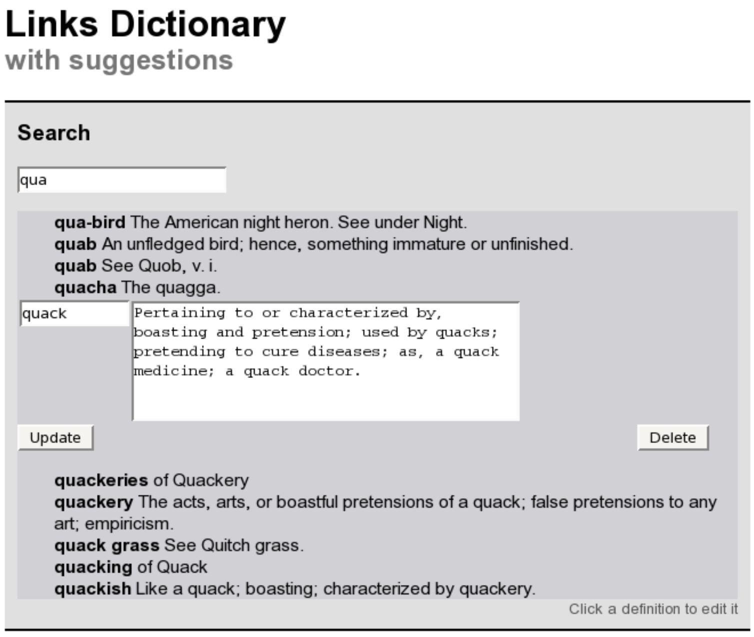
Word: Add

Meaning:

Figure 2.8: Links Dictionary in action. The user has entered the letters “qua” and is shown the first 10 entries beginning with that prefix.

formance of the Links system, this seems to indicate acceptable performance. Response time for the ASP.NET version over a dozen trials averaged 0.1s after subtracting a human reaction time measured the same way (the stopwatch was stopped and started as quickly as possible). Measurements of the ASP.NET version were made against a remote server, with its own distinct (and unknown) performance characteristics. Thus, essentially, the ASP.NET version was faster than effectively measurable given the equipment at hand, despite the added network delay, not present in the Links Dictionary experiment.

The code for the application is shown in Figures 2.10–2.13; let’s walk through it. When the page is loaded, the final expression, *main*() is evaluated; *main*



Links Dictionary
with suggestions

Search

qua

qua-bird The American night heron. See under Night.
quab An unfledged bird; hence, something immature or unfinished.
quab See Quob, v. i.
quacha The quagga.

quack

```
Pertaining to or characterized by,
boasting and pretension; used by quacks;
pretending to cure diseases; as, a quack
medicine; a quack doctor.
```

Update   Delete

quackeries of Quackery
quackery The acts, arts, or boastful pretensions of a quack; false pretensions to any art; empiricism.
quack grass See Quitch grass.
quacking of Quack
quackish Like a quack; boasting; characterized by quackery.

Click a definition to edit it

Figure 2.9: Links Dictionary in action. The user has clicked on the entry for "quack" and can now edit it.

---

spawns a thread and binds its identifier to the variable *eventLoop* (Figure 2.13). This process will manage the suggestion list. The function *main* also returns an HTML document which is installed in the browser window; this HTML contains event handlers which will be invoked by user activity.

On each keystroke in the searchbox, one of the event handlers, the `l:onkeyup` expression (Figure 2.13) is evaluated for its side-effects. In this case it sends a `Suggest` message, containing the current searchbox text, to the *eventLoop* process. (The expression $e_1 \; ! \; e_2$ denotes the operation of sending the message denoted by e_2 to the process denoted by e_1.) On receipt of a `Suggest` message, the *eventLoop* process calls *suggestView* to fetch the new suggestions and update the view and then calls its own handler function in tail position to remain receptive. The *suggestView* function in turn calls *completions* to fetch the data, and *formatDefView* to format that data. The HTML returned by

```
var defsTable =
  table "definitions" with
  (id:String, word:String, meaning:String)
  where id readonly from database "dictionary";

fun newDef(def) server { insert defsTable values [def] }

fun updateDef(def) server {
  update (var d <-- defsTable) where (d.id == def.id)
  set (word=def.word, meaning=def.meaning)
}

fun deleteDef(id) server {
  delete (var def <-- defsTable) where (def.id == id)
}
```

Figure 2.10: Links Dictionary (1)

---

formatDefView includes the material for both the static and editable versions of the entry; the editable version is initially hidden and the routines *editDefView* and *cancelEditView* respectively toggle the visibility of the two versions. The call to *completions* will force a transfer of control to the server—an *RPC call*—since the *completions* function is labeled with the `server` keyword and is here called in client context. When its return value is passed to the *format* function, labeled `client`, this will force a transfer of control back to the client.

By convention, Links event handlers, such as the `l:onkeyup` handler in this example, send a message to another process to do any heavy work. This is because all such handlers execute within one event-handling thread, and consume all control on the JavaScript virtual machine, which also blocks the browser window from doing anything else. This blocking design was chosen to allow processing events serially, that is, to prevent them from interfering concurrently with one another. When we want concurrency, as in this case, we always use a separate process. Moreover, in this case we use a *single* additional process because we want the events to be processed serially: each will queue up in the *eventLoop* process' mailbox, in the order they were sent, until previous events have finished

```
fun completions(s) server {
  if (s == "") [] else {
    take(10, for (var def <-- defsTable)
              where (def.word ~ /s.*/) orderby (def.word)
                [def])
  }
}

fun redraw(xml, defId) client {
  replaceChildren(xml, getNodeById("def:" ++ defId))
}

fun suggestView(s) client {
  replaceChildren(formatDefsView(completions(s)), getNodeById("suggestions"));
  ignore(showElement(getNodeById("def-instr"), "block"));
}

fun editDefView(def) client {
  ignore(showElement(getNodeById("def:" ++ def.id ++ "-form"), "block"));
  ignore(hideElement(getNodeById("def:" ++ def.id)));
}

fun cancelEditView(id) client {
  ignore(hideElement(getNodeById(id ++ "-form")));
  ignore(showElement(getNodeById(id), "block"));
}

fun cancelEditsView() client {
  var defs = getElementsByClass(getNodeById("suggestions"), "def");
  ignore(for (def <- defs) {
         cancelEditView(domGetAttributeFromRef(def, "id")); []});
}
```

Figure 2.11: Links Dictionary (2)

```
fun formatDefView(def) {
  <#>
    <div id="{"def:" ++ def.id}" class="def"
         l:onclick="{cancelEditsView(); editDefView(def)}">
     <b>{stringToXml(def.word)}</b>
     {stringToXml(def.meaning)}<br/>
    </div>
    <form id="{"def:" ++ def.id ++ "-form"}" class="hidden edit"
     method="POST"
     l:onsubmit="{
       var def = (id=def.id, word=w, meaning=m); updateDef(def);
       redraw(formatDefView(def), def.id)}">
     <table><tr>
       <td><input l:name="w" value="{def.word}" class="word-edit" /></td>
       <td>
       <textarea l:name="m" rows="5" cols="40">{
        stringToXml(def.meaning)}</textarea></td>
       </tr>
     </table>
     <button type="submit"> Update </button>
     <button l:onclick="{deleteDef(def.id); redraw([], def.id)}"
      style="position:absolute; right:100px;" type="button">
       Delete </button>
    </form>
  </#>
}

fun formatDefsView(defs) {
  for (def <- defs)
    formatDefView(def)
}

fun newdefFormView(eventLoop) client {
 <form l:onsubmit="{eventLoop!NewDef((word=w, meaning=m))}">
  <table>
  <tr><td class="label">Word:</td><td>
   <input type="text" l:name="w"/>
   <button type="submit">Add</button></td></tr>
  <tr><td>Meaning:</td><td>
   <textarea l:name="m" rows="5" cols="80"/></td></tr>
  </table>
 </form>
}
```

Figure 2.12: Links Dictionary (3)

```
fun main() {
  var eventLoop = spawn {
   fun receiver(s) {
    receive {
     case Suggest(s) -> suggestView(s); receiver(s)
     case NewDef(def) ->
       newDef(def);
       replaceChildren(newdefFormView(self()), getNodeById("add"));
       suggestView(s); receiver(s)
     case GetDefs(sender) ->
       sender ! s;
       receiver(s);
    }
   }
   receiver("")
  };

  <html>
   <head>
    <title>Links Dictionary</title>
    <link href="dict.css" rel="StyleSheet" type="text/css" />
   </head>
   <body>
    <h1>Links Dictionary  <div class="lesser">with suggestions</div></h1>
    <div class="searchbox">
      <h3>Search</h3>
      <form l:onkeyup="{eventLoop!Suggest(s)}">
        <input type="text" l:name="s" autocomplete="off"/>
      </form>
      <div id="suggestions"/>
      <div id="def-instr" class="note hidden">
        Click a definition to edit it</div>
    </div>
    <div id="newdef">
      <h3>Add a definition</h3>
      <div id="add">{newdefFormView(eventLoop)}</div>
      </div>
   </body>
  </html>
}
main()
```

Figure 2.13: Links Dictionary (4)

processing. This prevents out-of-order effects where fetching suggestions for one prefix could return after fetching those for a prefix entered later.

Clicking on a definition invokes the function `editDefView`, which calls the function `redraw` in order to replace the (static) definition with its editable counterpart. Clicking 'Update' or 'Delete' performs the corresponding modification to the definition by calling `updateDef` or `deleteDef` on the server, and then updates the view by calling the function `redraw` on the client.

Finally, the "Add a definition" box at the bottom of the page is produced by the function `newdefFormView`. Clicking 'Add' sends a `NewDef` message with the new data to the `eventLoop` process. The `eventLoop` process in turn calls the server function `newDef` to add the definition to the database, then resets the form and updates the view (that's in case the new definition appears in the current suggestion list).

This example demonstrates many features of Links: basic syntax, the RPC mechanism, message-passing concurrency, and the DOM interface.

Draggable lists

The next demo illustrates how a more interactive application can be written in Links. It shows in more depth the use of the concurrency primitives to encapsulate and manage the state of an interactive GUI component. It displays several itemized lists and the user may drag an item in any list to another position within the same list (see Figure 2.14). The code for the draggable list demo is shown in Figures 2.15–2.16.

Each draggable list is monitored by its own process. For each relevant event on an item (e.g. mouse-up, mouse-down, and mouse-out) a handler sends a message to the list's owning process, indicating the event. The process itself is coded as two mutually recursive functions, corresponding to two states. The process starts in the waiting state; when the mouse button is pressed it changes to the dragging state; and when the button is released it reverts to the waiting state. When in the dragging state and the mouse is moved (the `MouseOut` event indi-

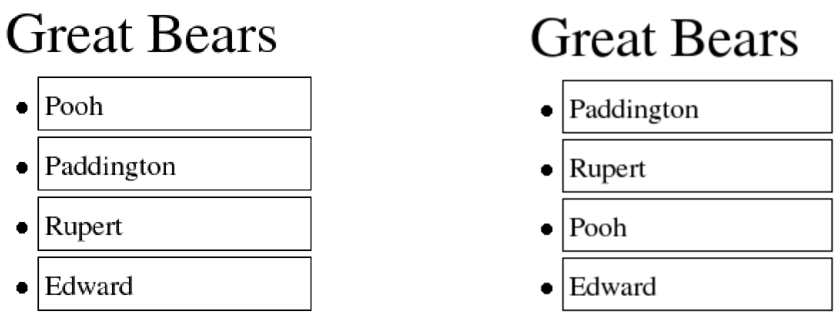


Figure 2.14: Draggable list: before and after dragging

cates the mouse position has left the original element), the dragged item and the other item are swapped.

Both functions take, as parameter, the DOM `id` of the draggable-list element, thus they know what part of the DOM tree they control. The *`dragging`* function takes an additional parameter, designating the particular item that is being dragged. Both functions are written in tail recursive style, each one calling either itself (to remain in that state) or the other (to change state). This style of using tail-recursion to hold state in a process was taken from the Erlang community.

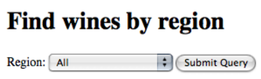
Eventually we will want to extract the ordering chosen by the user. This example does not show it, but it only a requires the process to maintain a list of the elements: whenever it swaps two in the UI, it swaps two in its list as well. Another form of message can ask it to reply with the contents of this list. It can be made to respond to another message that asks it to reply with the contents of that list.

```
fun waiting(id) {
 receive {
  case MouseDown(elem)   ->
   if (isElementNode(elem)
             && (parentNode(elem) == getNodeById(id)))
     dragging(id,elem)
   else waiting(id)
  case MouseUp           -> waiting(id)
  case MouseOut(newElem) -> waiting(id)
 }
}

fun dragging(id,elem) {
 receive {
  case MouseUp          -> waiting(id)
  case MouseDown(elem)  ->
   if (isElementNode(elem)
            && (parentNode(elem) == getNodeById(id)))
     dragging(id,elem)
   else
     waiting(id)
  case MouseOut(toElem) ->
   if (isElementNode(toElem)
            && (parentNode(elem) == getNodeById(id)))
    {swapNodes(elem,toElem); dragging(id,elem)}
   else dragging(id,elem)
 }
}
```

Figure 2.15: Draggable lists in Links (1)

```
fun draggableList (id,items) client {
 var dragger = spawn { waiting(id) };
 <ul id="{id}"
  l:onmouseup =     "{dragger ! MouseUp}"
  l:onmouseuppage = "{dragger ! MouseUp}"
  l:onmousedown =   "{dragger ! MouseDown(getTarget(event))}"
  l:onmouseout =    "{dragger ! MouseOut(getToElement(event))}">
   { for (item <- items) <li>{ item }</li> }
 </ul>
}

<html><body>
 <h1>Draggable lists</h1>
 <h2>Great Bears</h2>
 {
  draggableList("bears", ["Pooh", "Paddington", "Rupert", "Edward"])
 }
</body></html>
```

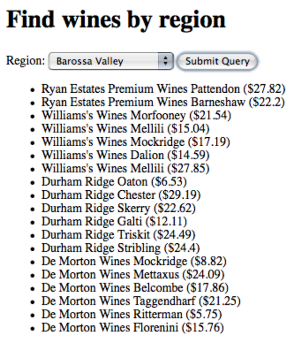
Figure 2.16: Draggable lists in Links (2)



Find wines by region

Region: All | Submit Query

Figure 2.17: Initial page.

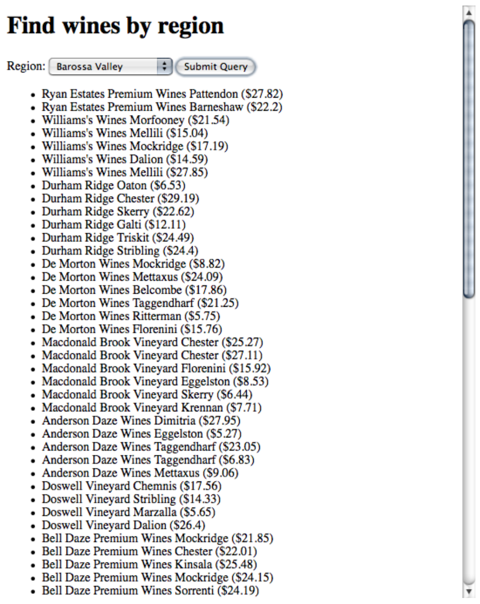


Find wines by region

Region: Barossa Valley | Submit Query

- Ryan Estates Premium Wines Pattendon ($27.82)
- Ryan Estates Premium Wines Barneshaw ($22.2)
- Williams's Wines Morfooney ($21.54)
- Williams's Wines Mellili ($15.04)
- Williams's Wines Mockridge ($17.19)
- Williams's Wines Dalion ($14.59)
- Williams's Wines Mellili ($27.85)
- Durham Ridge Oaton ($6.53)
- Durham Ridge Chester ($29.19)
- Durham Ridge Skerry ($22.62)
- Durham Ridge Galti ($12.11)
- Durham Ridge Triskit ($24.49)
- Durham Ridge Stribling ($24.4)
- De Morton Wines Mockridge ($8.82)
- De Morton Wines Mettaxus ($24.09)
- De Morton Wines Belcombe ($17.86)
- De Morton Wines Taggendharf ($21.25)
- De Morton Wines Ritterman ($5.75)
- De Morton Wines Florenini ($15.76)

Figure 2.18: Search in action.



Find wines by region

Region: Barossa Valley | Submit Query

- Ryan Estates Premium Wines Pattendon ($27.82)
- Ryan Estates Premium Wines Barneshaw ($22.2)
- Williams's Wines Morfooney ($21.54)
- Williams's Wines Mellili ($15.04)
- Williams's Wines Mockridge ($17.19)
- Williams's Wines Dalion ($14.59)
- Williams's Wines Mellili ($27.85)
- Durham Ridge Oaton ($6.53)
- Durham Ridge Chester ($29.19)
- Durham Ridge Skerry ($22.62)
- Durham Ridge Galti ($12.11)
- Durham Ridge Triskit ($24.49)
- Durham Ridge Stribling ($24.4)
- De Morton Wines Mockridge ($8.82)
- De Morton Wines Mettaxus ($24.09)
- De Morton Wines Belcombe ($17.86)
- De Morton Wines Taggendharf ($21.25)
- De Morton Wines Ritterman ($5.75)
- De Morton Wines Florenini ($15.76)
- Macdonald Brook Vineyard Chester ($25.27)
- Macdonald Brook Vineyard Chester ($27.11)
- Macdonald Brook Vineyard Florenini ($15.92)
- Macdonald Brook Vineyard Eggelston ($8.53)
- Macdonald Brook Vineyard Skerry ($6.44)
- Macdonald Brook Vineyard Krennan ($7.71)
- Anderson Daze Wines Dimitria ($27.95)
- Anderson Daze Wines Eggelston ($5.27)
- Anderson Daze Wines Taggendharf ($23.05)
- Anderson Daze Wines Taggendharf ($6.83)
- Anderson Daze Wines Mettaxus ($9.06)
- Doswell Vineyard Chemnis ($17.56)
- Doswell Vineyard Stribling ($14.33)
- Doswell Vineyard Marzalla ($5.65)
- Doswell Vineyard Dalion ($26.4)
- Bell Daze Premium Wines Mockridge ($21.85)
- Bell Daze Premium Wines Chester ($22.01)
- Bell Daze Premium Wines Kinsala ($25.48)
- Bell Daze Premium Wines Mockridge ($24.15)
- Bell Daze Premium Wines Sorrenti ($24.19)

Figure 2.19: Search finished.

Progressive Query

This application demonstrates symmetric client and server calls, which are here used to interactively display the result of some time-consuming server-side activity such as searching a large database.

A familiar model for this demo is the kind of flight-search website that searches multiple independent airlines for flights matching some criteria—which altogether can take some time—and displaying batches of results as they become available, rather than all at once.

This version simply retrieves a list of fictional wines in a given region. It is slightly contrived in that, to slow down the query and simulate fetching data from multiple external sources, it artificially splits up a simple SQL query into several pieces (one for each winery) and executes them sequentially. The application is meant to demonstrate how, in Links, chunks of data which independently become available can be incrementally *pushed* to the client for display. We can imagine a similar app drawing from multiple data sources in parallel, pushing data to the client when each parallel request finishes.

The code for the progressive query application is shown in Figures 2.20–2.22. The application runs against the database schema of Hugh and Dave's Online Wines, the case study application from a book about web application development [Williams and Lane, 2004]. The results in the screenshots (Figures 2.17–2.19) illustrate the freely downloadable data from the book's website.

The application initially presents a form where the user can select one of the available wine regions. Upon submitting this form, results begin to appear, in small batches, on the page.

Submitting the form evaluates, as usual, the expression in the form's `l:onsubmit` attribute. This calls the server function *progressiveSearch*, which iterates over all the wineries for the given region, and for each one it fetches the corresponding wines from the database, then calling the *showProgress* function to display these to the user. The *showProgress* function is a `client` function, so when it is invoked, the Links runtime will transfer control temporarily back to the client to execute the function body. Here the function uses DOM commands to insert the data into the page document. When the function completes, Links returns control to the server, where it continues by going around to the next iteration of the wineries loop. This goes on until the server runs out of wineries, and then the thread doing this work reaches its end and terminates.

Viewed at the low level, a call to a client function like *showProgress* is implemented as follows: The server responds to the client's HTTP request, which had been placed earlier. The HTTP response includes a reference to the function, its arguments, and a representation of the server-side continuation to be invoked when the client call is finished. No state whatsoever need remain on the server once the computation is moved to the client. The unusual execution model supporting this client-server mobility is discussed in more detail in Section 2.9.

As an advantage of this stateless-server design, if the client quits at some point during the computation (whether by surfing to another page, terminating the browser, or pulling the plug out of the wall), then no further work will be

```
fun showProgress(wines) client {
  appendChildren(
        for (w <- wines)
          <li>{stringToXml(w.winery)} {stringToXml(w.name)}
              (${floatToXml(w.cost)})</li>,
        getNodeById("listing")
  );
}
var db = database "winestore";
var inventoryTable = table "inventory" with
                     (wine_id : Int, cost : Float)
                     from db;
var regionTable =    table "region" with
                     (region_id : Int, region_name : String)
                     from db;
var wineTable =      table "wine" with
                     (wine_id : Int, wine_name : String,
                      wine_type : Int, year : Int, winery_id : Int)
                     from db;
var wineryTable =    table "winery" with
                     (winery_id : Int, winery_name : String,
                      region_id : Int)
                     from db;
```

Figure 2.20: Progressive query example (1).

```
fun wineriesByRegion(regionID) server {
  query { for (winery <-- wineryTable)
          where (winery.region_id == regionID)
            [winery] }
}

fun wineCost(wineID) {
  for (i <- asList(inventoryTable))
  where (wineID == i.wine_id)
    [i.cost]
}

fun winesByWinery(winery) {
  query {
    for (wine <-- wineTable)
    where (wine.winery_id == winery.winery_id)
    for (cost <- wineCost(wine.wine_id))
      [(name=wine.wine_name, cost=cost, winery=winery.winery_name)]
  }
}
```

Figure 2.21: Progressive query example (2).

```
fun progressiveSearch(regionID) server {
  foreach(wineriesByRegion(regionID), fun (winery) {
    showProgress(winesByWinery(winery))
  });
}

var blankListing = <ul id="listing" />;

page
 <html>
  <body>
   <h1>Find wines by region</h1>
   <form l:onsubmit="{replaceNode(blankListing, getNodeById("listing"));
                      progressiveSearch(stringToInt(regionID))}">
    Region: <select l:name="regionID">
              {for (r <- asList(regionTable))
                 <option value="{intToString(r.region_id)}">
                 {stringToXml(r.region_name)}</option>}
            </select>
    <input type="submit" />
   </form>
   {blankListing}
  </body>
 </html>
```

Figure 2.22: Progressive query example (3).

required of the server. To keep the computation going, the client must keep invoking these continuations.

2.4 Language Description

Now that we've seen Links in action, we'll take a tour of its features as a language.

At its core, Links is a fairly standard ML-like functional programming language with Hindley-Milner type inference. One conspicuously absent feature is exception handling, which Links should be able to accomodate albeit with a good deal of additional engineering.

A Links program consists of a series of definitions followed by a main expression. Running the program evaluates the main expression, which might of course make use of the definitions. The definitions are all defined mutually-recursively with respect to one another.

Syntax Links syntax was designed to resemble JavaScript and C-like languages. Thus we use curly-braces `{ }` to delimit code blocks, function bodies, and clauses of conditional (`if`) expressions. The appearance of the code is imperative (looking like a sequence of commands), rather than declarative (denoting a value)—but in fact, it is functional: every expression has a value. What looks like the initialization of a variable, `var` x `= 7;` is in fact just the head of a traditional let-binding, which binds over the whole rest of the block (until the close of the nearest containing curly-brace `{ }` pair). Names like x defined this way denote an *immutable value* rather than a *mutable location*. There are no built-in mutable-location constructs in Links.

Conditionals—which always have a value and must therefore contain both positive and negative branches—are written as in C, but return a value as in functional languages:

```
if (condition)
  trueBranch
else
  falseBranch
```

The two branches are simple expressions, but these can be blocks if we wish to execute multiple expressions therein and bind them to local variables.

We can perform structural pattern matching on any value with the `switch` construct. Unlike the switch of C or JavaScript, this performs pattern matching like ML, so the outermost constructors of the value are matched to a pattern and inner values are bound to variables.

```
switch (scrutinee) {
  case pat1 -> e1
  case pat2 -> e2
  ...
}
```

Like the conditional, this expression returns the value of the matching branch.

Functions are declared in any scope as follows:

```
fun f(x1, x2) { body }
```

This declares a function called f taking parameters $x1$ and $x2$ and whose body expression is $body$. Function values can also be created anonymously simply by omitting the name. So

```
fun (x1, x2) { body }
```

is an expression whose value is a two-argument function. All variables are scoped lexically, so such values may capture the values of variables from the environment that are free in the function expression. Links functions are *multi-argument*, so the above examples define functions of two arguments, which cannot be applied to a single pair.

The lexical scope of a `var` or `fun` declaration includes all of the following expressions within the enclosing `{ }` block. Thus in this example the scope of the underlined binder x consists of the underlined expressions.

```
fun f(x) {
 var x = x+1;
 var y = x;
 fun g(z) { x + z };
 x + 7
}
```

Note that var bindings are not recursive: the x appearing in the right-hand side of the var x binding refers to the parameter x above it, not to the x being defined.

Types and values Links uses Hindley-Milner type inference with row variables [Rémy, 1993, Pottier and Rémy, 2005]. Basic Links types include integers, floats, characters, booleans, lists, functions, records, and variants. The syntax of types and value constructors is as follows:

- A list is written `[`e_1`, ..., `e_k`]`, and a list type is written `[A]`.
- A record is written `(`f_1`=`e_1`, ..., `f_k`=`e_k`)`, and a record type is written `(`f_1`:A`$_1$`, ..., `f_k`:A`$_k$` |`r`)`, where r is an optional row variable. Field names, like f_1, must begin with a lower-case letter.
- A variant value is written $\mathtt{F}_i$`(`e_i`)` and a variant type is written `[|F`$_1$`:A`$_1$`, ..., F`$_k$`:A`$_k$`|`r`|]`. Variant tags, like $\mathtt{F}_1$, must begin with an upper-case letter.
- A lambda abstraction is written `fun (`x_1`, ..., `x_k`) {`e`}`, and a function type is written `(A`$_1$`, ..., A`$_k$`) -> B`. Parentheses around argument types are mandatory, even when only a single argument is present. This unfortunately necessitates double parenthesis when a single argument is a record type.

The `String` type is simply a list of characters (`Char`s), and tuples are records with sequential natural numbers as labels.

XML Links includes special syntax for constructing and manipulating XML data. XML data is written in ordinary XML notation, using curly braces to indicate embedded code; these expressions are called “quasiquotes.” Embedded code may occur in attributes, where it has string type, as well as in the body of an element, where it has XML type. The direct inclusion of XML syntax makes it easy to paste XML into Links code. The parser begins parsing XML when a < is immediately followed by a legal tag name. (A space must always follow < when it is used as a comparison operator; it so happens that legal XML does not permit a space after the < that opens a tag.) Links also supports the syntactic sugar `<#>` ··· `</#>` for specifying an unrooted list, or hedge, of XML trees, as in the function *`formatDefView`* in Figure 2.12. These hedges have type `Xml` while single trees have type `XmlValue`.

Links XML quasiquote expressions only generate *well-formed* XML; the parser rejects quasiquotes that would lead to ill-formed XML. The *validity* of an XML document, which Links does not guarantee, can only be established by reference to a schema, which specifies which XML tags are allowed in which combinations and what attributes each can have. The XML variant of HTML, called XHTML, has its own schema, and Links programs are informally bound to produce valid XHTML documents. Yet Links goes no further than checking well-formedness; validity checking for quasiquotes in the setting of higher-order programming languages is already well-studied and tractable [Hosoya and Pierce, 2003, Frisch, 2006], so it was not a focus of the Links project.

Parallel to the `Xml` type, Links also has a `DomNode` type, holding references to DOM nodes. These reference mutable nodes with identity, so two distinct nodes with the same structure are unequal, and each non-null `DomNode` value refers to a DOM node which is actually part of the current document (although it may be invisible for various reasons). Links provides library functions to access and modify the DOM, mirroring the functions specified by the W3C DOM standard [World Wide Web Consortium, 2004].

Note the relationship between pure XML and impure DOM trees: `DomNode` is a mutable DOM reference, while `Xml` is an immutable list of XML trees. We can convert the former to the latter with the *`getValue`* primitive, which makes a deep copy of the tree rooted at the node, returning it as a singleton XML-tree list. Other operations convert the latter to the former by *installing* XML in the active DOM, which creates new DOM node instances for the nodes of the XML tree.

There is also a null value in the `DomNode` type, reflecting the fact that W3C DOM functions can return a null value. (Perhaps it would be more in keeping with the statically-typed philosophy to treat these as functions into an option type.)

The operations on `Xml` and `DomNode` values are shown in Figure 2.23 and 2.24; they behave as follows.

Given an XML value, we can project its component data with *getTagName*, *getAttributes* and *getChildNodes*. We can test for and project particular attributes using *hasAttribute* and *getAttribute*. And we can extract its text content—the concatenation of all the descendant text nodes—with *getTextContent*.

We can fetch the root `DomNode` of the document with *getDocumentNode*, and fetch one by its ID attribute using *getDomNodeById*. The functions *parentNode*, *firstChild*, *nextSibling* provide navigation, returning the DOM node with the corresponding relation to the given one, or the null node if none exists. The *getValue* function extracts the structure of the given node as XML and *isNull* tests whether the argument is the special null value. The operations *removeNode*, *insertBefore*, *appendChildren*, *replaceChildren*, *replaceNode*, *swapNodes* and *replaceDocument* alter the document; those which take an `Xml` parameter first create a new DOM node from the list of XML trees and then installs this somewhere in the document.

```
getTagName       : (XmlValue) -> String
getChildNodes    : (XmlValue) -> [XmlValue]
getAttributes    : (XmlValue) -> [(String,String)]
hasAttribute     : (String, XmlValue) -> Bool
getAttribute     : (String, XmlValue) -> String
getTextContent   : (XmlValue) -> String
```

Figure 2.23: Library functions for `Xml`.

```
getDocumentNode : () -> DomNode
getDomNodeById  : (String) -> DomNode

getValue        : (DomNode) -> XmlValue
isNull          : (DomNode) -> Bool
parentNode      : (DomNode) -> DomNode
firstChild      : (DomNode) -> DomNode
nextSibling     : (DomNode) -> DomNode

removeNode      : (DomNode) -> ()
insertBefore    : ([XmlValue], DomNode) -> ()
appendChildren  : ([XmlValue], DomNode) -> ()
replaceChildren : ([XmlValue], DomNode) -> ()
replaceNode     : ([XmlValue], DomNode) -> ()
swapNodes       : (DomNode, DomNode) -> ()
replaceDocument : (XmlValue) -> ()
```

Figure 2.24: DOM interface.

```
for (var x <- e1)         concatMap(fun(x){e2},
 e2                                 e1)
for (var x <- e1)
 where (e2)               concatMap(fun(x){if (e2) e3 else []},
  e3                                e1)
for (var x <- e1)         concatMap(fun(x){e3},
 orderBy (e2)                       sortBy(fun(x){e2},
  e3                                e1))
for (var x <- e1)
 orderBy (e2)             concatMap(fun(x){if (e3) e4 else []},
 where (e3)                         sortBy(fun(x){e2},
  e4                                e1))

                          concatMap(fun((x,y)){if (e4) e5 else []},
for (x <- e1, y <- e2)          sortBy(fun((x,y)){e3},
 orderBy (e3)                   concatMap(fun(x){
 where (e4)                       concatMap(fun(y){ [(x,y)] },
  e5                                        e2)}, e1)))
```

Figure 2.25: List comprehension syntax examples and their desugared forms.

List comprehensions The `for (var` *x* `<-` *e1*`)` *e2* construct is a *list comprehension*. It evaluates *e2* once for each element of *e1* and concatenates the lists produced by *e2* into a result list. Links comprehensions also support several clauses which modify their behavior, including `where` and `orderBy`. The examples in Figure 2.25 show syntactic forms on the left, corresponding to expressions on the right that use only library functions.

Regular expressions To match a string against a regular expression we write $e \sim /r/$ where r is a regular expression. Curly braces may be used to substitute a string into a regular expression, so for example `/{`*s*`}.*/` matches any string that begins with the value bound to the variable *s*.

Concurrency A concurrent programming style has been found useful in programming user interfaces, where many UI components may independently react to user input. A user action may invoke a lengthy computation, such as fetching mail from a certain mailbox on the server, yet we want other controls to remain

`spawn {` *e* `}`	spawn a new thread
`spawnWait {` *e* `}`	spawn thread, wait for completion
`recv``()`	receive one message
`receive {` *cases* `}`	receive one message and pattern match
`self``()`	return process identifier
proc `!` *msg*	send message to a process

Table 2.1: Links concurrency primitives.

responsive; for example, the user may wish to view other mailboxes while the first one is fetching. Concurrency is a useful way to structure such a program.

With this in mind, Links supports concurrent programming, under a model with no shared memory between threads. (In this thesis, the terms "process" and "thread" are used interchangeably.) Following Erlang [Armstrong et al., 1993], each thread has a single incoming mailbox, which no other thread can read. The concurrency primitives are shown in Table 2.1.

The expression `spawn`$\{e\}$ creates a new process and returns a fresh process identifier. The primitive *`self`*`()` returns the identifier of the current process. The statement e_1`!`e_2 sends message e_2 to the process identified by e_1. The function *`recv`*`()` simply returns the next message from the caller's mailbox. And finally, the expression

```
receive {
  case p1 -> e1
  ...
  case pn -> en
}
```

extracts the next message waiting for the current process, or blocks (suspends) the process until there is such a message, and executes the first case whose pattern matches the message. (Unlike in Erlang, it is a static error if no case matches, which fits better with Links' discipline of static typing.) The `receive` expression is just syntactic sugar: `receive {` *cases* `}` is equivalent to `switch (`*`recv`*`()) {` *cases* `}`.

The spawnWait form spawns a thread as does spawn, but then blocks the parent until this child thread completes, returning the value of e to the parent. This is useful in particular because the child thread has a fresh mailbox, thus the type of its messages does not "pollute" the parent thread's mailbox, which was a problem observed in Links programs lacking spawnWait.

By convention, the type of messages sent to a process is a variant type, but it can in fact be any type. Using a variant allows tagging different sorts of messages, each of which contains different parameters.

There is a distinguished *main process* which executes the initial expression and all event handlers. Running all event handlers in a single process guarantees that events are processed in the order in which they are received.

Concurrency is discussed further in Sections 2.6 and 2.7.

User events A Links program specifies its interest in user events through HTML, using the event types defined by the DOM Events interface [World Wide Web Consortium, 2003b]. The association of event listeners with DOM nodes is similar to that of HTML 4.0 [World Wide Web Consortium, 1999], where event listeners were specified through element attributes.

In Links, then, event listeners can be specified as element attributes onclick, onkeyup, etc., but prefixed with l: to echo the use of namespaces in XML, like a special "Links namespace" (but this namespace exists only in Links XML quasiquotes and is not defined through the XML namespacing mechanism). Further, such listeners can refer to the values of form input fields carrying an l:name attribute (which also generates an HTML name attribute on the element). The value of an input element's l:name attribute becomes a bound variable for the scope of all the event listeners in any descendent element of that input element's containing form element. Event listeners can be seen here in the Links Dictionary code:

```
<form l:onkeyup="{eventLoop!Suggest(s)}">
  <input type="text" l:name="s" autocomplete="off"/>
</form>
```

The l:onkeyup attribute is the event listener, and the input element's l:name attribute defines a variable *s* that will be bound, when the listener is invoked, to the `String` value of that input element (the text entered therein) at that moment.

A subtle semantic detail needs explaining. Whereas in other XML-quasiquote contexts, curly braces introduce Links code that is substituted (i.e., evaluated immediately, with the result then taking the place of the curly-braced code), the `l:`*`event`* attributes are different: their code is only evaluated when the event listener is invoked, and only for side-effects.

Because of the scoping rules, Links can statically ensure that form-input fields are referenced from code in a consistent manner. This differs from dynamic web frameworks, where the form fields are typically indexed by string values at runtime, exposing them to runtime failure.

User interface and concurrency Event handlers like `l:onkeyup` are run by the main thread. Normally, threads can pre-empt one another at any time, as one would expect from concurrent threads. However, when the main thread is handling a user event, the handler is executed atomically—it effectively freezes the scheduler, preventing thread switches until the handler. This atomicity allows the programmer to be sure events are handled in the order they occur; without this, a second event could pre-empt an ongoing one, and could be completely processed, before the first one had taken action. However, this means that a long-running event handler could starve or delay other threads, so an event handler should execute quickly.

Event handlers are expected to execute quickly, so we follow a convention that each handler either sends a message or spawns a fresh process. For instance, in the Links Dictionary application, each event occurence sends a message to the handler process that is responsible for finding and displaying the completions. This puts any delay resulting from the database lookup into a separate process, so the user-interface thread remains responsive, and any future keystrokes are echoed immediately. Furthermore, the messages received by the handler process are processed sequentially, ensuring that the updates to the DOM happen in the order the keystrokes were received.

Database access Database queries are expressed directly in Links code, particularly with list comprehensions; so the programmer uses “native” operations rather than an external API for database access. In particular there are Links expressions representing *database handles* and *table handles*. The latter can be coerced directly to lists, giving the list of rows of the table at that moment; this effectively entails a database query.

Semantically, the Links compiler may evaluate *any* expression through the database system, provided it can preserve the value and any side-effects of the expression—but it seems never a good choice to do so unless the expression actually involves the database—and of course, if an expression depends on a database table, we must execute *something* in the database system. But the line where data from a database query is returned to the Links runtime for further processing is, by default, unspecified.

In order to regain some control over this division of labor, Links offers an expression annotation, `query {···}`, which requires its content to be executed as a query—and if this is not possible, the compiler must indicate a static error. This allows the programmer to detect flaws which prevent an expression from executing completely in the database system as part of a single query.

For example, if the programmer wishes to equijoin two tables with a Links expression, but Links cannot create a database query out of the expression, then normally it may obey the semantics by reading both tables completely and performing the join itself with a naive algorithm. Yet if the join expression is wrapped in a `query` annotation, Links is obliged to evaluate the equijoin in the database (which, we assume, is an efficient way of doing so, exploiting indexes and query planning). If Links cannot meet that obligation—for example, if the expression is not actually equivalent to an SQL statement—then it fails at compile-time, giving the programmer a chance to improve the expression so that it can be evaluated in the database system.

Database connections are introduced with the `database` keyword, giving the instance name and optional configuration data, which can also be read from a configuration file.

Table handles are introduced with the `table` keyword and the table name, the type signature of a row in the table, and the database connection.

The coercion operator *asList* takes a table handle to the list of its rows, and the syntax `for (var` x `<--` e_1`)` e_2 (with a long arrow) is syntactic sugar for `for (var` x `<-` *asList*`(`e_1`))` e_2 (with a shorter arrow).

In the following example, we use a table of words, as in the Links Dictionary example. For example, the expression

```
for (var def <-- defsTable)
where (def.word ~ /s.*/) orderBy (def.word)
  [def]
```

compiles into the equivalent SQL statement

```
SELECT def.meaning AS meaning, def.word AS word
FROM definitions AS def
WHERE def.word LIKE '{s}%'
ORDER BY def.word ASC
```

This feature, called language-integrated query, was pioneered in systems such as Kleisli [Buneman et al., 1995, Wong, 2000], and is also central to Microsoft's LINQ for .NET [Microsoft Corporation, 2005]. Chapter 5 formalizes a query-compilation system like the one employed by Links.

The regular-expression matching operator ~ is in this case compiled into the `LIKE` operator of SQL. At run time, the phrase `{`s`}` in the SQL is replaced by the string contained in the variable s, including escaping of special characters where necessary. We cannot translate all regular expressions to uses of the `LIKE` operator, and this is one place where query translation can fail at runtime. (A better design might be to offer a "like" operator in the Links library, with semantics equivalent to that of SQL, and translate it directly.)

Links also has statements to update, delete from, and insert into a database table, closely modeled on the same statements in SQL. These can be seen used in the functions *newDef*, *updateDef* and *deleteDef* in Figure 2.10.

Serializable continuations In Links, all values are serializable—and that includes closures and continuations. A continuation is represented as a stack of pointers to control contexts with associated environments. To serialize one of these, Links generates a label for each control context, and replaces each one with its label, also capturing the data environment for each one. The label must be generated stably, or deterministically, so that it can be located again on another invocation of the server, after parsing the program from scratch. Serializing other values is easier: each base type has a direct representation and the other constructed types are represented by a pair of an identifier of the constructor and the serialization of the contents.

This method transforms a continuation into a compact bit string that, together with the original code, contains all the information present in the continuation. As long as the code itself remains unchanged, we can safely resolve the code references in the bit string.

This mechanism can be distinguished from one which retains something at the server to correspond to each continuation—for example, one which keeps a record in memory or a process running for each client session or each branch thereof.

The choice to replace code pointers by computed strings is critical. By omitting a serialization of the code itself, we save a great deal of space and also preserve the secrecy of server-side code. And by indicating code points with computable values, we ensure that the representation is stable across distributed instances of the web server and across reboots. The only thing that can break a Links link is a change to the original program. Failing gracefully in this situation is an important area of future work.

The size of a serialized continuation depends on the amount of data captured in it, which can vary dynamically at runtime. Normally, these representations are very small, since they do not represent long call stacks. A typical continuation is like any other web link: it designates a program point with a small number of parameters, usually strings or integers. However, since the capture of data is implicit, it could easily overflow, perhaps even as a programming accident. It would be possible to accidentally capture a continuation whose representation is larger than the reasonable size of a URL (typically around 1024 characters for existing browsers and servers). This risk begs for additional techniques, both to warn of the possibility of large continuations and to provide means of wrangling large amounts of data.

The representations used in RPC calls (as opposed to continuations captured for web links) are particularly likely to grow large. The issue is far less pressing, however, since these are delivered in HTTP request and response bodies, which are limited only by memory and disk space.

Security also becomes a concern when values are automatically serialized. In particular, this serialization might leak data that the programmer never intended to be revealed outside the program, or allow attackers to tamper with data that has already been computed. While the Links implementation described here makes absolutely no effort to confront these problems, work by Baltopoulos and Gordon [2009] shows how such serializations can be encrypted and signed cryptographically to prevent against both kinds of problems, and proves the transformations correct with respect to a semantics of a core language. Such protections would be necessary in any security-sensitive use of a Links-like language.

Page flow A distinctive feature of web programming, as compared with batch and event-loop programming, is the need to describe the relationship between pages, as experienced by the user—which we'll call *page flow*.

The execution model of Links spans across web pages and so it incorporates page flow as a static feature of programs. It has a variety of facilities for supporting different kinds of page-flow designs, which are described in Section 2.8.

2.5 The Web environment

To understand the special execution model of Links, we need to understand the environment around a web program, as it is generally understood by practitioners; this section defines some terms and concepts from web standards and engineering practice.

Except for personal and experimental systems, nearly every web system includes several (middle-tier) server machines—a “server cluster” of at least two—across which the load of incoming requests is balanced by a special network router called a load balancer. Load balancing is ubiquitous in web systems for a host of reasons: the need for fault-tolerance, as well as the typical hit rate of “web-scale” application, coupled with a drive for performance and the fact that web requests typically comprise little hard computational work.

As such, one cannot assume that two successive web requests from a particular client will be received by the same server machine. While load balancers can be configured as “sticky,” meaning they attempt to route requests from the same client to the same server over time, this cannot be relied upon, for several reasons. First, the information available to the load balancer is unreliable: the principal identifier of the client is IP address, which can change during a session for a particular client (either because the client moves or because it is going through a proxy that hides it behind a bevy of IP addresses). HTTP cookies can be used instead of IP addresses, but extracting these requires parsing the HTTP protocol, requiring more complexity in the router software, and anyway these can be rejected or spoofed by clients. Further, a great number of clients can be represented by a single IP address, particularly with very large service providers, and making them all sticky to one server undermines the opportunity to balance the load. Next, load balancers are under pressure to handle a lot of requests very quickly and with a modest amount of storage, so that keeping track of the mapping between IP addresses to server machines may be too expensive. System-engineering experience has shown that the best method for load-balancing tends to be to assign requests to servers either probabilistically, based on load information from the servers, or simply in round-robin.

These facts determine the special execution environment of web applications. Critically, they must handle individual requests without relying on having any state information in memory, because any server in the cluster might handle any given request. Generally speaking, the means for sharing state is to write to shared concurrent storage media, perhaps an RDBMS or a distributed filesystem, although more ephemeral means, such as distributed in-memory caches, are sometimes employed.

Web applications also live in an *information-architecture environment*. The architecture of the web is characterized [World Wide Web Consortium, 2003a, Berners-Lee, 1998] as a collection of *web resources*, each identified by one or more *uniform resource locators* or URLs. A URL is thus posited to refer stably to a particular resource, which encompasses the collection of *documents* that may be returned upon requesting the URL; these documents are also called *representations of* the resource. Documents can be in any format (including images, videos, and other media); in this thesis we are mainly concerned with resources represented as HTML documents. Besides fetching representations, web standards also accomodate requests that *post* information to a resource; what this means is generally up to the application designer.

