Designing for Extensibility and Planning for Conflict: Experiments in Web-Browser Design

Benjamin S. Lerner

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2011

Program Authorized to Offer Degree: UW Computer Science & Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Benjamin S. Lerner

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

_____

Daniel Grossman

Reading Committee:

_____

Daniel Grossman

_____

Steven Gribble

_____

John Zahorjan

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Designing for Extensibility and Planning for Conflict: Experiments in Web-Browser Design

Benjamin S. Lerner

Chair of the Supervisory Committee:
Associate Professor Daniel Grossman
UW Computer Science & Engineering

The past few years have seen a growing trend in application development toward "web applications", a fuzzy category of programs that currently (but not necessarily) run within web browsers, that rely heavily on network servers for data storage, and that are developed and deployed differently from traditional desktop applications. Where (typical) traditional applications are *compiled* pieces of code, written in *arbitrary* languages, that implement both an application's user interface and its functionality, web apps by contrast are written in *three interpreted* languages: HTML to define the *structure* or content of the UI, CSS to define the *appearance*, and JavaScript (JS) to define the *behavior*. These three languages feel nothing alike, and are used for different facets of the applications.

The last decade has also seen the rise of Mozilla Firefox, a web browser whose UI and functionality are themselves written in (dialects of) HTML, CSS, and JS, making Firefox one of the first fully-fledged web apps. Part of Firefox's appeal is its strong support for *extensions*, which are downloadable, third-party pieces of code (i.e., not written by Mozilla or with Mozilla's cooperation) that enhance the browser with additional functionality or customizations. Firefox extensions are wildly popular: over six thousand distinct extensions have been downloaded over 2.5 billion times [193], and all major browser vendors have added varying degrees of support for extensions to their own products. Crucially, these extensions are also written in HTML, CSS, and JS: writing an extension feels fundamentally similar to writing a web page or web app.

Thanks to the dynamic, interpreted nature of these three languages, it is mostly straightforward to incorporate the contents of an extension into the existing browser. There are, however, some caveats. Not all programs are equally amenable to *post-hoc* extension, and there are currently no guarantees that multiple extensions do not *conflict*, destabilizing each other or the base browser.

In this dissertation, I aim to provide better support for rich extensibility for web apps. In particular, I claim that

Language-specific extension mechanisms are needed for each of HTML, CSS, and JS, and such mechanisms are needed for building useful diagnostic tools to address inter-extension conflicts.

To support this thesis, I first present C3, the "Cloud Computing Client", an implementation of the HTML/CSS/JS platform architected explicitly to support experimentation with extensibility. I then define two such extension mechanisms for HTML and for JS: *overlays* and *aspects*, respectively. I develop conflict analyses for HTML overlays, and evaluate them on a sample of Firefox extensions. Conflict analyses for JS are sketched, and extension mechanisms for CSS are left for future work.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would not have reached this point but for the support and friendship of many advisors, friends, colleagues and family, without whom graduate school would not have been possible on an academic or personal level.

First, thanks go to my advisor, Dan Grossman, for teaching me how to formalize and clearly explain ideas that seem intuitive, while still encouraging me to follow those flights of fancy and see where they may lead. More importantly, by example he helped me refine my presentational and teaching skills, for which my sincere thanks.

Thanks also go to my committee members, Steve Gribble and John Zahorjan, in whose classes and seminars I learned the impact of asking—and answering—the right questions. Discussions with them have helped guide and refine this work.

Thanks must also go to the tireless Lindsay Michimoto, for her always-ready support, advice, and hard work.

This dissertation is in large measure the fruit of a very successful collaboration with Microsoft Research, particularly with Herman Venter and Wolfram Schulte, and fellow intern Brian Burg. I greatly enjoyed working with the `RiSE` team, and look forward to continuing our collaboration.

Heartfelt thanks go to all my friends, who remind me that work can be left at school and life goes on afterward. In particular, thank you to my housemates, Steven Balensiefer, Kate Everitt, Stef Schoenmackers, Charlie Reis, and Kevin Wampler, for all the challenges, stimulating conversations, and fun times we've shared. Thanks also go to my long-distance friends, particularly Sara and Jeff Gordon, Jon Kirsch, Vardit Samuels, and Mike Seplowitz, who all made the effort never to lose touch. I am grateful also to my officemates, Neva Cherniavsky, Brian van Essen, Ethan Katz-Basset, Martha Kim, Andrew Putnam and Qi Shan, for enlivening the work day.

A final thanks go to my family, who have listened to me ramble *ad nauseum* about the minutiae of my work, and have even bravely attempted to read my papers. All I can say is, I'm glad you were listening.

DEDICATION

To my family, who never doubted me even when I did.

To Pop-pop, who is greatly missed.

Chapter 1

INTRODUCTION[1]

1.1 From browsers to platforms

Intuitively, an *extensible system* is one that permits later revision of the previously-designed base system: additions to, improvements upon, or replacements of existing functionality. In the past few years, people have come to expect their *web browser* to be an extensible system, adding toolbars, social-network customizations, interface tweaks, and many other personalizations to adapt the browser to their needs. Browser extensions are wildly popular: as of July 2011, Mozilla hosts over 6,000 Firefox extensions downloaded over 2.5 billion times [167, 168], while Google already hosts nearly 11,500 extensions for Chrome [95].

Writing individual browser extensions is not difficult: much like basing an application off an existing, rich library, extensions can leverage some or all of the functionality of the browser and its user interface. Indeed, these extensions frequently need to interact with the browser in fine-grained, non-trivial ways. The challenge in writing browser extensions *well* lies in making them robust in the face of other extensions: unlike standalone client applications, extensions do not get to monopolize the browser that hosts them. This is both a hardship and a bonus, as extensions can use this common host cooperatively to extend one another. However, currently there is no adequate support for ensuring that extensions are *compatible* with each other.

Most recently, the browser has begun to "evaporate", as browser vendors have competed in streamlining the browser interface to the point where it nearly disappears in everyday use. Production-quality and experimental systems such as Google's Chrome, HP's webOS, Mozilla Labs' Chromeless, Boot to Gecko, and the Webian shell all provide browser-like functionality with progressively less "browser" UI—indeed, Chromeless provides no default UI at all! The browser is morphing from a *program* into a *platform* for hosting web applications: the crucial elements of a browser are its support for HTML, CSS and JavaScript (JS), rather than its particular user interface. Accordingly, the emphasis is shifting from writing browser extensions to writing extensions on and for the *web platform*.

---

[1] This chapter is based on an earlier work: Language Support for Extensible Web Browsers, in Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications, {978-1-60558-913-8, (June 5)} © ACM, 2010. http://doi.acm.org/10.1145/1810139.1810146 [136]

My goal in this dissertation is to promote extension development on the web platform as a programming model worthy of study, and to focus attention on language support for improving extension development, and especially extension compatibility efforts. In particular, this thesis argues that:

1. *A powerful extension mechanism for the web platform is justified and desirable.* The browser itself has become an extensible system whose extensions often interact with, or are themselves, web applications. Often, these extensions need to interact with each other as well. Extensions are a hybrid: they are (pieces of) separate programs and so reasonably are self-contained entities distinct from the browser, but are also shared tenants of a browser environment and so must coexist with each other.

2. *The browsers (i.e., implementations of the web platform) that currently implement extensions do so poorly.* Firefox provides a flexible and powerful framework that yields essentially no reasonable semantics or security guarantees about multiple extensions: they are as privileged and unrestricted as the browser itself. Chrome has the reverse problem: in sandboxing extensions into a reasonable security framework, it has prevented useful and fine-grained interactions among them. The remaining mainstream browsers (Internet Explorer, Opera and Safari) all are adopting support for extensions, to varying degrees, but they all are less capable than either Firefox or Chrome; consequently, I mostly ignore them in the remainder of this dissertation.

3. *Programming languages research is an appropriate tool to help.* Defining how extensions may or may not interact with one another is a problem of semantics. Detecting and resolving conflicts among those interactions can benefit from declarative, language-based security techniques.

The rest of this chapter demonstrates the above points by example, with further details elaborated in subsequent chapters, and is organized as follows. Section 1.2 justifies the need for extensions by describing the breadth of extensions today and at a high level what browser support they require. Section 1.3 takes a closer look at the extension models of Firefox and Chrome, focusing on the limitations mentioned above. Section 1.4 presents an overview of several of my projects that help define a reasonable semantics for extensions, which will be described in detail in the subsequent chapters. Section 1.5 summarizes this chapter, and describes the structure of the rest of this dissertation.

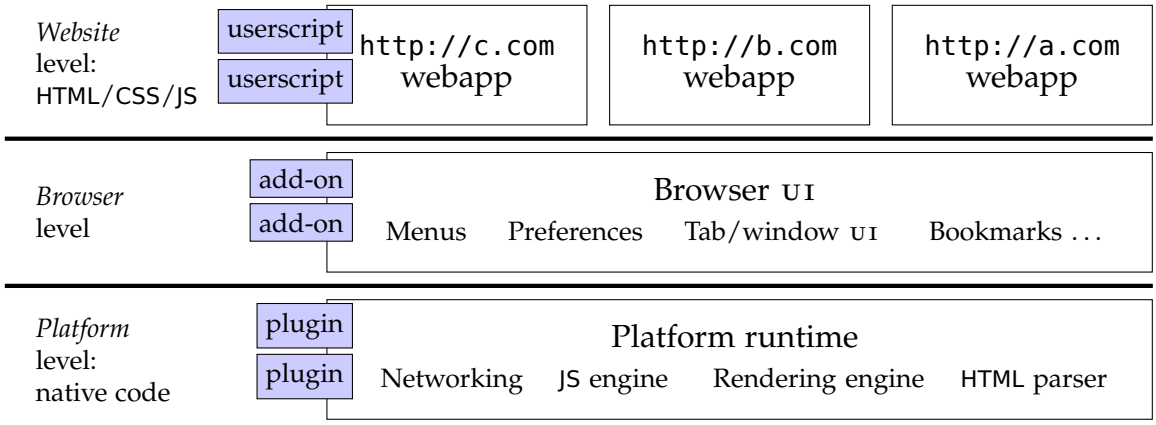


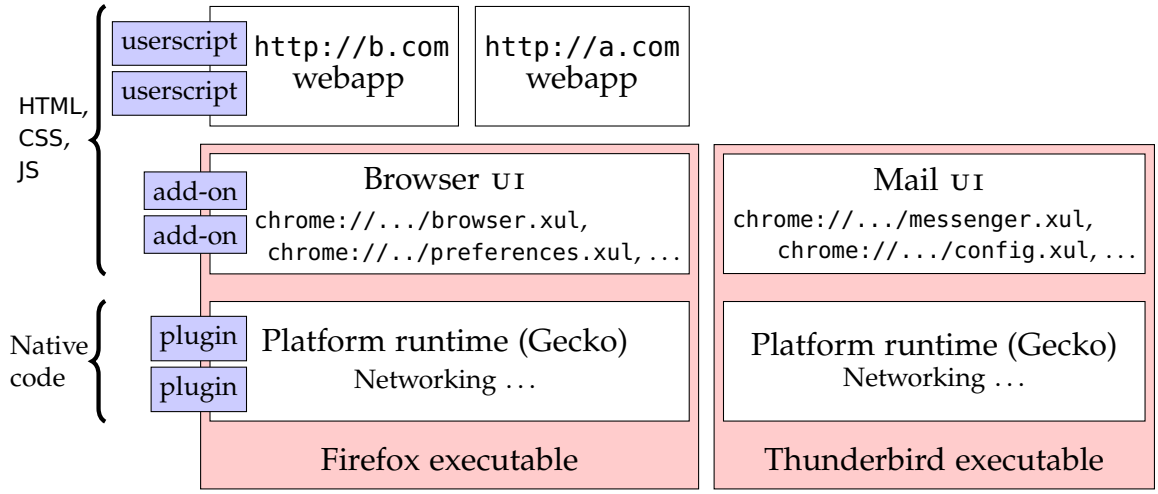
Figure 1.1: Schematic overview of the current web platform. Each website, the browser UI, and the platform runtime are conceptually distinct entities. Thick lines separate the three levels of the "web stack".

1.2 Positioning extensions

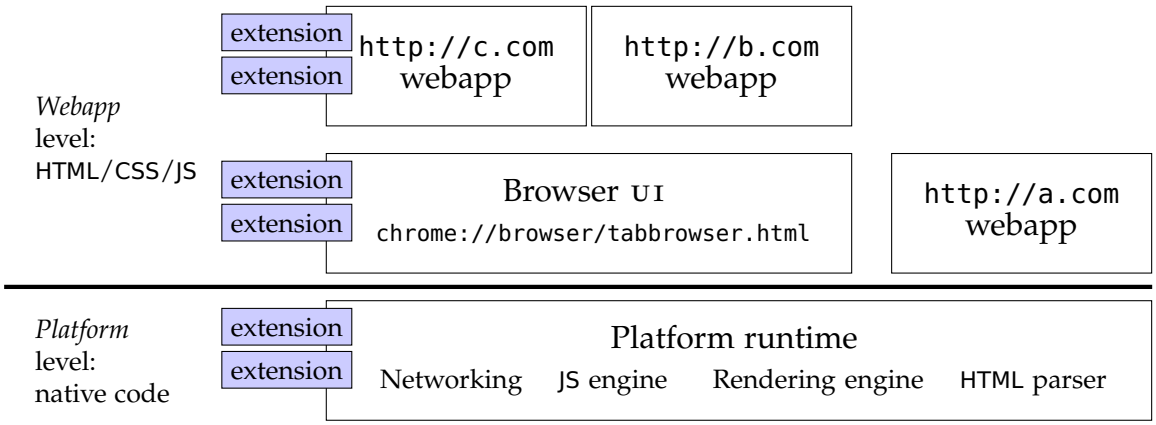
Properly framing the discussion of web-platform extensions requires identifying three distinct layers in the current client-side "web stack", each of which has supported technologically distinct extension mechanisms. A schematic diagram of this stack is shown in Fig. 1.1. I revisit these distinctions in Chapter 3, and give an overview here:

1. The lowest *platform level* implements support for HTML parsing and rendering, JS execution, and so on. This layer supports extensions, more commonly known as *plugins*, that add support for rendering new content types: for example, Flash, Silverlight, and Java. Plugins in this layer are site-agnostic: they are loaded to handle the relevant content, whichever site it comes from. Additionally, plugins typically do not interact with one another: for example, Java applets do not manipulate the state of PDFs viewed elsewhere in the page. This sort of extensibility is of interest to researchers experimenting with new capabilities for the web; I ignore them for now, but see Section 3.3 for further treatment.

2. The middle *browser level* implements the familiar idioms of browsing the web, such as page navigation, address bars, history and bookmarks, etc., and hosts the website level. Implementing the browser level relies on the platform level. Extensions at this level, also called *add-ons*, typically enhance the browsing experience in various ways; I give examples below.



(a) Schematic overview of Mozilla products' architecture. The platform runtime (known as Gecko) and the browser UI (perceived as "Firefox") are bundled into a single executable. That same runtime can be used to run other programs such as Thunderbird, but it is duplicated in each separate executable.



(b) Schematic overview of future web platforms, where from the perspective of the platform, the browser is "just another webapp", perhaps with elevated privileges compared to normal web content.

Figure 1.2: Schematic overview of current browser architectures compared to future web platform architectures

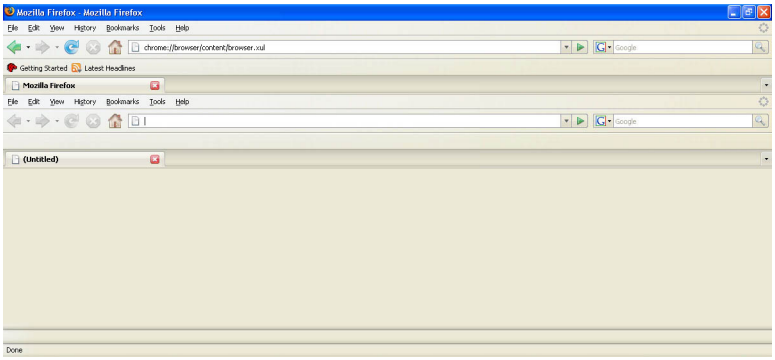


Figure 1.3: Firefox running within Firefox: because its UI is essentially a web page, available at a special URL (`chrome://browser/content/browser.xul`), Firefox-the-application can render Firefox-the-UI within itself.

These extensions are capable of manipulating the state of the browser, as a whole, and can (sometimes) interact with one another.

3. The upper *website level* is where individual web sites execute. They are hosted by the browser level. Extensions to web sites, commonly referred to as *userscripts*, are site-specific modifications that enhance the experience of a particular site. All userscripts on a single page can interact with one another.

Historically, "web pages" and "web browsers" have been technologically distinct—one was written in HTML/CSS/JS, one was not—and so were strictly separated as two levels of the stack. With the development of Firefox and the recent rise of web applications, this distinction is rapidly dissolving, shown schematically in Fig. 1.2. Everything that browsers use in their UIs (e.g., menus, toolbars, scrollbars, and controls) can be defined using HTML, CSS and JS to separate the UI structure, appearance and behavior—in precisely the same manner that developers define the widgets and controls used in webapps today. In fact, as I argue more fully in Chapter 3, from the perspective of the platform level—i.e., the piece of the stack that implements the core engines for interpreting HTML/CSS/JS—the following do not need to be distinguished:

1. a browser written in HTML/CSS/JS and allowing third-party extensions

2. a web-app written in HTML/CSS/JS and allowing third-party extensions, hosted inside a browser window

3. a web-app written in HTML/CSS/JS and allowing third-party extensions, running directly on the underlying web platform

Consequently, the historical distinction between *userscripts*, which extend individual web pages, and *browser extensions*, which extend the browser itself, should also vanish: only web application extensions remain.

Since this dissertation is about implementing an extensible web platform and therefore takes the perspective of that platform, I treat (1) as simply the most common example of (3), specifically one that happens to include the familiar idioms of web browsing, e.g., back/forward buttons, an address bar, etc.; see Fig. 1.3. (Indeed, projects like Mozilla Prism, Chromeless and Boot to Gecko [66, 169, 170] are already starting along this approach, and simplify the browser layer into a nearly-invisible wrapper: web applications can run in them without the intercession of a browser.)

The C3 architecture described in Chapter 3 implements the schematic architecture of Fig. 1.2b, and (except where otherwise noted) I generally include both userscripts and browser extensions under the term "extensions". Most examples of extensions are drawn from browser extensions, simply because there are more substantive and interesting examples from which to choose, but my conclusions apply more broadly.

1.2.1 The case for extensions

Web applications are becoming large collaborations of software spanning server-side logic and data storage, client-side code, data storage and UI elements, network interactions (both social and otherwise), and more—the boundaries of such a web application are often indistinct, and in general do not fit neatly into the legacy HTML page-as-document model of web interaction. Browser extensions form a new facet of this space that runs neither on servers in the cloud nor as web content within a browser. Rather, they run as *part* of the browser, transcending any one web page to enhance the browsing experience itself. This "intermediate" execution model blurs further the distinctions between client- and server-side code.

To highlight how inappropriate the page-as-document model has become, consider the various clients for social networks, which are far more about the content being generated and shared by their members than they are about the pages used to view that content. StumbleUpon[2] is a browser toolbar that lets its users group sites by their personal interests, then use those groupings to find other pages matching their current interest quickly. This experience is inherently *about* pages,

[2] All extensions mentioned here are available from http://addons.mozilla.org.

rather than *within* pages: a function of the browser and not its content. Likewise, Twitter clients such as TwitterBar or Yoono focus on letting users quickly publish tweets, often in response to what they are browsing; note that these clients are written to target the Twitter protocol, but are not applications written by Twitter itself. RSS aggregators such as Feedly explicitly create client-side views of data published from multiple sources, without needing a dedicated web site such as Google Reader to view the aggregated content.

Not all browser extensions are necessarily a part of web applications as these examples are. Many "only" modify the browser's UI, or stay purely on the client's machine. Others are primarily client-based but extend into network services. Mozilla Weave, for example, synchronizes multiple clients' history, bookmarks, and passwords with cloud-backed storage: it improves the client-side experience by transparently using the network. There is no clear line separating "client" extensions from "web application" ones. Moreover, in many cases, there may not even be a clear line separating one extension from another. For example, Firebug is an extremely popular extension permitting developers to debug web content, capturing page content, script, network requests, cache behavior and more. FireDiff is another extension that explicitly extends Firebug with a new view showing diff-like traces of page events.

These seven examples demonstrate how broadly extensions may behave. Moreover, no one user will ever install all available extensions: a web developer might install Firebug and scoff at using Yoono, while a social-networker might take the exact opposite stance. Not only do extensions let users customize the browser to their own needs without bloating it for others, they let users add features that *may not have existed* when the browser was first developed.

1.2.2 *Designing for power, flexibility and stability*

Current browser designs trade extension capability and flexibility for security and stability. On the one hand, extensions should be able to customize the appearance and behavior of the browser in a fine-grained, pervasive way, while multiple extensions should be able to interact with, build upon, or complement each other. This is the essence of Firefox's approach. On the other, extensions must be restrained sufficiently that they cannot destabilize the base system; in particular, malicious extensions should not be able to subvert the browser. This is the essence of Chrome's approach.

An extension system such as Firefox's poses an additional stability challenge: multiple interacting extensions must coexist compatibly, or detect when they cannot do so. Extensions are in some ways "selfish" code: they want to have unfettered access to the internals of the mainline code or other extensions, and yet want to protect their own code from that same unfettered ac-

cess. Since extensions are intended to themselves be extensible, they require a design that by default leaves them as extensible as the underlying browser, but that lets them (and the browser) protect their critical internals.

1.3 Contrasting two extension models

Both Firefox and Chrome let authors extend two fundamental resources, namely the UI of the browser and the functionality of the browser, to varying levels and in different ways. These two resources are very different—one is declarative, one imperative and free-form—so both browsers define separate mechanisms for extending each. I compare each browser's approach to extending each resource, and give examples of actual problems that extensions face in each browser.

1.3.1 Extending the user interface

Chrome: Chrome deliberately aims to minimize the user interface (the "chrome" surrounding the web content) in its browser, and so offers a limited selection of interface elements to extensions: Chrome extensions use 16 fields to describe themselves, and of those only four are UI extension points. This is a "pull" model: Chrome collates extensions' self-descriptions, and creates the new UI elements (e.g., toolbar buttons) accordingly. Because extending the UI is intentionally limited, extensions cannot extend each other's UI.

StumbleUpon: Working around limitations The four UI extension points supported in Chrome intentionally do not include a facility for creating toolbars. Some extensions, such as StumbleUpon, rely on a custom toolbar as their exclusive user interface. For these extensions, the only solution is to inject HTML into web pages that mimics the appearance of a toolbar positioned at the top edge of the content area. Unfortunately, this workaround cannot be fully robust: the vertical scrollbars in the chrome will reveal that the "toolbar" is part of the page (see Fig. 1.4), CSS in the page may inadvertently restyle the toolbar, frame-busting code may be triggered to remove the toolbar, and the toolbar cannot be injected until after the page has finished loading (yielding a visually-jarring flicker). StumbleUpon lists these as known issues, unsolvable without a richer extension API [205].

Firefox: By contrast, Firefox defines its entire UI using XUL, a markup language much like HTML, and CSS. It then exposes *everything* in its user interface to extensions via *overlays*, a reflective mechanism that lets extension authors "patch" XUL documents at runtime. Given a base document, an overlay can select nodes in the document by their `id` attribute and define new content

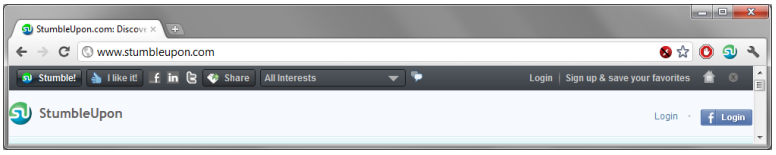


Figure 1.4: Faking the StumbleUpon toolbar in Chrome: the vertical scrollbar reveals that the toolbar (dark gray bar) is actually injected into the document

to be inserted into them. Since nearly every element in Firefox's UI has an id, this allows developers to target the whole UI freely and to insert any UI element they choose. This is a "push" model: Firefox does not decide *a priori* which UI elements to expose, but rather extensions force themselves into their targets.