Now let us make a broad, imperfect definition of a web application. We can view it is as a graph (perhaps implicit, perhaps infinite) of web resources. We specialize those web resources to *dynamic web pages*—HTML documents with associated code and document structure. Each dynamic web page has its own virtual machine (VM) for running the page's code, and at any moment has its own state, including the document structure, which can mutate over time. This virtual machine is nothing other than the browser's instantiation of its JavaScript engine and is limited, by the browser, to existence within that page. This containment means, inter alia, that the virtual machine suspends execution when the browser window is closed or the user navigates to another page—and its state can be reset completely when the page is no longer active. The virtual machine is empowered to make HTTP requests to fetch other web resources, which it can then process as it likes. It is also empowered to manipulate the document and listen for user activity using the DOM interface [World Wide Web Consortium, 2004, 2003b]. The web application can move suddenly between web pages, e.g. when a user clicks a link, thus abruptly moving between program points, and moreover the active pages, which are effectively threads of control, can be cloned and eliminated as a user opens and closes new windows on the site—a phenomenon we call *re-entrancy* and which the programmer must handle.

Because of all this, a web application is crucially different from batch and event-loop programs. A batch program starts at a unique entry point and runs to termination. It follows a specific flow of control, designed by the programmer, until completion.

An event-loop program does dispatch user events to registered event listeners, as does the DOM virtual machine. But missing is the graph of separate pages which the user can browse freely and clone. Desktop GUI programs, for example, need not handle the possibility of re-entrance. Such programs have a much more constrained execution model.

2.6 Links execution model

The Links execution model slightly extends the general web execution model. We keep the idea of a page-contained virtual machine but replace it with a *Links virtual machine* which has extra capabilities, including concurrency; also the constitutent pages or resources are defined by *program points* and their parameters (essentially, by *closures*), so that they are statically-defined entities. The Links virtual machine is only a thin addition to the JavaScript virtual machine, and inherits most of its behavor from its JavaScript counterpart.

The principal novelty is that the Links virtual machine spans both the browser's VM and a server-side execution environment. Threads in the Links VM can move smoothly from client to server and back again by means of RPC calls, which are simply calls to location-annotated functions. The location of execution makes a difference for a number of reasons: some operations may be more efficient when executed in one location or the other; some code may be secret so that it must never be transmitted to the browser; and some operations may be available only at one location, so choosing the point at which the transfer of control happens might make a big difference to efficiency. The Links language and runtime are designed to make these transfers of control as seamless as possible.

Now, we can describe the execution of a Links program broadly as follows. Semantically, fetching the base URL of the program begins executing the program's main expression. Operationally, this delivers enough JavaScript to the client for it to begin executing this main expression, ultimately producing a document (in HTML) which is installed into the browser's DOM. Any of the links or buttons on the page may be associated with a Links expression or continuation; invoking one of these moves the VM to a new page and, again, delivers enough JavaScript to the client to run that expression. We can imagine that the virtual Links machine for each page starts up upon delivery of the JavaScript and terminates when the user leaves that page through the browser's controls.

We do not try to create a virtual machine that transcends the page limitation of the browser's engine, both because it would be difficult, given the browser's security policies, which are explicitly designed to prevent widespread interpage communication, and also because the limitation to a page is a useful structuring mechanism. Because users can browse an unbounded series of pages and leave the browser open for a very long time, there is no other natural limit on the execution of client-side web code. Limiting the virtual machine to a page thus gives a natural cleanup point and saves the programmer from managing threads and state across the long lifetime of the browser itself.

Figure 2.26 depicts the execution environment of a Links program including several web pages, each with its own virtual machine, each making RPC calls to the server cluster (or cloud). This drawing represents the fact that each page has a separate virtual machine and does not share state with, nor can it send messages to, other pages, and interacts with the database or other pages only through the server cloud, which itself can be distributed across many machines and thus holds no state of its own.

Concurrency model

Among the facilities of the Links virtual machine is the ability to fork and communicate between concurrent threads. Furthermore, each thread can move back and forth between client and server independently. This means that any thread,

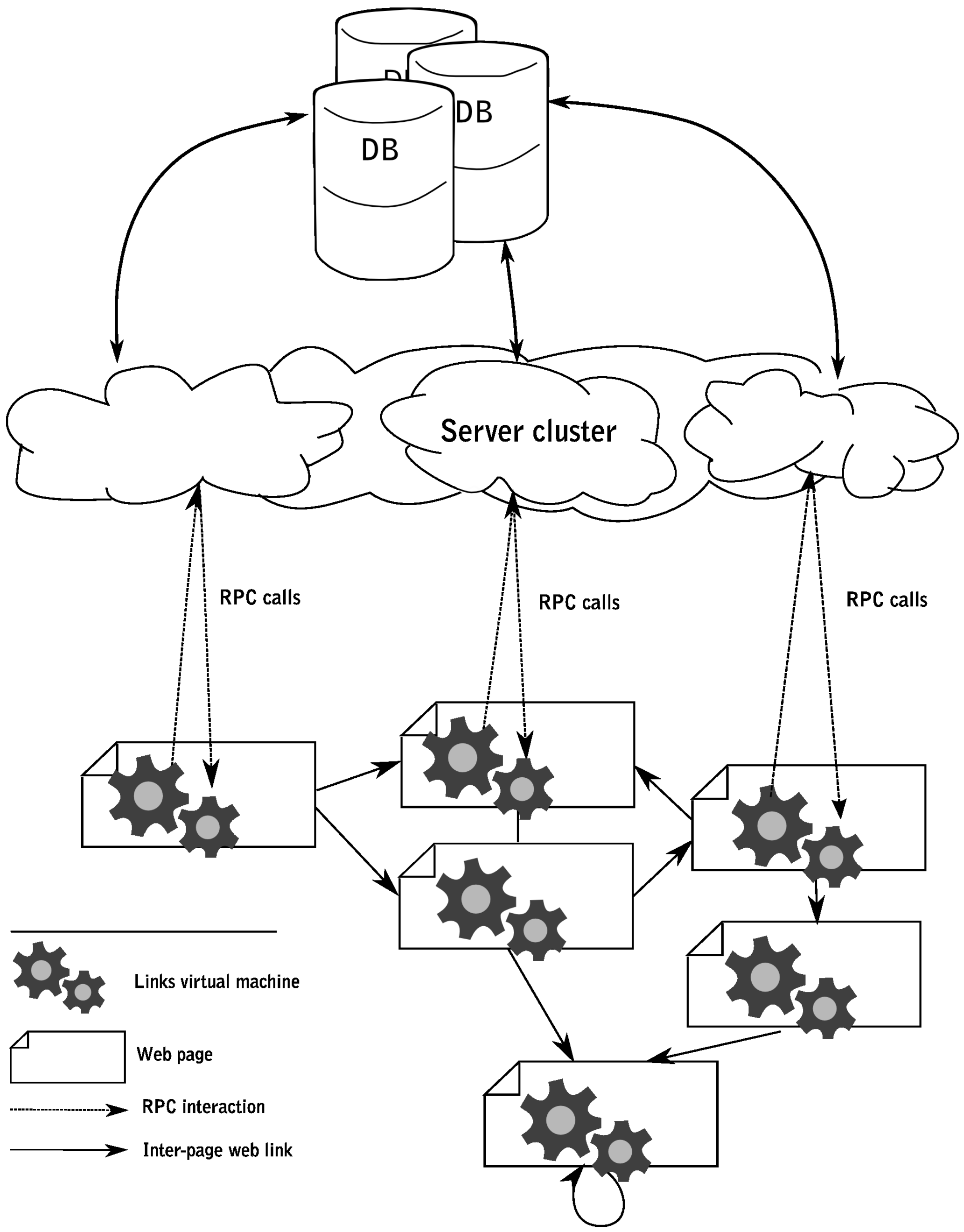


Figure 2.26: Links VMs live within Links web pages and communicate with the server.

even if it is presently running at the client, can access the program's server-located resources (which might include secret algorithms or access to back-end resources like the database) by means of RPC calls, which are executed sequentially within that thread.

Threads can be spawned at any time, regardless of the execution location. This is semantically simple, but delicate from the implementation point of view. Since each thread belongs to a particular VM, it belongs to a particular active web page, and so each thread is "homed" to a particular active page. Among the consequences of this fact is that if the user closes the corresponding browser window, or navigates away, its Links virtual machine will stop abruptly, and thus none of its threads are guaranteed to continue. This is surprising from the point of view of conventional programming, but here we view the client as king, and computation is principally performed only on the client's behalf. To drive the point home, the server, in the Links model, is really just that: it merely *serves*, and has no rights of its own.

Figure 2.27 shows an example VM with threads moving between client and server, as it is defined semantically. A thread (Thread 1) begins in the client and forks a second thread (Thread 2), which then moves to the server, where it forks another (Thread 3). Later, Thread 2 finishes its server-side business and returns to the client while Thread 3 keeps running on the server. Still later, Thread 3 returns to the client, does more business, and finishes.

Figure 2.28 shows the same execution, as it occurs at the low level. At this level, when Thread 2 needs to return to the client, the server must serialize its complete state to the client, so both threads are suspended, serialized, and delivered; the client, however, immediately sees that one of these threads (Thread 3) is still marked as running at the server, so it revivifies this suspended thread, making a server request to continue it; meanwhile it revivifies Thread 2 simply by continuing it directly at the client. Thread 3, because it spawns no further threads, is living in its own server-side execution environment, so when it returns to the client it need not disturb any other threads.

We need to suspend and serialize all the server-side threads when any one

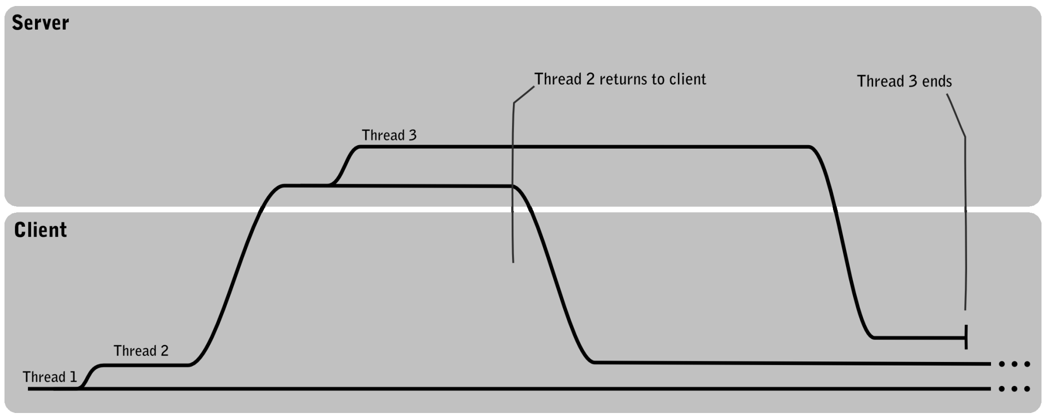


Figure 2.27: Links threads moving independently between client and server.

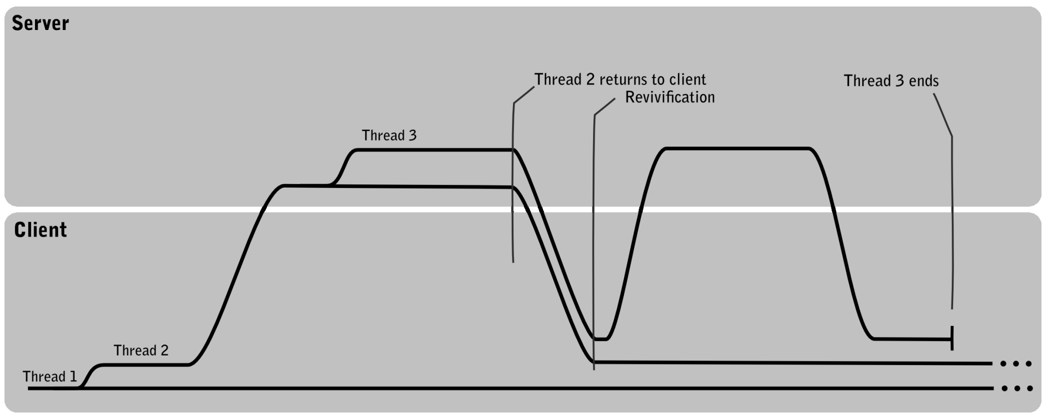


Figure 2.28: Mobile threads (low-level view).

of them moves to the client because of the need to tunnel our communications through the HTTP protocol. Recall that the server threads are all living in the confines of a particular HTTP request. Once we return a response to that request (and hence to that client) we'll have no good way of sending further data. So, whenever we need to return some data to the client, we completely stop the server-side VM and serialize its state over to the client. The client extracts the needed data and then re-starts the server-side VM with a new HTTP request. This way we always keep open a channel of communication from server to client.

Semantically, we view Thread 3 as continuing transparently on the server (rather than taking a detour to the client) since it makes no progress between the time it is bundled with Thread 2 for delivery to the client and the time it is revivified on the server. Such a detour, which may arise from technical details, can cause the thread to be delayed for quite a long time while this operation is taking place.

The concurrency implementation is described in detail in Section 2.7.

Re-entrancy

Section 2.5 noted the re-entrancy property of the web environment. How does this affect the Links virtual machine? Again, a server serving a Links application will route incoming requests to any of several internal program points, corresponding to web links, and they can be triggered in any sequence. The program has no predefined end and the client can always give rise to new server computations, at any number of entry points, by invoking a link or an RPC call.

Semantically, we wish for these entrances not to have unexpected side-effects, that is, an entrance at a given expression should have just the side-effects of that expression. On the other hand, we want top-level definitions to be in scope everywhere, according to the lexical scoping principle—so we have to take care to make sure that their semantics are well-defined.

To ensure this, we choose to require that top-level definitions are pure, that is, they cannot bind to an expression that has side-effects or depend on the state of the external world. Generally, they will be constants or functions (the functions themselves need not be pure). This way, when the server is invoked to evaluate a given expression, it can always read in the program from scratch and execute the desired expression without worrying about whether and how to evaluate top-level side-effects. In fact, there is no question about when those top-level side-effects would be run; top-level definitions are simply “always already” there.

2.7 Concurrency implementation

Since JavaScript’s execution model is sequential, implementing concurrency takes some doing. Links uses the following techniques:

- compiling to continuation-passing style (CPS),
- inserting explicit context-switch instructions during compilation,
- performing server calls asynchronously (with `XMLHttpRequest`),
- eliminating the stack frequently, using a trampoline.

Using CPS allows us to reify each process’s control state, but since JavaScript does not perform tail-call elimination, classic CPS would quickly overflow the JavaScript stack after a small number of function calls (hundreds or thousands); thus we use trampolined style [Ganz et al., 1999] to regularly eliminate the stack. Fortuitously, trampolined style is also a traditional part of a CPS-based thread implementation. The idea is that instead of invoking a function directly, each callsite returns immediately to an outer loop, the trampoline, which re-invokes one of the existing threads–this is called *bouncing* the trampoline.

So the Links JavaScript compiler replaces every function and continuation application with a call to special *yield* functions, which take care of trampolining. Most of these are simply no-ops—they just apply the function or continuation immediately—but periodically one will actually perform a bounce. We need to perform a bounce often enough to prevent the JavaScript stack from overflowing; its size varies by implementation but generally seems to allow on the order of 1000 nested function calls. The parameter `_yieldGranularity` controls the number of direct calls made by the yield functions before the bounce occurs.

We have two yield functions, one inserted around function applications, `_yield`, and another inserted around continuation applications, `_yieldCont`; the JavaScript code for these is given in Figures 2.29 and 2.30. In fact the two are nearly identical and could be implemented with a common function that accepted a thunk (zero-argument function) instead of separate arguments for the various applicands; but this would require creating new JavaScript thunks for every application, which would be very slow; so we partition the function into two to avoid creating the thunk.

Links uses two different trampolines, to implement the two concurrency modes (normal and atomic) described earlier (Section 2.4, p. 38). Essentially, Links has a trampoline of its own, based on exceptions, while it also uses the JavaScript interpreter's event loop as a trampoline. Bouncing the latter trampoline allows event handling and running other threads; the exception-based trampoline disallows them. In various situations we adjust a global flag, `_handlingEvent`, which indicates to the yield functions which trampoline mode to use.

To understand the event-loop trampoline, let's take a close look at the browser's mechanisms. (The following is a working model, not necessarily an accurate description of a real browser.) The browser is continually running an event loop which checks whether certain events have happened and dispatches to appropriate code (perhaps JavaScript code from the page or perhaps its own internal code). These events include user actions, network activity (e.g. responses to asynchronous XMLHttpRequest invocations), and *timeout* events. Timeout events arise because the browser continually maintains a list of callback functions with associated timeouts—times at which they become eligible to run. The `setTimeout` function, used in our trampoline, adds a callback to this list, which allows us to schedule code to be run from the browser's event loop. Any of these events (timeouts and user- and network-activity) might run in a given iteration of the event loop. The whole event loop, including all the JavaScript it invokes, is single-threaded. This loop acts as one of our two trampolines.

```
function _yield(f, a, k) {
  ++_yieldCount;
  if ((_yieldCount % _yieldGranularity) == 0) {
    if (!_handlingEvent) {
      var current_pid = _current_pid;
      setTimeout((function () {
                    _current_pid = current_pid;
                    f(a,k) }),
                 _sched_pause);
      return;
    } else
      throw new _Continuation(function () { f(a,k) });

  } else {
    return f(a,k);
  }
}
```

Figure 2.29: The yield function for function application.

The second argument to *setTimeout* is effectively a lower bound on how soon the callback is eligible to run. Ideally this value should be zero, so there would be no delay between the moment a thread yields and the moment it is eligible to run again. But in some browsers, passing zero bypasses the event loop completely, thus blocking event handling, so we pass a configurable value, *_sched_pause* and set it to a small non-zero value.

The two trampoline modes are bounced as follows:

- The timeout-callback trampoline is bounced by returning directly to the event loop, but only after registering the continuation thunk with *setTimeout*. (Because the code is in CPS, returning at all returns from all callers, right to the event loop.) The job of the thunk is to maintain the "current process ID" global and then continue with the thread's next application, either of a function or a continuation.

- The exception-based trampoline is bounced by throwing an exception containing the current continuation. Throwing the exception unwinds the

```
function _yieldCont(k, arg) {
  ++_yieldCount;
  if ((_yieldCount % _yieldGranularity) == 0) {
    if (!_handlingEvent) {
      var current_pid = _current_pid;
      setTimeout((function () {
                      _current_pid = current_pid;
                      k(arg) }),
                 _sched_pause);
      return;
    } else
      throw new _Continuation(function () { k(arg) });

  } else {
    return k(arg);
  }
}
```

Figure 2.30: The yield function for continuation application.

stack; the trampoline then catches the exception and invokes the continuation.

The JavaScript code for the exception-based trampoline is shown in Figure 2.31. Given a continuation, `_exceptionTrampoline` runs it, catching any bounces that occur, as `_Continuation` exceptions, and re-invokes the continuation each carries. It starts by setting the global `_handlingEvent` to tell the yield functions to use the exception method of bouncing. This global must be restored on any of the paths that leave the function, whether by a true exception or a normal exit. The trampoline uses a loop with no termination condition (`for (;;)`), which keeps reinvoking the next continuation at each iteration. If one of the `cont` invocations eventually returns, corresponding to the thread's terminatation, we break the loop, restore the `_handlingEvent` global and return.

There is no code for the timeout-callback trampoline because, as noted before, this is handled by the browser's machinery.

```
function _Continuation(v) { this.v = v }
                         // An exception constructor.

function _exceptionTrampoline(initialCont) {
  var _cont = initialCont;
  _handlingEvent = true;      // Globally indicate exception mode.
  for (;;) {                  // Looping until we break,
    try {
      _cont();                //   try invoking the continuation;
      break;                  //   when it returns, exit the loop.
    } catch (e) {                       // Upon exception,
      if (e instanceof _Continuation) { //   if it's a bounce,
         _cont = e.v;                   //   use its continuation &
         continue;                      //   start the loop again.
      } else {                          // a real exception,
        _handlingEvent = false;         //   clean up and
        throw e;                        //   re-throw.
      }
    }
  }
  _handlingEvent = false;   // Exiting normally, clean up.
}
```

Figure 2.31: The exception-based trampoline

2.8 Page flow definition

There are at least two approaches to planning *page flow* in a web application. One is to plan a central sequence of pages (a “linear page flow”) treating other paths as second-class; the other approach is to view the application as having a fundamentally *web-like* (or graph-like) structure, with each edge equally important. Both viewpoints are common in application design, and are not mutually exclusive: they can be mixed within one design. Links has special features supporting each style, as described here.

Branching page flow

Most web pages have many links, all of more-or-less equal importance. The application presents the user with a spectrum of choices for what to do next, the user drives, and the programmer is indifferent to the user's choice. We call this *branching page flow*.

In Links, such links are specified as expressions embedded in the HTML, whose value is the target page. Because of the usual scoping rules for expressions, the language can statically check that their referents actually exist and are well-typed.

To illustrate, consider an application for managing a set of kitchen recipes and its listing page which displays a list of recipes by name, each linked to a page showing the details of that recipe. When displaying the listing, suppose we have variables `recipeID` and `recipeName` in context. The link for the recipe details page might look as follows:

```
<a l:href="{viewRecipe(recipeID)}">{recipeName}</a>
```

The `l:href` attribute contains an expression whose value will become the next page; by convention this would be a function call, to a function specializing in that sort of page. The content of the anchor tag (`<a>`) is, as always in HTML, the text of the link—in this case just the value of the `recipeName` variable.

The `l:href` attribute is replaced, at runtime, with an ordinary `href` attribute containing a URL that identifies the expression and its environment; this is done using Links' serializable continuations (see Section 2.4).

To support a similar style of page-flow for forms, Links offers the `l:action` attribute for `form` elements. The expression contained in the `l:action` attribute produces the page to display after the form is submitted. It also has access to the form-field values via local variables: it has a variable corresponding to each form field with an `l:name` attribute, as described above under "User interface" in Section 2.4.

So, if we want to allow the user to create a new recipe by entering a name into a form, we could code the form in Links as follows:

```
<form l:action="{createRecipe(recipeName)}">
  New recipe name: <input l:name="recipeName">
</form>
```

This form, when submitted, will invoke the `createRecipe` function, passing as its argument the string entered into the form's single input field.

This method of form construction is useful for simple forms but is severely limited. For one thing, the `l:name` fields are strictly lexically scoped—thus there is no way to create a form fragment and reuse it within several other forms. Similarly, there is no way to compose these forms into larger forms. A more flexible mechanism for form construction, called "Formlets," is also available; it is described in detail in Chapter 4.

Branching page flow is the default in web-application design, and Links helps write well-formed branching page-flows, without broken links, by means of this static scoping. When linear flow is needed, Links offers an additional layer of abstraction, as follows.

Linear page flow

Linear page-flow designs—sometimes called a "pipeline," a "wizard," or an experience "on-rails"—can be seen in applications that drive the user through a sequence of forms—for example, one that collects your credit card, billing address, and shipping address on sequential pages. The user may click other links, such as a "help" link, or abandon the pipeline to start again at a home page, but still we would like to structure the program around the "normal" path. (An analogy can be drawn to the use of exceptions, which are used when we do not want unusual conditions to dominate the shape of the code. Relegating unusual conditions to exception handlers keeps the main line of control clean. Similarly, writing the linear page flow as a direct line of program execution keeps the focus on what's most important.)

With conventional web-programming techniques, there is no way to indicate such a normal path through the control flow of the program. Instead, a conventional web program is structured as a set of independent page handlers, each of which simply emits a string. Each handler must be treated as a new starting point, and must re-establish any needed context when the request comes in. In this approach, the relationship between successive pages in a pipeline is no stronger than that of any two arbitrary pages.

Yet the idea of "web continuations," pioneered by Queinnec [2000] and Graunke et al. [2001a], provides a flexible static control structure to internalize web pipelines. The idea is to extend the language with a function

$$\mathit{sendSuspend} : ((a \to \mathtt{Page}) \to \mathtt{Page}) \to a.$$

The *sendSuspend* function is similar to *call/cc*; invoking *sendSuspend*(*f*) passes to *f* the current continuation, getting back an HTML page which is immediately served to the client. This continuation can be transformed into a URL (see "Serializable continuations," in Section 2.4) and placed in a link, and it represents the user's next option along the pipeline. Following the link causes the computation to continue from the point to which *sendSuspend*(*f*) would return. This way, when the link is clicked, the context surrounding the call to *sendSuspend* (including the value of local variables, for example) is automatically re-established and we jump directly back into the lexical scope we left when we served the page. (As a historical note, Mawl [Atkins et al., 1999] preceded the above papers in offering a facility for, essentially, sending and suspending; however, Mawl's facility did not treat the continuation as a first-class value, the way *sendSuspend* does, and did not permit any independent links on the sent pages.)

The page that *f* produces may, of course, contain other links, produced by other mechanisms; these links depart from the "normal" line of control and take the user to some other part of the program. The link generated from the continuation passed to *f* is only distinguished by the fact that dereferencing it causes control to pass to the continuation of the *sendSuspend* call.

This feature becomes particularly useful when you want to capture a truly dynamic control context, as when you have a subroutine that displays a page in *arbitrary* control context and which needs eventually to display a link back into the original context. Two examples follow.

Example: Modularizing authentication Imagine that, at a certain point in a program, you require the user to be logged in—you need authentication—but if she is not, you want to give her the chance to log in. However, the code you are writing is concerned primarily with some other task, not authentication. Hence you do not want to write special code at this point for the case where the user still needs to log in, which would also necessitate handling detours such as failed logins. Naturally, you'd like to write the login code once and use it for any tasks in need of authentication: you want to *modularize authentication*. Concretely, modularizing the login interaction requires passing to it a representation of the calling context, and creating this representation may be a nuisance.

Figures 2.32–2.33 show how to use `sendSuspend` to solve this problem. The heart of the code is in Figure 2.33. The `loginForm` routine (Figure 2.32) simply acts as an HTML template for displaying the login form. The function `authenticate` (Figure 2.33) is the entry point; it is what you call to ensure that the user is authenticated. When it returns a username, you can be sure that this is the user logged in to the current session. If a user is already logged in on the current web session (established by cookies), then `authenticate` simply returns this user's identity. If not, it holds a conversation with the user to get the login credentials, displaying failure pages as necessary.

By using `sendSuspend`, the `authenticate` routine automatically procures a representation of the calling context (binding it locally to the variable `return`), so there is no need to create and pass an explicit representation by hand. And of course, the `sendSuspend` mechanism automatically captures any data that is in scope in the calling context, so manually capturing these is also avoided.

```
sig loginForm : (String, Handler((username:String, pass:String)))
                -> Page
fun loginForm(msg, return) {
  page
    <html>
      <head> <style>
      label {{float: left; width: 90pt; text-align: right;
              display: inline; margin-right: 3pt; }}
      </style> </head>
      <body>
      <div class="error">{stringToXml(msg)}</div>
      <form l:action="{return((username=name, pass=pass))}"
            method="post">
        <label>Name:
          <div> <input l:name="username" /></div></label>
        <label>Password:
          <div><input l:name="pass" type="password" /></div></label>
        <input type="Submit" />
      </form>
      <a l:href="{main()}">Start over</a>
      </body>
    </html>
}
```

Figure 2.32: Example use of sendSuspend (1): Login form with continuation argument.

```
sig validAuth : (String, String) -> Bool
fun validAuth(name, pass) {   # A simple authentication policy.
  name == "ezra" && pass == "knockknock"
}

sig authenticate : (String) -> String
fun authenticate(msg) {
  var current_user = getCookie("loginname");
  if (current_user <> "")   # User recognized; just return creds.
    current_user
  else {                    # User not recognized, show login page.
    var (username=name, pass=pass) =
      sendSuspend(fun (return) {loginForm(msg, return)});
    if (validAuth(name, pass)) {
      # User logged in successfully; set cookie, return creds.
      setCookie("loginname", name);
      name
    } else
      # User login failed; show this page again with error message.
      authenticate("The password you entered was incorrect")
  }
}
```

Figure 2.33: Example use of sendSuspend (2): Authentication loop.

```
sig sendSuspend : (((a) -> Page) -> Page) -> a
fun sendSuspend(makePage) {
  escape esc in {
    exit(renderPage(makePage(esc)))
  }
}

sig freshResource : () -> ()
fun freshResource() {
  escape esc in {
    redirect("?_cont=" ++ reifyK(esc)); exit(0)
  }
}
```

Figure 2.34: The definitions of sendSuspend and freshResource.

Example: Preventing duplicate actions OR Freshening the resource OR Internalizing POST-redirect-GET Consider another common problem, this time from a user's point of view: You have just submitted a purchase form, your credit card has been charged, and you are given a confirmation page. Later, you decide to refresh this page—or perhaps you simply want to view the page after it has left your browser's cache, and so the browser needs to reload it. You expect it to simply "refresh" the information on the page, but instead the browser asks whether you want to *re-submit the request* which got you to this page, which happens of course to be the purchase request. As a result, refreshing the information requires *purchasing* again—an undesirable situation. This happens because the confirmation page was served directly in response to the purchase request—seemingly a natural thing to do, but with awful consequences.

The *POST-redirect-GET* pattern [Wikipedia, 2008, Jouravlev, 2004] describes how to avoid the problem: After performing any side-effects associated with an HTTP POST request, the pattern advises not to serve a response page immediately, but instead, to redirect the client to a GET request on a different URL; thus the next page the user sees is actually served in response to the GET request, and is safe to reload at any time. (An HTTP POST request is allowed to have side-effects, whereas the more common GET request is required not to have them, or more specifically, not to entail any *obligations* [World Wide Web Consortium, 2003a].) But just as with the login-loop example we just saw, creating the second URL (and its code entry-point) is a nuisance: it requires re-establishing the control and data context, which may require manually marshaling the data into the latter URL.

The *freshResource* routine, defined in the Links prelude, internalizes the POST-redirect-GET pattern, using the continuation-capture and -serialization technology, making it easy to apply.

To use *freshResource*, the programmer simply calls it after doing some destructive action (such as committing a purchase order to a database). After this, if the user reloads the next page, this has the effect of resuming the program from the (most recent) call to *freshResource*. To sketch a use:

```
var confirmationNum = commitPurchase(details);
freshResource();
displayConfirmation(confirmationNum)
```

The implementations of *sendSuspend* and *freshResource* are shown in Figure 2.34. The construct `escape` *esc* `in {` *e* `}` binds *esc* to the continuation of the whole construct, within the expression *e* (the `escape` construct is called `let/cc` in Scheme). The *reifyK* function serializes such a continuation as a string. The *exit* function ends the HTTP request, sending its argument to the browser as the HTTP response body. The *redirect* function issues to the browser an HTTP 302 Redirect response to the given URL, thus effectively taking the user to that URL, taking effect once we call *exit* or exit the program. And *renderPage* simply converts a `Page` structure to an HTML value (type `Xml`). (The `Page` type contains HTML and acts as a container for additional data that could be associated with the page.) The `?_cont=` string is the signal for the Links internals to treat the incoming request as one that resumes a continuation—as opposed to starting at the beginning or performing an RPC call.

Web continuations: a comparison of techniques

The idea of using continuations as a language-level technique to structure web programs has been discussed in the literature [Queinnec, 2000, 2003, Graunke et al., 2001b, Matthews et al., 2004] and used in several web frameworks (such as Seaside, Borges, PLT Scheme, HOP, RIFE and Jetty) and applications, such as ViaWeb's e-commerce application [Graham, 2001b], which became Yahoo! Stores. Most of these take advantage of a call/cc primitive in the source language, although some implement a continuation-like object in other ways. Each of these systems creates URLs out of continuation values. The most common technique for establishing this correspondence is to store the continuation in memory, indexed by a generated unique identifier (a nonce) which is included as part of the URL. This is the technique used by PLT Scheme, Ocsigen, and Seaside for example.

Relying on in-memory tables makes such a system vulnerable to system failures and difficult to distribute across multiple servers. Whether stored in memory or in a database, the continuations can require a lot of storage. Every link on every page served may correspond to one of these continuations. Since URLs can live long in bookmarks, emails, and other media, it is impossible to predict how long a continuation should be retained. Most of the above frameworks destroy a stored continuation after a set period of time. Even with a modest lifetime, the storage cost can be quite high, as each page request may generate many continuations and there may be many requests per minute (maybe hundreds or thousands). No pre-set timeout period seems appropriate, since a user might reasonably leave a page open for a weekend, or bookmark it to return months later; yet the vast majority will be clicked or discarded within moments, never to be needed again.

The Links approach differs by following the strategy described by Graunke et al. [2001a]. It serializes the internal structure of continuations, embedding them in the page, albeit with reference to points in the original program. The structure of a continuation is like a stack of these pointers together with their data environments. Strictly speaking, then, continuations are not *completely* serialized, since the code pointers can only be decoded together with the full program code. But program code is vastly more stable than the data and control contexts that arise during execution: programs normally run for some time before being upgraded.

2.9 Location-aware distributed computing

As noted, a Links thread can move fluidly between client and server. The programmer need not *necessarily* worry about a thread's locality, yet can use client and server annotations to control it when necessary. This feature amounts to a remote-procedure-call system, where remote calls look like any other calls and located functions look like any other functions except for their location annotations.

Attaching client and server annotations has various benefits. Functions labeled with the `server` annotation never have their code transfered to the client, so they remain secret, and also data that does not pass out of server-annotated context will not be transferred to the client. Functions labeled `client` have access to client-side facilities, such as the DOM, and will not execute using server resources, which can be precious. Functions not annotated with a location keyword are located everywhere and thus can be called locally from either location. The Links team suggests that location annotations provide an appropriate level of detail at which to manage the (moderately expensive) transfers of control between client and server.

Typical network infrastructure only permits client-driven requests—the server cannot initiate a transaction with a browser and can only respond to requests. Yet in Links, calls between client and server are symmetrical (either location can call the other). This symmetry is implemented on top of the asymmetric HTTP channel.

The implementation is efficient in the sense that session state, when it is captured automatically, is preserved in the client only, thus requiring no server

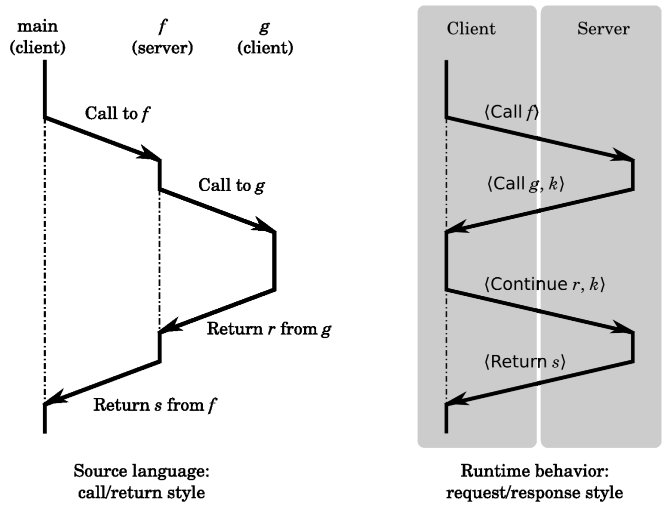


Figure 2.35: Semantics and runtime behaviour of client/server annotations.

resources except when the server is actively working. This should help Links programs to scale well from small to very large numbers of users, since server resources are not held for idle users.

All of this is accomplished by using *continuation-passing style* together with *fully serializable continuations*, and *tunnelling requests through responses*. The technique is shown formally in Chapter 3.

Figure 2.35 shows how the call/return style of programming offered by Links differs from the standard request/response style, and how it uses the latter to emulate the former. The left-hand diagram shows a sequence of calls between functions annotated with `client` and `server`. The solid line indicates the active line of control as it enters these calls, while the dashed line indicates a control context which is waiting for a function to return. In this example, *main* is a client function which calls a server function *f* which in turn calls a client function *g*. The semantics of this call-and-return pattern are familiar to every programmer.

The right-hand diagram shows the same series of calls as they actually occur in Links. The dashed line again indicates a control context. During the time when *g* has been invoked but has not yet returned a value, the server need not locally store the control context—or for that matter anything else about the ongoing computation—though the language provides an illusion that f is "still waiting" on the server for the value from g. Instead, the server's state is encapsulated in the value k, which it sends to the client along with a specification of the client-side call to perform, including a function reference and any arguments.

To orchestrate this interaction, the compiler translates the single source program into two targets, one for the client and one for the server. In this compilation step, non-local functions for each side are replaced by a *stub function*, which, rather than implementing the function directly, implements it by means of a remote procedure call (RPC) to the other location. On the client side, this is easy: the client simply makes an HTTP request indicating the server function and its arguments; the client-side thread then waits while the server function executes.

On the server side, the stub function does not make a new HTTP request but instead uses the existing HTTP connection as a channel on which to communicate (of course, the server-side program, to be running at all, must already be handling a client request). So the stub simply gives an HTTP response including a representation of the call (the callee and its arguments) and the server's own continuation. Upon returning in this way, all of the state of the Links program is present at the client, so the server need not store anything more.

The client recognizes this form of return and carries out the requested RPC call. When it has completed the call, it places a new request to the server, passing the function result and the server continuation. The server then resumes its computation by applying the continuation to the result.

If a top-level function definition does not have a location annotation then both locations receive a full implementation, not a stub. Location annotations are presently only allowed on top-level function definitions.

In short, a location annotation $a \in \{\texttt{client}, \texttt{server}\}$ requires that "every operation appearing lexically within the function must be executed at location a." The semantics of these annotations is formalized in Chapter 3.

2.10 Composable form abstraction

Ordinary HTML forms are flat and uncomposable: all fields within a form share the same namespace, so combining two sets of fields in a form can cause clashes—particularly so if the programmer wants to reuse a component more than once within a form.

The Links library offers a set of composable abstractions for building forms out of sub-forms, an abstraction called *formlets*. The Links system also implements syntactic sugar for constructing formlets in a concrete, HTML-like way. Formlets are discussed in detail in Chapter 4.

2.11 Language-integrated query

As noted, Links supports querying relational databases from within the language: suitable expressions are automatically transformed into SQL queries.

Not every expression can be translated safely to an SQL query: some have side-effects, and others just perform computations beyond the SQL's bourne of expressiveness. So, we call those expressions that can be translated as "queryizable."

The original version of Links recognized queryizable expressions in an ad-hoc fashion, using a custom algorithm. There was no easy way for programmers to predict in advance which expressions would be translated, which could lead to very bad performance.

Cooper et al. [2006] displayed a grammar for queryizable expressions and a rewrite system for doing the translation; but this grammar was very conservative. The system as implemented translated some expressions beyond the grammar given; and the grammar did not admit many useful constructs—such as functional abstraction—because of the difficulty of deciding SQLizability statically in the presence of these restrictions.

At last, then, Links has implemented a source annotation which asserts that the annotated expression ought to be queryizable—and if not, the programmer gets a static error. The static analysis is now much more permissive, allowing functional abstraction, for example.

The analysis and translation are formalized in Chapter 5.

2.12 Related work

Web frameworks and languages

The effort to simplify web development through programming abstractions is by no means new. A plethora of server-side “web frameworks” (essentially, libraries) exist to provide structure and common tools for making web applications. Popular examples include Ruby on Rails, Django, Catalyst, and liftweb. The level of abstraction provided by these is modest. They sit squarely within the server part of the web execution model. They typically offer services including URL dispatching (to top-level functions with no session-specific data context), functions providing a cushion on top of the HTTP protocol (e.g. for reading/writing HTTP request/response parameters), and data encoding (e.g. for delivery to the client through formats like JSON).

Client-side web frameworks Libraries for JavaScript, DOM and AJAX development have gone a long way toward simplifying the task of making dynamic web applications. Typically, these offer a browser-independent suite of operations for things like HTTP requests, CSS, DOM tree and DOM event access and manipulation, visual effects, periodic execution, observers, and event listening. Such frameworks include prototype.js, mochikit, dojo, and YUI, the Yahoo! UI library.

Research languages Pushing farther ahead, many research languages aim to provide, as Links does, higher-level abstractions and stronger static guarantees than what a library can offer. The following is not an exhaustive list, but aims to cover most of those that stand out in some special way.

Mawl [Atkins et al., 1999] is perhaps the first language system to abstract from HTTP request/response interactions to a more structured control flow across pages. A Mawl session is a sequential program where certain operations emit a form and continue the program after form submission. The Mawl system serializes program state (which includes mutable session-specific variables) between requests, using a session identifier embedded in each page to track this.

The <bigwig> project [Brabrand et al., 2002, 1999] is another early project to develop a web-centric programming language, also following a session-centric approach. In <bigwig>, session state is held in a server-side thread kept running for each session (this contrasts with the continuation-storing approach seen in most of the other systems). It is difficult to see how this approach would work in a typical load-balanced server environment (see Section 2.5). Within a particular <bigwig> session, every HTML page is served at the same URL, so that whatever the user does within that session she always sees a "most recent" state of the session; due to this design, <bigwig> does not support many common web interactions—at least not without extensive coding—for example, those where the user uses the browser's back button, opens multiple windows onto the site, or sends a URL to a friend. However, <bigwig> pioneers a number of novel, interesting features: a domain-specific language for form validation, called PowerForms [Brabrand et al., 2000], and a temporal-logic-based language for controlling the synchronization between concurrent threads; this powerful language allows expressing a variety of interesting synchronization patterns. The <bigwig> system also incorporates HTML syntax and statically verifies the validity under HTML 4 (*not* an XML format) of the documents it serves.

The Java-based language JWIG [Christensen et al., 2003], a successor to <bigwig>, follows that system in its session-centric model, also using running threads to manage client sessions.

WASH/CGI [Thiemann, 2002, 2005] is a Haskell library for web development that, among many other things, supports designing linear page-flow. It does this using a monad which, when generating a page, embeds the history of the session. This allows it to replay the session when a subsequent request comes in, resuming at the point where the page was issued. For data persistence, WASH provides a transactional interface to a relational database as well as its own data-persistence model, which stores native Haskell values. WASH supports form design and composition by offering input-field primitives and field-tupling combinators which act as input fields themselves.

The PLT Scheme webserver pioneered the definition of linear page-flow through continuation capture [Graunke et al., 2001c], using the `send/suspend` primitive on which Links' `sendSuspend` was based. PLT Scheme also comes with an implementation of the formlet abstraction, described in Chapter 4.

Ocsigen [Balat, 2006], is an OCaml web-programming framework; it associates URLs with closures, using the server-side storage technique. It has a sophisticated URL dispatcher with a library of routines that associate closures with URL path-parts, thereby giving programmers some control over the text of URLs (the *path part* of a URL is, loosely, the part following the domain name). This associates a path with a program point; the data environment needed to represent the closure is attached automatically to the request. This dispatcher library provides several ways to dynamically overlay the set of registered paths; for example, it supports establishing a URL path for a particular user session.

HOP [Serrano et al., 2006], a Scheme-like language, compiles to JavaScript and permits declaring server-based "services," analogous to Links' `server`-annotated functions, as well as an asynchronous notification service which allows events at the server to signal to the client. HOP has an interesting "dual evaluation" strategy with two "strata" (main, or server, and GUI, or client). HOP evaluates the program once at the server to produce a web page which is sent to the client. Quasi-quoted code within this page, comprising the GUI stratum, is compiled to JavaScript. The two strata can syntactically intermingle, so main-stratum code embedded in the GUI stratum is evaluated to become a value that the GUI stratum can use directly. HOP *services* are main-stratum functions, to be run at the server, which the GUI stratum can invoke.

The functional logic programming language Curry has a library called WUI (for Web User Interfaces) which allows constructing composable HTML forms, which can include validation conditions on the input [Hanus, 2006, 2007]. The validation conditions can be compiled to the client for more efficient testing, and are also evaluated on the server since clients are not trusted.

The iTasks library for the language Clean supports creating a certain type of web-based interface. The objects called iTasks represent work flows (which are loosely related to linear page-flows in our sense) and admit combinators, for example, to set them in parallel or in sequence. iTasks work flows support combinations that page-flows do not, for example where two user sessions synchronize—each must complete parts of the work in order for both to continue together.

The haXe language (`http://haxe.org/`) is a statically-typed web-oriented language which compiles to several relevant targets: JavaScript, Flash, and PHP, to name a few. It provides a convenience library for making network connections, to aid in writing applications with client–server interaction, rather than an integrated, language-based approach.

Work by Petříček and Syme [2007] extends the language F♯ to provide location-aware distribution. This work uses monads to capture the continuation at each point, which allows a runtime passing of continuations between client and server. The typing of monadic computations also forces a location constraint consistently throughout a monadic code block.

Flapjax [Meyerovich, 2007] departs from all the above, taking a rather different approach to interactivity, following *functional-reactive programming* [Elliott and Hudak, 1997]. A Flapjax program defines a single page as a time-varying function of time-varying signals, such as the state of the keyboard and mouse or even network-fetched data. In this model it is easy to have various indicators that continually read as functions of some other controls, without worrying about the mechanics of updating them.

The Ur/Web system, based on the Ur language by Chlipala [2008] provides XML quasiquotation syntax and statically-checked branching page flow. It directly embeds SQL syntax for accessing databases. For describing the client-side behavior of an application, it uses functional-reactive programming. It provides seamless interaction between client and server using a “location-oblivious” syntax—location annotations are not required or allowed.

The issue of page-flow design is an implicit, rather than explicit, theme of much of this research. In Mawl, WASH, <bigwig>, and JWIG, linear page-flow is the default, while in the others surveyed here, branching page-flow is the default. In the former set, the programmer must explicitly branch when she wants to offer the user a choice of outward links. In the latter set, re-establishing context after a page transition is the more difficult thing to code.

Another theme is client-server integration. Some of the languages provide integration in a location-oblivious way (Ur/Web) and some with location-awareness (HOP, F♯). There are also those that generate client and server code without language-based integration (haXe) and those that target only the client or only the server (Mawl, <bigwig>, JWIG, Ocsigen, WASH, Flapjax).

Ocsigen, WASH, and JWIG all validate their output pages as XHTML.

2.13 Conclusion

Links has a set of unique or unusual features which fit together to create a novel web-programming experience. Most significantly, it advances a "web execution model" of programming, an environment where user events, private server-side logic, and back-end services (such as data persistence) are all immediately available to one integrated program, thus ameliorating the notorious impedance mismatch.