The overlay mechanism suffers from several important flaws. First, its current implementation has no well-defined semantics and exposes race conditions: the loading order of multiple overlays determines the document order of their composite result (see Fig. 1.5). This might change or break the event handling order of keyboard shortcuts, or cause UI elements to be pushed outside the visible space of the window. Second, nothing prevents multiple extensions from using overlays to create elements with duplicate identifiers, breaking the crucial uniqueness property of identifiers and potentially fooling other extensions into overlaying the wrong portion of the UI. Third, no practical way exists for an extension (or Firefox) to declare some document nodes as frozen for overlaying. Finally, no error is raised if an overlay *fails* to find a target element to extend, which masks errors among different versions of Firefox.

1.3.2 *Extending the functionality*

The program logic for both Firefox and Chrome extensions is written in JavaScript, but beyond that the two extension approaches differ.

Firefox: As with its UI, Firefox defines much of its functionality in JS and permits extensions to modify that code arbitrarily. Firefox extensions typically use two idioms, *wrapping* and *monkeypatching*, to inject themselves into existing functionality. These idioms use a combination of JS quirks, `eval`, and runtime rebinding of variables to change existing code's behavior. Since all chrome code (from Firefox and extensions alike) lives in a common namespace, it is frequently impossible to prevent one extension from modifying another's code. Said another way, there is no

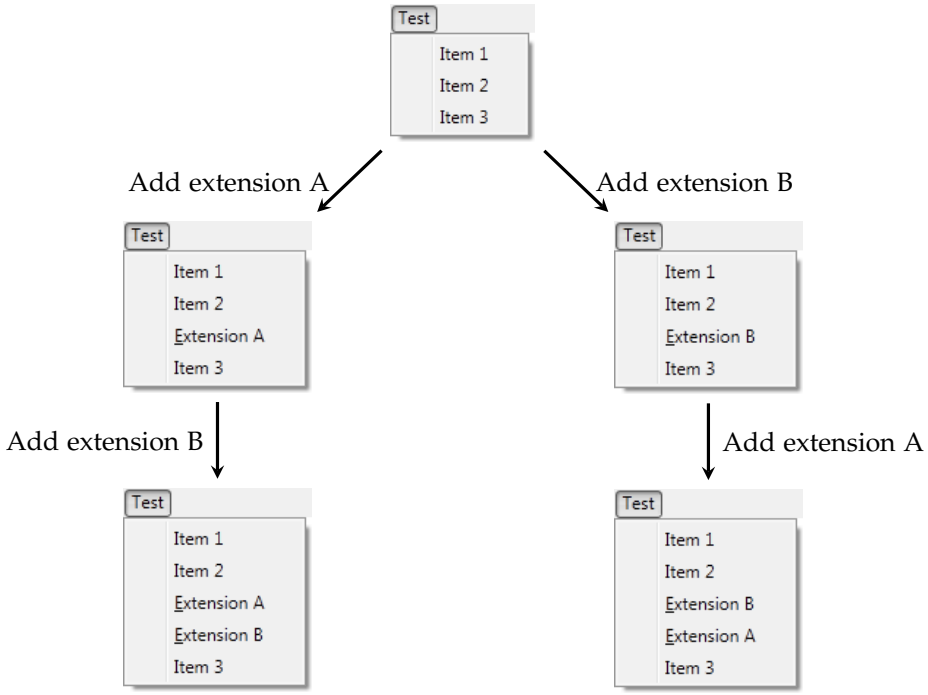


Figure 1.5: The effect of composition order on XUL document structure: Both extensions A and B insert their menu item *just before Item 3*. Adding either one alone to the base menu yields a predictable composite result. Adding *both* extensions yields two different results, depending on load order—and the menu hotkeys no longer work.

notion of extensions as distinct security principals, and hence extensions (and Firefox too) cannot ensure their own integrity at runtime.

AdBlock Plus and NoScript: A cautionary tale AdBlock Plus and NoScript are two extremely popular Firefox extensions that between them block content from user-selected ad providers and script from all but user-whitelisted sites. AdBlock collaborates with several authors who maintain subscription lists of ad domains to block; these subscriptions update without user intervention. However, NoScript's development is supported by first-run ads that display each time an updated version of the extension is installed. For the many users running both extensions, this posed a problem for NoScript. In May 2009 a brief "arms race" began between the two extensions. NoScript used a known bug in AdBlock to hide its ads from detection; AdBlock asked for NoScript's domains to be added to the most common subscription list. NoScript countered by obfuscating the sources of its ads; the list was updated to match, and so on several times a day for most of a week. Eventually the filters were so over-broad as to break legitimate script on NoScript's in-

stallation page. Consequently, NoScript released an update that modified some internal functions of AdBlock to permanently construct a whitelist for NoScript's ads; that modification destabilized and broke AdBlock on legitimate sites [152, 175].

While this particular spat was resolved successfully and without harm to users' machines or data, the implications are chilling. The same ability to extend another extension amicably can be abused intentionally to disable, cripple or subvert other extensions.

Chrome: By contrast, Chrome focuses heavily on robustness: extensions should not be able to subvert or destabilize the user's browsing session. To achieve this, Chrome extensions are segregated into individual security principals by the same-origin policy (where an origin is either a triple $\langle host, port, protocol \rangle$ or a unique token equal to nothing but itself) [188], and are further split into three layers: 1) JS running in the context of the web pages being viewed, 2) additional JS and HTML running in a separate process, and 3) binary components running in a third process. Each layer can communicate with the next via message passing. Only the topmost layer can manipulate web pages; only the middle layer can coordinate the extension across multiple pages or access network resources, and even then it is constrained only to resources declared in the extension manifest. (The third layer is irrelevant for my purposes.)

One strong advantage of the Chrome approach is that each extension runs in its own JS namespace: extensions cannot accidentally interfere with each other's code. No wrapping or monkey-patching is necessary—or possible. The layered architecture is both a strength and weakness: it helps ensure stability and prevent capability leaks, but is perhaps too limiting in preventing extensions from collaborating with each other: extensions cannot deliberately interfere with another's code, either. For instance, it would be impossible to implement a generic messaging client extension and later write supplemental extensions supporting specific protocols.

1.4 Proposed support for improving extensions

I claim that neither the free-wheeling sharing of Firefox nor the overly-partitioned approach of Chrome are appropriate designs for web-platform extensions. The former provides no aid in determining or ensuring extension compatibility, while the latter is too limiting for the compelling but unanticipated extensions that have been developed.

An idealized extensible web platform permits extensions to modify the state, functionality, and appearance of client webapps in a fine-grained, pervasive manner. Interactions between extensions are possible: extensions may communicate, may adapt to each other's presence, and may modify the mainline webapp

in benign ways. Extensions must have a mechanism for controlling their composition to resolve conflicts, either automatically or through user intervention.

Defining the notion of extension compatibility must account for the inherent differences between imperative script extension and declarative UI extension. Perhaps unsurprisingly, the latter will be much easier. For both facets of extension, I focus on *commutativity* as a first-order approximation for extension independence: if two extensions "produce the same effect" on the webapp regardless of their execution order, they might reasonably be considered independent of, and thus compatible with, each other. I expect many unrelated extensions to commute with each other. But when one extension relies on the presence or absence of another, commutativity is insufficient.

1.4.1 Necessary platform support

Unfortunately, no current browser implementation is an appropriate vehicle for research and experimentation on extension mechanisms themselves. Firefox is tightly wedded to XUL and the particular idiosyncrasies of its overlay mechanism; replacing or modifying these would require rewriting Firefox to accommodate the changes. Chrome has deeply ingrained notions of security principals and limits on extension abilities that are coupled to its treatment of websites; revising these policies would likewise require a large overhaul of Chrome's internals. And all other browsers are saddled with legacy codebases that are not easily amenable to adding hooks for extensions.

Instead, to facilitate the research in this dissertation, it was necessary for me to start from scratch and write a *new* implementation of the web platform, C3, that will be described in detail in Chapter 3. C3 is designed from the outset to redress the challenges listed above: it is highly modular, and therefore less explicitly dependent on particular mechanisms or security policies. At the same time, the default implementation includes several mechanisms for extending many parts of the system: among others, overlays for HTML to extend the structure of web pages or the browser chrome itself, and aspects for JS to extend their scripts; both of these are outlined below.

Crucially, these extension points are tailored for analyzability: this thesis develops several analyses for overlay compatibility, and proposes others for aspects. Additionally, the design of the extension mechanisms can easily be further refined to support additional guarantees. For instance, the current implementation presumes that overlay- and aspect-based extensions can extend one another for maximum flexibility, but in practice extensions may want to "seal" portions of themselves to safeguard their behavior against further extension. In this thesis I have chosen one form of such safeguarding, and future work may indicate better such forms; C3 is designed to make such experiment-driven revisions easy, with little modification to the rest of the platform.

1.4.2 *Code extension via aspects*

Extensions change behavior by overwriting existing code to call new functions in the extensions instead. Currently this is done by redefining top-level functions or rewriting code using `eval`, both of which are hard to analyze and have semantic problems. In Chapter 4, I define an aspect-oriented system for JavaScript where these code-injection points are made declarative and explicit. While that chapter focuses on designing and implementing the aspect primitives, I highlight a key benefit here: if all extension points used by extensions are declaratively specified, then it is much easier to detect a broad (but not exhaustive) class of conflicts by checking if two extensions' aspects advise the same targets. while this benefit is likely of great help in practice, it nonetheless has two fundamental limitations.

First, extensions may have disjoint effects on the code, yet still cause behaviors that are broken "at a higher level". Consider the following code:

```
var x = 1;
function f() { x++; }
function g() { x++; }
function h() { f(); g(); assert(x != 6); }
```

And consider the following two extensions:

1. `f = function() { x *= 2; }`
2. `g = function() { x *= 3; }`

While both extensions change different pieces of code (`f` and `g`, respectively), if both extensions are installed, then the assertion in `h` will fail—and such a failure would not happen if at most one extension were installed. It is beyond the scope of this thesis to attempt to detect such application-specific semantic problems.

Second, since `JS` is a Turing-complete language, extensions' code cannot easily be analyzed for conflict because whether extension points *alias* can depend on arbitrary run-time behavior: in the previous example, it is non-trivial merely to determine whether `f` and `g` are distinct. Moreover, pairwise analyses may not be correct for larger sets of extensions. For example, suppose the baseline browser has three distinct functions bound to `a`, `b` and `c` and consider the following three extensions:

1. *At some point, extend `a` to always return 42.*
2. *At some point, extend `b` to always return 53.*
3. *Extend `c` to set `a = b`.*

Any two of these extensions are compatible with each other, as they advise different functions. However, if all three extensions are installed together, if extension 3 loads first then extensions 1 and 2 *might* conflict, depending on whether `h` happens to execute before the other extensions install their advice. Thus precisely detecting this type of conflict is undecidable statically.

Consequently, the only program with perfect information about installed extensions is the user's browser itself. Only it can correctly analyze running extensions and raise warnings if, when weaving advice into a function, the advice bodies do not commute: some alternate weaving order might yield different overall program execution. This dynamic approach will certainly yield more detailed information than is currently available, because it can track which extensions installed the advice, and so be of use to extension developers. However it may not be very helpful to end users, as the only actions available to them at runtime are either to abort the offending extensions or permit an unanticipated action, neither of which may be palatable.

Far better would be to detect statically, i.e., at extension installation time or earlier, whether several extensions conflict. The example above of racing to apply extensions is admittedly far-fetched: perhaps an unsound analysis could assume such strange code isn't used, and pragmatically detect valid weaving errors. Additionally, Douence et al. [59, 60] have made progress in analyzing aspects without reference to a base system. They define notions of strong and weak compatibility among aspects, which may be enough in practice to make an approximate static dependency analysis feasible, or to decide exactly what runtime checks are needed for a sound analysis. (I discuss this and other related efforts in more detail in Chapter 2.)

Finally, as mentioned earlier, extensions may want to "seal" some of their functionality against modification by other extensions. I am not the first to notice this; Aldrich [6] proposed the notion of "open modules" to achieve such sealing, and the technique should fit well with my own aspect work and the browser setting. In this dissertation, I address only the language-implementation challenges of integrating aspects into JavaScript. My implementation makes it easy to seal a function against further advice. I have not yet implemented the dynamic compatibility analyses alluded to above; they are left for future work.

1.4.3 UI *extension via semantic overlays*

As a reasonable minimum requirement, the platform must ensure that the net UI effect of a set of installed extensions is defined solely in terms of the elements of the set: it should not depend on installation order, or naming conventions, or anything external to the extensions themselves. Therefore extensions must provide sufficient information to specify uniquely their loading order

up to commutativity: for each extension, which other extensions must precede or follow it, and which other extensions do not matter?

The design for overlays presented in this thesis starts in a manner based upon Firefox's overlays, by permitting extension authors to inject new content into specified locations in the base document, and generalizes the selection mechanism for additional flexibility. Next, they permit extension authors to seal portions of the overlay against further overlaying. Finally, extension authors specify a specific composition order for the various overlays in their extension, which includes details on which overlays may be optional, or which overlays are mutually exclusive variations that target different versions of the base system. These abilities to guard overlays or to specify composition order are missing from Firefox's approach. Ultimately, the design presented here identifies each extension as a single composition, which must as a whole succeed or fail.

Determining whether a set of compositions is internally compatible happens within the platform, as it launches. A *document* is a base document followed by an ordered sequence of compositions. I abstract the *state* of the document as four lists describing which requirements 1) must be defined in the document, 2) must be undefined, 3) have not yet been overlaid by a composition, or 4) must never again be overlaid, corresponding to the four guard types g. Using this machinery, I can interpret compositions c as *document transformers* that relate an initial document state before c is applied to a final state afterward, assuming the application succeeds. These abstract transformers are used to define a notion of when two compositions commute: if the output state of composition c_1 cannot satisfy the needed input state of composition c_2, then c_2 must precede c_1 and they do not commute. Any compositions not related by (chains of) such dependencies commute with each other. Such dependencies can then be used to find a valid loading sequence, if one exists, or find a set of conflicting extensions, if one does not.

Note that extension authors can modularly specify their compositions without knowing about any other extensions. Further, the compatibility analysis can be done by the browser without input from the user, and can simply inform the user (or developer) whether installing a new extension will yield a compatible composition or not. Thus in theory, with minimal developer overhead and no user effort, and without restricting UI extension points, I can solve the problems with Firefox overlays described above. In practice, defining the commutativity of compositions can be much harder, depending on the expressiveness of the overlay language. As shown in Chapter 5, the full language with all the composition operators and where targets may be described by arbitrary CSS selectors, the approach described above gradually breaks down, yielding first approximate and then incorrect results. I define the approach above more carefully in Chapter 5, and in Sections 5.7 and 5.8 I explore the failure modes of this algorithm.

1.4.4 Enforcing security policies

I have said nothing about enforcing security policies on these well-composed extensions; in fact, this dissertation will completely ignore security concerns for extensions. Instead, I see all of the challenges above for simply defining a clearer semantics for extensions and compatibility as being prerequisites to security efforts.

Chrome's extension security model takes a capability-based approach, wherein extensions declare which resources they require (e.g., "browser history", "network access to `foo.com`") and users grant those permissions at extension install-time. I do not yet address such policies, except to note that when extensions may interact freely (unlike in Chrome); more work must be done to prevent a confused-deputy attack or outright collusion between extensions; this extends the threat model in [16]. Additional care must be taken to protect such a composite system from runaway extensions, as in [84].

1.5 Summary

I have described the three main layers of the "web stack", and the types of extensions applicable to each layer. In particular I have described the space of webapp/browser extensions, and argued that they are in need of programming-language research. I have sketched the essential features of two browser extension systems, and identified key strengths and flaws of each. Finally, I have proposed two language mechanisms for improving the development of extensions, and suggest that together they provide a more compelling platform for supporting extensions.

The rest of this dissertation fleshes out these claims and sketches, and is organized as follows. In Chapter 2, I examine the space of *extensibility* in broad, and define a vocabulary for discussing *extension models*. Chapter 3 describes the implementation of C3, a new implementation of the web framework (HTML, CSS and JS) that is engineered to support easy extensibility and research experimentation throughout the system. Chapter 4 defines the aspect-oriented extension to JS described in Section 1.4.2, which I have implemented in C3, and evaluates both its performance and its effectiveness in streamlining actual Firefox extension code. Chapter 5 defines the overlay language from Section 1.4.3 and conflict-detection algorithms over that language, which have also been implemented in C3, and examines a survey of Firefox extensions to detect potential errors. Finally, Chapter 6 summarizes the contributions of this thesis and positions several promising avenues of future work. The technical details of proofs are relegated to Appendix A, while the precise formalism of the conflict-detection algorithm, and a step-by-step walkthrough of it, are in Appendix B.

Chapter 2

DEFINING AN EXTENSION MODEL

2.1 *Defining extensibility*

The preceding chapter claimed that the current state of extensibility in web browsers is insufficient, and that improvements must be developed. This chapter explores the mechanisms and failings of current browsers' extension approaches in more detail, and draw inspiration for their improvement from widely disparate areas of previous research. Sections 2.2 and 2.3 lay out crucial background information for the remainder of this dissertation. Section 2.4 introduces terminology and a classification scheme for defining *extension models*. Sections 2.5 to 2.8 discuss various extension models in related areas, and can be skimmed and re-read in any order relative to the following chapters. Section 2.9 contrasts these areas with the web platform area, and Section 2.10 summarizes the chapter.

2.2 *Extensibility in web platforms*

As sketched in Section 1.2, a browser is a multi-layered architecture, but it is more instructive to view it as a two-layered webapp running atop a two-layered platform. The browser itself is responsible for running individual sites at the *website level*, where each site is properly isolated from each other; those sites are hosted within the *browser level*, which uses the familiar chrome idioms of the browser to enable user interaction with the sites. That chrome—essentially the application logic of the browser—relies upon all the functionality of the *platform level* beneath it to parse and render the HTML and CSS of the sites, to execute their JS code, to handle all network or other resource requests, and even to enforce security policies. In some browsers, such as Firefox, the chrome of the browser is itself coded like a website, so that the same rendering and execution engines used to display and protect individual sites are reused to display and protect the browser chrome itself. Consequently, from the platform's perspective, the browser and website levels are really just a single *webapp level*—though obviously not all webapps require or employ such a distinction.

This perspective, that a browser is merely one webapp instance, is certainly not the commonly held view: surely webapps are things such as Gmail or Office365 that are run *in* browsers, not *instead of* browsers! But this is a historical accident, as it is already becoming clear that typical

application interactions are not easily shoehorned into page-based back/forward navigation of URLs. Many webapps have no real need of the browser's chrome, and indeed projects like Mozilla Labs' Chromeless [169] can run such applications "outside" the browser, i.e., directly on top of the platform level. And at a software-engineering level, Firefox partially embodies this split, by building upon Gecko, Mozilla's platform layer that implements support for HTML/CSS/JS as well as XUL, the XML UI Language in which all Gecko-based applications (among others, Thunderbird, a mail client; BlueGriffon, a WYSIWYG HTML editor; Songbird, a music manager) construct their UI.

The primary advantage to this perspective is that each layer supports its own characteristic extension mechanisms—some of which are well-known, and others are novel and will be elaborated in Chapter 3. Moreover, the kinds of extensions applicable to web browsers and to websites use *identical* mechanisms, rather than their current unnatural split into separate technologies. In the discussion that follows, *extensions* are defined to be additional pieces of code that are downloaded individually and then dynamically merged into the base platform.

2.2.1 The extension development model

Extensions are written by *third-party* developers, who only have publicly available information on which to base their code. For closed-source browsers, this information may be nothing more than documentation on extension APIs; for open-source browsers, the code itself is available. Beside source code access, developers may be limited by the extension mechanisms themselves: Chrome's extensions are far less able to modify Chrome than Firefox's extensions are.

Extensions are used by arbitrary people, who may have zero knowledge of how the browser or extensions are implemented. Extension users typically install a handful of extensions simultaneously [194]. They expect to install extensions into their browser, perhaps setting some options, and then simply have the extensions work compatibly with one another.

An idealized extensible browser permits extensions to modify the state, functionality, and appearance of the browser in a fine-grained, pervasive manner. Extensions are written by amateur developers external to the browser, and should require minimal cooperation from the browser to run successfully. Interactions between extensions are possible: extensions may communicate, may adapt to each other's presence, and may modify the mainline browser in benign ways. Conflicts may occur when extensions try to modify each other or the mainline browser in incompatible ways, by changing UI *or invariants upon which the other relies. It is not acceptable that extension conflicts first be found by end-users at runtime, so conflicts must be detectable by load-time and preferably at install time. Extensions must have a mechanism for controlling their composition to resolve conflicts, either automatically or through user intervention.*

Any discussion of extensions therefore must be framed in terms useful to these two groups of people: understanding what extensions may *do* must be defined by the publicly available information about the browser's implementation and its extension mechanisms, and understanding how extensions interact must be determined without any input from users beyond which set of extensions are installed simultaneously.

2.2.2 *The platform level*

The lowest level of the web platform is essentially a collaboration of file type-specific "interpreters" that combine to render content to the display and respond to user interaction. By default, explicit interpreters exist for HTML, CSS and JS. Implicitly, interpreters exist for the supported image formats; in browsers supporting HTML5, additional ones exist for supporting ⟨**audio**/⟩ and ⟨**video**/⟩ content. At any given time, each file format is handled by exactly one interpreter.

As long ago as Netscape Navigator 2.0, browsers have supported extension in the form of *plugins*, binary modules that add new interpreters to the platform's collaboration: the granularity is fairly coarse. Common examples of browser plugins include Adobe Flash Player, Microsoft Silverlight, Apple Quicktime, and many others. Support for these plugins comes from two standard APIs, ActiveX (for Internet Explorer) and NPAPI (for every other browser), that define the (intended) interactions between plugins and the rest of the browser. These plugins run in the browser process, and while they should behave according to the specified interfaces, they in fact are unconstrained in their behavior. They are analogous to kernel drivers in commodity OS designs: like drivers, a poorly-written plugin can destabilize the entire browser. And just as OS projects like SPIN [23] and Singularity [108, 109] try to tame drivers by corralling them behind interfaces or separate processes, a new plugin API is being developed [93] to push these plugins out of the browser kernel into separate processes. Regardless of the API, as with the default interpreters, only one plugin can be active for a given file type at a given time, which neatly avoids any conflicts between well-behaved plugins.

Depending on how the platform is architected, a finer-grained extension may be possible. As I will demonstrate in Chapter 3, the default interpreters within the platform may themselves be extensible: an extension might extend the languages recognized by the interpreter, adding support for new HTML tag names, new CSS properties, or new layout routines (though only the first of these is currently implemented). These extensions are intended to interact in the same manner as the underlying components: layout routines collate tags and their styles and render them in various ways. Importantly, multiple HTML tag-name extensions, say, are *not* intended to interact: all such

extensions must contribute disjoint sets of new tag names. Confirming that these extensions are in fact non-conflicting can be detected as they are loaded into the platform.

2.2.3 The webapp level

Webapps and web pages are written in HTML, CSS and JS, and therefore extensions at this level must be written within those languages as well. The HTML tree is exposed to JS via the Document Object Model (DOM), a suite of APIs for manipulating that structure. Since the DOM is mutable, it is always possible to inject new nodes, including ⟨**style**/⟩ and ⟨**script**/⟩ nodes, into a page—at which point they execute as if they had been part of the page ever since it was loaded. Extensions at this level simply amount to idiomatic ways of causing these injections.

Userscripts and bookmarklets: Like user-mode applications in an OS, web pages from different origins are isolated from one another. This level supports two closely related kinds of extensions: *bookmarklets* and *userscripts*. A bookmarklet is a bookmark containing a snippet of script that, when selected by a user, runs in the context of the current page. Userscripts do the same, but are activated automatically upon navigating to a page that the script selects as relevant. These extensions are analogous to, and as powerful as, remote thread injection in operating systems. If multiple bookmarklets or userscripts run on a page, they may conflict with one another and break the behavior of scripts on that web page, but cannot interact with or do any harm to the browser as a whole (assuming proper page isolation [182]). All modern browsers support bookmarklets; nearly all have support for userscripts either built into the browser or (in an ironic technical twist) patched in as an extension.

At a finer granularity, injected scripts may wish to modify or patch *existing code* that has already been loaded into a page. This is made possible due to various quirks of JS, including its ability to rebind functions at runtime (known as *wrapping*) and to manipulate functions' source code and to `eval` the resulting strings (known as *monkeypatching*). Unfortunately, these two idioms are semantically broken and cannot work in all cases. As I will explore in detail in Chapter 4, a modest enhancement to the JS engine to support dynamic aspect weaving (note: not an extension, in the technical sense being used here, but rather a new version of the engine) can provide a semantically robust mechanism for such code injection that avoids these faults.