Amongst other things, it offers specialized annotations for client–server interaction, a library and syntactic extension for client-side forms programming, and language-integrated query with statically-detectable queryizability.

The next chapters present specific technical contributions to each of these latter three functional areas—that is, to the technology of web-centric programming.

Chapter 3

The RPC Calculus

(This chapter represents sole work by the author, advised by Philip Wadler.)

3.1 Introduction

Designing a web server requires thinking carefully about user state and how to manage it. Unlike a desktop application, which deals with one user at a time, or a traditional multi-user networked system, whose environment is more controlled, even a modest web system can expect to deal with tens or hundreds of thousands of users in a day, each one can have multiple windows open on the site simultaneously, and these users can disappear at any time without notifying the server. This makes it infeasible for a web server to maintain state regarding a user's session. The mantra of web programming is: Get the state out!—get it out of the server and into the client. An efficient web server will respond to each request quickly and then forget about it even quicker.

Nonetheless, several recent programming language designs [Murphy, 2007, Neubauer and Thiemann, 2005, Neubauer, 2007] allow the programmer the illusion of a persistent environment encompassing both client and server; let us call these "location-aware languages." This allows the programmer to move control back and forth freely between client and server, using local resources on either side as necessary, but still expressing the program in one language. This chapter shows how to implement a location-aware language in an environment with a *stateful client* and a *stateless server*.

Murphy et al. [2004] introduced a location-aware core calculus, Lambda 5, and Murphy [2007] built upon this a full-fledged programming language, ML5, also showing how to compile it to separate code for client and server. Neubauer and Thiemann [2005] also introduced a variant of ML with location annotations and showed how to perform a splitting transformation to produce code for each location. However, each of these works relied on concurrently-running stateful peers. The present work is novel in implementing a location-aware language on a stateless-server substrate.

The technique of this chapter involves three essential steps: defunctionalization *à la* Reynolds, CPS translation, and a trampoline [Ganz et al., 1999] that allows tunnelling server-to-client requests within server responses. CPS and defunctionalization were used by Matthews et al. [2004] to implement linear page-flow design (Section 2.8). This chapter adds a trampoline to the toolbox, supporting the distinct problem of implementing RPC calls in a location-aware language.

A version of this feature is built into the Links language [Cooper et al., 2006]. The current Links version is limited in that only calls to top-level functions can pass control between client and server. This chapter formalizes the implementation and shows how to relax the top-level-function restriction. This has some impact on the compiler: the current, limited version requires just a CPS translation and a trampoline; defunctionalization is needed in implementing nested remote-function definitions.

This chapter This chapter presents a simple higher-order λ-calculus enriched with location annotations, allowing the programmer to indicate the location where a fragment of code should execute. The semantics of this calculus clarifies where each computation step is allowed to take place. This can be seen as a semantics of a language with Remote Procedure Call (RPC) features built in.

It then gives a translation from this calculus to a *first-order* abstract machine that models an asymmetrical client–server environment, and show that the translation preserves the locative semantics of the source calculus.

As a stepping stone to the full translation, we formally define defunctionalization. By isolating this phase of the larger translation, this aims to clarify the notation and formal techniques that will be used in the full translation. While there are many compact definitions of CPS translations in the literature, no such compact and formal definition of defunctionalization has appeared; this chapter gives a complete formal definition of defunctionalization in eleven lines.

Syntax $\boxed{\lambda_{\text{src}}}$

$$\begin{array}{rrcl}
\text{constants} & c & & \\
\text{variables} & x & & \\
\text{terms} & L,M,N & ::= & LM \mid \lambda x.N \mid x \mid c \\
\text{values} & V,W & ::= & \lambda x.N \mid x \mid c
\end{array}$$

Semantics $\boxed{M \Downarrow V}$

$$V \Downarrow V \qquad \text{(VALUE)}$$

$$\frac{L \Downarrow \lambda x.N \qquad M \Downarrow W \qquad N\{W/x\} \Downarrow V}{LM \Downarrow V} \qquad \text{(BETA)}$$

Figure 3.1: Source language λ_{src}, a higher-order λ-calculus.

3.2 Defunctionalization

Source calculus, λ_{src}

Figure 3.1 shows an entirely pedestrian call-by-value λ-calculus, called λ_{src}, whose semantics is defined by a big-step reduction relation $M \Downarrow V$, stating that term M reduces to value V. We write $N\{V/x\}$ for the capture-avoiding substitution of a value V for the variable x in the term N, and $N\{V_1/x_1,\dots,V_n/x_n\}$ for the simultaneous capture-avoiding substitution of each V_i for the corresponding x_i. We let σ range over these substitutions. We identify α-equivalent terms.

First-order machine

The defunctionalized machine, DM, defined in Figure 3.2, is a first-order abstract machine, in contrast to the λ_{src} calculus which allowed arbitrary expressions in the function position of an application. In DM, terms, ranged over by L, M, and N, are first-order; they are built from constants c, variables x, constructor applications $F(\vec{M})$, function applications $f(\vec{M})$ and case expressions $\text{case}\, M \text{ of } \mathscr{A}$. A list $\mathscr{A}$ of case alternatives is well-formed if it uniquely maps each name.

Syntax $\boxed{\text{DM}}$

$$\begin{array}{rrcl}
\text{variables} & x, y, z & & \\
\text{function names} & f, g & & \\
\text{constructors} & F, G & & \\
\text{values} & V, W, U & ::= & c \mid x \mid F(\vec{V}) \\
\text{terms} & M, N & ::= & c \mid x \mid F(\vec{M}) \mid \\
 & & & f(\vec{M}) \mid \mathsf{case}\, M \,\mathsf{of}\, \mathscr{A} \\
\text{alternative sets} & \mathscr{A} & & \text{a set of } A \text{ items} \\
\text{case alternatives} & A & ::= & F(\vec{x}) \Rightarrow M \\
\text{eval. contexts} & E & ::= & [\ ] \mid f(\vec{V}, E, \vec{M}) \mid \\
 & & & F(\vec{V}, E, \vec{M}) \mid \mathsf{case}\, E \,\mathsf{of}\, \mathscr{A} \\
\text{function def.} & D & ::= & f(\vec{x}) = M \\
\text{definition set} & \mathscr{D} & ::= & \mathsf{letrec}\, D \,\mathsf{and} \cdots \mathsf{and}\, D
\end{array}$$

Semantics $\boxed{M \longrightarrow_{\mathscr{D}} N}$

$$\begin{array}{rcll}
E[f(\vec{V})] & \longrightarrow_{\mathscr{D}} & E[M\{\vec{V}/\vec{x}\}] & \text{if } (f(\vec{x}) = M) \in \mathscr{D} \\
E[\mathsf{case}\,(F(\vec{V}))\,\mathsf{of}\,\mathscr{A}] & \longrightarrow_{\mathscr{D}} & E[M\{\vec{V}/\vec{x}\}] & \text{if } (F(\vec{x}) \Rightarrow M) \in \mathscr{A}
\end{array}$$

Figure 3.2: First-order target, DM (the Defunctionalized Machine).

The machine also uses a set $\mathscr{D}$ of function definitions. Each has the form $f(\vec{x}) = M$, defining a function called f taking parameters $\vec{x}$ which are then bound in the function body M. A definition set is well-formed if it uniquely defines each name. The definitions are mutually recursive, so the scope of each definition extends throughout all the other definitions as well as the term under evaluation.

The semantics is defined as a small-step reduction relation $M \longrightarrow_{\mathscr{D}} N$ stating that the term M reduces to the term N in the context of definition-set $\mathscr{D}$. The relation $\twoheadrightarrow_{\mathscr{D}}$ is the reflexive, transitive closure of $\longrightarrow_{\mathscr{D}}$, with the definitions $\mathscr{D}$ held fixed through the reduction sequence.

$$
\begin{array}{rcll}
[\![\lambda x.N]\!] & = & \ulcorner \lambda x.N \urcorner(\vec{y}) & \text{where } \vec{y} = \textsc{fv}(\lambda x.N) \\
[\![x]\!] & = & x & \\
[\![c]\!] & = & c & \\
[\![LM]\!] & = & \mathit{apply}([\![L]\!], [\![M]\!]) & \\
\\
\text{coll } f\ LM & = & f(LM) \cup \text{coll } f\ L \cup \text{coll } f\ M & \\
\text{coll } f\ \lambda x.N & = & f(\lambda x.N) \cup \text{coll } f\ N & \\
\text{coll } f\ V & = & f(V) & \text{if } V \neq \lambda x.N \\
\\
[\![M]\!]^{\text{top}} & = & \mathsf{letrec}\ \mathit{apply}(\mathit{fun}, \mathit{arg}) = \mathsf{case}\ \mathit{fun}\ \mathsf{of}\ [\![M]\!]^{\text{fun}} & \\
\\
[\![\lambda x.N]\!]^{\text{fun,aux}} & = & \{\ulcorner \lambda x.N \urcorner(\vec{y}) \Rightarrow [\![N]\!]\{\mathit{arg}/x\}\} & \text{where } \vec{y} = \textsc{fv}(\lambda x.N) \\
[\![M]\!]^{\text{fun,aux}} & = & \{\} & \text{if } M \neq \lambda x.N \\
\\
[\![M]\!]^{\text{fun}} & = & \text{coll } [\![-]\!]^{\text{fun,aux}}\ M &
\end{array}
$$

Figure 3.3: Defunctionalization.

Defunctionalization

Defunctionalization is a translation from the terms of λ_{src} to the terms and definition-sets of DM; it is defined in Figure 3.3. From a term M we compute a defunctionalized term $[\![M]\!]$ and a corresponding definition set, $[\![M]\!]^{\text{top}}$. Let *arg* be a special reserved variable name not appearing in the source program. The coll function is used by $[\![M]\!]^{\text{top}}$ to traverse a term: it applies a function f to each subterm of its argument, collecting the results.

A special feature of our translation is the use of an injective function that maps source terms into the space of constructor names. We write $\ulcorner M \urcorner$ for the name assigned to the term M by this function. One example of such a function is the one that collects the names assigned to immediate subterms and uses a hash function to digest these into a new name. (In this case, the issue of possible hash collisions would have to be treated delicately.) Previous formalizations of

$$\begin{array}{rcl}
[\![F(\vec{M})]\!]^{-1}_{\mathscr{D}} & = & \lambda x.([\![N\{x/arg\}]\!]^{-1}_{\mathscr{D}})\{[\![\vec{M}]\!]^{-1}_{\mathscr{D}}/\vec{y}\} \\
 & & \text{if } (F(\vec{y}) \Rightarrow N) \in \text{cases}(apply, \mathscr{D}), \text{ where } x \text{ fresh for } \vec{y}, \vec{V} \\
[\![x]\!]^{-1}_{\mathscr{D}} & = & x \\
[\![c]\!]^{-1}_{\mathscr{D}} & = & c \\
[\![apply(L,M)]\!]^{-1}_{\mathscr{D}} & = & [\![L]\!]^{-1}_{\mathscr{D}}\ [\![M]\!]^{-1}_{\mathscr{D}}
\end{array}$$

Figure 3.4: A reverse translation for defunctionalization.

defunctionalization treat the abstractions as already carrying labels; this chapter allows any injective function, which might in fact depend on context.

However, in a system like Links, it is essential to have a *stable labeling*, that is, one which always gives the same label for a given term, regardless of its context. This would not be the case for, say, a serial-numbering algorithm, which assigns labels serially as it traverses a term. With a stable labeling, RPC calls are robust even if the server has to reboot while the client is working, or if successive calls enter at different server machines in a server cluster, or even if an unrelated part of the program changes between calls.

We also use a reverse translation, $[\![-]\!]^{-1}$, a retraction of $[\![-]\!]$, defined in Figure 3.4. Here $[\![M]\!]^{-1}_{\mathscr{D}}$ denotes the retraction of a DM term M in the context of definitions $\mathscr{D}$. The reverse translation is undefined when the definition-set $\mathscr{D}$ does not define all the constructors appearing in M. We lift the reverse translation to lists of values and to substitutions:

$$\frac{\sigma = \{V_1/x_1, \dots, V_n/x_n\}}{[\![\sigma]\!]^{-1}_{\mathscr{D}} = \{[\![V_1]\!]^{-1}_{\mathscr{D}}/x_1, \dots, [\![V_n]\!]^{-1}_{\mathscr{D}}/x_n\}}$$

To extract the alternatives of case-expressions from function bodies, we use a function cases, defined as follows.

Notation is carefully (ab)used when we write things like $[\![\vec{M}]\!]^{-1}$ to indicate the vector obtained by applying a translation $[\![-]\!]^{-1}$ to each member of $\vec{M}$ in place.

Definition. Define a meta-function cases that returns the branches of a given function when that function is defined by case-analysis. Formally, define cases by

$$\frac{(f(x,\vec{y}) = \mathsf{case}\, \mathit{fun}\ \mathsf{of}\ \mathscr{A}) \in \mathscr{D}}{\mathrm{cases}(f,\mathscr{D}) = \mathscr{A}}$$

This completes the definition of the reverse translation $[\![-]\!]^{-1}$.

The central claim of this section is that the translation is sound and complete: the DM correctly simulates every λ_{src} computation. Before getting to the proof we establish some preliminaries.

During reduction we may lose subterms which would have given rise to defunctionalized definitions; thus the reduction of a term does not have the same definition-set as its ancestor. Still, all the definitions it needs were generated by the original term; we formalize this as follows.

Definition (Definition containment)**.** We say that a definition set $\mathscr{D}$ *contains* $\mathscr{D}'$, written $\mathscr{D} \geqslant \mathscr{D}'$, iff

$$\mathrm{cases}(\mathit{apply},\mathscr{D}) \supseteq \mathrm{cases}(\mathit{apply},\mathscr{D}').$$

Next, the function $[\![-]\!]^{-1}_{-}$ inverts the functions $([\![-]\!], [\![-]\!]^{\mathrm{top}})$, as follows.

Observation (Retraction)**.** If $\mathscr{D} \geqslant [\![M]\!]^{\mathrm{top}}$ then $[\![[\![M]\!]]\!]^{-1}_{\mathscr{D}} = M$. Note that the forward translation replaces each variable bound by a λ-abstraction with the variable arg, and the reverse translation generates fresh names when generating abstractions. This still produces an equality, since we identify α-equivalent terms.

Contrariwise, $[\![[\![M]\!]^{-1}_{\mathscr{D}}]\!]$ does not always equal M, because $[\![-]\!]^{-1}_{-}$ is not injective. Injectivity fails when translating a term $F(\vec{M})$, at the point where we perform the substitution, because many substitutions can give rise to the same term.

The reverse translation commutes with substitution, as follows.

Lemma 1 (Substitution–Reverse translation)**.** Given definition-set $\mathscr{D}$, term M and value V, we have $[\![M\{V/x\}]\!]^{-1}_{\mathscr{D}} = [\![M]\!]^{-1}_{\mathscr{D}}\{[\![V]\!]^{-1}_{\mathscr{D}}/x\}$.

Proof. By induction on the structure of M. The proof of each case is a simple matter of pushing the substitutions down through the terms, applying the inductive hypothesis, and pulling them back up the terms. □

Corollary. The reverse translation commutes with substitutions. Given definition-set $\mathscr{D}$, term M and substitution σ, if $\mathscr{D} \geqslant [\![M]\!]^{\text{top}}$, we have $[\![M\sigma]\!]^{-1}_{\mathscr{D}} = [\![M]\!]^{-1}_{\mathscr{D}}[\![\sigma]\!]^{-1}$.

Observation. If $\mathscr{D} \geqslant [\![M]\!]^{\text{top}}$ then $\mathscr{D} \geqslant [\![M']\!]^{\text{top}}$ for any subterm M' of M.

Lemma 2 (Closure, definition sets)**.** Given $\mathscr{D}$ in the image of $[\![-]\!]^{\text{top}}$ and N such that $F(\vec{y}) \Rightarrow [\![N]\!]\{arg/x\}$ is in cases($apply$,$\mathscr{D}$), we have $\mathscr{D} \geqslant [\![N]\!]^{\text{top}}$.

Proof. Let M be the term such that $\mathscr{D} = [\![M]\!]^{\text{top}}$. Each element of cases($apply$,$\mathscr{D}$) that has a right-hand side of the form $[\![N]\!]\{arg/x\}$ is produced by a term $\lambda x.N$, a subterm of M. As N is a subterm of M, then, $\mathscr{D} \geqslant [\![N]\!]^{\text{top}}$. □

Corollary. Given $\mathscr{D}$ in the image of $[\![-]\!]^{\text{top}}$ and N' such that $F(\vec{y}) \Rightarrow N'$ is in cases($apply$,$\mathscr{D}$), we have that $[\![N']\!]^{-1}_{\mathscr{D}}$ is defined.

Correctness of the translation from λ_{src} to DM

The correctness comes in two halves: soundness and completeness.

First we will show that DM correctly simulates λ_{src}: the translation of a λ_{src} term reduces in DM to a value which maps back to the value of the original term.

The soundness lemma uses a common trick used when relating big- and small-step semantics, of generalizing the statement to handle arbitrary substitutions, which facilitates the induction.

In proofs, the interjection *huzzah!* marks the end of a case in a proof by cases; at this point any metavariables introduced for that case go out of scope.

Lemma 3 (Soundness)**.** Given a λ_{src} term M and a DM value V', substitution σ and definitions $\mathscr{D}$ with $\mathscr{D} \geqslant [\![M]\!]^{\text{top}}$,

$$[\![M]\!]\sigma \twoheadrightarrow_{\mathscr{D}} V' \quad \text{implies} \quad M[\![\sigma]\!]^{-1}_{\mathscr{D}} \Downarrow [\![V']\!]^{-1}_{\mathscr{D}}$$

Proof. By induction on the length of the reduction $[\![M]\!]\sigma \twoheadrightarrow V'$.

Throughout all the reduction sequences, the definition set $\mathscr{D}$ stays fixed.

We take cases on the structure of the term M.

CASE V. The reduction is of zero steps, $[\![V]\!]\sigma \twoheadrightarrow [\![V]\!]\sigma$, and $[\![[\![V]\!]\sigma]\!]^{-1}_{\mathscr{D}} = V[\![\sigma]\!]^{-1}_{\mathscr{D}}$.
The judgment $V[\![\sigma]\!]^{-1}_{\mathscr{D}} \Downarrow V[\![\sigma]\!]^{-1}_{\mathscr{D}}$ is by VALUE. *huzzah!*

CASE LM. By hypothesis, $[\![LM]\!]\sigma \twoheadrightarrow V'$. Recall that

$$[\![LM]\!]\sigma = \mathit{apply}([\![L]\!]\sigma, [\![M]\!]\sigma).$$

It must be that $[\![L]\!]\sigma$ and $[\![M]\!]\sigma$ each reduce to some value, for if not the reduction would get stuck or diverge, when we know it reduces to V', and by the same token $[\![L]\!]\sigma$ must reduce to a constructor application $F(\vec{V})$, or it would make the whole term stuck. Let W be the value to which $[\![M]\!]\sigma$ reduces. Let x be a fresh variable and let N and $\vec{y}$ be such that $(F(\vec{y}) \Rightarrow [\![N]\!]\{\mathit{arg}/x\}) \in \text{cases}(\mathit{apply}, \mathscr{D})$. (We know the body of the case for F has the form $[\![N]\!]\{\mathit{arg}/x\}$ because it is in the image of the translation $[\![-]\!]^{\text{top}}$.) As such we have

$$[\![F(\vec{V})]\!]^{-1}_{\mathscr{D}} = \lambda x.N\{[\![\vec{V}]\!]^{-1}_{\mathscr{D}}/\vec{y}\}.$$

The reduction breaks down as follows:

$$\begin{array}{rl} & \mathit{apply}([\![L]\!]\sigma, [\![M]\!]\sigma) \\ \twoheadrightarrow & \mathit{apply}(F(\vec{V}), W) \\ \twoheadrightarrow & [\![N]\!]\{\vec{V}/\vec{y}, W/x\} \\ \twoheadrightarrow & V' \end{array}$$

Now we apply the inductive hypothesis three times. The inductive hypothesis applies because (1) the reductions used are shorter than the present one, and (2) the definitions in $[\![L]\!]^{\mathrm{top}}$, $[\![M]\!]^{\mathrm{top}}$ and $[\![N]\!]^{\mathrm{top}}$ are all contained in $\mathscr{D}$. The definitions $[\![L]\!]^{\mathrm{top}}$ and $[\![M]\!]^{\mathrm{top}}$ are contained because they are subterms, and $[\![N]\!]^{\mathrm{top}}$ is contained by dint of Lemma 2.

From $[\![L]\!]\sigma \twoheadrightarrow F(\vec{V})$ we get

$$L[\![\sigma]\!]^{-1}_{\mathscr{D}} \Downarrow \lambda x.N\{[\![\vec{V}]\!]^{-1}_{\mathscr{D}}/\vec{y}\}.$$

From $[\![M]\!]\sigma \twoheadrightarrow W$ we get

$$M[\![\sigma]\!]^{-1}_{\mathscr{D}} \Downarrow [\![W]\!]^{-1}_{\mathscr{D}}.$$

From $[\![N]\!]\{\vec{V}/\vec{y}, W/x\} \twoheadrightarrow V'$ we get

$$N\{[\![\vec{V}]\!]^{-1}_{\mathscr{D}}/\vec{y}, [\![W]\!]^{-1}_{\mathscr{D}}/x\} \Downarrow [\![V']\!]^{-1}_{\mathscr{D}}.$$

By the freshness of x, we can separate the substitutions:

$$N\{[\![\vec{V}]\!]^{-1}_{\mathscr{D}}/\vec{y}\}\{[\![W]\!]^{-1}_{\mathscr{D}}/x\} \Downarrow [\![V']\!]^{-1}_{\mathscr{D}}.$$

The judgment $(LM)[\![\sigma]\!]^{-1}_{\mathscr{D}} \Downarrow [\![V']\!]^{-1}_{\mathscr{D}}$ follows by BETA. *huzzah!* □

The above lemma can be summarized by this diagram, which shows that whatever the translation of a λ_{src} term reduces to, that value encodes the "correct answer" as given by the semantics of λ_{src} (dotted lines indicate relationships guaranteed by the lemma when the solid lines are present):

$$\begin{tikzcd}
M[\![\sigma]\!]_{\mathscr{D}}^{-1} \arrow[r, dashed, "\Downarrow"] & {[\![V']\!]_{\mathscr{D}}^{-1}} \\
{[\![M]\!]\sigma} \arrow[r, "\longrightarrow\!\!\!\!\!\twoheadrightarrow_{\mathscr{D}}"] \arrow[u, dashed, "-" ', no head] \arrow[u, dashed] & V' \arrow[u, dashed]
\end{tikzcd}$$

We could show that $M'\sigma \twoheadrightarrow_{\mathscr{D}} V'$ implies $[\![M'\sigma]\!]_{\mathscr{D}}^{-1} \Downarrow [\![V']\!]_{\mathscr{D}}^{-1}$ but this would not give a nice symmetry between this and the following lemma.

Next we see that every λ_{src} computation can be simulated by one in DM.

Lemma 4 (Completeness)**.** Given any DM term M', definitions $\mathscr{D}$ and λ_{src} value V,

$$[\![M']\!]_{\mathscr{D}}^{-1} \Downarrow V \text{ implies there exists } V' \text{ with } [\![V']\!]_{\mathscr{D}}^{-1} = V \text{ and } M' \twoheadrightarrow_{\mathscr{D}} V'.$$

Proof. By induction on the derivation $[\![M']\!]_{\mathscr{D}}^{-1} \Downarrow V$. Take cases on the final step of the derivation:

CASE VALUE. The high-level reduction is $V \Downarrow V$. The initial low-level term must be a value, V', in order to reverse-translate to a value. Then the DM reduction is of zero steps: $V' \twoheadrightarrow V'$. *huzzah!*

CASE BETA. Recall the rule:

$$\frac{L \Downarrow \lambda x.N \qquad M \Downarrow W \qquad N\{W/x\} \Downarrow V}{LM \Downarrow V}$$

Because the starting DM term maps to LM under $[\![-]\!]_{\mathscr{D}}^{-1}$, it must be of the form $apply(L', M')$ with $[\![L']\!]_{\mathscr{D}}^{-1} = L$ and $[\![M']\!]_{\mathscr{D}}^{-1} = M$.

By IH we have normal forms

$$L' \twoheadrightarrow_{\mathscr{D}} F(\vec{V}) \qquad M' \twoheadrightarrow_{\mathscr{D}} W'$$

satisfying

$$[\![F(\vec{V})]\!]_{\mathscr{D}}^{-1} = \lambda x.N \qquad [\![W']\!]_{\mathscr{D}}^{-1} = W$$

with N' such that

$$\begin{array}{ll} & (F(\vec{y}) \Rightarrow N') \in \text{cases}(\mathit{apply}, \mathscr{D}) \\ \text{and} & [\![N'\{x/\mathit{arg}\}]\!]^{-1}_{\mathscr{D}}\{[\![\vec{V}]\!]^{-1}_{\mathscr{D}}/\vec{y}\} = N \qquad (\text{def. of } [\![F(\vec{V})]\!]^{-1}_{\mathscr{D}}), \end{array}$$

Therefore

$$\begin{array}{rcl} [\![N'\{\vec{V}/\vec{y}\}\{x/\mathit{arg}\}]\!]^{-1}_{\mathscr{D}} & = & N \quad \text{and} \\ [\![N'\{\vec{V}/\vec{y}\}\{x/\mathit{arg}\}]\!]^{-1}_{\mathscr{D}}\{[\![W']\!]^{-1}_{\mathscr{D}}/x\} & = & N\{W/x\} \\ & = & [\![N'\{\vec{V}/\vec{y}\}\{x/\mathit{arg}\}\{W'/x\}]\!]^{-1}_{\mathscr{D}} \end{array}$$

And so, by the IH,

$$\begin{array}{rcl} N'\{\vec{V}/\vec{y}\}\{W'/\mathit{arg}\} & = & N'\{\vec{V}/\vec{y}\}\{x/\mathit{arg}\}\{W'/x\} \twoheadrightarrow_{\mathscr{D}} V' \\ & & \text{with } [\![V']\!]^{-1}_{\mathscr{D}} = V. \end{array}$$

So the term reduces as follows:

$$\begin{array}{rcl} \mathit{apply}(L', M') & \twoheadrightarrow_{\mathscr{D}} & \mathit{apply}(F(\vec{V}), W') \\ & \longrightarrow_{\mathscr{D}} & N'\{\vec{V}/\vec{y}\}\{W'/\mathit{arg}\} \\ & \twoheadrightarrow_{\mathscr{D}} & V' \end{array}$$

which was to be shown. *huzzah!* □

The above lemma can be summarized with this diagram, which shows that the evaluation of a term in the high-level language can be simulated by translating the term to the low-level language, reducing it there, and translating it back.

$$\begin{tikzcd} {[\![M']\!]^{-1}_{\mathscr{D}} = M} \arrow[r, "\Downarrow"] & {V = [\![V']\!]^{-1}_{\mathscr{D}}} \\ M' \arrow[u, dashed] \arrow[r, dashed, "\twoheadrightarrow_{\mathscr{D}}"] & V' \arrow[u, dashed] \end{tikzcd}$$

(The existence of M' with $[\![M']\!]^{-1}_{\mathscr{D}} = M$ comes from the surjectivity of $[\![-]\!]^{-1}$.)

As the diagram suggests, it is true that $M \Downarrow V$ implies there exists V' such that $[\![M]\!] \twoheadrightarrow_{\mathscr{D}} V'$ with $[\![V']\!]^{-1}_{\mathscr{D}} = V$, but Lemma 4 makes a stronger statement which is more amenable to the inductive proof given.

We cannot show that $M \Downarrow V$ implies $[\![M]\!] \twoheadrightarrow_{\mathscr{D}} [\![V]\!]$ because the $[\![-]\!]$ translation picks out just one of the low-level encodings for V. This may not be the version that $[\![M]\!]$ reduces to. By inverting the relationship on the value end of the reduction, we "forget" the differences between them and arrive at the desired λ_{src} value.

Proposition 1 (Correctness)**.** For any closed λ_{src} term M, value V and definitions $\mathscr{D} = [\![M]\!]^{\text{top}}$,

$$M \Downarrow V \quad \Longleftrightarrow \quad \text{exists } V' \text{ s.t. } [\![M]\!] \twoheadrightarrow_{\mathscr{D}} V' \text{ and } [\![V']\!]^{-1}_{\mathscr{D}} = V$$

Proof. The ($\Leftarrow$) direction follows immediately from Lemma 3. To show the ($\Rightarrow$) direction from Lemma 4 we need to show that the given M has M' with $[\![M']\!]^{-1}_{\mathscr{D}} = M$. Construct $M' = [\![M]\!]$ and the needed relationship follows from the retraction. $\square$

3.3 The RPC Calculus

The RPC calculus, λ_{rpc}, is defined in Figure 3.5. This calculus extends the pedestrian calculus of Section 3.2 by tagging λ-abstractions with a location; we use the set of locations $\{\mathsf{c}, \mathsf{s}\}$, because we are interested in the client-server setting, but the calculus would be undisturbed by any other choice of location set.

The annotation on a λ-abstraction indicates the location where its body must execute. Thus an abstraction $\lambda^{\mathsf{c}}x.N$ represents a function that, when applied, would evaluate the term N at the client (c), binding the variable x to the function argument as usual. Constants are assumed to be universal, that is, all locations treat them the same way, and they contain no location annotations.

The semantics is defined by a big-step reduction relation $M \Downarrow_a V$, which is read, "the term M, evaluated at location a, results in value V." The reader can verify that the lexical body N of an a-annotated abstraction $\lambda^a x.N$ is only ever evaluated at location a, and thus the location annotations are honored by the semantics. During evaluation, however, that body may invoke other functions that evaluate at other locations.

Again we write $N\{V/x\}$ for the capture-avoiding substitution of a value V for the variable x in the term N. We assume terms are equal under α-equivalence. The annotation a on $\lambda^a x.N$ has no effect on the binding behavior of names.

Client/server machine

Our target abstract machine models a pair of interacting agents, a client and a server, that are each first-order computing machines. Figure 3.6 defines the machine, called CSM.

Being a first-order machine, the application form $f(\vec{M})$ is n-ary and allows only a function name, f, in the function position. The machine also introduces constructor applications of the form $F(\vec{M})$, which can be seen as a tagged tuple. Constructor applications are destructed by the case-analysis form $\mathsf{case}\ M\ \mathsf{of}\ \mathscr{A}$. A list $\mathscr{A}$ of case alternatives is well-formed if it defines each name only once.

Syntax $\boxed{\lambda_{\mathrm{rpc}}}$

$$\begin{array}{rrcl}
\text{constants} & c & & \\
\text{variables} & x & & \\
\text{locations} & a, b & ::= & \mathsf{c} \mid \mathsf{s} \\
\text{terms} & L, M, N & ::= & LM \mid V \\
\text{values} & V, W & ::= & \lambda^a x.N \mid x \mid c \\
\\
\text{evaluation contexts} & E & ::= & [\ ] \mid VE \mid EN
\end{array}$$

Semantics (big-step reduction) $\boxed{M \Downarrow_a V}$

$$V \Downarrow_a V \qquad \text{(VALUE)}$$

$$\frac{L \Downarrow_a \lambda^b x.N \qquad M \Downarrow_a W \qquad N\{W/x\} \Downarrow_b V}{LM \Downarrow_a V} \qquad \text{(BETA)}$$

Figure 3.5: The RPC calculus, λ_{rpc}.

The client may make requests to the server, using the form $\mathsf{req}\, f(\vec{M})$. The server cannot make requests and can only run in response to client requests. Note that the $\mathsf{req}\, f(\vec{M})$ form has no meaning in server position; it may lead to a stuck configuration.

A configuration $\mathcal{K}$ of this machine comes in one of two forms: a client-side configuration $(M; \cdot)$ consisting of an active client term M and a quiescent server (represented by the dot), or a server-side configuration $(E; M)$ consisting of an active server term M and a suspended client context E, which is waiting for the server's response. Although the client and server are in some sense independent agents, they interact in a synchronous fashion: upon making a request, the client blocks waiting for the server, and upon completing a request, the server is idle until the next request.

Reduction takes place in the context of a pair of definition sets, one for each agent, thus the reduction judgment takes the form $\mathcal{K} \longrightarrow_{\mathscr{C},\mathscr{S}} \mathcal{K}'$. Each definition $f(\vec{x}) = M$ defines the function name f, taking arguments $\vec{x}$, to be the term

Syntax

function names	f, g		
constructor names	F, G		
values	U, V, W	$::=$	$x \mid c \mid F(\vec{V})$
terms	L, M, N, X	$::=$	$x \mid c \mid f(\vec{M})$
			$\mid F(\vec{M}) \mid \mathsf{case}\, M\, \mathsf{of}\, \mathscr{A}$
			$\mid \mathsf{req}\, f\, (\vec{M})$
alternative sets	$\mathscr{A}$		a set of A items
case alternatives	A	$::=$	$F(\vec{x}) \Rightarrow M$
evaluation contexts	E	$::=$	$[\ \ ] \mid f(\vec{V}, E, \vec{M})$
			$\mid F(\vec{V}, E, \vec{M})$
			$\mid \mathsf{case}\, E\, \mathsf{of}\, \mathscr{A}$
			$\mid \mathsf{req}\, f\, (\vec{V}, E, \vec{M})$
configurations	$\mathscr{K}$	$::=$	$(M; \cdot) \mid (E; M)$
function definitions	D	$::=$	$f(\vec{x}) = M$
definition set	$\mathscr{D}, \mathscr{C}, \mathscr{S}$	$::=$	$\mathsf{letrec}\, D\, \mathsf{and} \cdots \mathsf{and}\, D$
continuation values	J, K	$::=$	$k \mid App(V, K) \mid F(\vec{V}, K)$

Semantics

$$\boxed{\mathscr{K} \longrightarrow_{\mathscr{C},\mathscr{S}} \mathscr{K}'}$$

Client:

$$\begin{array}{rcll}
(E[f(\vec{V})]; \cdot) & \longrightarrow_{\mathscr{C},\mathscr{S}} & (E[M\{\vec{V}/\vec{x}\}]; \cdot) & \text{if } (f(\vec{x}) = M) \in \mathscr{C} \\
(E[\mathsf{case}\, (F(\vec{V}))\, \mathsf{of}\, \mathscr{A}]; \cdot) & \longrightarrow_{\mathscr{C},\mathscr{S}} & (E[M\{\vec{V}/\vec{x}\}]; \cdot) & \text{if } (F(\vec{x}) \Rightarrow M) \in \mathscr{A}
\end{array}$$

Server:

$$\begin{array}{rcll}
(E; E'[f(\vec{V})]) & \longrightarrow_{\mathscr{C},\mathscr{S}} & (E; E'[M\{\vec{V}/\vec{x}\}]) & \text{if } (f(\vec{x}) = M) \in \mathscr{S} \\
(E; E'[\mathsf{case}\, (F(\vec{V}))\, \mathsf{of}\, \mathscr{A}]) & \longrightarrow_{\mathscr{C},\mathscr{S}} & (E; E'[M\{\vec{V}/\vec{x}\}]) & \text{if } (F(\vec{x}) \Rightarrow M) \in \mathscr{A}
\end{array}$$

Communication:

$$\begin{array}{rcl}
(E[\mathsf{req}\, f\, (\vec{V})]; \cdot) & \longrightarrow_{\mathscr{C},\mathscr{S}} & (E; f(\vec{V})) \\
(E; V) & \longrightarrow_{\mathscr{C},\mathscr{S}} & (E[V]; \cdot)
\end{array}$$

Figure 3.6: Definition of the Client/Server Machine (CSM).

M. The variables $\vec{x}$ are thus bound in M. A definition set is only well-formed if it uniquely defines each name. This does not preclude the other definition set, in a pair $(\mathscr{C},\mathscr{S})$, from also defining the same name.

The reflexive, transitive closure of the relation $\longrightarrow_{\mathscr{C},\mathscr{S}}$ is written with a double-headed arrow $\twoheadrightarrow_{\mathscr{C},\mathscr{S}}$, where the definition-sets are fixed throughout the sequence.

Observation. CSM reduction steps on the server side can be made in any evaluation context, encompassing both sides of the client/server divide:

$$([\ ];M) \twoheadrightarrow ([\ ];N) \text{ implies } (E;E'[M]) \twoheadrightarrow (E;E'[N]).$$

This is a direct consequence of the reduction rules, which are already defined in terms of evaluation contexts.

Lemma 5 (Substitution-Evaluation)**.** For any substitution σ, we have

$$\begin{array}{lcl} (M;\cdot) \twoheadrightarrow_{\mathscr{C},\mathscr{S}} (N;\cdot) & \text{implies} & (M\sigma;\cdot) \twoheadrightarrow_{\mathscr{C},\mathscr{S}} (N\sigma;\cdot) \quad \text{and} \\ (E;M) \twoheadrightarrow_{\mathscr{C},\mathscr{S}} (E;N) & \text{implies} & (E;M\sigma) \twoheadrightarrow_{\mathscr{C},\mathscr{S}} (E;N\sigma). \end{array}$$

Proof. We show that $\mathscr{K} \longrightarrow_{\mathscr{C},\mathscr{S}} \mathscr{K}'$ implies $\mathscr{K}\sigma \longrightarrow_{\mathscr{C},\mathscr{S}} \mathscr{K}'\sigma$. The result for $\twoheadrightarrow_{\mathscr{C},\mathscr{S}}$ then follows by induction on the reduction sequence.

Whether $\mathscr{K}$ is of the form $(M;\cdot)$ or $(E;M)$, decompose M into $E'[M']$. The proposition then follows by induction on M'. The proof of each case is a simple matter of pushing the substitutions down through the terms, applying the inductive hypothesis, and pulling them back up the terms. □

Translation from λ_{rpc} to CSM

Figures 3.7–3.10 give a translation from the client-server λ_{rpc} to CSM. Figure 3.7 gives term-level translations $(-)^{\circ}$, $(-)^{*}$ and $(-)^{\dagger}$, which construct values, client terms, and server contexts, respectively. The $(-)^{\dagger}$ translation produces a context, which is expected to be filled by a continuation (which is a server value), so we

$$\boxed{(-)^{\circ}_{-} : V_{\lambda_{\mathrm{rpc}}} \to V_{\mathrm{CSM}}}$$

$$
\begin{array}{rcll}
(\lambda^a x.N)^{\circ} & = & \ulcorner \lambda^a x.N \urcorner(\vec{y}) & \vec{y} = \mathrm{FV}(\lambda^a x.N) \\
x^{\circ} & = & x & \\
c^{\circ} & = & c &
\end{array}
$$

$$\boxed{(-)^{*} : M_{\lambda_{\mathrm{rpc}}} \to M_{\mathrm{CSM|c}}}$$

$$
\begin{array}{rcl}
V^{*} & = & V^{\circ} \\
(LM)^{*} & = & \mathit{apply}(L^{*}, M^{*})
\end{array}
$$

$$\boxed{(-)^{\dagger}(-) : M_{\lambda_{\mathrm{rpc}}} \to V_{\mathrm{CSM}} \to M_{\mathrm{CSM|s}}}$$

$$
\begin{array}{rcll}
V^{\dagger}[\ ] & = & \mathit{cont}([\ ], V^{\circ}) & \\
(LM)^{\dagger}[\ ] & = & L^{\dagger}(\ulcorner M \urcorner(\vec{y}, [\ ])) & \text{where } \vec{y} = \mathrm{FV}(M)
\end{array}
$$

Figure 3.7: Term-level translations from λ_{rpc} to CSM.

will normally write $N^{\dagger}K$ for the translation of N to a server term with continuation K. The functions $(-)^{*}$ and $(-)^{\dagger}(-)$ are only defined for λ_{rpc} client and server terms, respectively. Functions $[\![-]\!]^{\mathrm{c,top}}$ (Fig. 3.9) and $[\![-]\!]^{\mathrm{s,top}}$ (Fig. 3.10) translate a source term to a definition set, making use of the generic traversal function coll (Fig. 3.8), which computes the union of the images under f of each subterm of a given term.

The translated terms make use of function definitions for *apply*, *tramp* and *cont*, as produced by $[\![-]\!]^{\mathrm{c,top}}$ and $[\![-]\!]^{\mathrm{s,top}}$. Intuitively, the *apply* functions handle all function applications, the *cont* function handles continuation application, and *tramp* is the trampoline, which tunnels server-to-client requests through responses. For this translation, let *arg* and *k* be special reserved variable names not appearing in the source program. The function coll $f\ M$ computes the union of the image under f of each subterm N of M.

The bodies of the two *apply* functions will have a case for each abstraction appearing in the source term. Each location will have a case for both locations' abstractions; for its own abstractions it gets a full definition but for the other's abstractions the case will be a mere stub. This stub dispatches a request to the

$$\boxed{\text{coll} : (M_{\lambda_{\text{rpc}}} \to \{\alpha\}) \to M_{\lambda_{\text{rpc}}} \to \{\alpha\}}$$

$$\begin{array}{lcll}
\text{coll}\ f\ (LM) & = & f(LM) \cup \text{coll}\ f\ L \cup \text{coll}\ f\ M & \\
\text{coll}\ f\ (\lambda^a x.N) & = & f(\lambda^a x.N) \cup \text{coll}\ f\ N & \\
\text{coll}\ f\ V & = & f(V) & \text{if } V \neq \lambda^a x.N
\end{array}$$

Figure 3.8: Generic traversal function for λ_{rpc} terms.

other location, to apply the function to the given arguments. The *cont* function is defined only on the server, because it arises from the CPS translation, which is only applied on the server side; it has a case for each continuation produced by CPS. This includes one for evaluating the argument subterm of each server-located application, one called *App* for applying a function to an argument, and one called *Fin* which returns a value to the client.

Recall the classic CPS translation for applications:

$$(LM)^{\text{cps}}K = L^{\text{cps}}(\underline{\lambda f.M^{\text{cps}}(\underline{\lambda x.fxK})}).$$

The outer underlined term corresponds to a continuation that is defunctionalized as $\ulcorner M \urcorner$. The inner one is defunctionalized as *App*. Finally, recall that CPS always requires a "top-level" continuation, usually $\lambda x.x$, to extract a value from a CPS term; this corresponds to *Fin*.

The *tramp* function implements the trampoline. Its job is to field responses from the server, determine whether they indicate function returns or server-to-client calls, and dispatch them as necessary. The protocol used by *tramp* is as follows: when the client first needs to make a server call, it makes a request wrapped in *tramp*. The server will either complete this call itself, without any client calls, or it will have to make a client call along the way. If it needs to make a client call, it returns a specification of that call as a value *Call*(*fun*,*arg*,*k*), where *fun* and *arg* specify the call and *k* is the current continuation. The *tramp* function recognizes these constructions and evaluates the necessary terms locally, then places another request to the server to apply *k* to whatever value resulted,

$$\boxed{[\![-]\!]^{\mathsf{c,top}} : M_{\lambda_{\text{rpc}}} \to \mathscr{D}_{\text{CSM}}}$$

$$\begin{aligned}
[\![M]\!]^{\mathsf{c,top}} \quad = \quad & \mathsf{letrec}\, \mathit{apply}(\mathit{fun}, \mathit{arg}) = \mathsf{case}\, \mathit{fun}\ \mathsf{of}\ [\![M]\!]^{\mathrm{c,fun}} \\
& \mathsf{and}\ \mathit{tramp}(x) = \mathsf{case}\ x\ \mathsf{of} \\
& \qquad \mid \mathit{Call}(f, x, k) \Rightarrow \\
& \qquad\qquad \mathit{tramp}(\mathsf{req}\ \mathit{cont}\ (k, \mathit{apply}(f, x))) \\
& \qquad \mid \mathit{Return}(x) \Rightarrow x
\end{aligned}$$

$$\begin{aligned}
[\![\lambda^{\mathsf{c}} x.N]\!]^{\mathrm{c,fun,aux}} &= \{\ulcorner \lambda^{\mathsf{c}} x.N \urcorner(\vec{y}) \Rightarrow N^{*}\{arg/x\}\} && \text{where } \vec{y} = \textsc{fv}(\lambda x.N) \\
[\![\lambda^{\mathsf{s}} x.N]\!]^{\mathrm{c,fun,aux}} &= \{\ulcorner \lambda^{\mathsf{s}} x.N \urcorner(\vec{y}) \Rightarrow \mathit{tramp}(\mathsf{req}\ \mathit{apply}\ (\ulcorner \lambda^{\mathsf{s}} x.N \urcorner(\vec{y}), \mathit{arg}, \mathit{Fin}()))\} \\
& && \text{where } \vec{y} = \textsc{fv}(\lambda x.N) \\
[\![M]\!]^{\mathrm{c,fun,aux}} &= \{\} && \text{if } M \neq \lambda^{a} x.N \\
[\![M]\!]^{\mathrm{c,fun}} &= \mathrm{coll}\ ([\![-]\!]^{\mathrm{c,fun,aux}})\, M
\end{aligned}$$

Figure 3.9: Definition construction, translation from λ_{rpc} to CSM (client).

$$\boxed{[\![-]\!]^{\mathsf{s,top}} : M_{\lambda_{\text{rpc}}} \to \mathscr{D}_{\text{CSM}}}$$

$$\begin{aligned}
[\![M]\!]^{\mathsf{s,top}} \ = \ & \mathsf{letrec}\, \mathit{apply}(\mathit{fun}, \mathit{arg}, k) = \mathsf{case}\, \mathit{fun}\ \mathsf{of}\ [\![M]\!]^{\mathsf{s,fun}} \\
& \mathsf{and}\ \mathit{cont}(k, \mathit{arg}) = \mathsf{case}\ k\ \mathsf{of} \\
& \qquad \mid\ [\![M]\!]^{\mathsf{s,cont}} \\
& \qquad \mid \mathit{App}(\mathit{fun}, k) \Rightarrow \mathit{apply}(\mathit{fun}, \mathit{arg}, k) \\
& \qquad \mid \mathit{Fin}() \Rightarrow \mathit{Return}(\mathit{arg})
\end{aligned}$$

$$\begin{aligned}
[\![\lambda^{\mathsf{c}} x.N]\!]^{\mathrm{s,fun,aux}} &= \{\ulcorner \lambda^{\mathsf{c}} x.N \urcorner(\vec{y}) \Rightarrow \mathit{Call}(\ulcorner \lambda^{\mathsf{c}} x.N \urcorner(\vec{y}), \mathit{arg}, k)\} \\
& && \text{where } \vec{y} = \textsc{fv}(\lambda x.N) \\
[\![\lambda^{\mathsf{s}} x.N]\!]^{\mathrm{s,fun,aux}} &= \{\ulcorner \lambda^{\mathsf{s}} x.N \urcorner(\vec{y}) \Rightarrow (N^{\dagger} k)\{arg/x\}\} && \text{where } \vec{y} = \textsc{fv}(\lambda x.N) \\
[\![M]\!]^{\mathrm{s,fun,aux}} &= \{\} && \text{if } M \neq \lambda^{a} x.N \\
[\![M]\!]^{\mathrm{s,fun}} &= \mathrm{coll}\ ([\![-]\!]^{\mathrm{s,fun,aux}})\, M \\
[\![LM]\!]^{\mathrm{s,cont,aux}} &= \{\ulcorner M \urcorner(\vec{y}, k) \Rightarrow M^{\dagger}(\mathit{App}(\mathit{arg}, k))\} && \text{where } \vec{y} = \textsc{fv}(M) \\
[\![N]\!]^{\mathrm{s,cont,aux}} &= \{\} && \text{if } N \neq LM \\
[\![M]\!]^{\mathrm{s,cont}} &= \mathrm{coll}\ ([\![-]\!]^{\mathrm{s,cont,aux}})\, M
\end{aligned}$$

Figure 3.10: Definition construction, translation from λ_{rpc} to CSM (server).

again wrapping the request in *tramp*. When the server finally completes its original call, it returns the value as the argument of the *Return* constructor; the *tramp* function recognizes this as the result of the original server call, so it simply returns x. As an invariant, *the client always wraps its server-requests in a call to tramp*. This way it can always handle *Call* responses.