Overlays: Much as userscripts attempt to patch the existing JS of a webpage, *overlays* attempt to patch the existing DOM of the page. As described in Section 1.3, overlays are a declarative (i.e., markup-based) way to select nodes in the document and insert new content into them. To date,

the only implementation of overlays is in XUL, which in turn is supported only in Firefox. As I will explain briefly below, and in more detail in Chapter 5, this implementation has a poorly defined semantics, which makes it particularly problematic for determining extension compatibility. Firefox's overlays are very fine-grained, as they can patch individual nodes, though they may be *too* fine-grained, as they cannot patch *groups* of nodes in a uniform way. Additionally, where overlays can interact by patching the same target nodes, there is no facility in Firefox for controlling the patch order (and therefore the resulting document order) to enforce desired ordering constraints.

2.3 Extension mechanisms in existing browsers

Firefox supports two extension formats: "traditional" extensions that are powerful, flexible and potentially error-prone, and newer "jetpacks" that are simpler to write, less powerful, but less error-prone. Chrome's extension model is closer to that of jetpacks, but is far more focused on sandboxing extensions securely. Internet Explorer supports a few *ad hoc* extension points, and Opera and Safari both support limited extensions that are a subset of Chrome's approach's functionality. I describe Firefox's, Chrome's and Internet Explorer's extension mechanisms below.

Traditional Firefox extensions: Traditional Firefox extensions consist of three primary pieces: a set of XUL overlays that define new UI content to be appended to existing UI that itself is defined in XUL, a set of JS files that define the functionality of the extension, and a simple manifest that declares some minimal metadata about the extension to Firefox. XUL, like HTML, exposes a DOM to scripts that manipulate it. Extensions can therefore define new UI with as rich capabilities as Firefox itself, and their scripts can manipulate both new and existing content. Additionally, Firefox exports a very wide API for internal functionality, which extensions can use to modify the browser's behavior.

Firefox extensions enjoy a uniquely powerful relationship with their host browser. While many of the extension mechanisms considered here are permitted to define limited forms of UI (e.g., toolbars, badges, or context-menu items), and some are capable of arbitrarily modifying existing UI (e.g., ActiveX controls), only Firefox extensions are given a sanctioned mechanism—overlays—to modify any portion of Firefox's UI. Accordingly, any discussion of web-platform extensions' design must be careful to distinguish UI-specific architectural choices from functionality ones, especially when considering abilities of and conflicts among extensions.

Firefox extensions' ability to manipulate the host browser's functionality is similarly unique: most of the extension mechanisms here are permitted modify some of the behavior of their host browser, and some may use unsanctioned (and unstable) methods to modify other behaviors, only

in Firefox is the vast majority of the browser's behavior deliberately available—exposed via JS—for modification and extension. Extensions have been used to modify core browser policies (e.g., blocking script execution), and to add new abilities (e.g., adding integration to various web services).

Extension authors are afforded an erratic level of cooperation from Firefox, with strong support for UI extension and weaker cooperation for code extension. A technical restriction in the implementation of overlays permits only XUL elements with identifiers to be extended: on the one hand, overlays would be effectively impotent without extensive cooperation from Firefox; on the other, giving identifiers to (nearly all) page elements is standard practice in HTML and so is no onerous burden. Further, no explicit support is given for extending Firefox's code; because JS is so mutable at runtime, at first glance it seems none is needed. But precisely because JS is so mutable, it is exceedingly difficult to describe exactly what interactions may occur between the code of multiple extensions.

The actual process of integrating overlays into Firefox's UI occurs at load-time. Code extensions may be integrated at any point after loading, since JS makes no essential distinction between load-time and runtime. Firefox only implements very rudimentary conflict checking: an extension's manifest can declare with which versions of Firefox it is compatible, and which other extensions it depends upon for successful execution; these checks can be enforced at extension installation and load time. However, all interactions over UI, code or execution state are not detectable until runtime, by users who likely cannot debug problems they encounter. Improvements to this status quo are the main contributions of Chapters 4 and 5.

Firefox permits users to install multiple extensions simultaneously; extensions load in some unspecified order as Firefox loads. Interactions between extensions can occur via code or overlays. Extensions may modify each other's UI, dynamically change each other's code, or interfere with each other's state. (These last two are particularly troublesome: since JS has no mandatory namespace mechanism for modularity, poorly-written extensions pollute the global namespace, which can silently redefine code added by other extensions. Recent community-driven standards such as CommonJS [47] attempt to define idioms and APIs to redress this lack.) Some of these interactions are intentional: for instance, TabMix Plus and Session Manager both implement saving and restoring of browser sessions (open windows and tabs), and the former changes both extensions' UI to enable exactly one or the other's ability, as both managers running simultaneously would break. However, many extensions have inadvertent conflicts: FoxyTunes inserts a small Flash object to play MP3 files directly from the browser, while FlashBlock eponymously disables this functionality. Other, more subtle and often surprising interactions are possible, due to quirks in Firefox's architecture.

Jetpacks: Mozilla Jetpack [165, 178–180] is a recent project to simplify the development of addons for Firefox. It aims to address several major pain points in addon authorship, particularly providing a stable API to use that will not change between versions of Firefox. This becomes even more crucial when rapid development and release cycles change the browser UI and internals more rapidly than extension authors can keep pace [213]. Additionally, Jetpack addons are *restartless*: they can be added and removed from the running browser without restarting it [212, 214].

Achieving both of these goals requires sacrificing much of the power of the traditional extensions: in particular, jetpacks do not include overlays. Instead, Jetpack provides an API for constructing common pieces of UI, such as context menu items, status-bar icons, new tabs, etc. Additionally, as it currently stands Jetpack asks extension authors to correctly write `main()` and `onUnload()` functions that are true inverses of each other, if `main()` does anything stateful that goes beyond the declarative Jetpack APIs. For example, while Jetpack lets you declare a new context menu item that it will automatically insert and remove as needed, with no need for `onUnload()`, if a jetpack uses DOM manipulation methods to insert content directly then it will require a purpose-built `onUnload()` to remove that content.

Fortunately, many simple extensions can comfortably use the APIs and not require much additional support. Consequently, jetpacks have eliminated one major source of conflicts between traditional extensions. However, the Jetpack APIs are limited (though growing steadily); as such jetpacks can never be as surgically targeted or as powerful as traditional extensions can be.

Chrome extensions: Chrome extensions are split into two or three pieces: one "background page" where the main extension logic occurs, userscripts injected into the content of relevant web pages, and an optional binary component for specialized computation better suited to native code. Both the userscripts and the binary component are given narrow message-passing APIs to communicate with the background page. At a high level, the APIs available to the background page of Chrome extensions look very similar to those of Jetpack (which indeed modeled itself on Chrome's approach): a set of limited but robust APIs for manipulating portions of Chrome's UI, e.g., adding toolbar buttons or context-menu items.

What distinguishes Chrome's approach from traditional Firefox extensions is that in Chrome, extensions are fully sandboxed from one another, from the browser, and from the pages being rendered. Each background page is considered a separate origin for purposes of the same-origin policy, which immediately implies that even if scripts from two extensions could see each other's background pages, they would not be able to manipulate each other. Additionally, if one extension were to crash, hang, or otherwise "go wrong", other extensions would not be affected. Similarly, if

scripts on some content page forced that page to go wrong, extensions could continue to interact with other content pages.

This sandboxing also distinguishes them from jetpacks: while their APIs are similar, Chrome extensions cannot use any platform functionality *except* those APIs. The primary benefit of the sandboxing is to provide the user with per-page and per-extension performance isolation. In this regard they are more advanced than either of Firefox's approaches. Again because of their sandboxing, extensions cannot extend each other, so interactions—and hence conflicts—are automatically minimized.

Internet Explorer: At the platform level, Internet Explorer has supported ActiveX controls (binary plugins) since version 3.0, so-called "browser helper objects" (similar to ActiveX controls but with a slightly different API) since version 4.0, and even supported the competing NPAPI plugins through version 5.5. These extensions were all fully-privileged native code that ran in the same process as the browser itself, leading to frequent crashes from faulty plugins. Version 5.0 formalized APIs for customized toolbars and context-menu entries, but these too were over-privileged native code.

Finally, Internet Explorer 8 introduced two new forms of extension, Web Slices and Accelerators [124, 158, 162], which are quicker ways of interacting with web content and provide the same functionality as many common plugins in a far more stable and secure manner. Analogues of both abilities have been implemented as Firefox extensions [77, 91, 164]. Both Accelerators and Web Slices offer limited extensibility: authors can define short `XML` files which add support for a new accelerator or slice; these extensions cannot interact with each other at all.

Internet Explorer does not include support for userscripts, though several (now defunct) ActiveX controls were written to create such support.

Security concerns: There are two primary security concerns relating to extensions: whether extensions can compromise the security of each other or the security of the browser.

In systems where extensions can communicate with and potentially extend one another, it is reasonable to ask whether extensions can safeguard their own integrity: can they ensure that other malicious or buggy extensions cannot compromise their own code and state? Currently, no extension system supports this: either extensions cannot communicate, as in Chrome, or they can trample each other's code freely, as in traditional Firefox extensions. Jetpacks pose an odd middle ground: by using the CommonJS framework, it becomes nearly impossible for one extension to modify another—but using that framework is optional and partial; jetpacks might still define

functionality without using the framework. It remains to be seen whether such opt-in behavior will be sufficient in general, or whether more robust enforcement mechanisms will be needed.

Beyond manipulating each other's code, extensions can violate security guarantees of the base browser. For example, all modern browsers support a "private browsing" or "incognito" mode, which is intended to leave no trace of a user's browsing on the computer once the browser session is closed. However, extensions have access to several means to persist state, including HTML 5 local storage, file-system APIs, or remote calls to servers. Because of these many information channels, no browser currently enforces that extensions automatically abide by the goals of incognito mode; in fact, the documentation for both Chrome and Firefox extensions include APIs to check whether incognito mode is active, and admonish extension authors to respect that flag by not recording sensitive data.

2.4 Defining extension models

The preceding sections described the current state of the art in browser extensibility. But extensibility is everywhere, in varying degrees. On the minimalist extreme, simple music players admit additional codecs to support new encoding formats, but for example no extension may define streaming media support when no prior internet functionality exists. More flexibly, office suites embed scripting languages that can encode fairly powerful computations, but for example cannot add support for new file formats. On the far extreme, some systems are practically nothing but extensions—other than a small runtime core, the emacs editor is a default collection of macro packages, written the same way as the non-default packages known as extensions.

The remainder of this chapter chapter examines the *extensibility problem*—understanding how extensions interact with one another and how best to resolve conflicts that arise—by examining prior work in four other areas of research: aspect-oriented programming and systems design, which focus primarily on defining extensions; and feature specification and security monitors, which focus mainly on resolving conflicts. (As this dissertation focuses primarily on defining extensions, it draws more heavily on the ideas from the former two; the latter are more useful for future work.) To frame the discussion, I introduce a set of criteria against which to describe an extensibility system. Previous efforts have classified system-specific criteria for operating systems [54, 200] and feature specification [122]; my criteria generalize and combine these to apply to all systems examined here. In this chapter, extensions will be examined based upon their:

- **Design considerations**: What *behavior* does the extension have—e.g., new policies, better performance, new functionality? Alternatively, what *semantics* are designed into the exten-

sion system? Who is the *author*—e.g., the base system's designer, an external developer, or an amateur? When is the extension *integrated* with the base system—e.g., at design, build, install, load, or run time? Finally, how much *cooperation* does the extension need from the base system—e.g., can the base system be written obliviously to future extensions, or must it accommodate them during its design?

- **Extension abilities**: What base system *resource* do extensions target—e.g., the set of supported hardware, or the set of security policies? Given that, how *pervasive* is the extension—how much of the system may be modified? How *granular* can extensions be—e.g., can they replace single lines of code, or only entire subsystems at once? In what ways can extensions *compose*? Finally, what *interactions* are possible between multiple extensions or between each other and the base system, and what guarantees can be made when no features interact?

- **Troubleshooting techniques**: What *conflicts* are possible—i.e., which interactions are undesirable? When can conflicts be *detected*—e.g., at design time, via runtime testing, or via user problems? Can individual extensions be checked *modularly* for conflicts? Finally, how are conflicts *resolved*—e.g., by restricting the action of the extensions, by manually composing them into a corrected composite, or by rewriting the extensions to cooperate?