To relate the two calculi, we again use a reverse translation, now from CSM to λ_{rpc}, given in Figure 3.11. All of the functions used in this translation are parameterized on the definition sets $\mathscr{C}$ and $\mathscr{S}$.

The function $(-)^{\bullet}_{\mathscr{C},\mathscr{S}}$ takes CSM values to λ_{rpc} values. Next $(-)^{\star}_{\mathscr{C},\mathscr{S}}$ and $(-)^{\ddagger}_{\mathscr{C},\mathscr{S}}$ take client-side and server-side CSM terms (respectively) to λ_{rpc} terms. And $(-)^{\$}_{\mathscr{C},\mathscr{S}}$ takes CSM values representing continuations to λ_{rpc} evaluation contexts.

These functions are defined only on CSM terms and definitions in the range of the corresponding forward translation. Moreover $M^{\ddagger}_{\mathscr{C},\mathscr{S}}$ and $M^{\star}_{\mathscr{C},\mathscr{S}}$ are not defined unless *both* definition sets $\mathscr{C}$ and $\mathscr{S}$ define all the constructors appearing in M.

The last case of $(-)^{\$}$ in Figure 3.11 is written using the assumption that the case defining each constructor F is in the image of the translation. It might seem that a more straightforward definition would simply apply $(-)^{\ddagger}$ to the right-hand side of the definition of F; but if that right-hand side is of the form $M^{\dagger}(App(arg,k))$, which is the only case we care about, then the result comes out as written here, which is simpler to read and work with.

As before, to extract the alternatives of case-expressions from function bodies, we use a function cases.

Definition. Define cases as follows:

$$\text{cases}(f,\mathscr{D}) = \mathscr{A} \text{ iff } (f(x,\vec{y}) = \text{case } x \text{ of } \mathscr{A}) \in \mathscr{D}.$$

This definition relies on the fact that each of our special functions dispatches on the first of its arguments, whether that be the argument *fun* for *apply*, or k for *cont*; the dispatching argument is conveniently always the first.

$$\boxed{(-)^{\bullet}_{-,-} : V_{\text{CSM}} \to \mathscr{D}_{\text{CSM}} \to \mathscr{D}_{\text{CSM}} \to V_{\lambda_{\text{rpc}}}}$$

$$\begin{array}{rcl}
c^{\bullet}_{\mathscr{C},\mathscr{S}} & = & c \\
x^{\bullet}_{\mathscr{C},\mathscr{S}} & = & x \\
(F(\vec{V}))^{\bullet}_{\mathscr{C},\mathscr{S}} & = & \lambda^{\mathsf{c}} x.(N\{x/arg\})^{\star}_{\mathscr{C},\mathscr{S}}\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\} \\
 & & \text{if } (F(\vec{y}) \Rightarrow N) \in \text{cases}(apply, \mathscr{C}) \text{ and } N \neq tramp(\mathsf{req} \cdot \cdot) \\
 & & \text{where } x \text{ fresh for } \vec{y}, \vec{V} \\
(F(\vec{V}))^{\bullet}_{\mathscr{C},\mathscr{S}} & = & \lambda^{\mathsf{s}} x.(N\{x/arg\})^{\ddagger}_{\mathscr{C},\mathscr{S}}\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\} \\
 & & \text{if } (F(\vec{y}) \Rightarrow N) \in \text{cases}(apply, \mathscr{S}) \text{ and } N \neq Call(\cdot,\cdot,\cdot) \\
 & & \text{where } x \text{ fresh for } \vec{y}, \vec{V}
\end{array}$$

$$\boxed{(-)^{\star}_{-,-} : M_{\text{CSM}|\mathsf{c}} \to \mathscr{D}_{\text{CSM}} \to \mathscr{D}_{\text{CSM}} \to M_{\lambda_{\text{rpc}}}}$$

$$\begin{array}{rcl}
V^{\star}_{\mathscr{C},\mathscr{S}} & = & V^{\bullet}_{\mathscr{C},\mathscr{S}} \\
(apply(L,M))^{\star}_{\mathscr{C},\mathscr{S}} & = & L^{\star}_{\mathscr{C},\mathscr{S}} M^{\star}_{\mathscr{C},\mathscr{S}}
\end{array}$$

$$\boxed{(-)^{\ddagger}_{-,-} : M_{\text{CSM}|\mathsf{s}} \to \mathscr{D}_{\text{CSM}} \to \mathscr{D}_{\text{CSM}} \to M_{\lambda_{\text{rpc}}}}$$

$$\begin{array}{rcl}
(cont(K,V))^{\ddagger}_{\mathscr{C},\mathscr{S}} & = & K^{\$}_{\mathscr{C},\mathscr{S}}[V^{\bullet}_{\mathscr{C},\mathscr{S}}] \\
(apply(V,W,K))^{\ddagger}_{\mathscr{C},\mathscr{S}} & = & K^{\$}_{\mathscr{C},\mathscr{S}}[V^{\bullet}_{\mathscr{C},\mathscr{S}} W^{\bullet}_{\mathscr{C},\mathscr{S}}]
\end{array}$$

$$\boxed{(-)^{\$}_{-,-} : V_{\text{CSM}} \to \mathscr{D}_{\text{CSM}} \to \mathscr{D}_{\text{CSM}} \to (M_{\lambda_{\text{rpc}}} \to M_{\lambda_{\text{rpc}}})}$$

$$\begin{array}{rcl}
k^{\$}_{\mathscr{C},\mathscr{S}}[\ ] & = & [\ ] \\
(App(V,K))^{\$}_{\mathscr{C},\mathscr{S}}[\ ] & = & K^{\$}_{\mathscr{C},\mathscr{S}}[V^{\bullet}_{\mathscr{C},\mathscr{S}}[\ ]] \\
(F(\vec{V},K))^{\$}_{\mathscr{C},\mathscr{S}}[\ ] & = & K^{\$}_{\mathscr{C},\mathscr{S}}[[\ ](M\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\})] \\
 & & \text{if } (F(\vec{y},k) \Rightarrow M^{\dagger}(App(arg,k))) \in \text{cases}(cont, \mathscr{S}) \\
 & & \text{and } F \neq Fin,\ F \neq App
\end{array}$$

Figure 3.11: Reverse translation from CSM to λ_{rpc}.

Observation. The range of the reverse translation $(-)^{\bullet}$ includes all values of λ_{rpc} and only values map to values under $(-)^{\bullet}$. The ranges of the reverse translations $(-)^{\star}$ and $(-)^{\ddagger}(-)$ include all λ_{rpc} terms.

Provided $(\mathscr{C},\mathscr{S})$ are in the image of $([\![-]\!]^{\text{c,top}},[\![-]\!]^{\text{s,top}})$, then $K^{\$}_{\mathscr{C},\mathscr{S}}$ is a term context for λ_{rpc}; by a simple inductive argument we can see that it is always an evaluation context: it meets the grammar $E ::= [\ \ ] \mid VE \mid EM$. $\square$

Correctness of the translation from λ_{rpc} to CSM

As before, CSM terms will disappear during reduction, so we again need a containment relation on definition-sets. This time we define the containment relation more generally: it holds just when the names defined in the right-hand side are all defined in the left-hand side and upon inspecting corresponding function definitions, either the bodies are identical or they are both case analyses where the left-hand side contains all the alternatives of the right-hand side.

Definition (Definition containment)**.** A definition set $\mathscr{D}$ *contains* $\mathscr{D}'$, written $\mathscr{D} \geqslant \mathscr{D}'$, iff for each definition $f(\vec{x}) = M'$ in $\mathscr{D}'$ there is a definition $f(\vec{x}) = M$ in $\mathscr{D}$ and either $M = M'$ or $\text{cases}(f,\mathscr{D}) \supseteq \text{cases}(f,\mathscr{D}')$.

Observation. For any subterm M' of M, if $\mathscr{C} \geqslant [\![M]\!]^{\text{c,top}}$ then $\mathscr{C} \geqslant [\![M']\!]^{\text{c,top}}$ and if $\mathscr{S} \geqslant [\![M]\!]^{\text{s,top}}$ then $\mathscr{S} \geqslant [\![M']\!]^{\text{s,top}}$.

Definitions produced by the top-level translations are *closed*: for each term that we find translated on the right-hand side of a definition case, all of that term's definitions can also be found amongst the definitions. More precisely:

Lemma 6 (Closure, definition sets)**.** Let $\mathscr{C}$ and $\mathscr{S}$ be in the range of $[\![-]\!]^{\text{c,top}}$ and $[\![-]\!]^{\text{s,top}}$ respectively.

- If $N^{*}\{arg/x\}$ is the right-hand side of an element of $\text{cases}(apply,\mathscr{C})$ then $\mathscr{C} \geqslant [\![N]\!]^{\text{c,top}}$.

- If $(N^{\dagger}k)\{arg/x\}$ is the right-hand side of an element of $\text{cases}(apply,\mathscr{S})$ then $\mathscr{S} \geqslant [\![N]\!]^{\text{s,top}}$.

- If $M^\dagger(App(arg,k))$ is the right-hand side of an element of cases$(cont,\mathscr{S})$ then $\mathscr{S} \geqslant [\![M]\!]^{\mathsf{s,top}}$.

Proof. Let X be the term such that $(\mathscr{C},\mathscr{S}) = ([\![X]\!]^{\mathsf{c,top}}, [\![X]\!]^{\mathsf{s,top}})$. Each element of cases$(apply,\mathscr{C})$ that has a right-hand side of the form $N^*\{arg/x\}$ is produced by a term $\lambda^{\mathsf{c}}x.N$, a subterm of X. As N is a subterm of X, then, $\mathscr{C} \geqslant [\![N]\!]^{\mathsf{c,top}}$. A similar argument holds for the other two consequents. □

Lemma 7 (Retraction)**.** When $\mathscr{C} \geqslant [\![M]\!]^{\mathsf{c,top}}$ and $\mathscr{S} \geqslant [\![M]\!]^{\mathsf{s,top}}$, we have

(i) $(M^\circ)^\bullet_{\mathscr{C},\mathscr{S}} = M$ provided M is a value,

(ii) $(M^*)^\star_{\mathscr{C},\mathscr{S}} = M$, and

(iii) $(M^\dagger K)^\ddagger_{\mathscr{C},\mathscr{S}} = K^\$_{\mathscr{C},\mathscr{S}} M$ for each K in CSM.

Proof. By induction on M. We omit the $(\mathscr{C},\mathscr{S})$ argument to each of the reverse-translation functions, since it never changes.

CASE x, c for (i). Trivial. *huzzah!*

CASE $\lambda^a x.N$ for (i). Let x' be a fresh variable. Take cases on a.

Case $a = \mathsf{s}$:

$$\begin{array}{rcll}
(\lambda^{\mathsf{s}}x.N)^{\circ\bullet} & = & (\ulcorner\lambda^{\mathsf{s}}x.N\urcorner(\vec{y}))^\bullet & \vec{y} = \text{FV}(\lambda x.N) \\
& = & & \text{because } \begin{array}{r}(\ulcorner\lambda^{\mathsf{s}}x.N\urcorner(\vec{y}) \Rightarrow N^\dagger k\{arg/x\}) \\ \in \text{cases}(apply,\mathscr{S})\end{array} \\
& & \lambda x'.((N^\dagger k)\{arg/x\}\{x'/arg\})^\ddagger & \\
& = & \lambda x'.((N^\dagger k)\{x'/x\})^\ddagger & \\
& =_\alpha & \lambda x.(N^\dagger k)^\ddagger & \\
& = & & \text{(IH)} \\
& & \lambda x.N &
\end{array}$$

Case $a = \mathsf{c}$:

$$\begin{array}{rcll}
(\lambda^{\mathsf{c}}x.N)^{\circ\bullet} & = & (\ulcorner \lambda^{\mathsf{c}}x.N \urcorner(\vec{y}))^{\bullet} & \vec{y} = \mathrm{FV}(\lambda x.N) \\
 & = & \text{because} & (\ulcorner \lambda^{\mathsf{c}}x.N \urcorner(\vec{y}) \Rightarrow N^{*}\{arg/x\}) \\
 & & & \in \text{cases}(apply, \mathscr{C}) \\
 & & \lambda x'.(N^{*}\{arg/x\}\{x'/arg\})^{\star} & \\
 & = & \lambda x'.(N^{*}\{x'/x\})^{\star} & \\
 & =_{\alpha} & \lambda x.N^{*\star} & \\
 & = & & \text{(IH)} \\
 & & \lambda x.N & \textit{huzzah!}
\end{array}$$

CASE V for (ii).

$$V^{*\star} = V^{\circ\star} = V^{\circ\bullet} = V$$

huzzah!

CASE V for (iii).

$$(V^{\dagger}K)^{\ddagger} = (cont(K, V^{\circ}))^{\ddagger} = K^{\$}(V^{\circ\bullet}) = K^{\$}V$$

huzzah!

CASE LM for (ii).

$$(LM)^{*\star} = (apply(L^{*}, M^{*}))^{\star} = L^{*\star}M^{*\star} = LM$$

huzzah!

CASE LM for (iii).

$$\begin{array}{rcll}
((LM)^{\dagger}K)^{\ddagger} & = & (L^{\dagger}(\ulcorner M \urcorner(\vec{y}, K)))^{\ddagger} & \vec{y} = \mathrm{FV}(M) \\
 & = & & \text{(IH)} \\
 & & (\ulcorner M \urcorner(\vec{y}, K))^{\$}L & \\
 & = & & \\
 & & \multicolumn{2}{l}{\text{because } (\ulcorner M \urcorner(\vec{y}, k) \Rightarrow M^{\dagger}(App(arg, k))) \in \text{cases}(cont, \mathscr{S})} \\
 & & K^{\$}(LM) & \textit{huzzah!} \ \square
\end{array}$$

The reverse translation commutes with substitution.

Lemma 8 (Substitution–Reverse Translation)**.** Given definition sets $\mathscr{C}$, $\mathscr{S}$ and terms M, V and W we have

$$V^{\bullet}_{\mathscr{C},\mathscr{S}}\{W^{\bullet}_{\mathscr{C},\mathscr{S}}/x\} = (V\{W/x\})^{\bullet}_{\mathscr{C},\mathscr{S}}$$
$$M^{\ddagger}_{\mathscr{C},\mathscr{S}}\{W^{\bullet}_{\mathscr{C},\mathscr{S}}/x\} = (M\{W/x\})^{\ddagger}_{\mathscr{C},\mathscr{S}} \text{ and}$$
$$M^{\star}_{\mathscr{C},\mathscr{S}}\{W^{\bullet}_{\mathscr{C},\mathscr{S}}/x\} = (M\{W/x\})^{\star}_{\mathscr{C},\mathscr{S}}.$$

Proof. By induction on M.

CASE c, x. Trivial. *huzzah!*

CASE $F(\vec{V})$. If $(F(\vec{V}))^{\bullet}_{\mathscr{C},\mathscr{S}}$ is defined, it equals $(\lambda^a x.N)\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\}$. And from this,

$$\begin{aligned}
(F(\vec{V}))^{\bullet}_{\mathscr{C},\mathscr{S}}\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\}\{W^{\bullet}_{\mathscr{C},\mathscr{S}}/x\} &= (\lambda^a x.N)\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\}\{W^{\bullet}_{\mathscr{C},\mathscr{S}}/x\} \\
&= (\lambda^a x.N)\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}\{W^{\bullet}_{\mathscr{C},\mathscr{S}}/x\}/\vec{y}\} \\
&= \qquad\qquad\qquad\qquad \text{(IH)} \\
&\quad (\lambda^a x.N)\{(\vec{V}\{W/x\})^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\} \\
&= (F(\vec{V}\{W/x\}))^{\bullet}_{\mathscr{C},\mathscr{S}} \\
&= (F(\vec{V})\{W/x\})^{\bullet}_{\mathscr{C},\mathscr{S}} \qquad \textit{huzzah!}\ \square
\end{aligned}$$

If $(F(\vec{V}))^{\bullet}_{\mathscr{C},\mathscr{S}}$ is not defined, then neither is $(F(\vec{V})\{W/x\})^{\bullet}$, and vice versa.

The cases for $(-)^{\star}$ and $(-)^{\ddagger}$ are simple uses of the inductive hypothesis, taking care with the use of $(-)^{\$}$ and noting, as in the above case for $F(\vec{V})^{\bullet}$, that the left-hand side is defined just when the left-hand side is.

Lemma 9 (Soundness)**.** For any term M and substitution σ in λ_{rpc}, together with definition sets $\mathscr{C}$ and $\mathscr{S}$ such that $\mathscr{C} \geqslant [\![M]\!]^{\text{c,top}}$ and $\mathscr{S} \geqslant [\![M]\!]^{\text{s,top}}$, we have the following implications for all V and K:

(i) $M^{*}\sigma \twoheadrightarrow_{\mathscr{C},\mathscr{S}} V$
implies $M\sigma^{\bullet}_{\mathscr{C},\mathscr{S}} \Downarrow_{\mathsf{c}} V^{\bullet}_{\mathscr{C},\mathscr{S}}$ and

(ii) $\mathit{tramp}([\ ]); (M^{\dagger}K)\sigma \twoheadrightarrow_{\mathscr{C},\mathscr{S}} \mathit{tramp}([\ ]); \mathit{cont}(K,V)$
implies $M\sigma^{\bullet}_{\mathscr{C},\mathscr{S}} \Downarrow_{\mathsf{s}} V^{\bullet}_{\mathscr{C},\mathscr{S}}$.

Proof. By induction on the length of the CSM reduction sequence. Throughout the induction, σ is kept general.

We make free use of Lemma 5, showing that we can apply a substitution to a reduction to get another reduction.

When using the inductive hypothesis, the preconditions that $\mathscr{C} \geqslant [\![M]\!]^{\text{c,top}}$ and $\mathscr{S} \geqslant [\![M]\!]^{\text{s,top}}$ will be maintained because we will only use the inductive hypothesis on subterms of the M and on terms N whose translations are part of the rhs of definitions in $\mathscr{C},\mathscr{S}$ and thus for which $\mathscr{C},\mathscr{S} \geqslant [\![N]\!]^{\text{c,top}}$ and $[\![N]\!]^{\text{s,top}}$ (by the closure of definition-sets).

In this proof we omit the subscripts $\mathscr{C}$, $\mathscr{S}$ on reductions, because they are unchanged throughout reduction sequences, and on the reverse-translation functions, because they are unchanged throughout the recursive calls thereof.

Take cases on the structure of the starting term, either (i) $M^*\sigma$ or (ii) $(M^\dagger K)\sigma$, and split the conclusion into cases for (i) and (ii);

CASE LM for (i).

By hypothesis, we have $(LM)^*\sigma \twoheadrightarrow V$.

By definition, $(LM)^*\sigma = apply(L^*\sigma, M^*\sigma)$.

In order for the reduction not to get stuck, it must be that

- $L^*\sigma$ reduces to a value $F(\vec{V})$ with $(F(\vec{V}))^\bullet = \lambda^a x.N\{\vec{V}^\bullet/\vec{y}\}$ for fresh x and some a and N.
- $M^*\sigma$ reduces to a value; call it W.

The freshness of x will allow us to equate simultaneous and sequential substitutions involving x.

The reduction begins as follows:

$$\begin{array}{rcl}(LM)^*\sigma & = & apply(L^*\sigma, M^*\sigma) \\ & \twoheadrightarrow & apply(F(\vec{V}), M^*\sigma) \\ & \twoheadrightarrow & apply(F(\vec{V}), W)\end{array}$$

Applying the inductive hypothesis twice, we get

$$\begin{array}{l}L\sigma^\bullet \Downarrow_{\mathsf{c}} (F(\vec{V}))^\bullet = \lambda^a x.N\{\vec{V}^\bullet/\vec{y}\} \text{ and} \\ M\sigma^\bullet \Downarrow_{\mathsf{c}} W^\bullet\end{array}$$

We now show the third leg of the BETA rule, that $N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_a V^\bullet$, by cases on a.

If $a = \mathsf{c}$ then N is such that $(F(\vec{y}) \Rightarrow N^*\{arg/x\}) \in \text{cases}(apply, \mathscr{C})$, by definition of $(-)^\bullet$.

Now the reduction finishes as:

$$\begin{aligned}
& apply(F(\vec{V}), W) \\
\twoheadrightarrow\; & N^*\{\vec{V}/\vec{y}, W/x\} = N^*\{\vec{V}/\vec{y}\}\{W/x\} \\
\twoheadrightarrow\; & V
\end{aligned}$$

So by the inductive hypothesis $N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_{\mathsf{C}} V^\bullet$.

If $a = \mathsf{s}$ then N is such that

$$(F(\vec{y}) \Rightarrow (N^\dagger k)\{arg/x\})) \in \mathsf{cases}(apply, \mathscr{S}) \quad \text{and}$$
$$(F(\vec{y}) \Rightarrow tramp(\mathsf{req}\, apply\ (F(\vec{y}), arg, Fin()))) \in \mathsf{cases}(apply, \mathscr{C}).$$

Now the reduction finishes as

$$\begin{aligned}
& apply(F(\vec{V}), W) \\
\twoheadrightarrow\; & tramp(\mathsf{req}\, apply\ (F(\vec{V}), W, Fin())) \\
\twoheadrightarrow\; & tramp([\ \ ]);\ apply(F(\vec{V}), W, Fin()) \\
\twoheadrightarrow\; & tramp([\ \ ]);\ (N^\dagger(Fin()))\{\vec{V}/\vec{y}, W/x\} = (N^\dagger(Fin()))\{\vec{V}/\vec{y}\}\{W/x\} \\
\twoheadrightarrow\; & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(IH)} \\
& tramp([\ \ ]);\ cont(Fin(), V) \\
\twoheadrightarrow\; & V
\end{aligned}$$

So by the inductive hypothesis $N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_{\mathsf{C}} V^\bullet$.

The judgment $(LM)\sigma^\bullet \Downarrow_{\mathsf{C}} V^\bullet$ follows by BETA. *huzzah!*

CASE LM for (ii).

By hypothesis, we have for some K that

$$tramp([\ \ ]); ((LM)^\dagger K)\sigma \twoheadrightarrow tramp([\ \ ]); cont(K, V)).$$

By definition, $(LM)^\dagger K = L^\dagger(\ulcorner M \urcorner(\vec{z}, K))$, letting $\vec{z} = \mathrm{FV}(M)$.

In order for the reduction not to get stuck, all of the following must hold:

1. $\ulcorner M \urcorner(\vec{z},k) \Rightarrow M^\dagger(App(arg,k))$ is in cases$(cont,\mathscr{S})$
2. $(M^\dagger K)\sigma$ reduces to a term of the form $cont(K,W)$ and
3. $(L^\dagger K)\sigma$ reduces to a term of the form $cont(K,F(\vec{y}))$, with $(F(\vec{y}))^\bullet = \lambda^a x.N$ for fresh x and some a and N.

The reduction begins as follows:

$$\begin{array}{rl}
 & tramp([\ \ ]);\, ((LM)^\dagger K)\sigma \\
= & tramp([\ \ ]);\, (L^\dagger(\ulcorner M \urcorner(\vec{z},K)))\sigma \\
\twoheadrightarrow & tramp([\ \ ]);\, cont((\ulcorner M \urcorner(\vec{z},K))\sigma, F(\vec{V})) \\
= & tramp([\ \ ]);\, cont(\ulcorner M \urcorner(\vec{z}\sigma,K), F(\vec{V})) \\
\longrightarrow & tramp([\ \ ]);\, (M^\dagger(App(arg,K)))\{F(\vec{V})/arg, (\vec{z}\sigma)/\vec{z}\} \\
= & tramp([\ \ ]);\, (M^\dagger k)\{App(F(\vec{V}),K)/k,\, \vec{z}\sigma/\vec{z}\} \\
\twoheadrightarrow & tramp([\ \ ]);\, cont(App(F(\vec{V}),K),W) \\
\longrightarrow & tramp([\ \ ]);\, apply(F(\vec{V}),W,K)
\end{array}$$

Applying the inductive hypothesis twice, we get

$$\begin{array}{l}
L\sigma^\bullet \Downarrow_{\mathsf{s}} (F(\vec{V}))^\bullet = \lambda^a x.N\{\vec{V}^\bullet/\vec{y}\} \text{ and} \\
M\sigma^\bullet \Downarrow_{\mathsf{s}} W^\bullet
\end{array}$$

Now we show the third leg of the BETA rule, that $N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_a V^\bullet$, by cases on a.

If $a = \mathsf{s}$ then N is such that $(F(\vec{y}) \Rightarrow (N^\dagger k)\{arg/x\})) \in$ cases$(apply,\mathscr{S})$. So the reduction continues as

$$\begin{array}{rl}
 & tramp([\ \ ]);\, apply(F(\vec{V}),W,k) \\
\longrightarrow & tramp([\ \ ]);\, (N^\dagger k)\{\vec{V}/\vec{y}, W/x\} = (N^\dagger k)\{\vec{V}/\vec{y}\}\{W/x\} \\
\twoheadrightarrow & tramp([\ \ ]);\, cont(k,V)
\end{array}$$

So by the inductive hypothesis, we get (as requred)

$$N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_{\mathsf{s}} V^\bullet.$$

If $a = \mathsf{c}$ then N is such that

$$(F(\vec{y}) \Rightarrow N^*\{arg/x\}) \in \mathsf{cases}(apply, \mathscr{C}) \text{ and}$$

$$(F(\vec{y}) \Rightarrow Call(F(\vec{y}), arg, k)) \in \mathsf{cases}(apply, \mathscr{S}).$$

So the reduction continues as:

$$\begin{array}{rl}
 & tramp([\ \ ]);\, apply(F(\vec{V}), W, k) \\
\longrightarrow & tramp([\ \ ]);\, Call(F(\vec{V}), W, k) \\
\longrightarrow & tramp(Call(F(\vec{V}), W, k)) \\
\longrightarrow & tramp(\mathsf{req}\, cont\,(k, apply(F(\vec{V}), W))) \\
\longrightarrow & tramp(\mathsf{req}\, cont\,(k, N^*\{\vec{V}/\vec{y}, W/x\})) \\
= & tramp(\mathsf{req}\, cont\,(k, N^*\{\vec{V}/\vec{y}\}\{W/x\})) \\
\twoheadrightarrow & tramp(\mathsf{req}\, cont\,(k, V)) \\
\twoheadrightarrow & tramp([\ \ ]);\, cont(k, V)
\end{array}$$

So by the inductive hypothesis, we get

$$N\{\vec{V}^\bullet/\vec{y}\}\{W^\bullet/x\} \Downarrow_{\mathsf{s}} V^\bullet.$$

The judgment $(LM)\sigma^\bullet \Downarrow_{\mathsf{s}} V^\bullet$ follows by BETA. *huzzah!*

CASE V for (i) and (ii). Write W for M, which must also be a value.

Because the starting term is a value, the reduction is of zero steps: In the client case: $M^*\sigma = V \twoheadrightarrow V$. We have that $M^* = W^\circ$ so $W^\circ\sigma = V$. In the server case: $(M^\dagger K)\sigma = cont(K, V) \twoheadrightarrow cont(K, V)$. We have that $M^\dagger K = cont(K, W^\circ)$ so $W^\circ\sigma = V$.

Using the substitution lemma (Lemma 8) and the inverseness of $(-)^\bullet$ to $(-)^\circ$, we get $(W^\circ\sigma)^\bullet = W\sigma^\bullet$. Now $(W^\circ\sigma)^\bullet = V^\bullet$ so $V^\bullet = M\sigma^\bullet$. And so the reduction $M\sigma^\bullet \Downarrow_a V$ follows by VALUE. *huzzah!* □

Next we turn to the completeness of the translation. First we show that continuations K in CSM are closely related to evaluation contexts E in λ_{rpc}. Using this we show the possible forms of CSM terms that map to λ_{rpc} application terms.

Lemma 10. Given definition-sets $\mathscr{C},\mathscr{S}$ and a continuation K, one of the following holds:

(a) The form of $K^{\$}_{\mathscr{C},\mathscr{S}}$ is $[\ \ ]$ and $K = k$.

(b) The form of $K^{\$}_{\mathscr{C},\mathscr{S}}$ is VE and there exist J, V' such that

$$\begin{aligned} K &= J\{App(V',k)/k\}, \\ V'^{\bullet}_{\mathscr{C},\mathscr{S}} &= V \quad \text{and} \\ J^{\$}_{\mathscr{C},\mathscr{S}} &= E. \end{aligned}$$

(c) The form of $K^{\$}_{\mathscr{C},\mathscr{S}}$ is EM and there exist $J, M', F, \vec{V}$ such that

$$\begin{aligned} K &= J\{F(\vec{V},k)/k\}, \\ &(F(\vec{y},k) \Rightarrow M'\{App(arg,k)/k\}) \in \text{cases}(cont,\mathscr{S}), \\ (M'\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathscr{C},\mathscr{S}} &= M \quad \text{and} \\ J^{\$}_{\mathscr{C},\mathscr{S}} &= E. \end{aligned}$$

Proof. The proof is by induction on K. Take cases on its form:

CASE k. By def., $K^{\$}_{\mathscr{C},\mathscr{S}} = [\ \ ]$, proving (a). *huzzah!*

CASE $App(U,K')$. By definition, $K^{\$}_{\mathscr{C},\mathscr{S}} = K'^{\$}_{\mathscr{C},\mathscr{S}}[U^{\bullet}_{\mathscr{C},\mathscr{S}}[\ \ ]]$.

If $K' = k$, then we prove (b). By def., $K'^{\$}_{\mathscr{C},\mathscr{S}} = [\ \ ]$. Letting $J = k$ and $E = [\ \ ]$ we get $K = J\{App(U,k)/k\}$ and $J^{\$}_{\mathscr{C},\mathscr{S}} = E$ as needed.

If $K' \neq k$ then the induction hypothesis gives us one of the cases (b) or (c); we prove the same case. The IH provides J' and E' with $J'^{\$}_{\mathscr{C},\mathscr{S}} = E'$. Let $J = App(U,J')$ and $E = E'[U^{\bullet}_{\mathscr{C},\mathscr{S}}[\ \ ]]$. By def., $J^{\$}_{\mathscr{C},\mathscr{S}} = E$. The required relation between K and J follows by manipulation of substitutions. The other needed items (V', or F and M') carry through from the inductive hypothesis. *huzzah!*

CASE $G(\vec{W},K')$. By definition of $(-)^{\$}_{\mathscr{C},\mathscr{S}}$, we have N such that

$$(G(\vec{y},k) \Rightarrow N^{\dagger}(App(arg,k))) \in \text{cases}(cont,\mathscr{S})$$

which gives us $K^{\$}_{\mathscr{C},\mathscr{S}} = K'^{\$}_{\mathscr{C},\mathscr{S}}[[\ \ ](N\{\vec{W}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\})]$.

If $K' = k$, then we prove (c). By def., $K'^{\$}_{\mathscr{C},\mathscr{S}} = [\ \ ]$. Let F be G. Letting $J = k$ and $E = [\ \ ]$ we get $K = J\{G(\vec{W},k)/k\}$ and $J^{\$}_{\mathscr{C},\mathscr{S}} = E$ as needed. Let M' be $N^{\dagger}k$. Then $(M'\{\vec{W}/\vec{y}\}))^{\ddagger}_{\mathscr{C},\mathscr{S}} = M'^{\ddagger}_{\mathscr{C},\mathscr{S}}\{\vec{W}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\}) = N\{\vec{W}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\}$ as needed.

If $K' \neq k$ then the induction hypothesis gives us one of the cases (b) or (c); we prove the same case. The IH provides J' and E' with $J'^{\$}_{\mathscr{C},\mathscr{S}} = E'$. Let $J = G(\vec{W}, J')$ and $E = E'[[\ \ ](N\{\vec{W}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\})]$. By def., $J^{\$}_{\mathscr{C},\mathscr{S}} = E$. The required relation between K and J follows by manipulation of substitutions. The other needed items (V', or F and M') carry through from the inductive hypothesis. *huzzah!* □

Lemma 11 (Application terms' inverse image under $(-)^{\ddagger}$)**.** Given a CSM term N' and λ_{rpc} terms L and M with $N'^{\ddagger}_{\mathscr{C},\mathscr{S}} = LM$, then at least one of the following hold:

(a) there exist CSM terms L', M', $\vec{V}$ and name F s.t.:

$$\begin{array}{rcl} N' & = & L'\{F(\vec{V},k)/k\}, \\ L'^{\ddagger}_{\mathscr{C},\mathscr{S}} & = & L \\ (M'\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathscr{C},\mathscr{S}} & = & M \quad \text{and} \\ & & (F(\vec{y},k) \Rightarrow M'\{App(arg,k)/k\}) \in \text{cases}(cont, \mathscr{S}). \end{array}$$

(b) L is a value and there exist CSM terms V' and M' s.t.:

$$\begin{array}{rcl} N' & = & M'\{App(V',k)/k\}, \\ V'^{\bullet}_{\mathscr{C},\mathscr{S}} & = & L \quad \text{and} \\ M'^{\ddagger}_{\mathscr{C},\mathscr{S}} & = & M. \end{array}$$

(c) L and M are values and there exist CSM terms V' and W' s.t.:

$$N' = apply(V', W', k) \text{ and}$$
$$V'^{\bullet}_{\mathscr{C},\mathscr{S}} = L \text{ and } W'^{\bullet}_{\mathscr{C},\mathscr{S}} = M.$$

Proof. Define two terms, K and Q, as follows: Consider the possible forms of N': either $cont(K,U')$ or $apply(U',W',K)$. In the first case, let $Q = U'^{\bullet}_{\mathscr{C},\mathscr{S}}$, and in the other let $Q = U'^{\bullet}_{\mathscr{C},\mathscr{S}} W'^{\bullet}_{\mathscr{C},\mathscr{S}}$. In each case, by def., $N'^{\ddagger}_{\mathscr{C},\mathscr{S}} = K^{\$}_{\mathscr{C},\mathscr{S}}[Q]$.

Take cases on the structure of $K^{\$}_{\mathscr{C},\mathscr{S}}$ as enumerated by Lemma 10.

CASE $K^{\$}_{\mathscr{C},\mathscr{S}} = [\ ]$. We show consequent (c).

Take cases on the form of N':

- Case $cont(K,U')$. Here $N'^{\ddagger}_{\mathscr{C},\mathscr{S}} = U'^{\bullet}_{\mathscr{C},\mathscr{S}}$; but this is not an application, a contradiction. *huz.*
- Case $apply(U',W',K)$

  Here $N'^{\ddagger}_{\mathscr{C},\mathscr{S}} = U'^{\bullet}_{\mathscr{C},\mathscr{S}} W'^{\bullet}_{\mathscr{C},\mathscr{S}} = LM$. By structural equality, then, $U'^{\bullet}_{\mathscr{C},\mathscr{S}} = L$ and $W'^{\bullet}_{\mathscr{C},\mathscr{S}} = M$. *huzzah!*

CASE $K^{\$}_{\mathscr{C},\mathscr{S}} = VE$. We show consequent (b). We have $L = V$ and $E[Q] = M$. From Lemma 10 we have J and V' such that $K = J\{App(V',k)/k\}$ with $J^{\$}_{\mathscr{C},\mathscr{S}} = E$ and $V'^{\bullet}_{\mathscr{C},\mathscr{S}} = V$. Let M' be the one of $cont(J,U')$ or $apply(U',W',J)$ that matches the form of N'. Then $N' = M'\{App(V,k)/k\}$. Calculate that $M'^{\ddagger}_{\mathscr{C},\mathscr{S}} = J^{\$}_{\mathscr{C},\mathscr{S}}[Q] = E[Q] = M$, as needed. *huzzah!*

CASE $K^{\$}_{\mathscr{C},\mathscr{S}} = EN$. We show consequent (a). We have $E[Q] = L$ and $N = M$. From Lemma 10 we have terms J, M' and $\vec{V}$ and name F so that $K = J\{F(\vec{V},k)/k\}$,

$$(F(\vec{y},k) \Rightarrow M'\{App(arg,k)/k\}) \in \text{cases}(cont,\mathscr{S}),$$

and $(M'\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathscr{C},\mathscr{S}} = M$ and $J^{\$}_{\mathscr{C},\mathscr{S}} = E$; this supplies the needed F, $\vec{V}$ and M'. Let L' be the one of $cont(J,U')$ or $apply(U',W',J)$ that matches the form of N'. Then $N' = L'\{F(\vec{V},k)/k\}$, as needed. Calculate that $L'^{\ddagger}_{\mathscr{C},\mathscr{S}} = J^{\$}_{\mathscr{C},\mathscr{S}}[Q] = E[Q] = L$, as needed. *huzzah!* □

Notation. Write $M \rightharpoonup_{\mathscr{C},\mathscr{S}} V$ for

$$tramp([\ ]);M \twoheadrightarrow_{\mathscr{C},\mathscr{S}} tramp([\ ]);cont(k,V).$$

The next lemma shows that the behavior of terms in CSM follows that of the corresponding λ_{rpc} terms.

Lemma 12 (Completeness)**.** Given any CSM terms M, V and definitions $\mathscr{C}$ and $\mathscr{S}$,

(i) If $M^{\star}_{\mathscr{C},\mathscr{S}} \Downarrow_{\mathsf{c}} V$ then there exists V' with $V'^{\bullet}_{\mathscr{C},\mathscr{S}} = V$ and $M \twoheadrightarrow_{\mathscr{C},\mathscr{S}} V'$, and

(ii) if $M^{\ddagger}_{\mathscr{C},\mathscr{S}} \Downarrow_{\mathsf{s}} V$ then there exists V' with $V'^{\bullet}_{\mathscr{C},\mathscr{S}} = V$ and $M \rightharpoonup_{\mathscr{C},\mathscr{S}} V'$

Proof. By induction on the derivation of the given $M^{\star}_{\mathscr{C},\mathscr{S}} \Downarrow_{\mathsf{c}} V$ or $M^{\ddagger}_{\mathscr{C},\mathscr{S}} \Downarrow_{\mathsf{s}} V$. Take cases on the final step of the derivation:

CASE VALUE. The low-level reduction is of zero steps. The initial low-level term must be a value, V', since its image under the reverse translation is a value. The initial and final low-level terms are the same because $V'^{\star}_{\mathscr{C},\mathscr{S}} = V'^{\bullet}_{\mathscr{C},\mathscr{S}}$ and $V'^{\ddagger}_{\mathscr{C},\mathscr{S}} = V'^{\bullet}_{\mathscr{C},\mathscr{S}}$ on values. *huzzah!*

CASE BETA. Recall the rule:

$$\frac{L \Downarrow_a \lambda^b x.N \qquad M \Downarrow_a W \qquad N\{W/x\} \Downarrow_b V}{LM \Downarrow_a V}$$

Take cases on a, the location where the BETA reduction takes place.

- Case $a = \mathsf{c}$.

  Because the starting CSM term maps to LM under $(-)^{\star}$, it must be of the form $apply(L', M')$ with $L'^{\star}_{\mathscr{C},\mathscr{S}} = L$ and $M'^{\star}_{\mathscr{C},\mathscr{S}} = M$.

  By IH we have normal forms

$$\begin{aligned} L' &\twoheadrightarrow F(\vec{V}) \\ M' &\twoheadrightarrow W' \end{aligned}$$

  satisfying

$$\begin{aligned} (F(\vec{V}))^{\bullet}_{\mathscr{C},\mathscr{S}} &= \lambda^b x.N \\ W'^{\bullet}_{\mathscr{C},\mathscr{S}} &= W \end{aligned}$$

So the term reduces as follows:

$$apply(L', M') \quad \twoheadrightarrow \quad apply(F(\vec{V}), W')$$

To finish the reduction, take cases on b.

If b is c then we have N' such that

$$(F(\vec{y}) \Rightarrow N') \in \text{cases}(apply, \mathscr{C})$$

Therefore

$$\begin{array}{rcll}
(N'\{x/arg\})^{\star}_{\mathscr{C},\mathscr{S}}\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\} & = & N & \text{(def. of } (F(\vec{V}))^{\bullet}) \\
(N'\{x/arg\}\{\vec{V}/\vec{y}\})^{\star}_{\mathscr{C},\mathscr{S}} & = & N & \\
(N'\{x/arg\}\{\vec{V}/\vec{y}\})^{\star}_{\mathscr{C},\mathscr{S}}\{W'^{\bullet}_{\mathscr{C},\mathscr{S}}/x\} & = & N\{W/x\} & \\
 & = & (N'\{x/arg\}\{\vec{V}/\vec{y}\}\{W'/x\})^{\star}_{\mathscr{C},\mathscr{S}} &
\end{array}$$

And so by IH

$$N'\{x/arg\}\{\vec{V}/\vec{y}\}\{W'/x\} \twoheadrightarrow V'$$
with $V'^{\bullet}_{\mathscr{C},\mathscr{S}} = V$.

Now we can finish the reduction:

$$\begin{array}{rl}
 & apply(F(\vec{V}), W') \\
\longrightarrow & N'\{\vec{V}/\vec{y}\}\{W'/arg\} \\
\twoheadrightarrow & V'
\end{array}$$

which was to be shown. *huz.*

If b is s then we have N' such that

$$\begin{array}{rl}
 & (F(\vec{y}) \Rightarrow tramp(\mathsf{req}\ apply\ (F(\vec{y}), arg, Fin()))) \\
 & \quad \in \text{cases}(apply, \mathscr{C}) \\
\text{and} & (F(\vec{y}) \Rightarrow N') \in \text{cases}(apply, \mathscr{S})
\end{array}$$

Therefore

$$\begin{array}{rcll}
(N'\{x/arg\})^{\ddagger}_{\mathscr{C},\mathscr{S}}\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\} & = & N & \text{(def. of } (F(\vec{V}))^{\bullet}) \\
(N'\{x/arg\}\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathscr{C},\mathscr{S}} & = & N & \\
(N'\{x/arg\}\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathscr{C},\mathscr{S}}\{W'^{\bullet}_{\mathscr{C},\mathscr{S}}/x\} & = & N\{W/x\} & \\
 & = & (N'\{x/arg\}\{\vec{V}/\vec{y}\}\{W'/x\})^{\ddagger}_{\mathscr{C},\mathscr{S}} &
\end{array}$$

And so by IH

$$N'\{x/arg\}\{\vec{V}/\vec{y}\}\{W'/x\} \rightharpoonup_{\mathscr{C},\mathscr{S}} V'$$

with $V'^{\bullet}_{\mathscr{C},\mathscr{S}} = V$.

Now we can finish the reduction:

$$\begin{array}{ll} & apply(F(\vec{V}), W') \\ \twoheadrightarrow & tramp(\mathsf{req}\, apply\ (F(\vec{V}), arg, Fin())) \\ \longrightarrow & tramp([\ \ ]);\, apply(F(\vec{V}), arg, Fin()) \\ \longrightarrow & tramp([\ \ ]);\, N'\{\vec{V}/\vec{y}\}\{W'/arg, Fin()/k\} \\ \twoheadrightarrow & tramp([\ \ ]);\, cont(Fin(), V') \\ \twoheadrightarrow & tramp(Return(V')) \\ \longrightarrow & V' \end{array}$$

which was to be shown. *huzzah!*

- Case $a = \mathsf{s}$. Let X be the term such that $X^{\ddagger}_{\mathscr{C},\mathscr{S}} = LM$. Lemma 11 nominates the possible forms of X.

  First consider the case of Lemma 11(a). This gives us terms L' and M' such that

$$\begin{array}{rcl} X & = & L'\{G(\vec{U}, k)/k\} \\ L'^{\ddagger}_{\mathscr{C},\mathscr{S}} & = & L, \\ (M'\{\vec{U}/\vec{z}\})^{\ddagger}_{\mathscr{C},\mathscr{S}} & = & M \text{ and} \\ (G(\vec{z}, k) \Rightarrow M'\{App(arg, k)/k\}) & \in & \text{cases}(cont, \mathscr{S}). \end{array}$$

  By IH we have these normal forms:

$$\begin{array}{rcl} L' & \rightharpoonup_{\mathscr{C},\mathscr{S}} & F(\vec{V}) \\ M'\{\vec{U}/\vec{y}\} & \rightharpoonup_{\mathscr{C},\mathscr{S}} & W' \end{array}$$

  satisfying

$$\begin{array}{rcl} (F(\vec{V}))^{\bullet}_{\mathscr{C},\mathscr{S}} & = & \lambda^{b} x.N \\ W'^{\bullet}_{\mathscr{C},\mathscr{S}} & = & W \end{array}$$

And so we can trace the reduction of our term:

$$\begin{array}{lll} & \mathit{tramp}([\ ]);\ L'\{G(\vec{U},k)/k\} & \text{(a)} \\ \twoheadrightarrow & \mathit{tramp}([\ ]);\ \mathit{cont}(G(\vec{U},k),F(\vec{V})) & \\ \twoheadrightarrow & \mathit{tramp}([\ ]);\ M'\{\mathit{App}(F(\vec{V}),k)/k\} & \text{(b)} \\ \twoheadrightarrow & \mathit{tramp}([\ ]);\ \mathit{cont}(\mathit{App}(F(\vec{V}),k),W') & \\ \twoheadrightarrow & \mathit{tramp}([\ ]);\ \mathit{apply}(F(\vec{V}),W',k) & \text{(c)} \end{array}$$

To finish the reduction, take cases on b.

If b is c then

$$\begin{array}{rll} & (F(\vec{y}) \Rightarrow \mathit{Call}(F(\vec{y}),\mathit{arg},k)) & \in \text{cases}(\mathit{apply},\mathscr{S}) \\ \text{and} & (F(\vec{y}) \Rightarrow N') & \in \text{cases}(\mathit{apply},\mathscr{C}) \end{array}$$

Therefore

$$\begin{array}{rcll} (N'\{x/\mathit{arg}\})^{\star}_{\mathscr{C},\mathscr{S}}\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\} & = & N & \text{(def. of } (F(\vec{V}))^{\bullet}) \\ (N'\{\vec{V}/\vec{y}\}\{x/\mathit{arg}\})^{\star}_{\mathscr{C},\mathscr{S}} & = & N & \\ (N'\{\vec{V}/\vec{y}\}\{x/\mathit{arg}\})^{\star}_{\mathscr{C},\mathscr{S}}\{W'^{\bullet}_{\mathscr{C},\mathscr{S}}/x\} & = & N\{W/x\} & \\ & = & (N'\{\vec{V}/\vec{y}\}\{x/\mathit{arg}\}\{W'_{\mathscr{C},\mathscr{S}}/x\})^{\star}_{\mathscr{C},\mathscr{S}} & \end{array}$$

And so by IH

$$N'\{\vec{V}/\vec{y}\}\{x/\mathit{arg}\}\{W'_{\mathscr{C},\mathscr{S}}/x\} \twoheadrightarrow V'$$
$$\text{with } V'^{\bullet}_{\mathscr{C},\mathscr{S}} = V$$

Now we can finish the reduction:

$$\begin{array}{ll} & \mathit{tramp}([\ ]);\ \mathit{apply}(F(\vec{V}),W',k) \\ \twoheadrightarrow & \mathit{tramp}([\ ]);\ \mathit{Call}(F(\vec{V}),W',k) \\ \longrightarrow & \mathit{tramp}(\mathit{Call}(F(\vec{V}),W',k)) \\ \twoheadrightarrow & \mathit{tramp}(\mathsf{req}\ \mathit{cont}\ (k,\mathit{apply}(F(\vec{V}),W'))) \\ \longrightarrow & \mathit{tramp}(\mathsf{req}\ \mathit{cont}\ (k,N\{\vec{V}/\vec{y},W'/\mathit{arg}\})) \\ \twoheadrightarrow & \mathit{tramp}(\mathsf{req}\ \mathit{cont}\ (k,V')) \\ \longrightarrow & \mathit{tramp}([\ ]);\ \mathit{cont}(k,V') \end{array}$$

which was to be shown. *huz.*

If b is s then we have N' such that

$$(F(\vec{y}) \Rightarrow N') \in \text{cases}(apply, \mathscr{S})$$

Therefore

$$\begin{aligned}
(N'\{x/arg\})^{\ddagger}_{\mathscr{C},\mathscr{S}}\{\vec{V}^{\bullet}_{\mathscr{C},\mathscr{S}}/\vec{y}\} &= N && (\text{def. of } (F(\vec{V}))^{\bullet}) \\
(N'\{x/arg\}\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathscr{C},\mathscr{S}} &= N \\
(N'\{x/arg\}\{\vec{V}/\vec{y}\})^{\ddagger}_{\mathscr{C},\mathscr{S}}\{W'^{\bullet}_{\mathscr{C},\mathscr{S}}/x\} &= N\{W/x\} \\
&= (N'\{x/arg\}\{\vec{V}/\vec{y}\}\{W'/x\})^{\ddagger}_{\mathscr{C},\mathscr{S}}
\end{aligned}$$

And so by IH

$$\begin{aligned}
N'\{x/arg\}\{\vec{V}/\vec{y}\}\{W'/x\} &\twoheadrightarrow_{\mathscr{C},\mathscr{S}} V' \\
&\quad \text{with } V'^{\bullet}_{\mathscr{C},\mathscr{S}} = V
\end{aligned}$$

Now we can finish the reduction:

$$\begin{aligned}
& tramp([\ \ ]); apply(F(\vec{V}), W', k) \\
\twoheadrightarrow\ & tramp([\ \ ]); N'\{\vec{V}/\vec{y}, W'/arg\} \\
\longrightarrow\ & tramp([\ \ ]); cont(k, V')
\end{aligned}$$

which was to be shown.

Now consider the other cases from Lemma 11, either (b) or (c). Then we use the above reduction sequence but beginning from the correspondingly marked line. *huzzah!* □

At last we can state and prove the full correctness result, compactly:

Proposition 2 (Soundness and Completeness)**.** For any closed λ_{rpc} term M, value V and definitions $(\mathscr{C}, \mathscr{S}) = (⟦M⟧^{\text{c,top}}, ⟦M⟧^{\text{s,top}})$,

$$M \Downarrow_{\text{c}} V \iff \text{exists } V' \text{ s.t. } M^{*} \twoheadrightarrow_{\mathscr{C},\mathscr{S}} V' \text{ and } V'^{\bullet} = V$$

Proof. The ($\Leftarrow$) implication is immediate from Lemma 9. To infer the ($\Rightarrow$) implication from Lemma 12 we need to show that the given M has an M' such that $M'^{\star}_{\mathscr{C},\mathscr{S}} = M$. We can construct $M' = M^*$ and the needed relationship follows directly from the retraction lemma. □

Syntax

$$\begin{array}{rrcl}
\text{constants} & c & & \\
\text{variables} & x & & \\
\text{locations} & a, b & & \\
\text{terms} & L, M, N & ::= & \langle M \rangle^a \mid \lambda x.N \mid LM \mid V \\
\text{values} & V, W & ::= & \lambda^a x.N \mid x \mid c
\end{array}$$

Semantics

$$\boxed{M \Downarrow_a V}$$

$$V \Downarrow_a V \qquad \text{(Value)}$$

$$\lambda x.N \Downarrow_a \lambda^a x.N \qquad \text{(Abstr)}$$

$$\frac{L \Downarrow_a \lambda^b x.N \qquad M \Downarrow_a W \qquad N\{W/x\} \Downarrow_b V}{LM \Downarrow_a V} \qquad \text{(Beta)}$$

$$\frac{M \Downarrow_b V}{\langle M \rangle^b \Downarrow_a V} \qquad \text{(Clothe)}$$

Figure 3.12: The bracket-located lambda calculus, $\lambda_{\langle\rangle}$.

3.4 Extension: Location Brackets

The calculus $\lambda_{\langle\rangle}$ in Figure 3.12 adds location brackets $\langle \cdot \rangle^a$ to λ_{rpc} and allows *unannotated* λ-abstractions. The interpretation of a bracketed expression $\langle M \rangle^a$ in a location-b context is a computation that evaluates every computation step lexically within M at location a and returns the value to the location b. Unannotated λ-abstractions are not treated as values: we want all values to be mobile, and yet the body of an unannotated abstraction should inherit its required location from the surrounding lexical context. Thus, to become a value, the abstraction itself must become tagged with this location, and the ABSTR rule attaches this annotation when it is not already provided.

Figure 3.13 gives a translation from $\lambda_{\langle\rangle}$ to λ_{rpc}. Bracketed terms $\langle M \rangle^a$ are

$$
\begin{array}{rcll}
[\![\langle M\rangle^b]\!]_a & = & (\lambda^b x.[\![M]\!]_b)() & x \text{ fresh} \\
[\![\lambda x.N]\!]_a & = & \lambda^a x.[\![N]\!]_a & \\
[\![\lambda^b x.N]\!]_a & = & \lambda^b x.[\![N]\!]_b & \\
[\![x]\!]_a & = & x & \\
[\![c]\!]_a & = & c &
\end{array}
$$

Figure 3.13: Translation from $\lambda_{\langle\rangle}$ to λ_{rpc}.

simply treated as applications of located thunks; and as expected, unannotated abstractions $\lambda x.N$ inherit their annotation from their lexical context.

To argue that this translation is correct in the same way as the previous translation, we would need a reverse translation, but there is a problem: the forward translation is not injective. For example, $\langle M\rangle^b$ and $(\lambda^b x.M)()$ go to the same term. We could resort to some hack to distinguish the terms, making it injective, or we could try to prove a looser relationship, perhaps using a simulation relation.

Location brackets such as these may be an interesting language feature, allowing programmers to designate the location of computation of arbitrary terms.

3.5 Related Work

Location-aware languages Inspired by modal logic, Lambda 5 [Murphy et al., 2004, Murphy, 2007] is a small calculus with constructs for controlling the location and movement of terms and values. The type of an expression indicates whether its result is mobile (packaged for transport), located but remotely manipulable (essentially a reference to which operations can be remote applied without explicitly), or simply local; and term formers box, unbox and transport these values. By contrast, λ_{rpc} trades this fine control for simplicity, such that the programmer needn't pack and unpack values into the special types. As in the present work, the translation of Lambda 5 to an operational model also involves a CPS translation; and closure conversion as an alternative to defunctionalization.

Neubauer and Thiemann [2005] describe techniques for splitting a location-annotated sequential program into separate concurrent programs that communicate over channels. By default each program maintains its own state, unlike the asymmetrical client-server pair used here. They note that "Our framework is applicable to [the special case of a web application] given a suitable mediator that implements channels on top of HTTP." The trampoline technique we have given provides such a mediator. Neubauer [2007] and Neubauer and Thiemann [2008] later add a location-assignment scheme which assigns locations to terms in order to minimize a certain worst-case running time metric. It would be interesting to try applying such a scheme to the RPC calculus.

For security purposes, Zdancewic et al. [1999] developed a calculus with location brackets, which inspired $\lambda_{\langle\rangle}$. Their results show how a type discipline, making a translation to and from an abstract type at the brackets, can be used to prove that certain *principals* (analogous to λ_{rpc}'s locations) cannot inspect certain values passed across an interface. Such a discipline could be applied to our calculus, to address information-flow security between client and server; Zdancewic et al.'s guarantee was based on the abstractness of the abstract type. In a networked setting, we would need to insert cryptographic protections at the boundaries, perhaps interpreting these as type coercions. Matthews and Findler [2007] give a nearly identical semantics, this time to model multi-language programs; here *languages* act like principals or locations.

Defunctionalization After first being introduced in a lucid but informal account by Reynolds [1972], defunctionalization has been formalized and verified in a typed setting in several papers [Bell and Hook, 1994, Bell et al., 1997, Pottier and Gauthier, 2004, Nielsen, 2000, Banerjee et al., 2001]. Defunctionalization is formalized here in an untyped setting, which is slightly easier because we need not segregate the application machinery by type. Danvy and Nielsen [2001] and Danvy and Millikin [2008] explore a number of uses and properties of defunctionalization. In particular, the connection between continuations and evaluation contexts, exploited in our completeness proof, was noted by Danvy and Nielsen [2001] in the context of work on defunctionalization.

Defunctionalization is very similar to lambda-lifting [Johnsson, 1985], but lambda-lifting does not reify a closure as an inspectable value. Thus it would not be applicable here, where we need to serialize the function to send across the wire.

As noted in the introduction, Murphy [2007] uses closure-conversion in place of our defunctionalization; the distinction here is that the converted closures still contain code pointers, rather than using a stable name to identify each abstraction. These code pointers are only valid as long as the server is actively running, and thus it may be difficult to achieve statelessness with such a system.

Continuation-Passing The continuation-passing transformation has a long and storied history, going back to the 1970s [Fischer, 1972, Plotkin, 1975] and including nice treatments by Danvy [Danvy and Filinski, 1992]. Our treatment owes much to the presentation and results of Sabry and Wadler [1997]. Reynolds [1993] tells an interesting tale of the many discoveries of the continuation idea.

Trampolined style Ganz et al. [1999] introduced the *trampolined style* of tail-form programs, whereby every tail call is replaced with the construction of a value containing a thunk for the tail call. Instead of performing the call, then, the program is returning a representation of the next tail call to be made. The program is then to be invoked from a loop, called the trampoline, which might treat the thunk in various ways, perhaps invoking it immediately, interjecting other actions, juggling several thunks or other possibilities. A program in trampolined style only does a bounded amount of work before returning the next thunk. The authors give a mechanical translation taking any program in *tail form* (which includes CPS) to one in trampolined style.

The system presented here is an instance of trampolined style in the sense that each remote call from the client is wrapped in a trampoline, and all remote calls from the server to the client are transformed to trampoline bounces. The fact that local function calls take place directly is a departure from earlier work.

3.6 Conclusions and Future Work

We've shown how to compile a symmetrical location-aware language to an asymmetrical, stateless, client-server machine by using three classic techniques (CPS-translation, defunctionalization and trampolining) to represent the server's dynamic context as a value on the client.

In the future, we hope to extend the source calculus by adding features such as exceptions and generalizing by allowing each annotation to consist of a *set* of permissible locations (rather than a single one). We also hope to implement the "richer calculus" with location brackets in the Links language.

The present work begins with a source calculus with location annotations, but the activity of annotation may burden the programmer. Considering that some resources are available only at some locations, it should be possible to automatically assign location annotations so as to reduce communication costs—as explored by Neubauer and Thiemann [2008]—rather than requiring the programmer to carefully annotate the program. Because the dynamic location behavior of a program may be hard to predict, and because there are a variety of possible communication and computation cost models, and perhaps other issues to consider, such as security, the problem is multifaceted and would be interesting to explore.

Remote-procedure call is not the only way to structure distributed programs. Extensions of this work might compare with or extend to other models, such as message-passing, shared memory (including transactional memory), join patterns, or synchronization techniques from the distributed programming literature.

Chapter 4

Formlets

(This chapter represents joint work together with Sam Lindley, Philip Wadler and Jeremy Yallop.)

4.1 Introduction

“Formlets” are a high-level abstraction of HTML forms, or fragments thereof, embodied as a language feature in Links (and also ported to some other languages). They encapsulate the appearance of and the data processing for a set of form fields, permitting arbitrary composition and, ultimately, delivery of the field data at any desired type. They build on raw HTML forms which are *first-order* (supporting no safe composition), *unscoped* (not forcing alignment between field definitions and their consuming code) and *unstructured* (delivering results only in the form of a flat string mapping). By contrast, formlets are *composable* (small formlets can be combined to produce larger ones), *scoped* (alignment between form fields and consumers is checked), and *restructurable* (the physical appearance and the data emitted can be manipulated arbitrarily).

As an example use case, imagine that you want to permit a user to enter a date, such as 31/1/2009. This control might appeart to the user in various ways, such as a text field, a set of pop-up menus, or a small calendar. Regardless of how the widget looks and feels, its role in the application is just to produce a date value, so the part of the program that consumes the data should be insulated from the choice of user interaction.

Now, formlets enable the following scenario: one piece of code defines a formlet, of type `Formlet(Date)`, which describes how the control appears and how it determines the `Date` value. Let's say it uses three pop-up menus. A separate piece of code, oblivious to the pop-up menus, uses this formlet by placing it on a web page, and attaches a continuation which accepts the resulting `Date` data. Later on, we can replace the three pop-up menus with a small calendar without affecting the page context or the continuation. Furthermore we can compose two or more formlets to create a new formlet which seals them inside, emitting data computed from the inner formlets. For example, we can compose two date formlets, and some others, inside a "travel" formlet which emits a sealed itinerary. The composition works as you might hope: duplicating the date formlet creates no clash between the fields, and we can again use the itinerary without caring how it was entered.

The twin benefits of encapsulation (of the user interface) and separation (of data input from data consumption), as given by formlets, serve the classic vision of reusable software components.

Example

Figure 4.1 shows formlets in use. Principally, it defines a formlet *`dateFormlet`*, as a value of type `Formlet(Date)` (and `Date` is defined as a tagged triple of integers). The example shows how *`dateFormlet`* is composed out of three component formlets, corresponding to the occurrences of *`intFormlet`* seen within the `formlet` $e1$ `yields` $e2$ construction. These are wrapped up in some surrounding HTML, and we have a way of binding the data they produce to the variables *`day`*, *`mo`* and *`year`*, which can be used within the `yields` clause to produce the output of *`dateFormlet`*.

More specifically, a formlet can be imagined as having a visible part, the *rendering*, and a data-processing part, the *collector*. The rendering is given in the

```
typename Date = Date(Int, Int, Int)   # as (day, month, year)
sig dateFormlet : Formlet(Date)
var dateFormlet =
    formlet
      <div class="date-input">
        <b>Date</b>
        {intFormlet -> day}
        {intFormlet -> mo}
        {intFormlet -> year}
        (day/month/year)
      </div>
    yields
      Date(day - 1, mo - 1, year `mod` 100)
```

Figure 4.1: Links code defining a formlet for entering a structured date.

`formlet` clause of the expression, which consists in this case of three component formlets surrounded by HTML, as noted. The component formlets here are primitive formlets (*`intFormlet`*) that each render as single input fields, and return `Int`-typed data. The surrounding HTML consists here of an outer `div` element (styled with CSS class `date-input`), and inside this a boldface label "Date," and finally a short legend for the user, explaining the order of the fields (apologies to my fellow Americans; the thesis committee is European).

The collector is specified by the `yields` clause. This is how we compute the value that will be delivered when this formlet is used as a component of another, or delivered to the final consuming code; the `yields` expression is evaluated at form-submission time. To use the values of component formlets, we use a variable binding mechanism called *formlet bindings*. Within the `formlet` clause you can find the formlet binding construct `{`*`fmlt`*` -> `x`}`, which also embeds at that point a component formlet's rendering. Its left-hand side, *`fmlt`*, is an expression with type `Formlet(`T`)`. When the form is submitted, the collector of *`fmlt`* will be evaluated to produce a value of type T, to which x is then bound, scoped to the present formlet's `yields` clause.

Figure 4.2 shows how this formlet might be used within yet another formlet, and thus how formlets can be freely recombined to create further formlets.

```
typename Itinerary = Itinerary(String, Date, Date)
sig travelFormlet : Formlet(Itinerary)
var travelFormlet =
    formlet
      <div>
        <h1>Book your stay</h1>
        Name:      {stringFormlet -> name}
        Arrival:   {dateFormlet -> arr}
        Departure: {dateFormlet -> dep}
      </div>
    yields
      Itinerary(name, arr, dep)
```

Figure 4.2: Links code defining a formlet for booking a hotel stay.

Consuming formlet data

Ultimately, we want to present further pages to the user, not just bundle data in a formlet. This is done with the function *handle*:

```
sig handle : (Formlet(a), (a) -> Xml) -> Xml
```

Given a `Formlet(a)` value and a continuation `(a) -> Xml` producing the next HTML page, *handle* binds them together to produce a bona-fide HTML form. The HTML produced by *handle* is a `form` element whose `action` attribute embeds the (serialization of the) continuation as well as all the component formlets' collectors.

4.2 Idioms

Underlying this syntax is a small set of operators, which also comprise the functional-programming interface known as an *idiom* [McBride, 2005, McBride and Paterson, 2008, Lindley et al., 2008], a generalization of the monad interface [Wadler, 1995]. An idiom, also known as an "applicative functor," is a type constructor I together with parametric operations *pure* and $(\circledast)$—pronounced "apply"—having types

$$\begin{aligned} \mathit{pure} &: a \to I a \\ (\circledast) &: I(a \to b) \to I(a) \to I(b) \end{aligned}$$

such that these equations apply (the $(\circ)$ operator is function composition):

$$\begin{aligned} \mathit{pure}\, \mathit{id} \circledast u &= u, \\ \mathit{pure}\,(\circ) \circledast u \circledast v \circledast w &= u \circledast (v \circledast w), \\ \mathit{pure}\, f \circledast \mathit{pure}\, x &= \mathit{pure}\,(f\,x) \qquad \text{and} \\ u \circledast \mathit{pure}\, x &= \mathit{pure}\,(\lambda f.f\,x) \circledast u. \end{aligned}$$

Intuitively, the *pure* function lifts a value into the idiom and the $(\circledast)$ operator applies a function, already embedded in the idiom, to an argument embedded in the idiom, producing a result embedded in the idiom. The idiom type $I a$ would typically carry additional information (or "effects") alongside data of type a. The equations ensure that *pure* values can be re-ordered within an expression (an *idiomatic* expression, one built from the idiom operators), preserving the result, but that impure ones cannot, in general; thus the order of these effects may make a difference to the idiom.

In our case, what effects are involved? We can consider name generation, HTML-accumulation, and environment reading (used for collecting the result) to be "effects." In the course of a formlet construction, we generate fresh names for input fields, we accumulate an HTML structure as we go, and we compose environment functions into larger environment functions.

So the idiom is defined as follows (see Figure 4.4): the *pure* operation takes a result value and returns a constant formlet (always yielding the given value), carrying an empty rendering, without HTML elements. The $(\circledast)$ operator, besides performing a function application, also composes the effects of the two formlets: it threads the name generator through the renderers (so that their generated names will not clash), concatenates the HTML renderings, and composes the collectors, passing the same environment to each. In so doing it co-ordinates the generated names between renderer and collector to keep them in synch.

$$\begin{aligned}\mathsf{xml} = \mathrm{Tag}\,\mathsf{of}\,(\mathsf{string} \times [(\mathsf{string} \times \mathsf{string})] \times [\mathsf{xml}])\\ \mid \mathrm{Text}\,\mathsf{of}\,\mathsf{string}\end{aligned}$$

Figure 4.3: Definition of an XML data type for use with formlets.

$$\mathit{env} = [\mathsf{string} \times \mathsf{string}]$$

$$\begin{array}{rcl}
\mathrm{I}_{\mathrm{formlet}}(a) & = & \mathsf{int} \rightarrow ([\mathsf{xml}] \times (\mathsf{env} \rightarrow a)) \times \mathsf{int} \\
\mathit{pure}_{\mathrm{formlet}}\, x & = & \lambda i.(([\,], \lambda e.x), i) \\
u \circledast_{\mathrm{formlet}} v & = & \lambda i.\mathsf{let}\,((\mathit{xml}_f, c_f), i') = u\,i\ \mathsf{in} \\
& & \quad \mathsf{let}\,((\mathit{xml}_x, c_x), i'') = v\,i'\ \mathsf{in} \\
& & \quad\ ((\mathit{xml}_f +\!\!+\, \mathit{xml}_x, \lambda e.c_f e(c_x e)), i'') \\
\\
\mathit{stringFormlet} & = & \lambda i.((\mathit{xmlTag}\ \texttt{"input"}\ [(\texttt{"name"}, \mathit{intToString}\ i)]\ [\,], \\
& & \quad \lambda e.\mathit{lookup}\ i\ e), i+1) \\
\mathit{intFormlet} & = & \mathit{pure}(\mathit{stringToInt}) \circledast \mathit{stringFormlet}
\end{array}$$

Figure 4.4: Definition of the formlet idiom.

$$\begin{array}{rcl} pure_{I \circ J} & = & pure_I \circ pure_J \\ f \circledast_{I \circ J} x & = & ((pure_I(\circledast_J))\, f) \circledast_I x \end{array}$$

Figure 4.5: The composition of idioms.

In fact, the three kinds of side effects are so crisply distinct that each can be defined as a separate idiom, and the formlet idiom falls out as their composition, with a few additional operators that entangle their effects. The three component idioms are essentially standard (they are derived directly from standard *monads*): the name-generation idiom (I_n), the monoid-accumulation idiom over the HTML monoid (I_x), and the environment-reader idiom (I_e). The formlet idiom decomposes as follows:

$$\mathrm{I}_{\mathrm{formlet}} = \mathrm{I}_n \circ \mathrm{I}_x \circ \mathrm{I}_e .$$

The composition operation on idioms is defined formally in Figure 4.5, and the component idioms in Figure 4.6. Additionally, we need some primitive formlets, which are defined directly in terms of the component idioms. Figure 4.7 defines the primitives *stringFormlet* and *intFormlet*.

So far we have not seen that the definitions in Figure 4.4 satisfy the idiom laws. The fact that they do follows from two general facts: first, that every monad is an idiom, and hence the three component idioms of Figure 4.6, which are derived from standard monads, are idioms. And second, the composition of idioms is also an idiom; hence our formlet structure is an idiom. Both of these general facts are noted in McBride and Paterson [2008].

Could formlets could be defined as a monad, rather than as an idiom? After all, the monad interface, being more strict, admits more uses, which might be beneficial. But the formlet idiom could *not* be defined as a monad, as can be seen by studying the type of the monadic "bind" operation:

$$(\gg\!=) : \mathrm{Formlet}(a) \to (a \to \mathrm{Formlet}(b)) \to \mathrm{Formlet}(b)$$

$$\begin{aligned}
\mathrm{I}_n(a) &= \mathsf{int} \to a \times \mathsf{int} \\
\mathit{pure}_n\, x &= \lambda i.(x, i) \\
u \circledast_n v &= \lambda i.\mathsf{let}\,(f, i') = u\,i \;\mathsf{in}\; \mathsf{let}\,(x, i'') = v\,i' \;\mathsf{in}\; (f\,x, i'') \\
\mathit{freshName} &= \lambda i.(i, i+1)
\end{aligned}$$

$$\begin{aligned}
\mathrm{I}_x(a) &= [\mathsf{xml}] \times a \\
\mathit{pure}_x\, x &= ([\,], x) \\
u \circledast_x v &= \mathsf{let}\,(x_1, f) = u \;\mathsf{in}\; \mathsf{let}\,(x_2, x) = v \;\mathsf{in}\; (x_1 +\!\!+\, x_2, f\,x) \\
\mathit{xmlText}\;\mathit{text} &= (\mathrm{Text}(\mathit{text}), ()) \\
\mathit{xmlTag}\;\mathit{tagName}\;\mathit{attrs}\;\mathit{contents} &= (\mathrm{Tag}(\mathit{tagName}, \mathit{attrs}, \mathit{contents}), ())
\end{aligned}$$

$$\begin{aligned}
\mathrm{I}_e(a) &= \mathsf{env} \to a \\
\mathit{pure}_e\, x &= \lambda e.x \\
u \circledast_e v &= \lambda e.u e(v e) \\
\mathit{getEnv}\; n &= \lambda e.\mathit{lookup}\; n\; e
\end{aligned}$$

Figure 4.6: The three standard idioms comprising formlets.

$$\begin{aligned}
\mathit{stringFormlet} &= \mathit{pure}_n\,(\lambda n.(\mathit{pure}_x\,(\lambda().\mathit{getEnv}\; n)) \\
&\qquad\qquad \circledast (\mathit{xmlTag}\; \texttt{"input"}\; [(\texttt{"name"}, n)]\; [\,])) \\
&\quad\; \circledast_n \mathit{freshName} \\
\mathit{intFormlet} &= (\mathit{pure}_{\mathrm{formlet}}\; \mathit{intOfString}) \circledast \mathit{stringFormlet} \\
\mathit{xmlText}_{\mathrm{formlet}}\; \mathit{str} &= \mathit{pure}_n\,(\mathit{xmlText}\; \mathit{str}, \mathit{pure}_e()) \\
\mathit{xmlTag}_{\mathrm{formlet}}\; \mathit{tagName}\; \mathit{attrs}\; \mathit{fmlt} &= \\
&\mathit{pure}_n(\lambda(\mathit{xml}, \mathit{envr}).(\mathit{xmlTag}\; \mathit{tagName}\; \mathit{attrs}\; \mathit{xml}, \mathit{envr})) \circledast_n \mathit{fmlt} \\
\mathit{xmlTree}_{\mathrm{formlet}}\; \mathit{xml} &= \mathit{pure}_n(\mathit{xml}, \mathit{pure}_e())
\end{aligned}$$

Figure 4.7: Primitive formlet operations.

Here the Formlet(b) produced by the second argument may depend on the a value contained within the first argument. But the a value in a Formlet(a) is the user's data; and yet we need to combine the two formlets on a page *before* presenting them to the user. So how could we pass the user's data to the function generating the second formlet? The data is not available at the right time, preventing the structure we've been studying from constituting a monad. Viewed another way, the monad interface requires us to support a dependency which we cannot support, that between the a and the Formlet(b) in the second argument to ($\gg=$).

We *could* define a "formlet monad" which presents formlets *in sequence*, each one after the user has submitted the previous one, but that is not our formlet structure, which composes formlets on a single page.

It may seem puzzling that we have taken three standard monads, considered them as idioms, composed them, and arrived at something which is not a monad. However, it is well-known that monads do not always compose to form monads, and indeed these particular ones do not.

Idioms, then, comprise a suitably-general yet minimally-powerful interface in which to express formlets. The factorization into standard idioms sheds light on their structure and suggests this definition is more basic than just an ad-hoc design.

4.3 Syntax

Figure 4.8 formally defines the formlet syntax. The additional constructs are added to the expression grammar of Links, or indeed any other suitable functional programming language. The extra syntactic forms are removed by the translation $[\![-]\!]$ of Figure 4.9.

The target of the translation uses the operations of the formlet idiom as defined in Figure 4.7, including the basic idiom operations.

In this chapter (as in the Links language), the notation `<#>`$e_1 e_2 \cdots e_n$`</#>` denotes an HTML forest: a sequence of HTML trees. The notation is needed to switch syntactic modes from the surrounding language, where juxtaposition

Expressions

$$e \ ::= \ \cdots \mid \mathsf{formlet}\ q\ \mathsf{yields}\ e \quad \text{(formlet)}$$

Formlet quasiquotes

$$\begin{array}{rcll} n & ::= & s \mid \{e\} \mid \{f \Rightarrow p\} \mid \texttt{<}t\ ats\texttt{>}n_1 \cdots n_k\texttt{</}t\texttt{>} & \text{node} \\ q & ::= & \texttt{<}t\ ats\texttt{>}n_1 \cdots n_k\texttt{</}t\texttt{>} \mid \texttt{<\#>}n_1 \cdots n_k\texttt{</\#>} & \text{quasiquote} \end{array}$$

Meta variables

e	expression	f	formlet-type expression	t	tag
p	pattern	s	string	ats	attribute list

Figure 4.8: Quasiquote syntax.

$$\begin{array}{rcl} [\![\mathsf{formlet}\ q\ \mathsf{yields}\ e]\!] & = & pure(\mathsf{fun}\ q^\dagger \to [\![e]\!]) \circledast q^\circ \\ \\ s^\circ & = & xmlText\,s \\ \{e\}^\circ & = & xmlTree\,[\![e]\!] \\ \{f \Rightarrow p\}^\circ & = & [\![f]\!] \\ (\texttt{<}t\,ats\texttt{>}n_1 \cdots n_k\texttt{</}t\texttt{>})^\circ & = & xmlTag\,t\,ats\,(\texttt{<\#>}n_1 \cdots n_k\texttt{</\#>})^\circ \\ (\texttt{<\#>}n_1 \cdots n_k\texttt{</\#>})^\circ & = & pure(\mathsf{fun}\ n_1^\dagger \cdots n_k^\dagger \to (n_1^\dagger, \ldots, n_k^\dagger)) \circledast n_1{}^\circ \cdots \circledast n_k{}^\circ \\ \\ s^\dagger & = & () \\ \{e\}^\dagger & = & () \\ \{f \Rightarrow p\}^\dagger & = & p \\ (\texttt{<}t\,ats\texttt{>}n_1 \cdots n_k\texttt{</}t\texttt{>})^\dagger & = & (n_1^\dagger, \ldots, n_k^\dagger) \\ (\texttt{<\#>}n_1 \cdots n_k\texttt{</\#>})^\dagger & = & (n_1^\dagger, \ldots, n_k^\dagger) \end{array}$$

Figure 4.9: Desugaring XML and formlets.

$e_1 e_2$ denotes function application, to HTML mode, where juxtaposition denotes construction of HTML node sequences. In particular, <#></#> denotes the empty HTML-forest value.

The idiom operators are complete for the syntax and vice versa. The former can be seen from the definition of the syntax; the latter is seen by letting

$$\begin{aligned} \mathit{pure}\, x &= \textsf{formlet <\#></\#> yields}\, x \\ f \circledast x &= \textsf{formlet}\, \{f \to f'\}\{x \to x'\}\, \textsf{yields}\, f'x'. \end{aligned}$$

4.4 Related Work

Web form abstraction has been tackled in several systems; formlets can be seen as a distillation of these systems into an essential core.

The WASH Haskell library [Thiemann, 2002, 2005] provides combinators for building up web forms with a typed result. It allows individual fields to be read at any desired type, by associating that type with a function to parse the incoming string. It also supports aggregating together multiple fields using tupling constructors, though these tuplings still reveal their component parts, rather than encapsulating the data at some other chosen type.

Two existing systems, iData and Curry/WUI, support the same degree of abstraction as formlets—they allow constructing form results at arbitrary type with arbitray computation. In fact, iData provided a guide in the development of formlets, while we were pleased to find how well Curry/WUI fit while finalizing our formlets work.

The iData framework [Plasmeijer and Achten, 2006], in the language Clean, offers form abstractions, each called *an iData*. Underlying iData is an abstraction much like formlets. As with formlets, iData permits constructing form results at arbitrary type in any computable way. Unlike formlets, an iData does not abstract from the field-name generation, so an iData exposes its internal structure. On the one hand, this exposes the programmer to field-name clashes. But on the other, it allows iDatas to be interdependent, each displaying a function of data entered into the other.

The WUI library [Hanus, 2006, 2007], in the functional logic language Curry, implements WUIs, an abstraction very similar to that of formlets. A WUI not only *emits* a value of the desired type, it also *consumes* an argument of that type, a default. This is useful since one commonly needs to pre-populate a form with values previously entered.

It is worth noting that, since the WUI abstraction uses its type argument both positively and negatively, it could not be defined as an idiom as such. This is because, to transform a $\mathit{Wui}\, a$ into a $\mathit{Wui}\, b$ requires not only a function $a \to b$ but also an inverse function $b \to a$. The direct analogy with functional application is thus strained, and WUIs would not fit the idiom interface. Perhaps this should incite research into another interface which applies *bidirectional* transformations to its objects.

The formlet abstraction itself has been implemented by other programmers, in Haskell [Eidhof, 2008] and Scheme [McCarthy, 2008]; the latter implementation includes the syntactic sugar and is now a part of the standard distribution of PLT Scheme [PLT]. Extensions to the idiom, supporting XHTML validation and user-input validation (with appropriate combinators to control the validation) are provided by Cooper et al. [2008], from which this chapter was derived.

Strugnell [2008] used formlets when porting a PHP project-management application to Links, and wrote a comparison of Links's with PHP's approach to form construction.

4.5 Conclusion

Formlets are both a high-level description of a safe, composable form abstraction and a usable system implemented in several programming languages. Formlets encapsulate form presentation and data packaging, and supports decoupling these from further transformation steps and the eventual consumption of the data, and thus aid in creating modular web software. The operators allow constructing and transforming formlets in a compositional style. An accompanying syntax allows writing and composing formlets with the familiar notation of HTML, making them easy to use.

As a new application of the idiom interface, formlets show once again that interface's utility. The Links team hopes that this work might lead to further applications and research on idioms, perhaps including a syntax like the formlet syntax but general enough to work with any idiom.

Chapter 5

SQL Compilation

(This chapter represents sole work by the author, advised by Philip Wadler.)

5.1 Introduction

We've had a look at user-interface programming and client-server interaction. Now what about the third leg of web programming: data persistence?

Data persistence in web applications is commonly provided by a relational database. But a programmer's interface to such a database is rarely comfortable.

Common practices for bridging the distance are either unsafe, or inflexible, or use a different data model, giving rise to an impedance mismatch problem. Programmers often form SQL queries simply by concatenating strings at runtime ("string interpolation"); this is risky, as it becomes easy to make malformed or undesired queries. Sometimes programmers develop a library of "prepared statements" which are parameterized only in controlled ways; this is safer but inflexible, requiring the programmer to dip into the library each time a prepared statement needs to be adjusted. Object–Relational mappings (ORMs) provide a safe and flexible object-oriented interface to data, but moving between the object and relational models can give rise to subtle problems, for example when object identity is relevant on the object side, or when complex join queries that are inexpressible on the object side are needed.

All common approaches fall short in abstraction: none allows applying abstracted query fragments, such as "where" conditions, in multiple contexts. And all of them bring their own query language, with its own peculiar syntax (this language may be SQL itself or the language comprised by the ORM library).

A newer approach is *language-integrated query*, as exemplified by Microsoft's LINQ [Microsoft Corporation, 2005], Edinburgh Links [Cooper et al., 2006], and the Kleisli system for querying bioinformatics databases [Wong, 2000]. With language-integrated query, the programmer writes iterations over the database tables directly in a host programming language, and the programming system takes care of converting these into SQL queries.

As an alternative to the common query interfaces, such a language-integrated query system offers:

- intermediate nested data structures,
- native host-language syntax,
- abstraction over query expressions (including predicates, data transformations, and join tables, to name a few).

Close language integration for queries has fundamental limitations, since general-purpose programming languages are more expressive than SQL. Besides the problem of nested data structures, programming languages normally have operations that cannot be expressed in SQL, including recursion, side-effecting statements, and primitive functions that simply aren't available. As a result, to fully and safely integrate query languages into general-purpose languages, we need an analysis that identifes "queryizable" expressions within programs that are not, as a whole, queryizable—an analysis which this chapter provides.

Contributions In all, this chapter makes several contributions to the science of language-integrated query; it shows

1. how to translate a suitable expression in a typical impure functional programming language, as long as it has flat relation type (i.e., it denotes a

bag of records of base values), to an equivalent single SQL query (where "suitable" includes the abjuration of recursive and effectful computations and non-SQL-expressible primitive functions). This suitable sublanguage is enough to express any query in a significant fragment of SQL.

2. how to support linguistic abstraction over query expressions by allowing them to contain free variables, possibly including predicates or other functions, and a type-and-effect system to enforce that these variables can only be given runtime values that make the expression queryizable.

3. that the programmer can use an annotation to insist that a particular expression be queryized, so that the compiler can fail or warn at compile-time if the expression is not surely queryizable.

Wong [1996] showed that any expression in a pure, first-order, nested relational algebra can be rewritten so that it produces no intermediate data structures deeper than the input and output relations (calling this property of the algebra "conservativity"). This chapter extends that result by making the relational algebra higher-order (functions are first-class), setting it in the context of an impure language, and providing a "queryizability" analysis.

Example Suppose Alice runs a local baseball league and tracks the teams with a database and language-integrated query system. First, she wants a list of the players with age at least 16. She might write this function:

```
fun overAgePlayers() {
  query { for (p <- players)
          where (p.age > 16)
            [(name = p.name)] }
}
```

The compiler can deduce that this expression is in fact equivalent to an SQL query (it is *queryizable*), so it accepts the function. However, the following code would give a compiler error:

```
fun overAgePlayersReversed() {
 query { for (p <- players)
         where (p.age > 16)
           [(name = reverse(p.name))] }          # ERROR!
}
```

This is because the *reverse* function has no SQL equivalent, and so no query is equivalent to this expression.

Now, it takes nine players to make a baseball team, but some “teams” in Alice’s league are short of players. One way to find them is to collect, for each team, a team roster (list of players) and then filter to those with a roster of size at least nine. She needs to generate a list of players that belong to a “short” team, to inform them that they won’t be able to play this season. She writes the following code:

```
fun teamRoster(name) {
  for (p <- players)
  where (p.team == name)
    [(playerName=p.name)]
}

fun unusablePlayers() {
  query {
    var teamRosters =
          for (t <- teams)
            [(name = t.name,
              roster = teamRoster(t.name))];
    for (t <- teamRosters)
    where (length(t.roster) < 9)
      t.roster
  }
}
```

(Recall that the `for`-comprehension takes the *union* of the collections produced by the body, so the last comprehension in this code takes the union of the rosters of all the short-handed teams.) This expression is equivalent to an SQL query, although not in a direct way, since it uses an intermediate data structure that is nested (the variable *teamRosters* has type `[(name:String, roster:[(playerName:String)])]`), and this is not supported by SQL. But since the final result is flat, our analysis accepts the query-bracketed expression and translates it into an equivalent SQL query, such as this one:

```
select t.name as team, p.name as player
  from players as p, teams as t
 where ((select count(*) from players as p2
         where p2.team = t.name) < 9)
```

In this latest example, the query compiler will effectively inline the call to *teamRoster* at runtime when forming the query corresponding to the query block in *unusablePlayers*. Functions called from within query brackets, such as *teamRoster*, do not need to be wrapped in query brackets, since it is only when they are used inside some query that the queryizability requirement appears. This way, *teamRoster* can be used equally well within or without a query.

Suppose now that Alice wishes to abstract the query condition on teams. That is, she wishes to write a function which accepts as argument a predicate, one that selects certain teams based on their rosters, and produces a list of the player-records belonging to those teams. With the following code, the query translator will produce a single SQL query each time *playersBySelectedTeams* is invoked:

```
fun playersBySelectedTeams(pred) {
  query {
    var teamRosters =
          for (t <- teams)
            [(name = t.name,
              roster = teamRoster(t.name))]);
    for (t <- teamRosters)
    where (pred(t.roster))
      t.roster
  }
}
```

This type of abstraction is particularly difficult to achieve in SQL, since the team rosters themselves cannot be explicitly constructed in SQL as part of a query. SQL places restrictions on how subqueries can be used (For example, in various contexts they must return just one column, just one row, or both) and has a non-orthogonal syntax so that the subquery itself must be changed depending on how it is used. Therefore, it may not be possible to implement this function simply as an SQL query string with "holes" for the predicate to be inserted into.

The compiler will ensure that any argument passed as *pred* is itself a queryizable function. If any call site tries to pass a non-queryizable predicate, it produces a compiler error.

For example, if Alice wants to apply *playersBySelectedTeams* to a predicate *shortTeam*, as follows,

```
playersBySelectedTeams(shortTeam)
```

this will be accepted if the definition of *shortTeam* is as follows

```
fun shortTeam(roster) {
  length(roster) < 9
}
```

but will produce an error if we try to produce some output within *shortTeam*, like this:

```
fun shortTeam(roster) {
  print(toString(roster));
  length(roster) < 9
}
```

The compiler will mark this definition unqueryizable, preventing it from being passed as a parameter to *playersBySelectedTeams*. The latter *shortTeam* function is still a perfectly good function, however, usable in other contexts.

How it works The recipe for compiling abstractable, higher-order language-integrated queries can be summarized as follows:

1. At compile time,

   (a) Check statically that query expressions have flat relation type,

   (b) Use a *type-and-effect system* to determine that query expressions are pure, and

   (c) Associate with each queryizable expression two representations, one used for direct evaluation and one used for queryization.

2. At runtime, to execute an expression via SQL,
   (a) insert the values for any free variables, forming a closed expression, and
   (b) reduce the expression to eliminate intermediate structures (that is, functions and nested data structures); this produces a normal form directly translatable to SQL.

Road map The rest of this chapter: (§5.2) defines a source language (a model of an impure functional language with comprehensions), (§5.3) gives a translation from this language to SQL, by rewriting terms into a normalized sublanguage which embeds directly into SQL, (§5.4) proves the correctness of the translation, (§5.5) extends to the language with recursion, and then (§5.8) gives the history of language-integrated query and query-unnesting.

5.2 The language

The source language resembles the core of an ordinary impure functional programming language, and is also a conservative extension of the (higher-order) Nested Relational Calculus with side-effects and a "query" annotation. Its grammar is given in Figure 5.1. All the term forms are pure, without side-effects, except for the primitives, $\oplus$, which may be given types with side-effects.

The terms $[M]$, $[]$ and $M \uplus N$ represent bag (multiset) operations: singleton construction, the empty bag, and bag-union. The bag comprehension $\mathsf{for}(x \leftarrow L)M$ computes the union of the bags produced by evaluating M in successive environments formed by binding x to the elements of L in turn. A table handle $\mathsf{table}\, s : T$ denotes a reference to a table, named s, in some active database connection; T designates the effective type of the table. In keeping with the flatness of SQL tables, we require that each table must have *relation type* (see definition below).