The next four sections discuss aspect-oriented programming (Section 2.5), operating systems and other platforms (Section 2.6), feature specification (Section 2.7), and security monitors (Section 2.8). The breadth of each area precludes defining *the* extension model for each domain, so I first give an overview and a strawman extension model of the four systems types mentioned above. I then describe salient research efforts that explore different facets of the organizing criteria above. Section 2.9 relates particularly useful efforts back to the motivating area of extensible web platforms. Section 2.10 summarizes the contributions of this chapter.

2.5 *Aspect-oriented programming*

Fundamentally, an *aspect* extends the control flow of an existing program by declaring *what* additional work must be done *when* certain conditions arise. The primitive hook for extension within AOP is the *joinpoint*, which describes an "interesting" moment of control-flow such as a function entry or exit. *Pointcuts*[1] define groups of joinpoints, usually via simple set operations. Specifying *what* action to take at a given pointcut is known as *advice*. Multiple pointcut/advice pairs may be

[1] Occasionally known as *crosscuts* [58], though that term is more often used as an adjective [186]; I avoid using it further.

combined to define a single aspect, which is defined compactly and separately from the *mainline* code (i.e., the non-aspect remainder of the program), until being *woven* together at compile time. Advice may apply *before*, *after*, or *around* a pointcut, though these forms may be simplified: as presented in Walker et al. [221], these can all be compiled to idioms of joinpoints, where advice is executed *at* the moment the joinpoint is reached. While AOP looks similar to event-based systems, AOP permits replacing the control flow of the advised program, which events cannot do. Concretely, a minimal aspect in AspectJ might define just one pointcut and one piece of advice:

```
public aspect HelloMain {
   pointcut runningMain() : execution(public static void main(String[]));
   after() returning : runningMain() { System.out.println("Now leaving main"); }
}
```

An idealized AOP *system permits extensions to modify the state and control flow of the mainline program via aspects that advise function calls. Advice is authored by the same developers as the mainline system, and is integrated at build time. The mainline code need not cooperate to enable advice, though care must be taken for the advice to work properly: conflicts may arise when multiple advice apply to a particular pointcut (detectable by the compiler), when aspects overlap in function, or when changes in the mainline break an aspect (detectable only by testing); all require programmer intervention. When all aspects are compatible with each other and the mainline, the semantics of the woven program is predictable and independent of the weaving order (modulo any explicitly specified ordering constraints.)*

The above definitions are very much simplified; each part of these definitions can be varied, yielding systems with greater or lesser flexibility—and impact on ease of analysis for conflict. The set of available joinpoint kinds (granularity) differs considerably between AOP implementations; while AspectJ contains a predefined set [123], they could be programmer-specified labels interspersed anywhere within the code [221]. Moreover, while AspectJ's joinpoints are focused on OO-style code (e.g., a special joinpoint for object constructors), there is nothing inherently object-oriented about aspects, and indeed aspects can be formalized for functional languages as well [49, 50, 58, 221]. Similarly, the flexibility of pointcut specification differs between implementations (e.g., [32, 58, 62, 123, 221]), and is a key factor in the utility of the AOP approach. More complex systems permit pointcuts that can examine the stack [48, 123]; others permit state machine-like inspection of the history of the computation. AOP languages may choose whether aspects are second-order, syntactic constructs baked into the language [123], are extensible but still second-order language constructs [32], or are first-order values definable and manipulable within the language itself [62]. Advice can be dynamically installed for a given scope, or lexically inserted at compile time. One point of common disagreement appears to be the appropriate weaving order

for aspects. Aldrich [6] applies advice in a LIFO order, while Dantas and Walker [48] take the exact opposite approach. AspectJ [123] defines a complex (and incomplete) set of rules governing weaving. Other systems (e.g., Douence et al. [60]) permit programmer-specified weaving orders. The only commonalities seem to be the need for clearly specifying the semantics of the system under study, and to choose the appropriate heuristic for the task at hand. Finally, the pervasiveness of the advice itself can be varied, by varying the environment available to the advice.

2.5.1 *Language design*

Common wisdom [81] suggests that aspects are most effective when the mainline code is *oblivious* to the effects of the aspects, i.e., written without consideration of or accommodation for future aspect extension, though this is somewhat oversimplified [106], and demands understanding what effects aspects can have on a program. There is extensive work on formalizing the foundations of aspects [5, 44, 49–51, 58, 62, 106, 107, 117, 147, 190, 195, 215, 221, 224, 225]. Several of these try to apply aspects to a typed (e.g., [49, 51, 147]) and/or higher-order functional [62, 153] setting. The former, while technically demanding, ensures that well-typed aspects cannot cause well-typed programs to go wrong. The latter adds enormous flexibility by permitting the definition of new forms of pointcuts as needed, though it makes compiling and optimizing the woven code much harder. At the moment, JS is dynamically typed, though proposals for ECMAScript 5 and beyond are looking to add some form of static typing to JS. Coupled with JS's support for higher-order functions, these research efforts provide a useful background for future aspect-like mechanisms for JS. There is also work that examines the semantics of extending one language with multiple aspect mechanisms themselves [127]. This level of extensibility does not bear on the browser extensibility model, and will not be addressed further here.

2.5.2 *Safe* AOP *idioms*

There are several efforts to define safer idioms of programming with aspects [6, 17, 48, 118, 174, 186]. This line of work seeks to place limits on either pointcuts [6] or advice [48] to restrict their unprincipled tampering with the mainline program. These two approaches roughly correspond, in other systems settings, to narrowing the client API and to restricting permissions on client actions.

Aldrich [6] takes as his starting point the fragility of pointcuts: by hooking into the control flow of a program, they necessarily rely on that structure remaining unchanged across versions. Moreover, such deep hooks prevent most local reasoning about abstraction boundaries. To address this, Aldrich restricts his set of primitive joinpoints to include only call sites of *declarations*, rather

than of all functions, a distinction that exposes only named, exported functions to advice, while anonymous lambda expressions and functions hidden behind module boundaries are immune. Within a module's definition, all call sites are available as joinpoints; "open modules" can choose to expose some of these otherwise-hidden joinpoints as part of their signature,[2] making them targetable by external advice. External calls to functions in the signature are advisable; intra-modular calls to those functions are not, unless they are exposed through an explicit addition to the module signature. Doing this makes the exposed joinpoints "a part of the API", implying they will be stable in the face of future internal changes. Note that this comes at a price: the module author is no longer oblivious to the potential for future advice.

This stability is formalized by Aldrich's treatment of the *equivalence of modules*, which for my purposes is the key contribution of this paper. A revision of a module is equivalent to the original when it does not break aspects that previously worked. More formally, "equivalent functions must not only produce equivalent results given equivalent arguments, they must *also* trigger advice on client-accessible labels in the same sequence with the same arguments"—revisions must preserve the behavior of their declared joinpoints, and the social conventions regarding API change come into effect. The essence of the open module approach has been adopted, for example, by the Eclipse platform [27], where plugin authors explicitly declare extension points for future plugins to use. Additionally, extension points must be deprecated before they are removed or changed. By contrast, in current browser designs, extensions frequently break between minor version changes of the browser; while effort is made to prevent needless problems, such brittleness is endemic when extension points are not promised to be stable.

Dantas and Walker [48] take the dual approach, placing restrictions on what advice can do with a source program. They focus on when is it possible to be certain that aspects do not *interfere* with the mainline program. So-called harmless aspects may observe the execution of a program and may influence its termination behavior (e.g., they can terminate it in response to an error condition) but they cannot influence its computed results; this definition preserves partial-correctness properties of the mainline program, in the face of changes to the aspects woven into it. To achieve this noninterference result, the authors define an information flow-like type system that ascribes protection domains to code. Their goal is an integrity property, so intuitively they ascribe a high-protection domain to the mainline code and low-protection domains to the aspects; the typing rules then guarantee that non-unit (i.e., information-carrying) values cannot flow back from the advice to the mainline code.

---

[2] The use of modules, opaque signatures, and functors derives heavily from ML's module system, but includes width, depth and transitivity subtyping rules among signatures, which will not be described here.

The key contribution here is their simplification of information-flow to the aspect setting. Harmless aspects are *consumers* of the state produced by the mainline program, and the type system enforces the producer/consumer split. But while their exposition in the paper only concerns separating aspects from mainline code via low- and high-integrity levels, their framework supports a standard lattice. Such a general stratification could support the layering of aspects, guaranteeing that for a weaving order compatible with the lattice order, aspects can be protected from each other as well. The primary drawback to this particular approach is its rejection of any other kind of aspect. Other work [42, 186] developed a classification of aspects supporting both consumers and the dual producers, as well as independent and interfering varieties. In the web browser setting, extension authors typically write extensions that interfere with the mainline browser, though potentially most extensions may not write to or interfere with each other.

2.5.3 Conflict detection among aspects

Another broad trend in AOP research focuses on giving developers tools to detect conflicts among aspects or to manage their composition manually [17, 59, 60, 121, 128, 129, 186]. Aspects may conflict with each other in two ways: directly, by overlaying the same joinpoint, or indirectly, by mutating shared state in ways the other extension does not expect. (In AspectJ, advice is arbitrary Java code that can be granted read/write access to the pointcut's parameters, such as function arguments, potentially even to private members of classes. Aspects can change the dispatching behavior of a program by adding new interface implementations to classes or even by changing the class hierarchy itself [104, 140, 207, 208].) Of the two, indirect conflicts will persist independently of the weaving order (two aspects may be disjoint in their pointcuts, yet still overlap in their effects on the program state), and are considered a coding flaw rather than an aspect-related problem. Therefore, research has focused on capturing weaving-related conflicts.

Douence et al. [59, 60] have developed a framework for modeling aspects whose pointcuts change over the course of the program, and which can carry runtime-derived information through their evolution. They model aspects via finite-state machines, where transitions between states correspond to triggering of pointcuts and associated advice. A global program monitor is responsible for weaving the advice into the mainline and updating the aspects' state as the program runs. The expressiveness of their pointcut language is carefully engineered to keep certain intersection problems decidable; as one consequence, their system cannot express AspectJ's `cflow` pointcut for recursive functions.

Practically all the main ideas of these papers are applicable to the web setting; I focus here on

just two. First, they permit aspects to "change interest" and focus on different pointcuts at different times, in a history-dependent way. This is far more general than AspectJ's approach, or even the stack-based inspection examined before, and it remains orthogonal to the safe-idiom techniques described above. Additionally, in the web setting, the behavior of extensions depends heavily on user interaction, so this facility is a natural fit. Second, the authors define notions of strong and weak independence: whether two aspects are always compatible with each other, or compatible within the confines of a specific mainline program. They also note that neither independence notion is necessary for aspects to be compatible; commutativity is sufficient as well. The work in later chapters will take the dual approach; commutativity is sufficient, but other independence notions are possible too. Further, they go on to describe an intermediate independence relation, where aspects may declare a minimal set of requirements on the base program; the aspects will be compatible with any mainline program satisfying those requirements.

2.6 Operating systems and other platforms

Operating systems are the lowest-level software interface between physical resources and software. They are traditionally responsible for multiplexing those limited resources between multiple clients, and for abstracting the details of those resources into a convenient programming interface. In one sense all operating systems are trivially extensible: they permit an unbounded set of user-level programs by exposing an abstracted view of the machine. In the following discussion, I am uninterested in these applications, and focus rather on more fundamental changes in functionality, which require operating beneath the user-visible abstraction layers. I am likewise uninterested in hypervisors sitting beneath the OS, since they expose no fundamentally new challenges for extensibility than OSes do. The case for kernel extensibility has been made numerous times (see for instance [8, 65, 68, 102, 135, 139, 191]) and can be summarized by the simple observation that no closed system can be all things to all people. An extensible OS therefore must expose enough control over its internal structures to permit tailoring their behavior to applications' needs. Such extensibility must be tempered by concerns for performance and safety—if the extensibility mechanisms are too expensive, functionality would be better written in user-mode on a system with a cheaper mechanism; if unsafe, the system behavior becomes unmanageable.