The conditional form $\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N$ evaluates to the value of either M or N depending on the value of B.

$$
\begin{array}{rrcl}
\text{terms} & B,L,M,N & ::= & [M] \mid [\,] \mid M \uplus N \\
 & & \mid & \mathsf{for}\,(x \leftarrow L)\,M \\
 & & \mid & \mathsf{table}\,s : T \\
 & & \mid & \mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N \\
 & & \mid & (\overrightarrow{l = M}) \mid M.l \\
 & & \mid & \lambda x.N \mid LM \mid x \mid c \\
 & & \mid & \oplus(\vec{M}) \\
 & & \mid & \mathsf{empty}(M) \\
 & & \mid & \mathsf{query}\,M \\
\text{primitives} & \oplus & & \\
\text{table names} & s,t & & \\
\text{field names} & l & & \\
 & & & \\
\text{types} & T & ::= & o \mid (\overrightarrow{l : T}) \mid [T] \mid S \xrightarrow{e} T \\
\text{base types} & o & ::= & \mathsf{bool} \mid \mathsf{int} \mid \mathsf{string} \\
\text{atomic effects} & E & ::= & \mathsf{noqy} \mid \cdots \\
\text{effect sets} & e & & \text{a set of atomic effects}
\end{array}
$$

Figure 5.1: Source language.

Records are constructed as a parenthesized sequence of field-name–term pairs $(\overrightarrow{l = M})$ and destructed with the field projection $M.l$. When speaking of a record construction $(\overrightarrow{l = M})$ we will indicate the immediate subterms by subscripting the metavariable M with labels so that M_l is a field of $(\overrightarrow{l = M})$ for each $l \in \vec{l}$. Similarly for record types $(\overrightarrow{l : T})$ the field types will be indicated T_l when $l \in \vec{l}$.

Functional abstraction $\lambda x.N$ and application LM are as usual. Variables are ranged by x, y, z and other italic alphabetic identifiers, but c ranges over constants.

The language is equipped with a suite of primitive operations, ranged by $\oplus$, which must appear fully-applied (this is not a significant restriction since one may abstract over such expressions). The primitives must include the boolean negation operation $(\neg)$. The $(\neg)$ operator is never recognized, only produced, by the rewrite system.

The form $\mathsf{empty}(M)$ evaluates to a boolean indicating whether the bag denoted by M is the empty bag or not.

The form query M forms a term which evaluates to the same value as M but which operationally must evaluate as an SQL query.

We assume that all bound variables in the source program are distinct.

Terms are assigned types which can be: *base* types ranged by o, *record* types $(\overrightarrow{l : T})$ where each field label l is given a type T_l, *bag* types $[T]$ and *function* types $S \xrightarrow{e} T$, where S is the function domain, T is the range, and e is a set of effects that the function needs permission to perform. The type system is monomorphic, so for example each appearance of the empty bag `[]` must be given some particular concrete type. Adding polymorphism presents some difficulties of its own and is left for future work.

Programming languages typically have something like a list type as primitive but *bags* are only provided, if at all, through a library implementation. To implement the results of this chapter, then, it is necessary to choose such a bag type and endow it with "primitive" status. List-typed (and array-typed) expressions cannot be treated by the methods given here, since they may use order-aware operations (such as head, tail, and ordinal indexing) that don't translate to the database. (But it may be possible to eliminate intermediate list-typed data by methods similar to those of this chapter.) Certain order-aware operations are important in database querying, such as SQL's `order by` clauses and the widely available `limit` and `offset` features. They might be handled by extending these results, but this is future work.

Effects are here considered abstractly: E ranges over some arbitrary set of effects, which includes at least an effect noqy (for "not a query") and may include flags representing other runtime actions such as I/O or reference-cell mutations. Every effect should represent some kind of runtime behavior that has no SQL equivalent; we use the distinguished effect noqy to mark nontranslatable operations when no other effect presents itself.

For this chapter, the single effect noqy is all we need, but this general treatment shows that we can mix this with other effect analyses. Since we only need the one effect, it would be enough to replace the *effect sets* with a simple flag

$e ::= \mathsf{qy} \mid \mathsf{noqy}$ with an operation ($\cup$) such that $\mathsf{qy} \cup \mathsf{qy} = \mathsf{qy}$, $\mathsf{noqy} \cup e = \mathsf{noqy}$ and $e \cup \mathsf{noqy} = \mathsf{noqy}$.

Every primitive $\oplus$ must either have an SQL equivalent $\oplus_{\mathrm{db}}$ or else carry an effect annotation (perhaps the catch-all noqy). The SQL equivalent is a macro which expands to some combination of primitives available in SQL. Thus primitives need not be in one-to-one correspondance with real SQL operations. We also insist that primitives have basic argument types and basic result type, or else have an effect annotation.

To be perfectly clear, this document defines the terms "row type" and "relation type" as follows:

Definition (Row and Relation Types)**.** A *row type* is a type of the form $(\overrightarrow{l = o})$, that is, a record each of whose fields has some base type. A *relation type* is a type of the form $[T]$ where T is a row type.

(This meaning of "row type" should not be confused with the completely different term "row types" for polymorphically extensible records [Pottier and Rémy, 2005]. The name "row" here evokes the rows of a database table.)

NRC Comparison Compare the given language with the Nested Relational Calculus (NRC) as given by Wong [1996], shown in Figure 5.2. The two languages are nearly the same. Some apparent differences are only notational. NRC's comprehension form $\bigcup\{M \mid x \in L\}$ is identical to ours, $\mathsf{for}\,(x \leftarrow L)\,M$. The NRC literature uses set notation, $\{\}$, $\{M\}$, and $M \cup N$, but they can denote any of the extended monads for bags, sets or lists. This chapter treats only bags; some of the transformations examined here do not preserve order and hence lists present some difficulty. We write table handles explicitly as $\mathsf{table}\, s : T$, while NRC uses free variables to refer to tables.

NRC uses tuples while we use records, a mild generalization. And our language extends NRC with the assertion for query-translatability, $\mathsf{query}\, M$.

The calculus of this chapter includes in its grammar an arbitrary set of primitives (ranged by the symbol $\oplus$) while NRC, at its core, includes no operations;

This work	NRC
$LM \quad \lambda x.N \quad x$	$LM \quad \lambda x.M \quad x$
$\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N$	$\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N$
$\mathsf{empty}\, M$	$\mathsf{empty}\, M$
c	c
$\mathsf{for}\,(x \leftarrow L)\, M$	$\bigcup\{M \mid x \in L\}$
$[\,] \quad [M] \quad M \uplus N$	$\{\} \quad \{M\} \quad M \cup N$
$\mathsf{table}\, s : T$	x
$(\overrightarrow{l = M}) \quad M.l$	$() \quad (L, M) \quad \pi_1 \quad \pi_2$
$\oplus(\overrightarrow{M})$	
	$M = N$
$\mathsf{query}\, M$	

Figure 5.2: Comparing with Nested Relational Calculus.

instead they are normally treated as extensions.

Wong's formulation of NRC includes an equality test at each type; we lump this under the primitives. Wong [1996] shows how to implement equality at other types in terms of the base equality.

Type-and-effect system

The static type-and-effect system for queryizability analysis is given in Figure 5.3. It is close to a standard type-and-effect system along the lines of Talpin and Jouvelot [1992] and their predecessors Gifford and Lucassen [1986] and Lucassen and Gifford [1988].

The judgment $\Gamma \vdash M : T \,!\, e$ can be read "With variables of types as in context Γ, the term M has type T and may perform effects in the set e."

The system permits no recursion, and thus is analogous to simply-typed λ-calculus. We add recursion later, in Section 5.5.

The type of each constant c is given by T_c, which must be a base type. Constant values at complex type can, of course, be constructed explicitly. Each primitive has a given type, denoted by a judgment $\oplus : S_1 \times \cdots \times S_n \xrightarrow{e} T$.

An immediate type annotation is required on table expressions. This may

.

$$\Gamma \vdash c : T_c \,!\, \varnothing \quad \text{(T-Const)}$$

$$\Gamma, x : T \vdash x : T \,!\, \varnothing \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : S \vdash N : T \,!\, e'}{\Gamma \vdash \lambda x.N : S \xrightarrow{e'} T \,!\, \varnothing} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash L : S \xrightarrow{e} T \,!\, e' \qquad \Gamma \vdash M : S \,!\, e''}{\Gamma \vdash LM : T \,!\, e \cup e' \cup e''} \quad \text{(T-App)}$$

$$\frac{\oplus : S_1 \times \cdots \times S_n \xrightarrow{e} T \qquad \Gamma \vdash M_l : S_l \,!\, e_l}{\Gamma \vdash \oplus(\vec{M}) : T \,!\, e \cup \bigcup_{l \in \vec{l}} e_l} \quad \text{(T-Op)}$$

$$\frac{\Gamma \vdash M : T \,!\, \varnothing \qquad T \text{ has the form } [(\overrightarrow{l : o})]}{\Gamma \vdash \mathsf{query}\, M : T \,!\, \varnothing} \quad \text{(T-Db)}$$

$$\frac{T \text{ has the form } [(\overrightarrow{l : o})]}{\Gamma \vdash (\mathsf{table}\, t : T) : T \,!\, \varnothing} \quad \text{(T-Table)}$$

$$\frac{\Gamma \vdash M : [S] \,!\, e \qquad \Gamma, x : S \vdash N : [T] \,!\, e'}{\Gamma \vdash \mathsf{for}\,(x \leftarrow M)\, N : [T] \,!\, e \cup e'} \quad \text{(T-For)}$$

$$\frac{\Gamma \vdash M_l : T_l \,!\, e_l \text{ for each } l \in \vec{l}}{\Gamma \vdash (\overrightarrow{l = M}) : (\overrightarrow{l : T}) \,!\, \bigcup_{l \in \vec{l}} e_l} \quad \text{(T-Record)}$$

$$\frac{\Gamma \vdash M : (\overrightarrow{l : T}) \,!\, e \qquad (l : T_l) \in (\overrightarrow{l : T})}{\Gamma \vdash M.l : T_l \,!\, e} \quad \text{(T-Project)}$$

$$\Gamma \vdash [\,] : [T] \,!\, \varnothing \quad \text{(T-Null)}$$

$$\frac{\Gamma \vdash M : T \,!\, e}{\Gamma \vdash [M] : [T] \,!\, e} \quad \text{(T-Singleton)}$$

$$\frac{\Gamma \vdash M : [T] \,!\, e \qquad \Gamma \vdash N : [T] \,!\, e'}{\Gamma \vdash M \uplus N : [T] \,!\, e \cup e'} \quad \text{(T-Union)}$$

$$\frac{\Gamma \vdash M : [T] \,!\, e}{\Gamma \vdash \mathsf{empty}(M) : \mathsf{bool} \,!\, e} \quad \text{(T-Empty)}$$

$$\frac{\Gamma \vdash L : \mathsf{bool} \,!\, e \qquad \Gamma \vdash M : T \,!\, e' \qquad \Gamma \vdash N : T \,!\, e''}{\Gamma \vdash \mathsf{if}\, L\, \mathsf{then}\, M_1\, \mathsf{else}\, M_2 : T \,!\, e \cup e' \cup e''} \quad \text{(T-If)}$$

$$\frac{\Gamma \vdash M : T \,!\, e \qquad e \subseteq e'}{\Gamma \vdash M : T \,!\, e'} \quad \text{(T-Eff-Weakening)}$$

Figure 5.3: Type-and-effect system.

seem a nuisance; in fact it is not strictly necessary, since an algorithm could infer the type at which the table is used. As a pragmatic matter, however, it provides a direct way for the programmer to check whether the usage type of a table agrees with the underlying table's schema type in the DBMS. Ensuring such agreement is beyond the scope of this theory; but at least this immediate type annotation permits a simple source-level check, where otherwise it would be necessary to run type inference first.

5.3 Making Queries

To make queries from the source language, we will rewrite source terms to a sublanguage which embeds easily in our SQL subset.

In this section we first examine the sublanguage and its relationship to our SQL fragment, then turn to the rewrite system.

The complete translation from source language to SQL is defined on terms M for which $\mathsf{query}\, M$ is well-typed. We will give a normalizing rewrite relation $\rightsquigarrow$ and a total function $[\![-]\!]$ on its normal forms. To translate a term M, first normalize $M \rightsquigarrow^* V$ using the rewrite system, then apply the $[\![-]\!]$ function to V to get an SQL query $Q = [\![V]\!]$.

SQL-like sublanguage Our chosen sublanguage is as follows:

$$\begin{array}{llcl}
\text{(normal forms)} & V, U & ::= & V \uplus U \mid [] \mid F \\
\text{(joined relations)} & F & ::= & \mathsf{for}\,(x \leftarrow \mathsf{table}\, s : T)\, F \mid Z \\
\text{(filtered relations)} & Z & ::= & \mathsf{if}\, B\, \mathsf{then}\, Z\, \mathsf{else}\, [] \mid [R] \mid \mathsf{table}\, s : T \\
\text{(row forms)} & R & ::= & (\overrightarrow{l = B}) \mid x \\
\text{(basic expressions)} & B & ::= & \mathsf{if}\, B\, \mathsf{then}\, B'\, \mathsf{else}\, B'' \mid \mathsf{empty}(V) \mid \\
 & & & \oplus(\vec{B}) \mid x.l \mid c
\end{array}$$

Observe that the "normal form" expressions ranged by V all have *relation type*: bag of record of base type, or $[(\overrightarrow{l : o})]$.

SQL fragment Our target SQL fragment is as follows:

$$\begin{array}{rcl}
Q,R & ::= & Q \;\mathsf{union\ all}\; R \mid S \\
S & ::= & \mathsf{select}\; \vec{c}\; \mathsf{from}\; \overrightarrow{t\;\mathsf{as}\;x}\; \mathsf{where}\; e \\
t & & \text{(table name)} \\
c & ::= & e\;\mathsf{as}\;l \mid x.* \\
e & ::= & \mathsf{case\,when}\, e \,\mathsf{then}\, e' \,\mathsf{else}\, e'' \,\mathsf{end} \mid \\
 & & c \mid x.l \mid e \wedge e' \mid \neg e \mid \mathsf{exists}(Q) \mid \oplus(\vec{e})
\end{array}$$

This includes all unions of queries on an inner join of zero or more tables, with result and query conditions taken from some given algebra of operations, including field projection, boolean conjunction, negation, the exists operator, and conditionals case...end.

SQL translation Now the type-sensitive function $[\![-]\!]$ (Figure 5.4) translates each closed term in the sublanguage directly into a query. Source phrases in V translate to those in target grammar Q, those in Z and F translate into S, those in R translate into $\overrightarrow{e\,\mathsf{as}\,l}$, and those in B translate into e. In two cases, those for empty lists and for table handles, the translation depends on the type of the source term; this allows listing the right columns in the select clause of the target query.

When we write a from clause with $\varnothing$, as in select $\overrightarrow{e\;\mathsf{as}\;l}$ from $\varnothing$ where B, we indicate the SQL query that omits the from clause.

(A fine point: SQL has no way of selecting an empty set of result columns in a select clause; to translate a singleton bag of a nullary record, [()], we need to offer some dummy value, or *, as the result column. An implementation can supply any such select clause, but should be consistent since unions of queries with different result columns are ill-typed.)

Because we assume that all bound variables in the source program are distinct, there will be no clashes among the table aliases (the identifiers following the as keywords) produced by this translation.

$$\begin{array}{rcl}
\llbracket V \uplus U \rrbracket & = & \llbracket V \rrbracket\ \textsf{union all}\ \llbracket U \rrbracket \\
\llbracket\, \textsf{[]} : \textsf{[}(\overrightarrow{l : T})\textsf{]} \rrbracket & = & \textsf{select}\ \overrightarrow{\textsf{null as}\ l}\ \textsf{from}\ \varnothing\ \textsf{where}\ \textsf{false} \\
\llbracket \textsf{for}\,(x \leftarrow \textsf{table}\, s : T)\, F \rrbracket & = & \textsf{select}\ \overrightarrow{e\ \textsf{as}\ l}\ \textsf{from}\ s\ \textsf{as}\ x, \overrightarrow{t\ \textsf{as}\ y}\ \textsf{where}\ B \\
 & & \quad \text{where } (\textsf{select}\ \overrightarrow{e\ \textsf{as}\ l}\ \textsf{from}\ \overrightarrow{t\ \textsf{as}\ y}\ \textsf{where}\ B) = \llbracket F \rrbracket \\
\llbracket \textsf{if}\ B\ \textsf{then}\ Z\ \textsf{else}\ \textsf{[]} \rrbracket & = & \textsf{select}\ \overrightarrow{e\ \textsf{as}\ l}\ \textsf{from}\ \vec{t}\ \textsf{where}\ B' \wedge \llbracket B \rrbracket \\
 & & \qquad \text{where } (\textsf{select}\ \overrightarrow{e\ \textsf{as}\ l}\ \textsf{from}\ \vec{t}\ \textsf{where}\ B') = \llbracket Z \rrbracket \\
\llbracket \textsf{table}\, s : \textsf{[}(\overrightarrow{l : o})\textsf{]} \rrbracket & = & \textsf{select}\ \overrightarrow{s.l\ \textsf{as}\ l}\ \textsf{from}\ s\ \textsf{where}\ \textsf{true} \\
\llbracket\, \textsf{[}R\textsf{]} \rrbracket & = & \textsf{select}\ \llbracket R \rrbracket\ \textsf{from}\ \varnothing\ \textsf{where}\ \textsf{true} \\
\llbracket (\overrightarrow{l = B}) \rrbracket & = & \overrightarrow{\llbracket B \rrbracket\ \textsf{as}\ l} \\
\llbracket x \rrbracket & = & x.* \\
 & & \\
\llbracket \textsf{if}\ B\ \textsf{then}\ B'\ \textsf{else}\ B'' \rrbracket & = & \textsf{case when}\ \llbracket B \rrbracket\ \textsf{then}\ \llbracket B' \rrbracket\ \textsf{else}\ \llbracket B'' \rrbracket\ \textsf{end} \\
\llbracket \textsf{empty}(V) \rrbracket & = & \neg\textsf{exists}(\llbracket V \rrbracket) \\
\llbracket \oplus(\vec{B}) \rrbracket & = & \oplus_{\mathrm{db}}(\overrightarrow{\llbracket B \rrbracket}) \\
\llbracket x.l \rrbracket & = & x.l \\
\llbracket c \rrbracket & = & c
\end{array}$$

Figure 5.4: Translation from normalized sublanguage to SQL.

Rewrite rules The translation of source terms into the SQL-isomorphic sublanguage is given as a strongly-normalizing rewrite system (Figure 5.5). We write $M[L/x]$ for the substitution of the term L for the free variable x in the term M. The rules are type-sensitive; we assume that every term has an attached type, but we only write the type for the topmost term on the left-hand side of the rules because this is the only one we need to look at to determine applicability of the rule.

The only type-sensitive rules are IF-SPLIT, IF-RECORD, and EMPTY-FLATTEN. IF-SPLIT turns a choice between two bags (an operation with no direct analogue in SQL) into a union of two oppositely-guarded bags. At record type the IF-RECORD rule applies, which turns a choice between two records into a record of choices at each field. The APP-IF rule, although it is not type-sensitive, serves to eliminate conditionals at function type, provided they are applied, by moving the application inside the conditional, giving a conditional at the result type. The EMPTY-FLATTEN rule ensures that the argument to empty has relation type and so can be modeled as an SQL subquery; it effectively discards any row data in the argument, while preserving its length.

These rules may duplicate or eliminate side-effects, but we intend to apply them only to pure terms. We will show later that the rules preserve purity.

Digression: SQL subqueries? Several of the rules may seem unnecessary if we are permitted to use SQL subqueries. For example, why employ the FOR-ASSOC rule if we can write an SQL query that uses a nested SQL select statement in its from clause? After all, we could more directly implement the expression

$$\mathsf{for}\,(y \leftarrow \mathsf{for}\,(x \leftarrow \mathsf{table}\,s : T)\ [(b = x.a)])\ [(c = y.b)]$$

with this SQL:

$$\mathsf{select}\,y.b\ \mathsf{as}\,c\,\mathsf{from}\,(\mathsf{select}\,x.a\ \mathsf{as}\ b\,\mathsf{from}\,s\ \mathsf{as}\ x)\ \mathsf{as}\ y.$$

In this case, the nested comprehension became a nested subquery. So why include FOR-ASSOC?

$$
\begin{array}{rcll}
(\lambda x.N)M : T & \rightsquigarrow & N[M/x] & (\textsc{abs-}\beta) \\
\mathsf{for}\,(x \leftarrow [M])\,N : T & \rightsquigarrow & N[M/x] & (\textsc{for-}\beta) \\
(\overrightarrow{l = M}).l_i : T & \rightsquigarrow & M_i & (\textsc{record-}\beta) \\
\mathsf{for}\,(x \leftarrow [\,])\,M : T & \rightsquigarrow & [\,] & (\textsc{for-zero-src}) \\
\mathsf{for}\,(x \leftarrow N)\,[\,] : T & \rightsquigarrow & [\,] & (\textsc{for-zero-body}) \\
\mathsf{for}\,(x \leftarrow \mathsf{for}\,(y \leftarrow L)\,M)\,N : T & \rightsquigarrow & \mathsf{for}\,(y \leftarrow L)\,(\mathsf{for}\,(x \leftarrow M)\,N) & \text{if } y \notin \textsc{fv}(N) \\
 & & & (\textsc{for-assoc}) \\
\mathsf{for}\,(x \leftarrow M_1 \uplus M_2)\,N : T & \rightsquigarrow & \mathsf{for}\,(x \leftarrow M_1)\,N \uplus \mathsf{for}\,(x \leftarrow M_2)\,N & \\
 & & & (\textsc{for-union-src}) \\
\mathsf{for}\,(x \leftarrow M)\,(N_1 \uplus N_2) : T & \rightsquigarrow & \mathsf{for}\,(x \leftarrow M)\,N_1 \uplus \mathsf{for}\,(x \leftarrow M)\,N_2 & \\
 & & & (\textsc{for-union-body}) \\
\mathsf{for}\,(x \leftarrow \mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[\,])\,N : T & \rightsquigarrow & \mathsf{if}\,B\,\mathsf{then}\,(\mathsf{for}\,(x \leftarrow M)\,N)\,\mathsf{else}\,[\,] & \\
 & & & (\textsc{for-if-src}) \\
(\mathsf{if}\,B\,\mathsf{then}\,L\,\mathsf{else}\,L')M : T & \rightsquigarrow & \mathsf{if}\,B\,\mathsf{then}\,LM\,\mathsf{else}\,L'M & (\textsc{app-if}) \\
\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N : (\overrightarrow{l : T}) & \rightsquigarrow & (\overrightarrow{l = L}) & (\textsc{if-record}) \\
 & & \multicolumn{2}{l}{\text{where } L_l = \mathsf{if}\,B\,\mathsf{then}\,M.l\,\mathsf{else}\,N.l \text{ for each } l \in \vec{l}} \\
\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N : [T] & \rightsquigarrow & \mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[\,] & \text{if } N \neq [\,] \\
 & & \uplus\,\mathsf{if}\,\neg B\,\mathsf{then}\,N\,\mathsf{else}\,[\,] & (\textsc{if-split}) \\
\mathsf{if}\,B\,\mathsf{then}\,[\,]\,\mathsf{else}\,[\,] : T & \rightsquigarrow & [\,] & (\textsc{if-zero}) \\
\mathsf{if}\,B\,\mathsf{then}\,(\mathsf{for}\,(x \leftarrow M)\,N)\,\mathsf{else}\,[\,] : T & \rightsquigarrow & \mathsf{for}\,(x \leftarrow M)\,(\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,]) & (\textsc{if-for}) \\
\mathsf{if}\,B\,\mathsf{then}\,M \uplus N\,\mathsf{else}\,[\,] : T & \rightsquigarrow & \mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[\,] & (\textsc{if-union}) \\
 & & \uplus\,\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,] & \\
\mathsf{empty}(M) : T & \rightsquigarrow & \mathsf{empty}(\mathsf{for}\,(x \leftarrow M)\,[()]) & \\
 & & & \text{if } M \text{ is not relation-typed} \\
 & & & (\textsc{empty-flatten}) \\
\mathsf{query}\,M : T & \rightsquigarrow & M & (\textsc{ignore-db})
\end{array}
$$

Figure 5.5: The rewrite system for normalizing source-language terms.

The answer is that such rules are critical to the unnesting of intermediate data structures. Consider this query which creates an intermediate result of nested bag-of-bag type, not an SQL-representable type:

$$\mathsf{for}\,(y \leftarrow \mathsf{for}\,(x \leftarrow \mathsf{table}\,s)\;[\,[x]\,])\;y$$

The expression rewrites using the FOR-ASSOC rule:

$$\begin{array}{rll}
\mathsf{for}\,(y \leftarrow (\mathsf{for}\,(x \leftarrow \mathsf{table}\,s)\;[\,[x]\,])\;y & \rightsquigarrow & (\text{FOR-ASSOC}) \\
\mathsf{for}\,(x \leftarrow \mathsf{table}\,s)(\mathsf{for}\,(y \leftarrow\; [\,[x]\,])\;y) & \rightsquigarrow & (\beta\text{-FOR}) \\
\mathsf{for}\,(x \leftarrow \mathsf{table}\,s)\;[x] & &
\end{array}$$

and now the expression is unnested. The FOR-ASSOC rule thus exposes β reductions which themselves eliminate constructor/destructor pairs and hence reduce the types of intermediate values.

5.4 Correctness

Soundness

The system is useless if rewriting changes the behavior of terms. In particular, it would be suspect if rewriting, which is intended to be applied to pure terms, produced terms with side-effects. This section shows that reduction preserves types and purity.

Normally, in an effect system with an operational semantics, the reductions in that semantics preserve types and effects. But since, for this rewrite system, the rewriting strategy is not specified (rewrites can be made anywhere in a term), types and effects are not always preserved. Indeed, the rules are only sound for pure terms, but this is all we plan to use them for—so we'll show that pure terms have their purity preserved by reduction.

(The problem is that substitution can modify the effects captured by a function type. For example suppose M has typing $\vdash M : S \,!\, e$ with e a non-empty effect set. We can rewrite $(\lambda x.\lambda y.x)M \rightsquigarrow \lambda y.M$. The redex types as $\vdash (\lambda x.\lambda y.x)M : T \xrightarrow{\varnothing}$

$S \,!\, e$ while the reduct types as $\vdash \lambda y.M : T \xrightarrow{e} S \,!\, \varnothing$: the reduction relocates the effect, disturbing the type.)

Define a function *push* which pushes effects into every arrow in a type:

$$\begin{array}{rcl}
push\ e'\ (S \xrightarrow{e} T) & = & push\ e'\ S \xrightarrow{e \cup e'} push\ e'\ T \\
push\ e'\ [T] & = & [push\ e'\ T] \\
push\ e'\ (\overrightarrow{l : T}) & = & (\overrightarrow{l : (push\ e'\ T)}) \\
push\ e'\ o & = & o
\end{array}$$

And a function *complete* which pushes effects into every arrow in a type and propagates effects in the type downwards:

$$\begin{array}{rcl}
complete\ e'\ (S \xrightarrow{e} T) & = & complete\ (e \cup e')\ S \xrightarrow{e \cup e'} complete\ (e \cup e')\ T \\
complete\ e'\ [T] & = & [complete\ e'\ T] \\
complete\ e'\ (\overrightarrow{l : T}) & = & (\overrightarrow{l : (complete\ e'\ T)}) \\
complete\ e'\ o & = & o
\end{array}$$

Define lifted versions of both of these functions for typing contexts by applying them to each type in the context.

Now, a typing can be weakened to give the same term a typing with all the types pushed or completed by any effect.

Lemma 13. If $\Gamma \vdash M : T \,!\, e$ then $\Gamma' \vdash M : T' \,!\, e \cup e'$ where $\Gamma' = complete\ e'\ \Gamma$ and $T' = complete\ e'\ T$.

Proof. By induction on the derivation $\Gamma \vdash M : T \,!\, e$. Showing the most interesting cases:

CASE $\Gamma \vdash x : T \,!\, \varnothing$. Then $x : T \in \Gamma$ and then $x : T' \in \Gamma'$ where $T' = complete\ e'\ T$. So $\Gamma' \vdash x : T' \,!\, \varnothing$. And T-EFF-WEAKENING gives $\Gamma' \vdash x : T' \,!\, \varnothing \cup e'$ as needed.

CASE $\Gamma \vdash LM : T \,!\, e$. Then $\Gamma \vdash L : S \xrightarrow{e_f} T \,!\, e_L$ and $\Gamma \vdash M : S \,!\, e_M$ with $e = e_f \cup e_L \cup e_M$.

By IH, $\Gamma' \vdash L : S' \xrightarrow{e'_f} T' \,!\, e_L \cup e'$ and $\Gamma' \vdash M : S' \,!\, e_M \cup e'$ where $\Gamma' = \mathit{complete}\; e'\; \Gamma$, $e'_f = e_f \cup e'$, $S' = \mathit{complete}\,(e' \cup e_f)\, S$ and $T' = \mathit{complete}\,(e' \cup e_f)\, T$.

And so by T-APP, $\Gamma' \vdash LM : T' \,!\, e_L \cup e_M \cup e_f \cup e'$ as needed.

CASE $\Gamma \vdash \lambda x.N : T \,!\, \varnothing$. Let $T = T_1 \overset{e_N}{\to} T_2$.

So $\Gamma', x : T_1 \vdash N : T_2 \,!\, e_N$. By IH, $\Gamma', x : T'_1 \vdash N : T'_2 \,!\, e_N \cup e'$ where $\Gamma' = \mathit{complete}\; e'\; \Gamma$, $T'_1 = \mathit{complete}\,(e_N \cup e')\, T_1$, $T'_2 = \mathit{complete}\,(e_N \cup e')\, T_2$.

By T-ABS, $\Gamma' \vdash \lambda x.N : T'_1 \overset{e_N \cup e'}{\to} T'_2 \,!\, \varnothing$.

And by T-EFF-WEAKENING, $\Gamma' \vdash \lambda x.N : T'_1 \overset{e_N \cup e'}{\to} T'_2 \,!\, e_N \cup e'$.

Also $T'_1 \overset{e_N \cup e'}{\to} T'_2 = \mathit{complete}\; e'\; (T_1 \overset{e_N}{\to} T_2)$. □

Lemma 14. If $\Gamma \vdash M : T \,!\, e$ then $\Gamma' \vdash M : T' \,!\, e \cup e'$ where $\Gamma' = \mathit{push}\; e'\; \Gamma$ and $T' = \mathit{push}\; e'\; T$.

Proof omitted. (Similar to Lemma 13)

Substitution of one effectful term into another gives us a term which might have the former's effect on any of its arrows; conservatively, we can derive a type which contains that effect on *any* arrow—the effect given by the *push* function.

Lemma 15. From $\Gamma, x : S \vdash N : T \,!\, e_N$
and $\Gamma \vdash P : S \,!\, e_P$
we get $\Gamma' \vdash N[P/x] : T' \,!\, e_N \cup e_P$
where $\Gamma' = \mathit{push}\; e_P\; \Gamma$ and $T' = \mathit{push}\; e_P\; T$.

Proof. By induction on the derivation of $\Gamma, x : S \vdash N : T \,!\, e_N$.

CASE T-VAR.

Case $N = x$. The typing is $\Gamma, x : S \vdash x : T \,!\, \varnothing$ with $e_N = \varnothing$. Here $S = T$. So we have $\Gamma \vdash P : T \,!\, e_P$. By Lemma 14, $\Gamma' \vdash P : T' \,!\, e_P$ where $\Gamma' \vdash \mathit{push}\; e_P\; \Gamma$ and $T' = \mathit{push}\; e_P\; T$ as needed.

Case $N = y$. The typing is $\Gamma, x : S \vdash y : T \,!\, \varnothing$ with $e_N = \varnothing$. Now $N[P/x] = y$ and so the typing of P becomes $\Gamma \vdash y : T \,!\, e_P$. By Lemma 14, $\Gamma' \vdash y : T' \,!\, e_P$.

CASE T-APP.

The typing is

$$\frac{\Gamma, x : S \vdash L : U \xrightarrow{e_f} T \,!\, e_L \qquad \Gamma, x : S \vdash M : U \,!\, e_M}{\Gamma, x : S \vdash LM : T \,!\, e_N}$$

with $e_N = e_f \cup e_L \cup e_M$.

By IH,

$$\Gamma' \vdash L[P/x] : U' \xrightarrow{e'_f} T' \,!\, e'_L \qquad \text{and} \qquad \Gamma' \vdash M[P/x] : U' \,!\, e'_M$$

with $U' = \textit{push}\ e_P\ U$, $T' = \textit{push}\ e_P\ T$, $e'_f = e_f \cup e_P$, $e'_L = e_L \cup e_P$, $e'_M = e_M \cup e_P$. Now by T-APP, $\Gamma' \vdash (LM)[P/x] : T' \,!\, e'_f \cup e'_L \cup e'_M = e_P \cup e_N$.

CASE T-ABS. If the term is $\lambda x.N'$ then the substitution is a no-op and the result follows from Lemma 14 and T-EFF-WEAKENING. So assume it is $\lambda y.N'$ with $y \neq x$.

The typing is

$$\frac{\Gamma, x : S, y : T_1 \vdash N' : T_2 \,!\, e_{N'}}{\Gamma, x : S \vdash \lambda y.N' : T_1 \xrightarrow{e_{N'}} T_2 \,!\, \varnothing}$$

with $T = T_1 \xrightarrow{e_{N'}} T_2$.

By IH, $\Gamma', y : T'_1 \vdash N[P/x] : T'_2 \,!\, e'_{N'} \cup e_P$ with $T'_2 = \textit{push}\ e_P\ T_2$ and $(\Gamma', y : T'_1) = \textit{push}\ e_P\ (\Gamma, y : T_1)$ so $T'_1 = \textit{push}\ e_P\ T_1$.

Then by T-ABS, $\Gamma' \vdash \lambda y.N' : T'_1 \xrightarrow{e_{N'} \cup e_P} T'_2 \,!\, \varnothing$ with $T'_1 \xrightarrow{e_{N'}} T'_2 = \textit{push}\ e_P\ (T_1 \xrightarrow{e_{N'}} T_2)$ and by T-EFF-WEAKENING, $\Gamma' \vdash \lambda y.N' : T'_1 \xrightarrow{e_{N'} \cup e_P} T'_2 \,!\, e_P$ as needed.

CASE T-EFF-WEAKENING.

The typing is

$$\frac{\Gamma, x : S \vdash N : T \,!\, e_1}{\Gamma, x : S \vdash N : T \,!\, e_1 \cup e_2}$$

By IH, $\Gamma' \vdash M : T' \,!\, e_1 \cup e_P$ with $\Gamma' = \textit{push}\ e_P\ \Gamma$ and $T' = \textit{push}\ e_P\ T$. Then by T-EFF-WEAKENING, $\Gamma' \vdash M : T' \,!\, e_1 \cup e_2 \cup e_P$ as needed. □

Completion commutes with union in the effect argument.

Lemma 16. For any e, e' and T: $\mathit{complete}\ e\ (\mathit{complete}\ e'\ T) = \mathit{complete}\ (e \cup e')\ T$ and $\mathit{push}\ e\ (\mathit{push}\ e'\ T) = \mathit{push}\ (e \cup e')\ T$.

Proof. By induction on the type. □

Completion by an effect subsumes pushing by the same or a lesser effect.

Lemma 17. For any $e \supseteq e'$ and T, $\mathit{complete}\ e\ (\mathit{push}\ e'\ T) = \mathit{complete}\ e\ T$.

Proof. By induction on the type. Take the case $S \xrightarrow{e''} T$.

$$\begin{array}{ll}
 & \mathit{complete}\ e\ (\mathit{push}\ e'\ (S \xrightarrow{e''} T)) \\
= & \mathit{complete}\ (e \cup e' \cup e'')\ (\mathit{push}\ e'\ S) \xrightarrow{e \cup e' \cup e''} \mathit{complete}\ (e \cup e' \cup e'')\ (\mathit{push}\ e'\ T) \\
= & \mathit{complete}\ (e \cup e'')\ (\mathit{push}\ e'\ S) \xrightarrow{e \cup e''} \mathit{complete}\ (e \cup e'')\ (\mathit{push}\ e'\ T) \\
= & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by IH)} \\
 & \mathit{complete}\ (e \cup e'')\ S \xrightarrow{e \cup e''} \mathit{complete}\ (e \cup e'')\ T \\
= & \mathit{complete}\ e\ (S \xrightarrow{e''} T)
\end{array}$$

The remaining cases are trivial. □

And finally, any reduction produces a term of the same type, completed by the effect of the term itself.

Lemma 18. If $\Gamma \vdash M : T\ !\ e$ and $M \rightsquigarrow M'$ then $\Gamma' \vdash M' : T'\ !\ e$ with $\Gamma' = \mathit{complete}\ e\ \Gamma$ and $T' = \mathit{complete}\ e\ T$.

Proof. By induction on M. Take cases on the reduction. We examine only the most interesting cases.

CASE $LM \rightsquigarrow LM'$.

The typing is

$$\frac{\Gamma \vdash L : S \xrightarrow{e_f} T\ !\ e_L \qquad \Gamma \vdash M : S\ !\ e_M}{\Gamma \vdash LM : T\ !\ e = e_f \cup e_L \cup e_M}$$

By IH, $\Gamma_M \vdash M' : S'\ !\ e_M$ where $\Gamma_M = \mathit{complete}\ e_M\ \Gamma$ and $S' = \mathit{complete}\ e_M\ S$.

Using Lemmas 13 and 16, we can generalize this to $\Gamma' \vdash M' : S'' \,!\, e$ where $\Gamma' = \mathit{complete}\ e\ \Gamma$ and $S'' = \mathit{complete}\ e\ S$.

Similarly, $\Gamma' \vdash L : S'' \xrightarrow{e_f \cup e} T'' \,!\, e_L$ where $T'' = \mathit{complete}\ e\ T$.

And so by T-App, $\Gamma' \vdash LM' : T'' \,!\, e$.

CASE $LM \rightsquigarrow L'M$.

The typing is

$$\frac{\Gamma \vdash L : S \xrightarrow{e_f} T \,!\, e_L \qquad \Gamma \vdash M : S \,!\, e_M}{\Gamma \vdash LM : T \,!\, e = e_f \cup e_L \cup e_M}$$

By IH, $\Gamma_L \vdash L' : S' \xrightarrow{e'_f} T' \,!\, e_L$ where $\Gamma_L = \mathit{complete}\ e_L\ \Gamma$ and $S' \xrightarrow{e'_f} T' = \mathit{complete}\ e_L\ (S \xrightarrow{e_f} T)$ so $e'_f = e_f \cup e_L$, $S' = \mathit{complete}\ (e_f \cup e_L)\ S$ and $T' = \mathit{complete}\ (e_f \cup e_L)\ T$.

Using Lemmas 13 and 16, we can generalize this to $\Gamma' \vdash L' : S'' \xrightarrow{e''_f} T'' \,!\, e$ where $\Gamma' \vdash \mathit{complete}\ e\ \Gamma$, $S'' = \mathit{complete}\ e\ S$, $T'' = \mathit{complete}\ e\ T$ and $e''_f = e \cup e_f = e$

Similarly, $\Gamma' \vdash M : S'' \,!\, e_M$

And so by T-App, $\Gamma' \vdash L'M : T'' \,!\, e''_f \cup e \cup e_M = e$.

CASE ABS-β. $(\lambda x.N)M \rightsquigarrow N[M/x]$. The typing is

$$\frac{\dfrac{\Gamma, x : S \vdash N : T \,!\, e_N}{\Gamma \vdash \lambda x.N : S \xrightarrow{e_N} T \,!\, \varnothing} \qquad \Gamma \vdash M : S \,!\, e_M}{\Gamma \vdash (\lambda x.N)M : T \,!\, e_N \cup e_M}$$

with $e = e_N \cup e_M$.

From this by Lemma 15 we get $\Gamma' \vdash N[M/x] : T' \,!\, e_N \cup e_M$ where $\Gamma' = \mathit{push}\ e_M\ \Gamma$ and $T' = \mathit{push}\ e_M\ T$.

We can also generalize this, using Lemmas 13, 16 and 17, to $\Gamma'' \vdash N[M/x] : T'' \,!\, e_N \cup e_M$ where $\Gamma'' = \mathit{complete}\ e\ \Gamma$ and $T'' = \mathit{complete}\ e\ T$.

CASE $\lambda x.N \rightsquigarrow \lambda x.N'$. The typing is $\Gamma \vdash \lambda x.N : S \xrightarrow{e} T \,!\, \varnothing$ from $\Gamma, x : S \vdash N : T \,!\, e$.

By IH, $\Gamma', x : S' \vdash N' : T' \mathbin{!} e$ with $\Gamma' = \mathit{complete}\ e\ \Gamma$, $S' = \mathit{complete}\ e\ S$ and $T' = \mathit{complete}\ e\ T$.

And so by T-ABS, $\Gamma' \vdash \lambda x.N : S' \xrightarrow{e} T' \mathbin{!} \varnothing$, as needed, noting $S' \xrightarrow{e} T' = \mathit{complete}\ e\ S \xrightarrow{e} T$. □

The goal is to rewrite closed, pure terms. The next lemma shows that all such terms have their type and effect exactly preserved by the rewrite system:

Lemma 19. If $M \rightsquigarrow M'$ and $\varnothing \vdash M : T \mathbin{!} \varnothing$ and T is a relation type then $\varnothing \vdash M' : T \mathbin{!} \varnothing$. Moreover $M \rightsquigarrow^* V$ gives then $\varnothing \vdash V : T \mathbin{!} \varnothing$.

Proof. A direct consequence of Lemma 18, since relation types contain no arrows and so $T = \mathit{complete}\ \varnothing\ T$. □

Totality

For the rewriting to be effective, it must in fact normalize every queryizable term to a normal form in the domain of the function $[\![-]\!]$. This section shows that the system strongly normalizes: every reduction sequence terminates.

The proof follows the plan of the argument used by Lindley and Stark [2005], which builds on the "reducibility" method of Tait [1967]. This proof uses some new tricks, for example to handle the consequences of the fact that some of these rules duplicate subterms.

Like the usual proof, this one uses "continuations" as a way of bookkeeping the progress made by certain rules, the so-called "commuting conversion" rules, which interchange two forms without eliminating any. A continuation can be thought of as a stack of contexts which need to be eliminated to normalize the term, along with any contexts that commute with those.

Definition (Continuations)**.** Define a set of *continuations* as follows:

$$\begin{aligned} K &::= Id \mid K \circ F \\ F &::= (x)N \mid (\mathsf{where}\, B) \mid (M \uplus) \end{aligned}$$

The notation $K@M$ denotes filling K with the term M:

$$\begin{array}{rcl}
Id@M & = & M \\
(K \circ (x)N)@M & = & K@(\mathsf{for}\,(x \leftarrow M)\,N) \\
(K \circ (\mathsf{where}\,B))@M & = & K@(\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[]) \\
(K \circ (M \uplus))@M' & = & K@(M \uplus M')
\end{array}$$

Definition. The length $|K|$ of a continuation K is the number of for-abstraction frames that it contains:

$$\begin{array}{rcl}
|Id| & = & 0 \\
|K \circ (x)N| & = & |K| + 1 \\
|K \circ (\mathsf{where}\,B)| & = & |K| \\
|K \circ (M \uplus)| & = & |K|
\end{array}$$

We only count the for-abstraction frames because only they represent contexts that need to be eliminated. The others are neutral contexts—they don't react with what's placed inside them. In fact, the other frames are only present to ensure that the continuations are closed under reduction, as we will see in a moment.

The proof works by showing that terms are *reducible*, a property stronger than strong normalization, but more amenable to induction. Intuitively, a term is reducible if it strongly normalizes in every well-typed context. The reducibility predicate is indexed by the type, and at each type it is defined in terms of smaller types. For bag types $[T]$, reducibility is defined in terms of continuations, because we need to track the commuting conversions; for record and function types, there are no commuting conversions to worry about so we opt for a simpler (non-inductive) definition. Reducible base-type terms are just the strongly-normalizing ones.

Definition (Reducibility)**.** For each type T, define a set red_T on closed terms of type T by the following rules:

- $M \in \mathsf{red}_o$ (base type) iff M strongly normalizes,
- $L \in \mathsf{red}_{S \to T}$ iff for every $M \in \mathsf{red}_S$ we have that $LM \in \mathsf{red}_T$.
- $M \in \mathsf{red}_{(\overrightarrow{l:T})}$ iff for each $(l : T) \in (\overrightarrow{l : T})$ we have that $M.l \in \mathsf{red}_T$
- $K \in \mathsf{redK}_{[T]}$ iff for every $M : T$ with $M \in \mathsf{red}_T$ we have that $K@[M]$ strongly normalizes.
- $M \in \mathsf{red}_{[T]}$ iff for every $K \in \mathsf{redK}_{[T]}$, $K@M$ strongly normalizes.

We will need some general properties of reducibility.

Lemma 20. There is some reducible term of every type.

Proof. A straightforward induction on the type constructs the term; we begin with a constant at base type; at type $S \to T$ we abstract a reducible N of type T with a dummy variable x to get $\lambda x.N$. For bags and records we can construct the term directly with the singleton and record constructors respectively. □

Next, reducibility implies strong normalization (the abbreviation "M s.n." is short for "M strongly normalizes"):

Lemma 21. If $M \in \mathsf{red}_T$ then M s.n.

Proof. By induction on T, the type of M:

CASE o. Immediate from the definition of reducibility.

CASE $S \to T$. By the definition of reducibility, $MM' \in \mathsf{red}_T$ for each $M' \in \mathsf{red}_S$, and then by the induction hypothesis MM' strongly normalizes. And so its subterm M strongly normalizes.

CASE $(\overrightarrow{l : T})$. By definition, $M.l$ is reducible for each $l \in \vec{l}$. Therefore by the IH, $M.l$ strongly normalizes, and so its subterm M strongly normalizes.

CASE $[T]$. By the definition of reducibility, $Id@M = M$ strongly normalizes. □

Lemma 22. If $M \in \mathsf{red}_T$ and $M \rightsquigarrow N$ then $N \in \mathsf{red}_T$.

Proof. By induction on T, the type of M:

CASE o. If M s.n. then so does its reduct N; this suffices.

CASE $S \to T$. For each $M' \in \mathsf{red}_S$, we have that $MM' \in \mathsf{red}_T$ and $MM' \leadsto NM'$; then by the inductive hypothesis, $NM' \in \mathsf{red}_T$. Since this is true for any $M' \in \mathsf{red}_S$, this satisfies the definition of $N \in \mathsf{red}_{S \to T}$.

CASE $(\overrightarrow{l : T})$. For each $(l : T) \in (\overrightarrow{l : T})$, we have that $M.l \in \mathsf{red}_T$ and $M.l \leadsto N.l$; then by the inductive hypothesis, $N.l \in \mathsf{red}_T$. Since this is true for each $l \in \vec{l}$, this satisfies the definition of $N \in \mathsf{red}_{(\overrightarrow{l:T})}$.

CASE $[T]$. For any $K \in \mathsf{red}_{[T]}$, we have that $K@M$ strongly normalizes; as a reduct thereof, $K@N$ strongly normalizes. Since this is true for any $K \in \mathsf{redK}_{[T]}$, this statisfies the definition of $N \in \mathsf{red}_{[T]}$. □

Definition (Term contexts)**.** A *term context* is a term with zero or more holes. A *hole* $[\ \ ]$ can stand in a context anywhere a term otherwise could. The notation $C[M]$ denotes the term formed by plugging M into *every* hole in C.

Please take note of the distinction between context brackets ($[\ \ ]$) and empty-list brackets ($[]$).

Observation. Continuations K are a subset of contexts C. The continuation plugging operation $K@M$ is a special case of context plugging $C[M]$.

Now, we will need to speak of reductions taking place "within the continuation part" of a plugged pair like $K@M$, so we will develop a notion of reduction for continuations. First we define reduction of contexts.

Definition (Context reduction)**.** If $C[M] \leadsto C'[M]$ for all M then we write $C \leadsto C'$. Then given a reduction $C[M] \leadsto C'[M]$ for some M, we can say the reduction is *within C*.

Definition (Reduction-in-context)**.** If we have $M \leadsto M'$ then we say that the reduction $C[M] \leadsto C[M']$ is *within M*.

Definition (Reduction at the interface)**.** If $C[M]$ reduces and the reduction is not within C and not within M then it is *at the interface* between C and M.

Note that reductions at the interface could actually alter the content of M. For example, if $C = (\lambda x.C')N$ then $C[M] \rightsquigarrow C'[N/x][M[N/x]]$. This reduction is at the interface but can involve changes in both C' and M.

Definition (Neutral term-context pairs)**.** A term M is *neutral for the context* C if the only reductions applicable to $C[M]$ are those that are applicable within C or within M. Otherwise it is *active for the context* C.

Since continuations are contexts, we can speak of a reduction "within K." However, reduction on continuations (defined by reduction of contexts) is not closed: given a continuation $K = C$, we may have $C \rightsquigarrow C'$ where C' is not a continuation.

This infelicity is palliated by Lemma 23 which, for any term $K@P$ with $K = C \rightsquigarrow C'$, produces a new continuation K' that we can use in place of C', because $C'[P] = K'@P$. The continuation K' will depend, however, on P, so it is not a general replacement for C'.

Lemma 23. When $K \rightsquigarrow C'$, then for each term P there exists K' such that $C'[P] = K'@P$ and $|K| \geq |K'|$.

Proof. Take cases on the reduction $K \rightsquigarrow C'$.

CASE $K = K' \circ (x)(M \uplus N) \rightsquigarrow$ (FOR-UNION-BODY)
$K'@((\mathsf{for}\,(x \leftarrow [\ \ ])\,M) \uplus (\mathsf{for}\,(x \leftarrow [\ \ ])\,N))$.

Then

$$\begin{aligned}&(K'@((\mathsf{for}\,(x \leftarrow P)\,M) \uplus (\mathsf{for}\,(x \leftarrow P)\,N)))\\ &= (K' \circ ((\mathsf{for}\,(x \leftarrow P)\,M)\uplus) \circ (x)N)@P.\end{aligned}$$

And $|K' \circ (x)(M \uplus N)| = |K' \circ ((\mathsf{for}\,(x \leftarrow P)\,M)\uplus) \circ (x)N|$ as needed.

CASE $K = K' \circ (x)N \circ (y)N' \rightsquigarrow K' \circ (y)(\mathsf{for}\,(x \leftarrow N')\,N)$ (FOR-ASSOC)

The reduct is already a continuation and

$$|K' \circ (x)N \circ (y)N'| \geq |K' \circ (y)(\mathsf{for}\,(x \leftarrow N')\,N)|.$$

CASE $K = K' \circ (x)N \circ (M \uplus) \rightsquigarrow K' \circ ((\mathsf{for}\,(x \leftarrow M)\,N)\uplus) \circ (x)N$ (FOR-UNION-SRC)

The reduct $K' \circ ((\mathsf{for}\,(x \leftarrow M)\,N)\uplus) \circ (x)N$ is already a continuation and the length is unchanged.

CASE $K = K' \circ (\mathsf{where}\,B) \circ (x)N \rightsquigarrow K' \circ (x)(\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,])$. (IF-FOR)

The reduct is already a continuation and the length is unchanged.

CASE $K = K' \circ (x)N \circ (\mathsf{where}\,B) \rightsquigarrow K' \circ (\mathsf{where}\,B) \circ (x)N$. (FOR-IF-SRC)

The reduct is already a continuation and the length is unchanged.

CASE $K = K' \circ (\mathsf{where}\,B) \circ (M \uplus) \rightsquigarrow$ (IF-UNION)
$K' \circ ((\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[\,])\uplus) \circ (\mathsf{where}\,B)$

The reduct is already a continuation and the length is unchanged.

Other reductions are of the form $K = (K' \circ F) \rightsquigarrow C''[F]$. Then by the IH there is a K'' with $C''[F[M]] = K''@F[M] = (K'' \circ F)@M$ and $|K'| \geq |K''|$. Thence $|K' \circ F| \geq |K'' \circ F|$ as needed. □

Now when we say that a reduction of $K@M$ is "within K," we mean that $K \rightsquigarrow C$ and that $C[M] = K'@M$ for some K'. The fact that $|K| \geq |K'|$ in this case means that we can use $|K|$ as part of an induction metric without fear that it will increase during reduction.

Continuing on, the definition of *neutrality for a given context* specializes to *neutrality for a given continuation*.

Definition (Neutral term-continuation pairs)**.** A term M is *neutral for the continuation K* if the only reductions applicable to $K@M$ are those within K (i.e. $K \rightsquigarrow C'$ for some C') or within M.

More generally, we want to recognize terms that are simply *neutral* toward continuations:

Definition (Neutral terms)**.** A term is *neutral* if it is neutral for all continuations.

Now then, we will use a few general properties of reduction, as follows:

Definition. Write *maxred*(M) for the maximum length of any reduction sequence from M, defined only for strongly normalizing M.

Lemma 24. If M strongly normalizes and $M \rightsquigarrow N$, then *maxred*(M) > *maxred*(N).

Proof. If the reduction is along a maximal path, then the reduct's maximal reduction sequence is shorter by 1. If it is along some path strictly shorter than a maximal one, then the reduct's maximal reduction sequence is also strictly shorter. □

Lemma 25. If M is neutral and each reduction $M \rightsquigarrow N$ has $N \in \mathsf{red}_T$ then $M \in \mathsf{red}_T$.

Proof. By induction on the type T:

CASE o. Every reduct of M s.n. so M s.n. and $M \in \mathsf{red}_o$ by def.

CASE $S \to T$. To show: that $MM' \in \mathsf{red}_T$ for each $M' \in \mathsf{red}_S$. Examine reductions of MM'. Because M is neutral, every reduction is within M or within M'. Every reduct of M' is reducible by Lemma 22. Thus every reduct of MM' is an application of a reducible term at $S \to T$ to a reducible term at S and is reducible. Because applications are neutral, the IH applies and $MM' \in \mathsf{red}_T$. This satisfies the definition of $M \in \mathsf{red}_{S \to T}$.

CASE $(\overrightarrow{l:T})$. To show: that $M.l \in \mathsf{red}_T$ for each $l : T \in \overrightarrow{l:T}$. Examine reductions of $M.l$. Because M is neutral, every reduction is within M. Thus every reduct of $M.l$ is a projection of a reducible term and hence is reducible. Because projections themselves are neutral, the IH applies and $M.l \in \mathsf{red}_T$. This satisfies the definition of $M \in \mathsf{red}_{(\overrightarrow{l:T})}$.

CASE $[T]$. Given $K \in \mathsf{redK}_{[T]}$ we want to show that $K@M$ s.n. By induction on *maxred*(K). Consider a reduction of $K@M$. If it is within M then $M \rightsquigarrow N$. By hypothesis $N \in \mathsf{red}_{[T]}$ so $K@N$ strongly normalizes. Since M is neutral, there are no reductions at the interface. And since K is reducible, each of its reducts, say K', is reducible, and so by the inner IH $K'@M$ strongly normalizes. Thus all reducts of $K@M$ s.n. and hence $M \in \mathsf{red}_{[T]}$. □

Definition. Write *size*(M) for the size of the term M, that is, one plus the sum of the sizes of the immediate subterms.

— ★ —

Now comes the heavy lifting: the next lemmas show for each syntactic form that it is strongly-normalizing (when its subterms are) or that it is reducible (when its subterms are).

Lemma 26. If $K@M$ strongly normalizes then $K@[\,]$ strongly normalizes.

Proof. By induction on $(|K|, \mathit{maxred}(K@M))$. Take cases on the reducts of $K@[\,]$:

CASE $(K' \circ (x)N)@[\,] \rightsquigarrow K'@[\,]$, in the case where $K = K' \circ (x)N$. This strongly normalizes, by the IH, noting $|K| > |K'|$.

CASE $(K' \circ (\mathsf{where}\, B))@[\,] \rightsquigarrow K'@[\,]$, in the case where $K = K' \circ (\mathsf{where}\, B)$. This strongly normalizes, by the IH, noting $|K| > |K'|$.

All other reductions are within K, and so reduce *maxred*$(K@M)$, and thus each strongly normalizes, by the IH. □

Lemma 27. If $M \in \mathsf{red}_T$ then $[M] \in \mathsf{red}_{[T]}$.

Proof. Given $K \in \mathsf{redK}_{[T]}$, we want to show that $K@[M]$ s.n. This follows directly from the definition of $\mathsf{redK}_{[T]}$ and the hypothesis $M \in \mathsf{red}_T$. □

Lemma 28. If $N[P/x] \in \mathsf{red}_T$ for every $P \in \mathsf{red}_S$ then $\lambda x.N \in \mathsf{red}_{S \to T}$.

Proof. Given $P \in \mathsf{red}_S$, we want to show that the application $(\lambda x.N)P \in \mathsf{red}_T$. Since applications are neutral, it suffices (by Lemma 25) to show that all the application's reducts are in red_T. This follows by induction on $\mathit{maxred}(N) + \mathit{maxred}(P)$. Take cases on the reducts:

CASE $(\lambda x.N)P \leadsto N[P/x] \in \mathsf{red}_T$ by hypothesis.

All other reductions are within N or P, preserving the respective reducibility property, by Lemma 22, and these reductions decrease the induction metric. □

Lemma 29. Given $(\overrightarrow{l = M}) : (\overrightarrow{l : T})$, if $M_l \in \mathsf{red}_{T_l}$ for each $l \in \vec{l}$ then $(\overrightarrow{l = M}) \in \mathsf{red}_{(\overrightarrow{l:T})}$.

Proof. By induction on $\sum_{M \in \vec{M}} \mathit{maxred}(M)$. We show that for any field $l \in \vec{l}$ with type T_l, the projection $(\overrightarrow{l = M}).l \in \mathsf{red}_{T_l}$. Since projections are neutral, it suffices to show that all the reducts are in red_{T_l}. The important case is $(\overrightarrow{l = M}).l \leadsto M_l$ (by RECORD-β) which by hypothesis is in red_{T_l}. Any other reductions are within one of the $\vec{M}$ and so the induction hypothesis applies. □

Lemma 30. Given a continuation K and terms M and N, if $K@M$ and $K@N$ strongly normalize then $K@(M \uplus N)$ strongly normalizes.

Proof. By lexicographic induction on $(|K|, \mathit{maxred}(K@M) + \mathit{maxred}(K@N))$. For the base case, when $K = \mathit{Id}$, the strong normalization of $M \uplus N$ follows directly from that of M and N, since no reduction applies to a union at the top level. Now take cases on the reducts of $K@(M \uplus N)$:

CASE $K@(M \uplus N) \leadsto$ (FOR-UNION-SRC)
$K'@((\mathsf{for}\,(x \leftarrow M)\,L) \uplus (\mathsf{for}\,(x \leftarrow N)\,L))$, when $K = K' \circ (x)L$.

The application $K'@(\mathsf{for}(x \leftarrow M)L) = K@M$ strongly normalizes by hypothesis and likewise does $K@N$. By IH, then, $K'@((\mathsf{for}(x \leftarrow M)L) \uplus (\mathsf{for}(x \leftarrow N)L))$ strongly normalizes. (IH applies because $|K| > |K'|$.)

CASE $K@(M \uplus N) \leadsto$ (IF-UNION)
$K'@((\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[\,]) \uplus (\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,]))$, when $K = K' \circ (\mathsf{where}\,B)$.

The same reasoning applies as in the first case.

All other reductions are within K, M or N, thus reducing the induction metric.

□

Lemma 31. If $K@(M \uplus N)$ strongly normalizes, then $K@M$ and $K@N$ strongly normalize, and

$$\begin{aligned} &\mathit{maxred}(K@(M \uplus N)) \geq \mathit{maxred}(K@M), \text{ and} \\ &\mathit{maxred}(K@(M \uplus N)) \geq \mathit{maxred}(K@N). \end{aligned}$$

Proof. We will show that any single reduction in $K@M$ has a corresponding nonempty reduction sequence in $K@(M \uplus N)$ which produces a new term of the same form (some $K'@(M' \uplus N')$). Therefore no reduction sequence of $K@M$ is longer than the maximal one of $K@(M \uplus N)$ and in particular $K@M$ must strongly normalize.

We construct the corresponding reduction sequence on $K@(M \uplus N)$ as follows. A reduction in $K@M$ must be in K, in M, or at the interface. If it is in K or in M, the reduction applies directly to $K@(M \uplus N)$. If it is at the interface, then $K@M = (K' \circ F)@M \rightsquigarrow K'@M'$. Take cases on the form of F:

CASE $F = (x)N'$. We have $F@M = \mathsf{for}\,(x \leftarrow M)\,N' \rightsquigarrow M'$.

$$\begin{aligned} & (K' \circ (x)N')@(M \uplus N) \\ \rightsquigarrow\ & K'@((\mathsf{for}\,(x \leftarrow M)\,N') \uplus (\mathsf{for}\,(x \leftarrow N)\,N')) \\ \rightsquigarrow\ & K'@(M' \uplus (\mathsf{for}\,(x \leftarrow N)\,N')) \end{aligned}$$

CASE $F = (\mathsf{where}\,B)$. We have $F@M = \mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[\,] \rightsquigarrow M'$

$$\begin{aligned} & (K' \circ (\mathsf{where}\,B))@(M \uplus N) \\ \rightsquigarrow\ & K'@((\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[\,]) \uplus (\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,])) \\ \rightsquigarrow\ & K'@(M' \uplus (\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,])). \end{aligned}$$

CASE $F = (M' \uplus)$. No reductions apply at this interface.

A symmetrical argument applies for $K@N$. □

Lemma 32. If $M \in \mathsf{red}_{[T]}$ and $N \in \mathsf{red}_{[T]}$ then $M \uplus N \in \mathsf{red}_{[T]}$.

Proof. Immediate from Lemma 30. □

Lemma 33. Given a continuation $K : [T]$, a strongly normalizing term $L : S$, and a frame $(x)N : [S] \to [T]$, if $K@(N[L/x])$ strongly normalizes then $K@(\mathsf{for}\,(x \leftarrow [L])\,N)$ strongly normalizes.

Proof. By lexicographic induction on

$$(|K|, \mathit{maxred}(K@(N[L/x]) + \mathit{maxred}(L), \mathit{size}(N)).$$

Take cases on the reducts of $K@(\mathsf{for}\,(x \leftarrow [L])\,N)$, to show that each strongly normalizes:

CASE $K@(\mathsf{for}\,(x \leftarrow [L])\,N) \rightsquigarrow K@(N[L/x])$ (FOR-β)

This s.n. by hypothesis.

CASE $K@(\mathsf{for}\,(x \leftarrow [L])\,[\,]) \rightsquigarrow$ (FOR-ZERO-BODY)
$K@[\,]$ if $N = [\,]$.

This s.n. by Lemma 26.

CASE $(K' \circ (y)M)@(\mathsf{for}\,(x \leftarrow [L])\,N) \rightsquigarrow$ (FOR-ASSOC)
$K'@(\mathsf{for}\,(x \leftarrow [L])\,(\mathsf{for}\,(y \leftarrow N)\,M))$, if $K = K' \circ (y)M$, with $x \notin \mathrm{FV}(M)$.

By the ind. hyp., using K' and $\mathsf{for}\,(y \leftarrow N)M$ for K and N, resp., we get that $K@(\mathsf{for}\,(x \leftarrow [L])\mathsf{for}\,(y \leftarrow N)\,M)$ s.n., as required. To see that the IH applies, note that

$$K'@((\mathsf{for}\,(y \leftarrow N)\,M)[L/x]) = K@(N[L/x])$$

by hypothesis (remembering $x \notin \mathrm{FV}(M)$), and that $|K| > |K'|$.

CASE $K@(\mathsf{for}\,(x \leftarrow [L])\,(N_1 \uplus N_2)) \rightsquigarrow$ (FOR-UNION-BODY)
$K@((\mathsf{for}\,(x \leftarrow [L])\,N_1) \uplus (\mathsf{for}\,(x \leftarrow [L])\,N_2))$ if $N = N_1 \uplus N_2$.

Because $K@(N_1 \uplus N_2)[L/x]$ s.n. we know $K@N_1[L/x]$ and $K@N_2[L/x]$ s.n. (Lemma 31).

Then by the IH, $K@(\mathsf{for}\,(x \leftarrow [L])\,N_1)$ and $K@(\mathsf{for}\,(x \leftarrow [L])\,N_2)$ s.n. (IH applies because the size of N decreases and preceding induction-metric components don't increase.) Finally, by Lemma 30, we conclude that

$$K@((\mathsf{for}\,(x \leftarrow [L])\,N_1) \uplus (\mathsf{for}\,(x \leftarrow [L])\,N_2)) \text{ s.n.}$$

CASE $(K' \circ (\mathsf{where}\,B))@(\mathsf{for}\,(x \leftarrow [L])\,N) \leadsto$ (IF-FOR)
$K'@(\mathsf{for}\,(x \leftarrow [L])\,(\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,])),$ if $K = K' \circ (\mathsf{where}\,B)$.

We have that $x \notin \mathrm{FV}(B)$ by the assumption that all bound variables are distinct. Therefore

$$\begin{aligned} K'@((\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,])[L/x]) &= K'@(\mathsf{if}\,B\,\mathsf{then}\,N[L/x]\,\mathsf{else}\,[\,]) \\ &= K@(N[L/x]) \end{aligned}$$

and this strongly normalizes by hypothesis.

Then by the IH, $K'@(\mathsf{for}(x \leftarrow [L])(\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,]))$ s.n. IH applies because $|K| > |K'|$.

Any other reduction takes place within K, N or L, so the claim for those cases follows by the IH, reducing the induction metric. □

Lemma 34 (Reducibility of for terms)**.** If $M \in \mathsf{red}_{[S]}$ and N is such that $N[L/x] \in \mathsf{red}_{[T]}$ for any $L \in \mathsf{red}_S$ then $\mathsf{for}\,(x \leftarrow M)\,N \in \mathsf{red}_{[T]}$.

Proof. Given any $K \in \mathsf{redK}_{[T]}$ we need to show that $K@(\mathsf{for}\,(x \leftarrow M)\,N)$ s.n. By hyp., for any $K' \in \mathsf{redK}_{[S]}$, we know that $K'@M$ s.n. Pick $K' = K \circ (x)N$ to show this condition, that $K' \in \mathsf{redK}_{[S]}$. To show this requires showing that $K'@[L]$, equivalently $K@(\mathsf{for}\,(x \leftarrow [L])\,N)$, strongly normalizes, for any reducible L. By Lemma 33 and hypotheses, this is the case. So $K'@M$ strongly normalizes, which is what we wanted to write. □

Definition. Define Q-contexts as follows:

$$Q \quad ::= \quad [\;] \mid \mathsf{for}\,(x \leftarrow Q)\,M \mid \mathsf{for}\,(x \leftarrow M)\,Q \mid M \uplus Q \mid \mathsf{if}\,B\,\mathsf{then}\,Q\,\mathsf{else}\,[\,]$$

Definition. Define a measure $|Q|$ as follows:

$$\begin{array}{rcl}
|[\ ]| & = & 0 \\
|\mathsf{for}\,(x \leftarrow Q)\,M| & = & 1 + |Q| \\
|\mathsf{for}\,(x \leftarrow M)\,Q| & = & |Q| \\
|M \uplus Q| & = & |Q| \\
|\mathsf{if}\,B\,\mathsf{then}\,Q\,\mathsf{else}\,[]| & = & |Q|
\end{array}$$

Definition. Define a function $\mathrm{BV}(Q)$, giving the *bound variables (over the hole) of Q* as follows:

$$\begin{array}{rcl}
\mathrm{BV}([\ ]) & = & \varnothing \\
\mathrm{BV}(\mathsf{for}\,(x \leftarrow Q)\,M) & = & \mathrm{BV}(Q) \\
\mathrm{BV}(\mathsf{for}\,(x \leftarrow M)\,Q) & = & \{x\} \cup \mathrm{BV}(Q) \\
\mathrm{BV}(M \uplus Q) & = & \mathrm{BV}(Q) \\
\mathrm{BV}(\mathsf{if}\,B\,\mathsf{then}\,Q\,\mathsf{else}\,[]) & = & \mathrm{BV}(Q)
\end{array}$$

Reduction of Q-contexts is just a case of reduction of contexts, so $Q \rightsquigarrow C'$ iff $Q@M \rightsquigarrow C'[M]$. As with K-continuations, we need to be able to refit a Q-context, when the reduction duplicates the hole, to leave a single hole.

Lemma 35. Whenever $Q \rightsquigarrow C'$ we have for each P some Q' such that $C'[P] = Q'@P$ with $|Q| \geq |Q'|$ and given a finite set X with $X \cap \mathrm{BV}(Q) = \varnothing$, Q' can be chosen such that $X \cap \mathrm{BV}(Q') = \varnothing$.

Proof. By cases on the reduction.

CASE $\mathsf{for}\,(y \leftarrow M \uplus M')\,Q \rightsquigarrow$ (FOR-UNION-SRC)
$(\mathsf{for}\,(y \leftarrow M)\,Q) \uplus (\mathsf{for}\,(y \leftarrow M')\,Q)$.

Given P we have

$$((\mathsf{for}\,(y \leftarrow M)\,Q) \uplus (\mathsf{for}\,(y \leftarrow M')\,Q))[P] =$$
$$((\mathsf{for}\,(y \leftarrow M)\,(Q@P)) \uplus (\mathsf{for}\,(y \leftarrow M')\,Q))@P,$$

with $(\mathsf{for}\,(y \leftarrow M)\,(Q@P)) \uplus (\mathsf{for}\,(y \leftarrow M')\,Q))$ a Q-context.

Meanwhile

$$|\mathsf{for}\,(y \leftarrow M \uplus M')\,Q| = |Q| = |(\mathsf{for}\,(y \leftarrow M)\,(Q@P)) \uplus (\mathsf{for}\,(y \leftarrow M')\,Q)|$$

and the bound variables are unchanged.

CASE $\mathsf{for}\,(y \leftarrow Q)\,(M \uplus M') \leadsto$ (FOR-UNION-BODY)

$(\mathsf{for}\,(y \leftarrow Q)\,M) \uplus (\mathsf{for}\,(y \leftarrow Q)\,M')$.

Given P we have

$$((\mathsf{for}\,(y \leftarrow Q)\,M) \uplus (\mathsf{for}\,(y \leftarrow Q)\,M'))[P] =$$
$$((\mathsf{for}\,(y \leftarrow (Q@P))M) \uplus (\mathsf{for}\,(y \leftarrow Q)\,M'))@P,$$

with the part before the @ a Q-context.

Meanwhile

$$|\mathsf{for}\,(y \leftarrow Q)\,M \uplus M'| = 1 + |Q| = |(\mathsf{for}\,(y \leftarrow (Q@P))M) \uplus (\mathsf{for}\,(y \leftarrow Q)\,M')|$$

and the bound variables are unchanged.

CASE $\mathsf{for}\,(y \leftarrow M \uplus Q)\,M' \leadsto$ (FOR-UNION-SRC)

$\mathsf{for}\,(y \leftarrow M)\,M' \uplus \mathsf{for}\,(y \leftarrow Q)\,M'$

The reduct is a Q-context, while both redex and reduct have size $1+|Q|$ and the bound variables are unchanged.

CASE $\mathsf{for}\,(y \leftarrow M)\,(M' \uplus Q) \leadsto$ (FOR-UNION-BODY)

$\mathsf{for}\,(y \leftarrow M)\,M' \uplus \mathsf{for}\,(y \leftarrow M)\,Q$

Similar to previous cases.

CASE $\mathsf{if}\,B\,\mathsf{then}\,\mathsf{for}\,(x \leftarrow Q)\,N\,\mathsf{else}\,[\,] \leadsto$ (IF-FOR)

$\mathsf{for}\,(x \leftarrow Q)\,(\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,])$.

The reduct is a Q-context and

$$|\mathsf{if}\,B\,\mathsf{then}\,\mathsf{for}\,(x \leftarrow Q)\,N\,\mathsf{else}\,[\,]| = 1 + |Q| = |\mathsf{for}\,(x \leftarrow Q)\,(\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[\,])|$$

and the bound variables are unchanged.

CASE $\mathsf{if}\,B\,\mathsf{then}\,(\mathsf{for}\,(x \leftarrow M)\,Q)\,\mathsf{else}\,[\,] \leadsto$ (IF-FOR)

$\mathsf{for}\,(x \leftarrow M)\,(\mathsf{if}\,B\,\mathsf{then}\,Q\,\mathsf{else}\,[\,])$.

The reduct is a Q-context and

$$|\mathsf{if}\,B\,\mathsf{then}\,\mathsf{for}\,(x \leftarrow M)\,Q\,\mathsf{else}\,[\,]| = |Q| = |\mathsf{for}\,(x \leftarrow M)\,(\mathsf{if}\,B\,\mathsf{then}\,Q\,\mathsf{else}\,[\,])|$$

and the bound variables are unchanged.

CASE $\mathsf{for}\,(x \leftarrow \mathsf{if}\,B\,\mathsf{then}\,Q\,\mathsf{else}\,[\,])\,M \leadsto$ (FOR-IF-SRC)

$\mathsf{if}\,B\,\mathsf{then}\,(\mathsf{for}\,(x \leftarrow Q)\,M)\,\mathsf{else}\,[\,]$

The reduct is a Q-context and

$$|\mathsf{for}\,(x \leftarrow \mathsf{if}\,B\,\mathsf{then}\,Q\,\mathsf{else}\,[\,])\,M| = 1 + |Q| = |\mathsf{if}\,B\,\mathsf{then}\,(\mathsf{for}\,(x \leftarrow Q)\,M)\,\mathsf{else}\,[\,]|$$

and the bound variables are unchanged.

CASE $\mathsf{for}\,(x \leftarrow \mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[\,])\,Q \leadsto$ (FOR-IF-SRC)

$\mathsf{if}\,B\,\mathsf{then}\,(\mathsf{for}\,(x \leftarrow M)\,Q)\,\mathsf{else}\,[\,]$

The reduct is a Q-context and

$$|\mathsf{for}\,(x \leftarrow \mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[\,])\,Q| = |Q| = |\mathsf{if}\,B\,\mathsf{then}\,(\mathsf{for}\,(x \leftarrow M)\,Q)\,\mathsf{else}\,[\,]|$$

and the bound variables are unchanged.

CASE $\mathsf{for}\,(x \leftarrow \mathsf{for}\,(y \leftarrow Q)\,N')\,N \leadsto$ (FOR-ASSOC)

$\mathsf{for}\,(y \leftarrow Q)\,\mathsf{for}\,(x \leftarrow N')\,N$

The reduct is a Q-context and

$$|\mathsf{for}\,(x \leftarrow \mathsf{for}\,(y \leftarrow Q)\,N')\,N| = 2 + |Q| \geq |\mathsf{for}\,(y \leftarrow Q)\,\mathsf{for}\,(x \leftarrow N')\,N| = 1 + |Q|$$

and the bound variables are unchanged.

CASE $\mathsf{for}\,(x \leftarrow \mathsf{for}\,(y \leftarrow M')\,Q)\,N \leadsto$ (FOR-ASSOC)

$\mathsf{for}\,(y \leftarrow M')\,\mathsf{for}\,(x \leftarrow Q)\,N$

The reduct is a Q-context and

$$|\mathsf{for}\,(x \leftarrow \mathsf{for}\,(y \leftarrow M')\,Q)\,N| = 1 + |Q| = |\mathsf{for}\,(y \leftarrow M')\,\mathsf{for}\,(x \leftarrow Q)\,N|$$

and the bound variables are unchanged.

CASE $\mathsf{for}\,(x \leftarrow \mathsf{for}\,(y \leftarrow M)\,N)\,Q \leadsto$ (FOR-ASSOC)

$\mathsf{for}\,(y \leftarrow M)\,\mathsf{for}\,(x \leftarrow N)\,Q$

The reduct is a Q-context and

$$|\mathsf{for}\,(x \leftarrow \mathsf{for}\,(y \leftarrow M)\,N)\,Q| = |Q| = |\mathsf{for}\,(y \leftarrow M)\,\mathsf{for}\,(x \leftarrow N)\,Q| = |Q|.$$

This time we use α-renaming to ensure $y \notin X$.

CASE $\mathsf{if}\, B\, \mathsf{then}\, M \uplus Q\, \mathsf{else}\, [] \rightsquigarrow$ (IF-UNION)

$\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, [] \uplus \mathsf{if}\, B\, \mathsf{then}\, Q\, \mathsf{else}\, []$.

The reduct is a Q-context and

$$|\mathsf{if}\, B\, \mathsf{then}\, M \uplus Q\, \mathsf{else}\, []| = |Q| = |\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, [] \uplus \mathsf{if}\, B\, \mathsf{then}\, Q\, \mathsf{else}\, []|.$$

□

Lemma 36. Given a Q-context Q and terms M and N, if $Q@M$ and $Q@N$ strongly normalize then $Q@(M \uplus N)$ strongly normalizes.

Proof. Similarly to Lemma 31, by induction on $(|Q|, \mathit{maxred}(Q@M) + \mathit{maxred}(Q@N))$. As before, reductions at the interface eliminate frames from Q, pulling them into the two arguments, and giving us a term of the same form, susceptible to the inductive hypothesis. □

Lemma 37. If $Q@(M \uplus N)$ strongly normalizes, then $Q@M$ and $Q@N$ strongly normalize, and

$$\mathit{maxred}(Q@(M \uplus N)) \geq \mathit{maxred}(Q@M), \text{ and}$$
$$\mathit{maxred}(Q@(M \uplus N)) \geq \mathit{maxred}(Q@N).$$

Proof. Similarly to Lemma 30, for any reduction in $Q@M$ or $Q@N$ we can construct a parallel, and no shorter, reduction in $Q@(M \uplus N)$. □

The next lemma examines contexts Q that don't bind any of the variables free in a term B. This is sufficient since externally we will use this lemma only for Q-contexts that are also K-continuations, and internally (when using the inductive hypothesis) the condition is an invariant.

Lemma 38. Given $Q@N$ and B with $\mathrm{FV}(B) \cap \mathrm{BV}(Q) = \varnothing$ and $Q@N$ and B s.n. then $Q@(\mathsf{if}\, B\, \mathsf{then}\, N\, \mathsf{else}\, [])$ s.n.

Proof. By induction on $(\mathit{maxred}(Q@N) + \mathit{maxred}(B), |Q|, \mathit{size}(N))$. Take cases on the reductions of $Q@(\mathsf{if}\, B\, \mathsf{then}\, N\, \mathsf{else}\, [])$:

CASE $Q@(\mathsf{if}\, B\, \mathsf{then}\, (\mathsf{for}\, (x \leftarrow M')\, N')\, \mathsf{else}\, [\,]) \rightsquigarrow$ (IF-FOR)

$Q@(\mathsf{for}\, (x \leftarrow M')\, (\mathsf{if}\, B\, \mathsf{then}\, N'\, \mathsf{else}\, [\,]))$ if $N = \mathsf{for}\, (x \leftarrow M')\, N'$.

The reduct s.n. by the inductive hypothesis, letting Q be $Q \circ (\mathsf{for}\, (x \leftarrow M')[\ \ ]$ and N be N'. We have $size(N) > size(N')$ and the length of the new Q is $|Q \circ (\mathsf{for}\, (x \leftarrow M')[\ \ ]| = |Q|$ while $Q@N$ is unchanged. We have $x \notin \mathrm{FV}(B)$ by the assumption of distinct binders. (This is the only case where $\mathrm{BV}(Q)$ changes in applying the IH.)

CASE $Q_1@(\mathsf{for}\, (x \leftarrow [L])\, (Q_2@(\mathsf{if}\, B\, \mathsf{then}\, N\, \mathsf{else}\, [\,]))) \rightsquigarrow$ (FOR-β)

$Q_1@(Q_2[L/x]@(\mathsf{if}\, B\, \mathsf{then}\, N\, \mathsf{else}\, [\,])[L/x])$ if $Q = Q_1 \circ \mathsf{for}\, (x \leftarrow [L])\, Q_2$.

Because $x \notin \mathrm{FV}(B)$ we have

$$Q_1@(Q_2[L/x]@(\mathsf{if}\, B\, \mathsf{then}\, N\, \mathsf{else}\, [\,])[L/x]) =$$
$$Q_1@(Q_2[L/x]@(\mathsf{if}\, B\, \mathsf{then}\, N[L/x]\, \mathsf{else}\, [\,]))$$

so we show the latter is s.n.

This follows by the inductive hypothesis, letting Q be $Q_1 \circ (Q_2[L/x])$ and N be $N[L/x]$. The induction metric decreases because

$$maxred(Q_1@(\mathsf{for}\, (x \leftarrow [L])\, (Q_2@N))) >$$
$$maxred(Q_1@(Q_2[L/x]@(N[L/x]))).$$

CASE $Q'@(\mathsf{for}\, (x \leftarrow \mathsf{if}\, B\, \mathsf{then}\, N\, \mathsf{else}\, [\,])\, M) \rightsquigarrow$ (FOR-IF-SRC)

$Q'@(\mathsf{if}\, B\, \mathsf{then}\, (\mathsf{for}\, (x \leftarrow N)\, M)\, \mathsf{else}\, [\,]$ if $Q = Q' \circ \mathsf{for}\, (x \leftarrow [\ \ ])\, M$.

The reduct s.n. by the inductive hypothesis, letting Q be Q' and N be $\mathsf{for}\, (x \leftarrow N)\, M$. We have $Q@N$ unchanged while $|Q| > |Q'|$.

CASE $Q@\mathsf{if}\, B\, \mathsf{then}\, N\, \mathsf{else}\, [\,] \rightsquigarrow$ (IF-ZERO)

$Q@[\,]$ if $N = [\,]$.

The reduct s.n. by assumption, since $Q@[\,] = Q@N$.

CASE $Q@(\mathsf{if}\, B\, \mathsf{then}\, (N_1 \uplus N_2)\, \mathsf{else}\, [\,]) \rightsquigarrow$ (IF-UNION)

$Q@((\mathsf{if}\, B\, \mathsf{then}\, N_1\, \mathsf{else}\, [\,]) \uplus (\mathsf{if}\, B\, \mathsf{then}\, N_2\, \mathsf{else}\, [\,]))$ if $N = N_1 \uplus N_2$.

By the inductive hypothesis, we have that $Q@(\mathsf{if}\,B\,\mathsf{then}\,N_1\,\mathsf{else}\,[])$ s.n. and likewise $Q@(\mathsf{if}\,B\,\mathsf{then}\,N_2\,\mathsf{else}\,[])$. To see that IH applies, note that Q is unchanged, $\mathit{size}(N_1) < \mathit{size}(N) > \mathit{size}(N_2)$ and (by Lemma 37)

$$\mathit{maxred}(Q@N_1) \leq \mathit{maxred}(Q@N) \geq \mathit{maxred}(Q@N_2).$$

Then Lemma 36 combines these two facts to show that

$$Q@((\mathsf{if}\,B\,\mathsf{then}\,N_1\,\mathsf{else}\,[]) \uplus (\mathsf{if}\,B\,\mathsf{then}\,N_2\,\mathsf{else}\,[]))\text{ s.n.}$$

All other reductions are within Q, B or N and so reduce $\mathit{maxred}(Q@N)+\mathit{maxred}(B)$. We use Lemma 35 to ensure that a reduction in $Q@P \rightsquigarrow Q'@P$ has $|Q| \geq |Q'|$ and $\textsc{bv}(Q')$ does not intersect $\textsc{fv}(B)$. □

Lemma 39. If $K@N$ s.n. and B s.n. then $K@(\mathsf{if}\,B\,\mathsf{then}\,N\,\mathsf{else}\,[])$ s.n.

Proof. The set of continuations K is a subset of the continuation-contexts Q, and in particular $\textsc{bv}(K) = \varnothing$. Thus the lemma follows immediately from Lemma 38. □

Lemma 40. Given a continuation $K : [T]$ and terms $B : \mathsf{bool}$, $M : [T]$ and $N : [T]$, if B, $K@M$ and $K@N$ strongly normalizes, then $K@(\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N)$ strongly normalizes.

Proof. By induction on $\mathit{maxred}(B) + \mathit{maxred}(K@M) + \mathit{maxred}(K@N)$. When $N = []$, Lemma 39 applies. So consider the case $N \neq []$. Take cases on the reducts of $K@(\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N)$.

CASE The reduct $K@(\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[] \uplus \mathsf{if}\,\neg B\,\mathsf{then}\,N\,\mathsf{else}\,[])$ if $N \neq []$.

Now $K@M$ and $K@N$ s.n. by hyp., so by Lemma 39, $K@(\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,[])$ s.n. and likewise $K@(\mathsf{if}\,\neg B\,\mathsf{then}\,N\,\mathsf{else}\,[])$. Then using Lemma 30 we combine these two facts to show that the reduct s.n.

Other reductions are within B, M, N or K, and so reduce the induction metric, and hence the reducts are s.n. by IH. □

Lemma 41 (Reducibility of conditionals at bag type)**.** If $M \in \mathsf{red}_{[T]}$ and $N \in \mathsf{red}_{[T]}$ and B strongly normalizes then $\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N \in \mathsf{red}_{[T]}$.

Proof. Immediate from Lemma 40. □

At last we can show that conditionals of any type are reducible.

Lemma 42 (Reducibility of conditionals at any type)**.** If $M, N \in \mathsf{red}_T$ and B s.n. then $(\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N) \in \mathsf{red}_T$.

Proof. By induction on T.

CASE $[T]$. By Lemma 41.

CASE $S \to T$. Given a term M' reducible at type S, we must show that the application $(\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N)M'$ is reducible at T. We show that all its reducts are reducible. This is by induction on $\mathit{maxred}(B) + \mathit{maxred}(M) + \mathit{maxred}(N) + \mathit{maxred}(M')$. Consider the reductions. First

$$(\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N)M' \rightsquigarrow \mathsf{if}\, B\, \mathsf{then}\, MM'\, \mathsf{else}\, NM' : T.$$

By (outer) IH, this reduct is in red_T.

All other reductions are within B, M, N, or M', producing a term that is reducible by the (inner) IH.

Because all its reducts are reducible, and applications are neutral, by Lemma 25, $(\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N)M'$ and then also $\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N$ are reducible.

CASE $(\overrightarrow{l : T})$.

We must show that for any $(l : T_l) \in (\overrightarrow{l : T})$, the projection $(\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N).l$ is reducible at T_l. We show that all its reducts are reducible. This is by induction on $\mathit{maxred}(B) + \mathit{maxred}(M) + \mathit{maxred}(N)$. Consider the reductions. First,

$$(\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N).l \quad \rightsquigarrow \quad (\overrightarrow{l = L}).l \qquad \text{(IF-RECORD)}$$

where for each l, $L_l = \mathsf{if}\, B\, \mathsf{then}\, M.l\, \mathsf{else}\, N.l$.

By (outer) IH, each L_l is in red_{T_l}, and so by Lemma 29, the construction $(\overrightarrow{l = L})$ is in $\mathsf{red}_{\overrightarrow{(l:T)}}$. By definition, this means that $(\overrightarrow{l = L}).l$ is in red_{T_l}.

All other reductions are within B, M, N, producing a term that is reducible by the (inner) IH.

Because all its reducts are reducible, and projections are neutral, we have by Lemma 25 that $(\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N).l$ and hence $\mathsf{if}\, B\, \mathsf{then}\, M\, \mathsf{else}\, N$ are reducible.

CASE *o*. All reductions are within B, M or N and strong normalization follows directly from the hypotheses. □

Lemma 43. If $M \in \mathsf{red}_T$ then $\mathsf{empty}(M) \in \mathsf{red}_{\mathsf{bool}}$.

Proof. We merely need to show that $\mathsf{empty}(M)$ strongly normalizes. If M is of relation type then no reductions apply except for those applying within M, which strongly normalizes. Any normal form V of M has the same type and thus we have a normal form $\mathsf{empty}(V)$ of $\mathsf{empty}(M)$.

If M is not of relation type then we apply induction on $\mathit{maxred}(M)$. Consider the reduct $\mathsf{empty}(M) \rightsquigarrow \mathsf{empty}(\mathsf{for}\,(x \leftarrow M)\ [()])$. Now $\mathsf{for}\,(x \leftarrow M)\ [()]$ strongly normalizes by virtue of $M \in \mathsf{red}_T$ (letting $K = \mathit{Id} \circ (x)\,[()]$). For any normal form V of $\mathsf{for}\,(x \leftarrow M)\ [()]$, we know that $\mathsf{empty}(V)$ is a normal form, because V is relation type. Any other reduction is within M and the goal follows by the IH. □

Proposition 3 (Reducibility)**.** Given any typed term

$$y_1 : S_1, \ldots, y_n : S_n \vdash M : T$$

and corresponding closed terms

$$L_1 \in \mathsf{red}_{S_1}, \ldots, L_n \in \mathsf{red}_{S_n},$$

we have $M[\overrightarrow{L/y}] \in \mathsf{red}_T$.

Proof. By induction on the structure of M.

CASE c. Trivially strongly normalizing, hence (because they have base type) reducible.

CASE $\mathsf{table}\, s : T$. Already normal; trivial.

CASE y_i. Each free variable of M is substituted with a reducible term; trivial.

CASE $[]$. Follows immediately from Lemma 26.

CASE $[M]$. Let T be the type of M. By IH, $M[\overrightarrow{L/y}]$ is in red_T, and then by Lemma 27, $[M][\overrightarrow{L/y}]$ is in $\mathsf{red}_{[T]}$.

CASE $\mathsf{for}\,(x \leftarrow M)\, N$. Let $[T]$ be the type of M and $[S]$ be the type of N. We can assume $x \notin \vec{y}$, using α-conversion as necessary.

By IH, $M[\overrightarrow{L/y}]$ is in $\mathsf{red}_{[T]}$ and

$$N[\overrightarrow{L/y}, P/x] = N[\overrightarrow{L/y}][P/x] \in \mathsf{red}_{[S]}$$

for each $P \in \mathsf{red}_T$ (recall the $\vec{L}$ are closed). Then by Lemma 34, $(\mathsf{for}\,(x \leftarrow M)\, N)[\overrightarrow{L/y}]$ is in $\mathsf{red}_{[S]}$.

CASE $M \uplus N$.

By IH, $M[\overrightarrow{L/y}]$ and $N[\overrightarrow{L/y}]$ are in $\mathsf{red}_{[T]}$; then by Lemma 32, $(M \uplus N)[\overrightarrow{L/y}]$ is, too.

CASE $MN : T$. By IH, we have $M[\overrightarrow{L/y}] \in \mathsf{red}_{S \to T}$ and $N[\overrightarrow{L/y}] \in \mathsf{red}_S$; together these directly imply $(MN)[\overrightarrow{L/y}] \in \mathsf{red}_T$.

CASE $\lambda x.N : S \to T$.

By IH, we have that all closing substitutions on N give a reducible term, so in particular $N[\overrightarrow{L/y}][P/x] \in \mathsf{red}_T$ for any $P \in \mathsf{red}_S$. Then by Lemma 28 we get that $\lambda x.N[\overrightarrow{L/y}] = (\lambda x.N)[\overrightarrow{L/y}] \in \mathsf{red}_{S \to T}$,

CASE $M.l$. By IH, $M \in \mathsf{red}_{(\overrightarrow{l:T})}$, which implies directly that well-formed $M.l \in \mathsf{red}_{T_l}$.

CASE $(\overrightarrow{l = M})$. By IH and Lemma 29.

CASE $\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N$.

By IH, $M[\overrightarrow{L/y}]$ and $N[\overrightarrow{L/y}]$ are in red_T, $B[\overrightarrow{L/y}] \in \mathsf{red}_B$; then by Lemma 42, $(\mathsf{if}\,B\,\mathsf{then}\,M\,\mathsf{else}\,N)[\overrightarrow{L/y}] \in \mathsf{red}_T$.