An idealized extensible OS *permits kernel extensions to expand the resources (e.g., new hardware or better performance) available to the rest of the system, via modules that expose new* APIS*. Extensions are typically authored by the vendors of those new resources, rather than the* OS *developer, and are loaded dynamically when the* OS *boots. The* OS *itself must expose some hooks so modules can rely upon a standard*

interface. Modules may interact through any shared state of the OS*, and may conflict if multiple modules expect exclusive control over some resource; such conflicts are detectable only through detailed testing, and require developer intervention to fix. Compatible extensions may be loaded in any order (unless otherwise specified), and will not destabilize the system.*

This focus on extensibility leads to the microkernel approach to OS architecture: as explained by Liedtke [143, 144], microkernels should include in the kernel only that which is necessary to implement the remainder of the system; everything else need not be in the kernel. (Note that this is distinct from whether extensions may *run* in the kernel memory space or not, a choice which varies among different microkernels—for instance, device drivers are not considered part of a microkernel, but may run with kernel privileges.) Since operating system efficiency is paramount, endless clever mechanisms have been proposed to provide extensibility.

Broadly, all OS extensions, regardless of mechanism, supply a new method of interacting with some resource of the system: they extend the set of resource access protocols. The space of extensions therefore depends heavily on the granularity of the resources exposed by the existing kernel. If a kernel exposes a file system abstraction, for instance, then extending it might entail permitting new access-control mechanisms or policies for files. If the kernel only exposes a disk-level abstraction, extensions might include file system implementations themselves.

2.6.1 *Static* OS *extensions: Aspects and code management*

There is a small line of research porting the code-management facilities of AOP to OS development. In a short position paper, Fiuczynski et al. [83] observe that while several research OS projects are obviously extensions to some mainline OS, they are equally well *aspects*—each one a single concern cutting across the codebase to implement a feature. Indeed, matching my taxonomy, these extensions are build-time integrated, pervasive and fine-grained, and provide new functionality and new policies. Like all aspects, they extend the control flow and state of the mainline kernel. However, the authors lament that the current notation of these extensions as patch sets—collections of the syntactic differences between the mainline and the extension—is inadequate. These extensions are semantic units, and therefore ought to be expressed explicitly in an AOP language. Further, patches are notoriously brittle to tiny perturbations in the mainline code, and a more semantics-driven weaving mechanism (explored also in [172]) would ease extension management considerably. (In this spirit, Lohmann et al. [149] built an aspect-oriented embedded OS, and described their successes implementing various memory protection schemes as aspects—unsurprisingly, the AOP approach permitted more flexibility than patch-sets would have.)

In follow-up work, Reynolds et al. [184] examine to what extent the Linux kernel can be "unwoven" into separate aspects, and how amenable it is to extension by advice. They note that *coding conventions* weakly approximate aspects, as preprocessor directives are used to separate code pertaining to only one use-case, and optimistically suggest that an AOP rephrasing of the code is feasible. As an added benefit, by making these aspects explicit, the analytical tools of AOP (some of which were described earlier) can be applied to kernel maintenance issues, including easing the brittleness of applying extensions and identifying when two extensions might cause conflicting changes to the mainline code.

2.6.2 *The Exokernel approach: Composable, pervasive but coarse*

In some sense, any time a computer spends "running the operating system" is time better spent running user programs, and therefore the design of the operating system should optimize for unobtrusiveness and speed. Further, since no operating system can suffice for all users' needs, surely it would be sheer hubris to select some subset of needs to be addressed and declare the result sufficient. Instead, the appropriate design should therefore be to address *no* needs beyond multiplexing the hardware resources of the system to all users. This rather extreme rationale motivates the Exokernel system.

Exokernel defines an operating system as "any piece of software that the application cannot either change or avoid". It views a kernel as a meticulous switchboard operator, whose sole function is to safely multiplex physical resources [68, 69]. Everything else is policy—and therefore relegated to an extension: how to use the compute, storage or communications resources is left to a "library OS", on a per-application basis, that defines the semantics connecting the low-level hardware and the application needs. Thus a databaseOS could implement a completely customized file system, while a webOS could implement a highly-tuned network interface [88]. If necessary, a common libOS could be implemented to provide emulation of a more traditional operating system [120]. The extension model here is application-centric and very pervasive, but not very granular: a user-mode extension can do anything it wants, using a nearly-native interface to the hardware of the system, but must reimplement an entire libOS to do so. The only conflicts relevant to an exokernel are when a libOS or application hoards resources the system as a whole requires for some other purpose. To that end, an exokernel defines a *secure binding*, a combination of a physical resource name (when possible) and capabilities granting privileges over that resource to the requesting process. These capabilities are enforced by hardware as often as possible, for sheer speed, and in software when unavoidable.

2.6.3 *The SPIN approach: fine-grained and wide*

On one level, the SPIN project can be seen as an experiment in mechanisms, showing that by requiring extensions to be written in a certain way, one could build a system with easier-to-use base abstractions, comparable performance, and stronger security guarantees than an exokernel approach [23]. On another level, SPIN is very similar to an exokernel, "one step up": providing as part of its ABI some abstractions above the hardware level, but still permitting extensive customization. Philosophically, SPIN adopts the more traditional model of actual kernel-mode extensions, which yields a non-degenerate kernel extension model.

The fundamental choice in SPIN, from which the rest of its architecture falls out as a consequence, is the choice of `Modula-3` as the only supported language for writing kernel extensions. `Modula-3` is an object-oriented language that has support for modules with opaque types: in a type-safe language, values of an opaque type may be used as capabilities, and SPIN exploits this for its security mechanism. Unlike prior capability systems that required complex data structures and capability checking (e.g., Hydra [229]), SPIN represents capabilities as bare pointers and capability checking degenerates to pointer dereference. A pointer of type `Console.T`, for instance, permits the holder of that pointer to do nothing beyond calling methods expecting a `Console.T`: this is a special case of a more general *parametricity theorem* that in essence states that type-safe code cannot violate the encapsulation of opaque types. SPIN therefore discharges most safety checks *at compile or link time*, rather than requiring runtime hardware or software support.[3]

SPIN defines higher-level abstractions than exokernel does: for instance, it defines the rudiments of a thread library and provides a default global scheduler, though it does not impose a specific threading discipline to applications.[4] It also chooses not to expose physical names for memory to extensions, but rather capabilities and virtual addresses; extensions may implement memory management policies manipulating these abstractions (for instance, [189]). To surface these interfaces to extensions, SPIN defines its extension model in terms of *events* and *event handlers*. Event handlers are merely procedures of a given interface; raising an event is essentially just a procedure call. SPIN permits multiple extensions to register handlers for an event.[5] The ordering of such handlers is explicitly undefined, so extensions cannot make assumptions about the presence of other extensions or of their relative ordering. For even finer granularity than

---

[3] One runtime precaution is needed: since user code is type-unsafe, raw pointers are wrapped as *externalized references* before escaping the kernel.

[4] At least in user-mode; the kernel controls scheduling while an application thread makes a system call.

[5] In general, with multiple handlers, event dispatch bears an overhead linear in the size of handlers and guards.

handling every event of a given type, extensions may provide guards that pass along only those events they deem relevant. In short, the SPIN model is less pervasive than that of Exokernel but of much finer granularity.

SPIN defines no explicit notion of conflict, though operationally, conflicts among extensions are simply those problems caught by the compiler and linker. But other problems are left unspecified: extensions may remove other extensions' handlers for a given event, and an event's dispatcher returns the value of the last-run handler, which is ill-specified (as handlers are unordered). However, if extensions are well-behaved (for example, when extensions are application-specific or disjoint), SPIN's semantics are well-defined: faulty extensions used by one application will not cause another application (or its extensions) to malfunction.

The SPIN model seems a closer fit to the web browser space than Exokernel's does. The ability to define customized protection domains is useful, especially in an environment where the best delineations of those boundaries are not yet known [182, 183]. Additionally, since web browsers currently distribute extensions in source-code form, using a language mechanism to reduce execution overhead is a timely and practical approach. These language-level benefits shine through even more clearly when considering the Singularity architecture.

2.6.4 The Singularity approach: Fine-grained, not too wide or narrow

Singularity is a recent research effort in a clean-slate redesign of an operating system [24, 55, 108–110, 203]. It is predicated on three primary techniques: the exclusive use of type-safe languages, a first-class notion of an application, and a carefully managed channel abstraction for interprocess communication. Together, these techniques permit a robust extensibility model: both user-mode extensions (applications) and kernel-mode extensions (drivers) play by the same rules [110].

An extension in Singularity is defined by a *manifest*, declaring what resources it must be granted, what resources it would like to use, and what channels it exports for interaction. (In fact, the only distinctions between applications and drivers are merely the claim of belonging to the ⟨**driverCategory**/⟩ of extensions, and the dependency on raw hardware resources.) Code is annotated with pre- and post-conditions using Spec# [14], thereby specifying requirements down to the individual procedure level. Channels are essentially two-way pipes that are annotated with state-based contract types, which encode the expected message protocol for the channel. These declarations and types are more than documentation: they are checked by the kernel repeatedly, at compile time, at install time, and at load time.[6] These repeated checks ensure that drivers that

[6] These declarations imply a need for some form of side-by-side versioning, to ensure that older drivers still have the

cannot run properly do not compile, that drivers that cannot ever successfully load on a particular machine are never installed, and that drivers that cannot run in the current machine state are never loaded. Singularity thereby provides feedback early and often, explaining the cause of an extension failure before it causes a crash. Essentially, the manifest becomes a model for the extension, making it a "self-describing artifact" [203].

For all that it formalizes, Singularity does not explicitly define the notions of extension compatibility or conflict. From the system invariants, compatibility among extensions amounts to requiring only resources already available from the system and existing extensions, and using those resources in type-safe ways. All extensions in Singularity are *sealed*, prohibiting runtime code generation or injection of data into a running process. Therefore *all* communication between processes takes place via channels. Forcing this communication pattern ensures that the temporal guarantees from the types apply in all circumstances. Thus type-safety here is a stronger claim than it is in SPIN. Note that while this does not prevent logic bugs from making extensions handle inputs incorrectly, it does ensure extensions never have to handle incorrect inputs. Dually, one can infer the kinds of conflicts avoided by the system: requiring unavailable resources, sending incorrect inputs to another extension, or breaking abstraction and directly modifying another extension's data.

Relative to the other operating systems discussed above, Singularity permits fewer lowest-level extensions than Exokernel or SPIN allow. For example, Singularity provides thread-manipulation and exchange-heap functions as part of the kernel ABI [109]; Exokernel permits a libOS to define its own threading model [11], while SPIN permits extensible memory management [23]. This is mostly because the projects goals are orthogonal—Singularity aims first to improve the dependability and security of operating systems, and this imposes certain design choices that require a more extensive kernel. This decision also simplifies the Sing# type system to ignore such low-level details.

As alluded to when discussing SPIN, language-level techniques seem particularly apropos for the browser space. The choice of manifest-based extensions is particularly appropriate: Firefox's extensions already use a manifest to describe some (minimal) information about themselves, and enhancing these to declare needed resources, à la Singularity, is a natural step. Further, Singularity's type system encodes communication patterns of the extension; currently such patterns in Firefox are merely unenforced documentation. Additionally, Singularity demonstrates a decent compromise between lowest-level extensibility and a strong, expressive type system: by analogy, a browser (or any webapp, for that matter) has no need for replaceable HTML rendering engines.

---

expected versions of dependencies and prevent "DLL Hell". This is merely an engineering problem; the .Net framework addresses this issue.

(But see Section 3.3, where this possibility is addressed.) The largest remaining challenge is the lack of a convenient but type-safe API for accessing web pages' contents: similarly to Singularity's drivers and applications, extensions and web pages are defined almost identically, differing only in what resources they are initially allowed to access.

2.6.5 Other platforms

As mentioned at the outset of this section, extensibility can be found to varying degrees in many systems. Though few formal research papers document these efforts, several of these systems bear many resemblances to idioms seen in browser extensions, or to trends highlighted in OSes above.

LaTeX: A patchwork of pervasive, mutable macros The "LaTeX document preparation system" [134, 163] describes both the program that compiles input markup into output documents and the ecosystem of packages that customize every aspect of the appearance of those documents. The core of the program is based on Knuth's TeX [125], which defines a startlingly simple algorithm for optimally typesetting pages, paragraphs, and mathematical content [126]. Other than bug-fixes and updates to incorporate multi-lingual fonts, the TeX kernel has hardly changed in over thirty years. By contrast, thousands of packages currently exist and more are being added constantly, and it is very common for packages to conflict with one another.