CASE $\mathsf{empty}(M)$. By IH, $M \in \mathsf{red}_T$ and so by Lemma 43 we get $\mathsf{empty}(M) \in \mathsf{red}_{\mathsf{bool}}$.

CASE $\mathsf{query}\,M$. Taking cases on the reducts, which are either just $M[\overrightarrow{L/y}]$ or produced by making reductions within $M[\overrightarrow{L/y}]$, their strong normalization follows directly by the inductive hypothesis. □

Lemma 44 (Unsubstitution)**.** If $M[\overrightarrow{L/y}]$ strongly normalizes then M does.

Proof. By induction on M; most cases are simple applications of the induction hypothesis. For the base case of a variable, x, the variable strongly normalizes regardless of the substitution, so we're done. □

Proposition 4. Any well-typed term strongly normalizes.

Proof. Given M which may have free variables $\vec{x}$, there is some well-typed substitution $[\vec{L}/\vec{x}]$ which produces a closed term $M[\vec{L}/\vec{x}]$, and by the reducibility of closed terms this is reducible. Reducible terms are strongly normalizing, so $M[\vec{L}/\vec{x}]$ strongly normalizes. Then the Unsubstitution Lemma shows that M itself strongly normalizes. □

Normal forms

Proposition 5. Closed, well-typed terms of effect-free relation type have normal forms that satisfy this grammar:

(normal forms)	V, U, W	$::=$	$V \uplus U \mid [\,] \mid F$
(comprehension NFs)	F	$::=$	$\mathsf{for}\,(x \leftarrow \mathsf{table}\,s : T)\,F \mid Z$
(comprehension bodies)	Z	$::=$	$\mathsf{if}\,B\,\mathsf{then}\,Z\,\mathsf{else}\,[\,] \mid [R] \mid \mathsf{table}\,s : T$
(row forms)	R	$::=$	$(\overrightarrow{l = B}) \mid x$
(basic expressions)	B	$::=$	$\mathsf{if}\,B\,\mathsf{then}\,B'\,\mathsf{else}\,B'' \mid \mathsf{empty}(V) \mid$
			$\oplus(\vec{B}) \mid x.l \mid x \mid c$

To prove the special case of closed, effect-free relation-type terms, we first characterize *all* the normal forms.

Lemma 45 (Normal forms). Every well-typed $\rightsquigarrow$-normal form falls in this grammar:

$$\begin{array}{rrcl}
\text{(normal forms)} & V,U,W & ::= & V \uplus U \mid [\,] \mid F \mid R \\
\text{(comprehension NFs)} & F & ::= & \mathsf{for}\,(x \leftarrow L)\, F \mid Z \\
\text{(table-like forms)} & L & ::= & \mathsf{table}\, s : T \mid B \\
\text{(comprehension bodies)} & Z & ::= & \mathsf{if}\, I\, \mathsf{then}\, Z\, \mathsf{else}\, [\,] \mid [V] \mid L \\
\text{(nonbag expressions)} & R & ::= & (\overrightarrow{l = V}) \mid I \\
\text{(nonbag, nonrecord expr'ns)} & I & ::= & \mathsf{if}\, I\, \mathsf{then}\, I'\, \mathsf{else}\, I'' \mid \lambda x.V \mid B \\
\text{(basic forms)} & B & ::= & BV \mid B.l \mid x \mid c \mid \oplus(\vec{V}) \mid \\
 & & & \mathsf{empty}(V)
\end{array}$$

Proof. The proof shows, by induction on the structure of terms, that each term is either ill-typed, non-normal, or matches the above grammar.

Observe that the nonterminal V encompasses all the others in this grammar. As a result we can take the IH as asserting that subterms match the grammar V.

CASE $M \uplus N$, $[\,]$, $[M]$, $\mathsf{table}\, s : T$, $(\overrightarrow{l = M})$, $\lambda x.N$, x, c. These meet the grammar, applying the inductive hypothesis where subterms are concerned.

CASE $\mathsf{for}\,(x \leftarrow L)\, M$. Take cases on L which by IH meets the grammar V:

CASE $V \uplus U$, $[\,]$, $[V]$, $\mathsf{for}\,(x \leftarrow L)\, F$ and $\mathsf{if}\, B\, \mathsf{then}\, Z\, \mathsf{else}\, [\,]$. Active for the context.

CASE $\mathsf{if}\, I\, \mathsf{then}\, I'\, \mathsf{else}\, I''$ with $I'' \neq [\,]$. This term is either bag-typed, and rewrites by IF-SPLIT, or else is not bag-typed and cannot be well-typed in this context.

CASE $\mathsf{table}\, s : T$, BV, $B.l.$ x, c. These meet the grammar.

CASE $\lambda x.N$, $(\overrightarrow{l = M})$. Ill-typed in this context.

Take cases on M, which by IH meets the grammar V:

CASE $V \uplus U$, $[\,]$. Active for the context.

CASE terms matching F. These meet the grammar.

CASE $(\overrightarrow{l = M})$, $\lambda x.N$. Ill-typed in this context.

CASE $\mathsf{if}\, I \,\mathsf{then}\, I' \,\mathsf{else}\, I''$ with $I'' \neq []$. If this term is bag-typed, it rewrites by IF-SPLIT; otherwise it is ill-typed in this context.

CASE $\mathsf{if}\, M' \,\mathsf{then}\, M \,\mathsf{else}\, N$.

The condition M' must be of type bool, and so must be a constant, a deconstruction, or a variable, hence it matches I.

Now take cases on whether the type of $\mathsf{if}\, M' \,\mathsf{then}\, M \,\mathsf{else}\, N$ is a bag type, a record type, or some other.

If it has bag type and N is not $[]$ then the term rewrites. So consider the case that N is $[]$. Take cases on M, which by IH must match the grammar V:

CASE $V \uplus U$, $[]$, $\mathsf{for}\,(x \leftarrow L)F$. Active for the context.

CASE terms matching Z. Meets the grammar.

CASE terms matching P. Ill-typed in this context

If it has record type, it rewrites.

If it does not have bag or record type, take cases on M and N, which must match the grammar V; we enumerate one set of cases since M and N are treated symmetrically:

CASE $V \uplus U$, $[]$, $(\overrightarrow{l = V})$, F. Ill-typed in this context.

CASE I. Meets the grammar.

CASE LM. Take cases on L, which must meet the grammar V:

CASE $V \uplus U$, $[]$, F. Ill-typed in this context.

CASE $(\overrightarrow{l = V})$. Ill-typed in this context.

CASE $\mathsf{if}\, I \,\mathsf{then}\, I' \,\mathsf{else}\, I''$, $\lambda x.N$. Active for the context.

CASE B. Meets the grammar.

CASE $\oplus(\vec{V})$. Meets the grammar.

CASE $\mathsf{empty}(V)$. Meets the grammar.

CASE $\mathsf{query}(V)$. Rewrites to V, hence non-normal.

CASE $V.l$. The only normal form of record type is the record construction $V = (\overrightarrow{l = V})$; but then $V.l$ forms a redex and is not in normal form, a contradiction. □

Next we tighten the grammar for the normal forms of relation-type terms.

First we need a lemma showing that basic forms B essentially inherit their type from the environment—this is because basic forms consist only of a series of destructors applied to a variable.

Definition. A type S is a *subformula* of a type T, written $S \curlyvee T$, iff S appears within T; the relation is the least one satisfying these laws:

$$\begin{array}{lcl} S \curlyvee S & & \\ S \curlyvee T & \Longrightarrow & S \curlyvee T' \xrightarrow{e} T \\ S \curlyvee T & \Longrightarrow & S \curlyvee T \xrightarrow{e} T' \\ S \curlyvee T & \Longrightarrow & S \curlyvee (l : T, \overrightarrow{l : T}) \\ S \curlyvee T & \Longrightarrow & S \curlyvee [T] \end{array}$$

Lemma 46. For any effect-free basic form B with typing $\Gamma \vdash B : T \,!\, \varnothing$ the type T is either a base type or a subformula of the type of one of the variables in Γ.

Proof. By induction on the structure of B. Each syntactic form of B either has base type, or is a deconstructor of a B form, or is a variable. By cases:

CASE BV. Here B must have type $S \to T$. By the IH, B has hereditary typing wrt Γ.

CASE $B.l$. Here B must have type $(\overrightarrow{l : S})$ with $(l : T) \in (\overrightarrow{l : T})$, and by IH B has hereditary typing wrt Γ.

CASE x. Here $x : T$ is in Γ.

CASE c. Constants have base type.

CASE $\oplus(\vec{V})$. By prescription, since this is effect-free it has base type.

CASE $\mathsf{empty}(V)$. Has base type. □

Now in an environment that assigns a row type to each variable, we can more tightly characterize the possible forms, as follows.

Lemma 47. Given any relation-type normal form V with typing $\Gamma \vdash V : T$ where Γ assigns row type to each x in $\mathrm{dom}(\Gamma)$, we have that all free and bound variables appearing in V have row type, and for any subterm $\mathsf{for}\,(x \leftarrow L)\, Z$ we have that L is of the form $\mathsf{table}\, s : [(\overrightarrow{l : o})]$.

Proof. By induction on the structure of V:

CASE $\mathsf{for}\,(y \leftarrow L)\, Z$.

Any free variables appearing in L come from the environment Γ and thus have row type. Thus L cannot itself be a variable. By an inductive argument, it cannot have either of the forms BV or $B.l$ since this would require variables with function type or whose type is a record with a bag field. Finally L cannot be a constant since constants are assumed to have base type. Thus L can only have the form $\mathsf{table} s : [(\overrightarrow{l : o})]$. By the inductive hypothesis, extending Γ with a binding $y : (\overrightarrow{l : o})$, we get the result for subterms in Z.

CASE $\mathsf{if}\, I\, \mathsf{then}\, Z\, \mathsf{else}\, []$, $[V]$, $\mathsf{table}\, s : T$, BV and $B.l$. Here the inductive hypothesis directly gives our proposition.

CASE x. By hypothesis, x has row type. □

Lemma 48. If Γ gives a row type to each variable and B is normal and effect-free with typing $\Gamma \vdash B : T \,!\, \varnothing$ then B is not an application form $B'V$.

Proof. If B had the form $B'V$ then B' would have a type $S \to T$ and this $S \to T$ would have to be a subformula of one of the types in Γ (by Lemma 46), yet these are all row types, a contradiction. □

Lemma 49. If Γ gives a row type to each variable and B is normal and effect-free with typing $\Gamma \vdash B : T \mathbin{!} \varnothing$ with T a base type then B cannot have the form $B'V$ or x.

Proof. Suppose B has the form $B'V$; then B' has type of the form $S \to T$ and $S \to T$ is a subformula of one of the types in Γ (by Lemma 46), yet these are all row-type, a contradiction. Suppose B is a variable, x. Then its type appears in Γ; yet x has base type and all the variables in Γ have row type, a contradiction. $\square$

Lemma 50. If Γ gives a row type to each variable and B is normal and effect-free with typing $\Gamma \vdash B : T \mathbin{!} \varnothing$ with T a row type then B cannot have the form $B'V$, $B'.l$, c, $\mathsf{empty}(V)$ or $\oplus(\vec{V})$.

Proof. The forms c, $\mathsf{empty}(V)$ and $\oplus(\vec{V})$ all have base type and so would be ill-typed. If B has the form $B'V$, the term B' has type of the form $S \to T$, and this is a subformula of one of the types in Γ (by Lemma 46), yet these are all row-type, a contradiction. If B has the form $B'.l$ then the field l of B' must have a row type, making B' something strictly larger than a row type; and as such it cannot be a subformula of one of the types in Γ, a contradiction. $\square$

Lemma 51. If Γ gives a row type to each variable and Z is normal with typing $\Gamma \vdash Z : T$ with T a relation type, then Z cannot have any of the forms ranged by B, and if it has the form $[V]$ then $V = (\overrightarrow{l = I})$ and each field member I has one of the forms $\mathsf{if}\, B\, \mathsf{then}\, B'\, \mathsf{else}\, B''$, $\mathsf{empty}(V)$, $\oplus(\vec{B})$, $x.l$, x or c.

Proof. Suppose Z falls in the set ranged by B. Then its type must be a subformula of one of the types in Γ (by Lemma 46). But since Z has relation type and none of the elements of Γ can have such a subformula, this is a contradiction.

Now suppose Z is a singleton list $[V]$. Because it is relation-typed, V must be row-typed. Thus it cannot have any of the forms $V \uplus U$, $[]$ or F (recalling that it cannot have any form ranged by B). This leaves the forms ranged by R. At row type, it cannot have a conditional form since this is not normal. It cannot be an abstraction because this would be ill-typed. Thus it can only be a record construction $(\overrightarrow{l = V})$ with each V_l having base type. The only normal forms which have base type are those ranged by B and the form $\mathsf{if}\, I \,\mathsf{then}\, I' \,\mathsf{else}\, I''$. Within the forms ranged by B, Lemma 48 shows that it cannot have the form $B'V$. And it cannot be a variable x because this would give it relation type in the context Γ. If it has the form $B'.l$, then B' can only be a variable (other possibilities being ill-typed due to the hereditary typing or given conditions). Remaining are the forms $\mathsf{if}\, B \,\mathsf{then}\, B' \,\mathsf{else}\, B''$, $\mathsf{empty}(V)$, $\oplus(\vec{B})$, $x.l$, x and c as we wanted to show. □

Lemma 52. Wherever $\mathsf{empty}(V)$ appears in a normal-form term, V has relation type.

Proof. The rule EMPTY-FLATTEN ensures this. □

Observation. Each type/effect inference rule other than that for functional abstractions $\lambda x.N$ is monotonic in its effects. That is, if the conclusion is $\Gamma \vdash M : T \,!\, e$ then each precondition $\Gamma' \vdash M' : T' \,!\, e'$ has $e \supseteq e'$.

Corollary. If a term has no functional abstractions, then its entire derivation is monotonic in the effects. That is, for any derivation of $\Gamma \vdash M : T \,!\, e$, any subderivation $\Gamma' \vdash M' : T' \,!\, e'$ has $e \supseteq e'$.

Lemma 53. In an effect-free normal form of relation type, any operation application $\oplus(\vec{V})$ has all its arguments of base type, thus each argument meets the grammar for B.

Proof. Because the term is pure and contains no abstractions, every subterm is pure. By the side condition that every primitive either has arguments all of base type or has an effect, we can infer that primitives in such a term have arguments all of base type, and thus meet the normal-form grammar for B. □

At last we can prove the full characterization of the normal forms of query-bracketed terms.

Proof of Prop. 5. Striking from the grammar of Lemma 45 the forms disallowed by Lemmas 47, 48, 49, 50, 51, 52, and 53 we are left with a grammar for the normal forms of closed relation-type expressions:

(normal forms)	V, U, W	$::=$	$V \uplus U \mid [] \mid F$
(comprehension NFs)	F	$::=$	$\mathsf{for}\,(x \leftarrow \mathsf{table}\, s : T) F \mid Z$
(comprehension bodies)	Z	$::=$	$\mathsf{if}\, B\, \mathsf{then}\, Z\, \mathsf{else}\, [] \mid [R] \mid \mathsf{table}\, s : T$
(record expressions)	R	$::=$	$(\overrightarrow{l = B}) \mid x$
(basic expressions)	B	$::=$	$\mathsf{if}\, B\, \mathsf{then}\, B'\, \mathsf{else}\, B'' \mid \mathsf{empty}(V) \mid$
			$\oplus(\vec{B}) \mid x.l \mid c$

Which is what we wanted to show. □

5.5 Adding Recursion

A general-purpose programming language without recursion would be severely hobbled; but in standard SQL, recursive queries are not expressible. Thus we need to add recursion to our language but ban it from query expressions.

We add recursion by introducing a recursive λ-abstraction, spelled recfun. It introduces a recursive function of one argument and forces the resulting function type to have a noqy effect.

$$\frac{\Gamma, f : S \xrightarrow{e \cup \{\mathsf{noqy}\}} T, x : S \vdash M : T \,!\, e}{\Gamma \vdash \mathsf{recfun}\, f\, x = M : S \xrightarrow{e \cup \{\mathsf{noqy}\}} T \,!\, \varnothing} \quad \text{(T-RecFun)}$$

Alternatively, we could introduce a fixpoint operator:

$$\frac{\Gamma \vdash M : (S \xrightarrow{e} T) \xrightarrow{\varnothing} (S \xrightarrow{e} T) \,!\, \varnothing}{\Gamma \vdash \mathsf{fix} M : S \xrightarrow{e \cup \{\mathsf{noqy}\}} T \,!\, \varnothing} \quad \text{(T-Fix)}$$

This prohibition is conservative—it forbids even primitive-recursive programs, for example, which might sometimes be translatable to SQL—but reasonable, since programmers are not in the habit of writing queries that require that much power.

5.6 Adding the length operator

The examples in Section 5.1 used a function *length* which we have not yet studied. This section shows how to extend the system to support it. In the source, it is much like *empty*, but SQL's nonuniformity forces us to handle it specially.

First we extend the source language with length:

$$M ::= \cdots \mid \mathsf{length}(M)$$

The typing rule is as you would expect, resulting in type int.

Extend the SQL-like sublanguage (the normal forms) as follows:

$$B ::= \cdots \mid \mathsf{length}(F)$$

Note that we will normalize the argument to a comprehension normal form, F, so that it gives a select query and not, say, a union all query. Next, augment the SQL target:

$$e ::= \cdots \mid \mathsf{select\ count}(*)\ \mathsf{from}\ \overrightarrow{t\ \mathsf{as}\ x}\ \mathsf{where}\ e$$

And hence we can translate it to SQL as follows:

$$\begin{aligned} [\![\mathsf{length}(F)]\!] &= \mathsf{select\ count}(*)\ \mathsf{from}\ \overrightarrow{t\ \mathsf{as}\ x}\ \mathsf{where}\ e \\ &\qquad \text{where } \mathsf{select}\ \vec{s}\ \mathsf{from}\ \overrightarrow{t\ \mathsf{as}\ x}\ \mathsf{where}\ e = [\![F]\!] \end{aligned}$$

Now we add the following rewrite rules:

$$\begin{array}{rcll} \mathsf{length}(M) : T & \rightsquigarrow & \mathsf{length}(\mathsf{for}\,(x \leftarrow M)\ [()]) & \text{(LENGTH-FLATTEN)} \\ & & & \text{if } M \text{ is not relation-typed} \\ \mathsf{length}([\,]) : T & \rightsquigarrow & 0 & \text{(LENGTH-ZERO)} \\ \mathsf{length}(M \uplus N) : T & \rightsquigarrow & \mathsf{length}(M) + \mathsf{length}(N) & \text{(LENGTH-UNION)} \end{array}$$

Thus requires, of course, that we have the constant 0 and the integer-addition operation (+) in our set of constants and primitives.

Other SQL aggregate functions (`avg`, `max`, `min`, and so on)—all of which take a bag to a scalar—can be handled in a similar fashion.

5.7 Language-integrated query systems

The query system Kleisli [Wong, 2000] is structured as a general-purpose programming language, which compiles (sometimes partially) into SQL. Those parts of a Kleisli program that the compiler cannot translate are executed directly by the interpreter; there is no *a priori* way to determine what programs, or parts thereof, would compile to SQL.

The LINQ project [Microsoft Corporation, 2005] is a set of extensions to the .NET framework which allow expressing database queries (targeting SQL, XML and other data models) in two ways: through an object interface or through SQL-like syntactic sugar. The object interface provides methods for query operations, such as mapping, filtering, and so on. Many of these methods accept code, in the form of "expression trees." For instance, the `Where` method, for filtering, takes as argument a predicate with which to filter. Up to a point, this facility permits arbitrary code from the host language to be added to queries. But not all code is successfully translated. For example, it is possible to use a predicate as a query condition; but it is not possible to compose these predicates. This is because functions for use in queries have a distinct type, `Expr<Function<A,B»`, rather than the function type `Function<A,B>`, and the former does not afford composition in a way that the query translator will recognize. So, for example if *pred* is a predicate of type `Expr<Function<B,Bool>>` and f is a function of type `Expr<Function<A,B>>`, and one wishes to filter a bag of rows to those rows x such that $pred(f(x))$ is true, it is necessary to declare a new function $pred2$ which implements the composition.

It bears noting that LINQ, like Kleisli, allows the expression of queries that can't be expressed in SQL, but whose results can be constructed from an SQL query. For example, a LINQ query can give its result rows in the form of an object type, which has no direct analogue in SQL; in this case the query generator may still perform the bulk of the query in SQL and simply repackage the results during a post-processing phase. Such a splitting is not afforded by the system of this chapter, because this system is based in a hard assertion of what code must translate to SQL. It may be desirable to find a flexible middle ground which would allow expressions to be split, as in LINQ and Kleisli, and still offers some static guarantees of queryization, as in this chapter.

The Links language [Cooper et al., 2006] also offers language-integrated query. As in Kleisli, the queries are expressed using the language's own native iteration constructs and conditionals. Kleisli and LINQ accomodate abstracted functions to a point: Kleisli allows them in positions where they are immediately applied, and LINQ allows them (unapplied) in particular roles, for example, as filtering conditions or mapping functions. More sophisticated combinations of abstraction and application (for example, currying and partial application) are not permitted. Links implements the higher-order normalization described in this chapter. The Links implementation, by Sam Lindley, also allows polymorphic types, using a *kinding* discipline to force polymorphic query expressions to have appropriate types.

The Ferry system [Grust et al., 2009] is a first-order functional programming language that translates a query, which may have a nested result type, into one or more flat SQL queries. Nested types will require a number of queries depending on the depth of nesting; the runtime system reassembles the multiple query results into the nested result value. As a functional programming language, it is restricted (to first-order functions and pure, non-recursive computations), allowing it to be totally translatable to SQL without further analysis.

Comprehensions Comprehensions as a syntax for database queries have a long history, going back at least to Trinder and Wadler [1989] and surveyed by Grust [2003]. Kleisli [Davidson et al., 1997], with its query language CPL, is an early commercial system using comprehensions as a query notation.

Other database systems that apply comprehensions for querying include the QLC query language of Erlang's Mnesia [Mattsson et al., 1998]. QLC (for *Query List Comprehensions*), rather than compiling to SQL, runs directly against an Mnesia database, using certain of its own optimizations. For example, a comprehension that draws from two Mnesia tables using a condition to relate them may be executed using joining algorithms, taking advantage of indexes. The LINQ special syntax can also be seen as a kind of comprehension.

5.8 History

The "first-normal-form" restriction in database theory—the restriction that every relation should be flat, containing only base types—dates back to E. F. Codd's original definition of the relational model. Schek and Scholl [1986] were the first to examine a non-flat relational model, defining a Nested Relational Calculus.

Paredaens and van Gucht [1988, 1992] gave the first unnesting result, showing that nested relational algebra, when restricted to flat input and output relations, is equivalent in power to traditional flat relational algebra. Wong [1996] soon extended this result, showing his "conservativity" result, that any first-order nested relational algebra expression can be rewritten so that it produces no intermediate data structures deeper than the greatest of its input and output relations. Fegaras [1998] also shows how to transform higher-order nested relational queries into flat ones using a rewrite system with similarities to ours; we extend this by offering a proof of normalization, taking the queries all the way to SQL, and showing how a type-and-effect system can separate the translatable and untranslatable fragments of a general-purpose language. van den Bussche [2001] extended Wong's first-order result to show that even nested-result-type expressions can be simulated with the flat relational algebra, if we allow for an interpretation function which assembles the flat results back into a nested relation. The Ferry system [Grust et al., 2009] expands on Van den Bussche's result and translates the queries all the way to SQL.

Hillebrand et al. [1993] prove an inverse result to the present one, that simply-typed λ-calculus itself subsumes other query languages (flat relational calculus, Datalog$^{\neg}$, and others), and in particular that PTIME queries become λ-terms evaluable in PTIME.

Wiedermann and Cook [2007] and Wiedermann et al. [2008] present a technique using abstract interpretation for extracting structured queries from imperative object-oriented programs using an ORM. Here object-oriented dispatch takes the place of higher-order functions. Again the extraction is partial.

The effort to harmonize query languages with programming languages is nearly as old as the two fields themselves. Atkinson and Buneman [1987] survey the early history of integration. The impedance mismatch problem between databases and programming languages is described by Copeland and Maier [1984].

The use of comprehension syntax was a breakthrough for language integration, since it gave an iteration construct that was both powerful enough for much general-purpose programming and also explicit enough to admit a more direct translation to a query language; this connection is explored by many authors [Trinder and Wadler, 1989, Trinder, 1992, Breazu-Tannen et al., 1992, Buneman et al., 1994, Grust and Scholl, 1999]. Grust [2003] summarizes much preceding work on monad comprehensions as a query language.

Chapter 6

Future Work

The essential Links project of creating an experimental language to ease web programming has more or less been achieved. The group developed a new language with an execution model that encompasses the complete span of a web application, from the browser environment through the server environment to a backend relational database.

Much more can be done to make Links more usable, as well as to further the goal of making web programming easier.

Links improvements

As mentioned in the body of the thesis, the location annotations on functions should be completed in several ways: they should be allowed on non-top-level functions, there should be a form that allows annotating arbitrary terms. The calculus should be extended to allow *sets of locations* as annotations.

Formlets are adequate for composition of static forms, but they don't offer any way to create dynamic, interactive controls that react immediately to user activity. Could the syntactic and semantic approach of formlets be extended to define reactivity for such controls?

The SQL compilation chapter showed how to detect expressions that can be compiled to a single SQL query; but Grust et al. showed how to translate each expression in a pure language into a fixed set of SQL queries, even when no single query is equivalent, as when the result has a nested collection type. It should be possible to accomodate such expressions in the framework of Chapter 5.

Furthermore, there are several features of SQL which we might wish to target in such a translation—particularly the `group by` and `order by` clauses. Peyton Jones and Wadler [2007] have developed a notation and semantics, integrated with that of comprehensions, for expressing these transformations, and Grust et al. [2009] have shown how to perform the translation for the `order by` clause.

The serialization of closures in Links is vulnerable to changes in the source code; yet, in the face of code changes, we should expect active web applications to keep working or to gracefully inform the user that certain existing links will no longer work. Conventional web systems can often handle old links after code updates, since their URLs are controlled manually anyway. Ideally, the programmer should be able to control the behavior of outdated links; some might have a sensible substitute in the new version, while others will become dead ends and require an apology. How best for the programmer to define these behaviors? Might outdated links cause an exception to be thrown in some global scope, which the programmer could choose to handle as she liked? Might the scope in fact be more targeted to permit even a convenient separation of error handling between application segments?

This last point leads to a larger set of concerns: How to make the language support truly *web-savvy* application development. Web application developers are typically very concerned about the ways their applications interact with the URL space of the web and their application's participation in the *web resource* concept.

Web developers often want to fine-tune the structure of their URLs; they may even see that structure as part of the public interface of their application. The ability for clients to form URLs might be an important part of a *web API*, that is, an HTTP interface that other software can use to fetch and manipulate information programmatically. A web API might be defined to take several parameters at various points in the URL, which clients can vary at will. This is ill-supported by the present design of Links, because all of Links' internal URLs are presently opaque. Certain web frameworks, notably Catalyst for Perl [Riedel et al.] and Ocsigen for OCaml [Balat, 2006], allow associating various kinds of URL patterns with functions, and the framework takes care of dispatching incoming requests to these. In Links, we would need a bi-directional mapping from code-points to URLs and back again, so that we could both construct URLs and dispatch upon them.

Extending the formalisms

Each of the formalisms of the thesis could be extended, perhaps providing guidance for a broader set of languages or applications.

The RPC calculus could be extended to show how exceptions could be handled by the same trampolining approach. Also, we saw a particular network topology, with client and server, but it would be interesting to see how location-aware languages could be implemented on another topologies, perhaps with many machines arranged in a client-server sequence or with arbitrary connections among them.

In showing how a stateful system can be implemented with a stateless server, this work examined only a limited form of state. Mutable data would be a compelling form of state to study. Different approaches to such data are possible: reference could point into a global store, shared between client and server (with a provision to expunge it from the server at the low level, of course), or client and server could have different stores with separate references

The concurrency and message-passing facilities in Links could also be modeled within the calculus with more work.

The RPC calculus, like Links, made it easy to move between client and server. But programmers might like a way of controlling this movement—for example, barring moves during performance-critical sections. Such control could perhaps be achieved using an effect system and annotations like in the SQL compilation analysis.

Previous studies of comprehensions for querying have generalized from the use of bags as the collection type to any monad with appropriate monoid operations [Watt and Trinder, Fegaras, 1998, Grust, 2003], sometimes called a *ringad* [Watt and Trinder]. Some of the rewrite rules in Chapter 5 are unsound for these structures in general, since the rewrites require something like commutativity of the ($\uplus$) operation. Kleisli offers three collection types (sets, bags, and lists); could we extend the totality result of this chapter to such a suite of types?

Design mistakes

Experience has suggested that, in the author's view, some choices made in the design of Links were mistakes. Emulating the surface syntax of a language with quite different semantics (namely JavaScript) was intended to lure programmers from 'the other side'; instead it served mainly to confuse people (where semantics differed) and lead to some painful compromises. Multi-argument functions are a cumbersome duplication of features already available and fit awkwardly with higher-order functions (which now must be sensitive to the arity of the argument function). Using the semicolon to force a unit-typed expression is a trick too subtle for normal use, and leads to lots of spurious type errors—and it has not proved itself by catching dangerous errors. Embedding XML in the language can be useful, particularly for short examples, but in software engineering one wants to put the user interface definition as far from the core logic as possible.

Comprehensions in Links have a body of list type and the evaluated bodies are concatenated; this is similar to the XQuery notion of comprehension but differs from that of Haskell, Python and JavaScript. The advantage of its approach is that further generators and guards can be expressed by nesting the existing comprehension (`for`) and conditional (`if`) constructs. But rarely, if ever, do we make use of the ability to put a non-singleton literal list in a comprehension body, and so the perpetual need for a singleton constructor becomes a potential source of errors. Also, using nested comprehensions instead of the multiple-generator ones interacts poorly with the `order by` and `group by` clauses proposed by Peyton Jones and Wadler [2007]: since these clauses must be *inside the scope* of one or more generators but affect the *result returned* by all of them together, we need some way to indicate which series of generators are to be affected. The traditional comprehension notation, which allows multiple generators, is a natural delimiter for `order by` and `group by`.

Using Erlang's process-specific mailboxes in our statically-typed setting required some way of breaking out of the process' fixed mailbox type. This could have been done with first-class (rather than process-specific) message queues; instead we did it with the `spawnWait` keyword, effectively allowing a process to create a new mailbox, handle some messages on it, and then continue—a dynamically-scoped mailbox.

Epi{gr,t}aph for Links:

Look closely at the most embarrassing details and amplify them.
(Oblique Strategy #116, Brian Eno and Peter Schmidt.)

Chapter 7

Conclusion

Links was a research experiment in creating a new language for the web, unifying the messy components that programmers commonly have to stitch together.

The challenge proved to be a difficult one. Across the tiers, there are many independent points where better programming abstractions could be desired. Trying to invent all of these and keep them integrated in the context of a brand-new language was an ambitious challenge—perhaps too ambitious.

Trying to integrate all aspects of a big area has the disadvantage that there is no partial success. Can any single language hope to keep up with all the external systems that are needed in an area? Can it hope to give unleaky abstractions that seal away the details of all of these?

The world outside our integrated garden is changing, and we can't keep up with it. It would be better to make bridges, to make advancements that interact well with the outside world.

Still, web programming is an exciting domain for which to invent abstractions, because its execution model poses unfamiliar challenges for programmers. Unlike batch and GUI event-loop programs, web programs are re-entrant: if we conceive of the whole application as one program, then control flow takes novel forms in the presence of the back button. Surely, many web programmers do not even perceive a new execution model—they see themselves as writing batch programs that spit out web pages and talk to databases—so it is a step forward if we have defined a sensible concept of what web programs are, in their natural habitat. And, optimistically, the facilities offered here for query integration and location-awareness might contribute to other spheres than web programming.

I'm most proud of the ways this thesis, and the Links research group, accomplished what Jeremy Yallop asked for in my introduction: internalizing "design patterns" within the language. Most of the useful features presented here are based on a handful of powerful pre-existing ideas: CPS translation, defunctionalization, trampolining, rewrite systems, and idioms, for example. Web programming is, admittedly, a simple task compared to what many people do with programs; but we were able to use these elegant artifacts of computer science to provide just the right abstractions for some web programming tasks.

Underneath all this, the Links work has borne out the idea that good ideas are best expressed in clear, concise formalisms. The definition of formlets, the compilation scheme for the RPC calculus, and the rewriting system for query translation are each expressible on one sheet of paper in 12-point type with whitespace to spare. This makes them much more attractive for re-implementing in other languages. The formalisms guided us to designs of low complexity and helped us communicate our ideas efficiently.

Here's hoping the future will produce more graceful formalisms that make computer programming easier and more powerful, decade after decade.

Bibliography

J. Armstrong, M. Williams, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

D. L. Atkins, T. Ball, G. Bruns, and K. C. Cox. Mawl: A domain-specific language for form-based services. *IEEE Trans. Softw. Eng.*, 25(3):334–346, 1999.

Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/62070.45066.

Vincent Balat. Ocsigen: typing web interaction with Objective Caml. In *ML '06*, September 2006.

I.G. Baltopoulos and A.D. Gordon. Secure compilation of a multi-tier web language. In *Proceedings of the 4th international workshop on Types in language design and implementation*, pages 27–38. ACM New York, NY, USA, 2009.

Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In *TACS '01*, volume 2215 of *Lecture Notes in Computer Science*, pages 420–447. Springer, 2001.

Jeffrey M. Bell and James Hook. Defunctionalization of typed programs. Technical report, Oregon Graduate Institute, 1994.

Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. *SIGPLAN Not.*, 32(8):25–37, 1997.

Tim Berners-Lee. Web architecture from 50,000 feet. Available at `http://www.w3.org/DesignIssues/Architecture.html.`, September 1998. URL `http://www.w3.org/DesignIssues/Architecture.html`.

C. Brabrand, A. Moller, A. Sandholm, and M. I. Schwartzbach. A runtime system for interactive Web services. *Computer Networks-the International Journal of Computer and Telecommunications Networkin*, 31(11):1391–1402, 1999.

C. Brabrand, A. Møller, M. Ricky, and M. I. Schwartzbach. PowerForms: Declarative client-side form field validation. *World Wide Web*, 3(4):205–214, 2000.

Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The <bigwig> project. *ACM Trans. Internet Techn.*, 2(2):79–114, 2002.

Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *ICDT '92*. Springer, 1992.

P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. 149(1):3–48, 1995.

Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23:87–96, 1994.

Adam Chlipala. The ur programming language family, 2008. URL `http://www.impredicative.com/ur/`. Fetched Mar 2009.

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level web service construction. *ACM Trans. Program. Lang. Syst.*, 25(6):814–875, 2003.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO '06*, 2006.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *APLAS '08*, 2008.

George Copeland and David Maier. Making smalltalk a database system. *SIGMOD Rec.*, 14(2):316–325, 1984. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/971697.602300.

Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

Olivier Danvy and Kevin Millikin. Refunctionalization at work. Technical Report RS-08-4, BRICS, June 2008.

Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *PPDP '01*, pages 162–174. ACM, 2001.

SB Davidson, C. Overton, V. Tannen, and L. Wong. BioKleisli: a digital library for biomedical researchers. *International Journal on Digital Libraries*, 1(1): 36–53, 1997.

Chris Eidhof. Haskell Formlets module, 2008. URL `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/formlets`.

Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP '97*, pages 263–273, New York, NY, USA, 1997. ACM Press.

Leonidas Fegaras. Query unnesting in object-oriented databases. In *SIGMOD '98*, pages 49–60, New York, NY, USA, 1998. ACM.

Michael J. Fischer. Lambda calculus schemata. *SIGACT News*, (14):104–109, 1972.

Alain Frisch. OCaml + XDuce. In *ICFP '06*, pages 192–200, 2006.

Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. Trampolined style. In *ICFP '99*. ACM Press, September 1999.

David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP '86*, pages 28–38, New York, NY, USA, 1986. ACM.

Google Inc. Google Suggest (application), December 2004. URL `http://labs.google.com/suggest`. Now incorporated into the `google.com` home page.

Paul Graham. Beating the averages, 2001a. URL `http://www.paulgraham.com/avg.html`.

Paul Graham. Method for client-server communications through a minimal interface. United States Patent no. 6,205,469, March 20 2001b. (filed May 27, 1997).

Paul Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In *ASE '01*, pages 211–222, Washington, DC, USA, 2001a. IEEE Computer Society.

Paul Graunke, Robert Findler, Shriram Krishnamurthi, and Matthias Felleisen. Modeling web interactions and errors. In *ESOP '03*, Warsaw, Poland, Apr 2003. Springer-Verlag.

Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the web with high-level programming languages. In *ESOP '01*, pages 122–136, London, UK, 2001b. Springer-Verlag.

Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the web with high-level programming languages. In *ESOP '01*, pages 122–136, 2001c.

T. R. G. Green. Cognitive dimensions of notations. In *Proc. of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, pages 443–460, New York, NY, USA, 1989. Cambridge University Press.

T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. A Ferry across the great database and programming languages divide. *Submitted for publication.*

Torsten Grust. *The Functional Approach to Data Management*, chapter Monad comprehensions, a versatile representation for queries. Springer Verlag, 2003.

Torsten Grust and Marc H. Scholl. How to comprehend queries functionally. *J. Intell. Inf. Syst.*, 12(2-3):191–218, 1999.

Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *SIGMOD '09*, June 2009.

Michael Hanus. Type-oriented construction of web user interfaces. In *PPDP '06*, pages 27–38, 2006.

Michael Hanus. Putting declarative programming into the web: Translating Curry to JavaScript. In *PPDP '07*, pages 155–166, 2007.

Gerd G Hillebrand, Paris C Kanellakis, and Harry G Mairson. Database query languages embedded in the typed lambda calculus. In *LICS '93*, 1993.

Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

Michael Jouravlev. Redirect after post, August 2004. URL `http://www.theserverside.com/tt/articles/article.tss?l=RedirectAfterPos%t`.

Sam Lindley and Ian Stark. Reducibility and $\top\top$-lifting for computation types. In *TLCA '05*, pages 262–277, 2005.

Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In Venanzio Capretta and Conor McBride, editors, *MSFP '08*, Reykjavik, Iceland., 2008.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88*, pages 47–57, New York, NY, USA, 1988. ACM.

John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD thesis, August 1987.

Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *POPL '07*, pages 3–10, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4.

Jacob Matthews, Robert Bruce Findler, Paul Graunke, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. *Automated Software Engineering*, 11:337–364, 10 2004. doi: 10.1023/B:AUSE.0000038936.09009.69. URL `http://www.springerlink.com/content/t552m86535518257`.

Håkan Mattsson, Hans Nilsson, and Claes Wikström. Mnesia—a distributed robust DBMS for telecommunications applications. In *PADL '99*, pages 152–163. Springer, 1998.

Conor McBride. Idioms, 2005. Presented at the Scottish Programming Languages Seminar, June 2005. `http://www.macs.hw.ac.uk/~trinder/spls05/McBride.html`.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.

Jay McCarthy. PLT Scheme Formlets module, 2008. URL `http://docs.plt-scheme.org/web-server/formlets.html`. Distributed with PLT Scheme [PLT].

Leo Meyerovich. Flapjax: Functional reactive web programming, 2007. URL `http://www.cs.brown.edu/lmeyerov/thesis8.pdf`.

Microsoft Corporation. The LINQ project: .NET language integrated query. White paper, September 2005.

Tom Murphy, VII. *Modal Types for Mobile Code*. PhD thesis, Carnegie Mellon University, 2007.

Tom Murphy, VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *LICS '04*, pages 286–295, Washington, DC, USA, 2004. IEEE Computer Society.

Gavi Narra. ObjectGraph dictionary (application). Described at `http://www.objectgraph.com/dictionary/how.html`, 2004.

M. Neubauer and P. Thiemann. Placement Inference for a Client-Server Calculus. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, page 86. Springer, 2008.

Matthias Neubauer. *Multi-Tier Programming*. PhD thesis, Universität Freiburg, 2007.

Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL '05*, pages 221–232, New York, NY, USA, 2005. ACM Press.

Lasse R. Nielsen. A denotational investigation of defunctionalization. Technical Report BRICS RS-00-47, DAIMI, Department of Computer Science, University of Aarhus, December 2000.

Jan Paredaens and Dirk van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *PODS '88*, pages 29–38, New York, NY, USA, 1988. ACM.

Jan Paredaens and Dirk van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1):65–93, 1992.

Tomáš Petříček and Don Syme. Rich client/server web applications in F$\sharp$, 2007. URL `http://tomasp.net/fswebtools/files/fswebtoolkit-ml.pdf`.

Simon L. Peyton Jones and Philip Wadler. Comprehensive comprehensions. In Gabriele Keller, editor, *Haskell*, pages 61–72. ACM, 2007. ISBN 978-1-59593-674-5.

Rinus Plasmeijer and Peter Achten. iData for the World Wide Web—programming interconnected web forms. In *FLOPS '06*, volume 3945 of *LNCS*, Fuji Susono, Japan, 2006. Springer Verlag.

Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

PLT. PLT Scheme. URL `http://plt-scheme.org/`.

François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *POPL '04*, pages 89–98, New York, NY, USA, 2004. ACM.

Franąois Pottier and Didier Rémy. *Advanced Topics in Types and Programming Languages*, chapter 'The essence of ML type inference', pages 389–489. MIT Press, 2005.

Christian Queinnec. Continuations to program web servers. In *International Conference on Functional Programming*, 2000.

Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, 2003.

Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, chapter 3, pages 67–95. MIT Press, 1993.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, New York, NY, USA, 1972. ACM Press.

John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3):233–247, 1993.

Sebastian Riedel, David Naughton, Marcus Ramberg, Jess Sheidlower, Danijel Milicevic, Kieren Diment, and Yuval Kogman. Introduction to Catalyst: Actions. URL `http://search.cpan.org/dist/Catalyst-Manual/lib/Catalyst/Manual/Intro.p%od#Actions`.

Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997.

H. J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11(2):137–147, 1986.

Manuel Serrano, Erick Gallesio, and Florian Loitsch. HOP, a language for programming the Web 2.0. In *Proc. of the First Dynamic Languages Symposium*, Portland, Oregon, USA, Oct 2006. URL `http://www-sop.inria.fr/members/Manuel.Serrano/publi/dls06/article.html%`.

Steve Strugnell. Creating linksCollab: an assessment of Links as a web development language. Bachelor's thesis, U of Edinburgh, 2008. Available at `http://groups.inf.ed.ac.uk/links/papers/undergrads/steve.pdf`.

W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

Jean-pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Information and Computation*, pages 162–173, 1992.

Peter Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *PADL '02*, pages 192–208, London, UK, 2002. Springer-Verlag.

Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Inter. Tech.*, 5(1):1–46, 2005.

Phil Trinder. Comprehensions, a query notation for DBPLs. In *DBPL '91*, San Francisco, CA, USA, 1992. ISBN 1-55860-242-9.

Phil Trinder and Philip L. Wadler. List comprehensions and the relational calculus. In *Glasgow Workshop on Functional Programming*, pages 187–202, 1989. URL `citeseer.ist.psu.edu/wadler99list.html`.

Jan van den Bussche. Simulation of the nested relational algebra by the flat relational algebra, with an application to the complexity of evaluating powerset algebra expressions. *Theoretical Computer Science*, 254(1-2):363–377, 2001.

Peter van Roy. Convergence in language design: a case of lightning striking four times in the same place. In *FLOPS*, 2006.

Philip Wadler. Monads for functional programming. In *Advanced Functional Programming '95*, volume 925 of *LNCS*, pages 24–52. 1995.

David A. Watt and Phil Trinder. Towards a theory of bulk types. Technical Report Fide Technical Report 91/26, Glasgow University.

Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL '07*, 2007.

Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. *SIGPLAN Not.*, 43(10):19–36, 2008. ISSN 0362-1340.

Wikipedia. Post/redirect/get, November 2008. URL `http://en.wikipedia.org/wiki/Post/Redirect/Get`.

Hugh E. Williams and David Lane. *Web Database Applications with PHP & MySQL*. O'Reilly, 2nd edition, 2004.

Limsoon Wong. Kleisli, a functional query system. *J. Functional Programming*, 10(1):19–56, January 2000.

Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3):495–505, 1996.

World Wide Web Consortium. DOM level 3 core, recommendation, April 2004. URL `http://www.w3.org/TR/DOM-Level-3-Core/`.

World Wide Web Consortium. Architecture of the World Wide Web 1.0, December 2003a. URL `http://www.w3.org/2001/tag/2003/webarch-20031128`.

World Wide Web Consortium. HTML 4.01 specification, 1999. URL `http://www.w3.org/TR/html4/`.

World Wide Web Consortium. Document Object Model (DOM) level 3 events specification, November 2003b. URL `http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107/`.

Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: a syntactic proof technique. In *ICFP '99*, pages 197–207, New York, NY, USA, 1999. ACM Press.

Glossary of Web Terms

control (user interface) An item with which a user can interact, such as a button, a tab, a scroll bar, etc.

Document Object Model (DOM) The tree-like data model used by web browsers to represent a web page; also, the interface defined for manipulating this data and that for receiving notification of user activity on the page.

server cluster A collection of servers all of which play the same role; for example, they might all provide a certain web application by responding to web requests from clients; typically (though not always), incoming requests are distributed amongst servers in a farm to balance the load and so no client or task is consistently assigned to any particular machine.

web resource An implicit information resource which may admit retrieval or modification using web protocols; that which is located by a URL or Uniform Resource Locator.

XMLHttpRequest The standard interface for making HTTP requests from JavaScript code in a browser.

Colophon

This thesis was typeset by the author with LaTeX. The text and headings are set in New Century Schoolbook, code in Computer Modern Typewriter, and math in Fourier with Computer Modern Sans Serif. Drawings were produced in OmniGraffle and Inkscape.