The TeX markup language is fundamentally macro based, and includes both primitives to describe page parameters (e.g., font size, page dimensions) and the ability to create user-defined macros. These macros are used to build up larger units of text, such as chapter titles or page headers; *packages* are nothing more than collections of macros. Often, packages produce *nearly* the desired results, so many macros (and packages) resort to *patching* the commands of other packages to modify their definitions:

```
\newcommand{\Hello}{Hello}
% \Hello --> "Hello"
\let\oldhello=\Hello          % saves an alias to original \Hello macro
\renewcommand{\Hello}[1]{\oldhello, #1} % "calls" original \Hello macro
% \Hello{world} --> "Hello, world"
```

Often, such patches insert additional content before or after existing macros; occasionally they will replace the original definitions entirely. It should therefore be completely unsurprising that such macro patches, which crudely emulate manual aspect weaving, are highly sensitive to weaving-order conflicts. Resolving these conflicts is often quite tedious, and package authors frequently

have to add "compatibility flags" that modify how the package is loaded in order to account for other packages' presence. In contrast to web extension conflicts, where the user of extensions has no expertise to resolve problems, the user of LaTeX packages often understands the rudiments of macros well enough to troubleshoot their particular problems.

Emacs, Eclipse and Visual Studio Integrated development environments (IDEs) are a frequent target for extension, perhaps because their primary users—developers—are precisely the same people capable of writing extensions. Different IDEs support different APIs for extensibility, and may permit varying amounts of functionality to be replaced.

Emacs (short for "editor macros") [204] is essentially a thin shell for editing buffers of plain text, interpreting LISP programs, and handling I/O. Much like an exokernel, its default functionality merely multiplexes the keyboard to the display or the disk. All of the other functionality that emacs can host—e.g., language-specific syntax highlighting, automatic word wrapping, a mail client, an IDE—is provided by LISP modules that are dynamically loaded and interpreted by that minimal shell. By convention, these functions are divided into "major" and "minor" modes: major modes are mutually exclusive, while minor modes embellish a major one. Consequently, major modes frequently do not conflict with each other, except perhaps by clashing over global hotkeys, while minor modes frequently do conflict. (Ironically, there exists a package that attempts to provide "multiple major modes" simultaneously, and this package is itself a frequent source of conflicts.) To accommodate packages that must adapt to each other's presence, the LISP interpreter includes aspect-oriented primitives to modify existing code. As with LaTeX, resolving conflicts between emacs packages often amounts to manipulating the package loading order, but if necessary users can modify the packages themselves, which are nearly always distributed in source-code form.

Eclipse is also an IDE that is entirely built up from packages (known as "plugins") interpreted by, in this case, a Java runtime. But in stark contrast to emacs, Eclipse plugins must *explicitly state* their dependencies and their extension points [27]; if the dependencies can be satisfied, then the set of plugins can be loaded. As noted earlier, this approach resembles that of open modules [6], and in practice works relatively well. However, the Eclipse architecture is fairly "rigid": if a needed extension point does not exist, there is no way for another plugin to hook into that point without reimplementing the target extension entirely.[7] Additionally, Eclipse tends to suffer from its own version of "DLL hell", where unless precisely the expected versions of all plugins are present, the system as a whole will not work. When packages do not work together, users have

[7] Recent versions of Eclipse include support for AspectJ, which permits some dynamic weaving of functions that were not exposed as extension points; this is very brittle and not recommended practice.

no recourse beyond disabling the broken extensions, since packages are compiled JAR files that cannot readily be modified.

Visual Studio permits far more limited extensibility than either emacs or Eclipse, but its extensions often are more stable [160]. Built atop a larger kernel than the others, many of its major UI elements are provided by a collection of packages [161] that expose additional APIs to extend various components of the platform [30, 31]. Visual Studio does not yet support any form of aspect weaving in its extensions, so unless an extension point is provided there is nothing for extensions to modify. Extensions are compiled code, so again when conflicts occur the user has no choice but to disable the troublesome extensions.

2.7 Feature specification

The larger a system grows, the greater the chance that seemingly unrelated parts interact in unexpected ways. At the concrete level of code, interactions may make the codebase difficult to improve over time; at a higher level, interactions may threaten the correctness of the system's user-visible behavior. *Feature specification* aims to address this higher-level concern: a *feature* is an informal, self-contained, user-visible piece of functionality, whose behavior may be rigorously *specified* using a number of techniques. These specifications may be checked against each other and against axioms of the base system, to detect unwanted interactions—conflicts. This *feature interaction problem* has been intensely studied by the telecommunications industry: as they roll out new features (e.g., call waiting, caller ID) they must ensure that previously-working features do not break unexpectedly.

Unlike AOP or operating system extension mechanisms, feature specification is a step removed from the extensible systems it is used to model. While the underlying system therefore has to contend with granularity, pervasiveness, integration and other architectural issues, feature specification simply uses a very expressive (and therefore granular) logic to express whatever properties the underlying system ought to have. Likewise, the integration of multiple feature specifications mimics whichever integration technique the underlying system uses. Feature specification concerns itself almost exclusively with the troubleshooting part of extensibility.

An idealized feature specification effort is a collection of abstracted, checkable formal models that represent the desired behavior of the implemented system. Features can specify fine-grained responses to individual situations, and may require behavior that spans the entire underlying system. New features can be added to the collection over time, and the collection can be checked anew to ensure different features do not break each other's specifications. Interactions occur when feature specifications require distinct reactions to the same situation; conflicts occur when interactions were not expected. Conflicts are resolved by prioritizing

features (i.e., restricting their scope) or rewriting them; these resolutions are then reflected in the eventual implemented program. When all features are compatible, all behaviors demanded of the system hold—a guarantee only as strong as the specification effort.

As should be no surprise, there are a wealth of techniques for defining and modeling features, and defining, detecting and resolving conflicts. Keck and Kuehn [122] and Calder et al. [36] give several axes to examine the literature; I will focus on their *causal view*, where the goal is to identify a witness to the cause of the interaction, rather than classifying interactions by when during development they might have been introduced or managed. Additionally, I will focus solely on design-time specification techniques. Though there are several efforts at run-time feature specification (e.g., [25, 26, 35]), these feature-managers look very similar to runtime security monitors.

The essential idea behind feature specification is to describe the behavior over time of the features in the system: a temporal property is true *now* if at some time *later* a sub-property holds. All temporal logics can express several powerful notions relating to time: e.g., a property p holds *always*, *eventually*, *until* a property q holds, or at the *next* moment in time. Deterministic automata encode (nearly) the same notions via sequences of states. (Some temporal logics can also express properties concerning nondeterminism; I ignore the details here.) For both mechanisms, all feature specification work focuses on predicates that will be satisfied *infinitely often*—temporally finite properties are a special case as they trivially can be converted to infinite ones. Depending on the problem formulation, conflicts arise either when two specifications are simultaneously enabled infinitely often (where each feature thinks it has sole control but doesn't), or when combining two specifications admits no solutions at all (no system could satisfy both).

2.7.1 *Logic choice*

As with aspects, nearly every feature specification effort proposes a slightly different mechanism (among others, [29, 61, 105, 154, 177, 196]), and the subtleties distinguishing one temporal logic from another are not important here (indeed, no real consensus has been reached; see for instance [1, 2, 67, 133, 216]). Choosing a particular logic ultimately depends on knowing what conflicts need to be expressed for a given problem.

2.7.2 *Termination conditions*

Felty and Namjoshi [79] adopt the use of temporal logic for specifying both the system and the features. (Since temporal logic will eventually be necessary, they view the use of additional mechanisms, such as state machines, as needless additional sources of error.) Specifications in their sys-

tem are divided into system axioms and feature properties. Their system is not modular: adding new features may require adding to or changing the set of system axioms. (They give the example of adding a new type of call resolution; existing axioms must be enriched to reason about this new status.) Feature properties as presented here are engineered to be intuitive to specify.

Recall that a specification details the behavior over time of a feature: once a feature has reached a given state (a precondition), it will exhibit a certain property until no longer applicable (a post-condition). Felty and Namjoshi give an intuitive internal structure to pre- and post-conditions. Preconditions are modeled as a sequence of events, between which a sequence of properties hold. Postconditions specify what behavior *persists* as a consequence, either *until* some normal resolution or until an exception *discharges* the situation. For example, an informal specification of patients visiting a doctor might say "the patient must *arrive* at the office, and *stay there* until the receptionist *greets* him; once greeted, the doctor *sees* the patient until *finished* or another *emergency* discharges the patient early". This rule schema is simple but redundant: in their case study, the authors never used multiple events in their preconditions, and instead encoded the ongoing properties as persistent postconditions of multiple rules. Additionally, distinguishing the release and the discharge conditions adds no expressive power to the rules, but does simplify their definition of conflict.

A conflict occurs in every execution where 1) two features are enabled simultaneously infinitely often, 2) the system axioms hold, 3) the features are not discharged, but 4) somehow a feature property does not hold. When conflicts occur, the authors either strengthen the precondition of one feature (making it apply *less* often) or weaken the discharge postcondition (making it apply *more* often), so as to prioritize the other feature. This is again not modular, as adding a feature requires editing the properties of the others to resolve conflicts. For browser extensions, the lack of modularity is problematic. However, this clean skeleton for specifying a weakening of one rule with respect to another will be very useful for resolving feature conflicts and, using the insights I explore below, may be achievable in a modular way.

2.7.3 Modular checking

Li et al. [141, 142] propose a modular approach to feature checking. They work with *open features*, which depend upon variables or predicates not under the control of the feature. Their key approach (shared by [33, 37]) is to interpret temporal logic formulas using a *three-valued logic*: variables may be *unknown*, indicating the lack of knowledge that one feature has about another's behavior. When validating feature composition, modularly guess the values for these unknown predicates. Pessimistically, if everything unknown is assumed false, and the feature still validates,

then the feature will always validate successfully. Conversely, if everything unknown is assumed true and the feature fails to validate, then the feature will never validate and there is an intrinsic conflict. Otherwise, manual intervention is needed to resolve possible conflicts, by modifying the specifications or by manually composing features.

To achieve *modular* checking, i.e., validation of a single open feature without knowing the precise set of other features being composed into the system, the authors require that specifications not be written obliviously: if a given property p plausibly may be reduced (resp. expanded) in scope by later features, it must be specified proactively as $p \wedge c_p$ (resp. $p \vee c_p$), where c_p is a symbol of unknown value. Moreover, they split the validation process into two: first, they *verify* that the feature's specifications hold in the current product. The same technique as above broadly applies: optimistically set all c_p that reduce scope, and pessimistically set the others, and see if any verifications fail.[8] Second, they ensure that any changes the new feature makes to the product *preserve* existing properties required by prior features. This is their modularity result: once composed and verified, it becomes *later* features' responsibilities to ensure they do not invalidate prior features. In short, while their modular checking does model only one feature at a time, it still requires the presence of the base program and hence is not as modular as might be desired.

The essential points for my purposes are the elaboration on the idea of cleanly weakening of one feature with respect to another from Felty and Namjoshi [79], and identifying a plausible way to modularly specify and reason about open features. Their modularity result depends heavily on "unknown" values and on the cooperation of specification authors—oblivious properties cannot be modularly checked. Additionally, as presented, the authors acknowledge that they assume a linear ordering of feature composition, but state that the work easily extends to composing features in the absence of a base product. If so, the essence of this approach may be very useful for browser extensions, indicating what cooperation levels may be necessary for independent extension authors to produce easily composable extensions.

2.7.4 Reified features

Plath and Ryan [176] propose making features an explicit construct of the language used to verify feature interactions. They also use a hybrid approach in which the system is specified as a state machine while the requirements over it are specified as temporal formulas. Together, these essentially form a feature aspect: a reified, syntactic component that may specify the presence

[8] The actual algorithm is much subtler for improved precision; the gist shown here omits the requisite but confusing bookkeeping.

of required data, functionality or other features; may introduce new data, functionality, or specifications; and may change the behavior of existing data. Like aspect weaving, this last step is inherently not modular. They identify four types of conflict between two extensions, reminiscent of aspect conflicts: a feature composed later (resp. earlier) breaks one composed earlier (resp. later); both features together break a property satisfied by the base system and either feature alone; finally, the two features do not commute.

Firefox extensions are already syntactically reified, declarative objects. (They are not quite first-class in the language sense, as they cannot be assigned as values.) Extending them to include verification information would be natural. The exact technical assumptions made in this work may be too expensive: the authors admit that even in their case study with a small number of simple features, the state-space explosion quickly made the verification intractable. Perhaps surprisingly, they argue for relative nonchalance over conflicts that break the original system (after all, extensions are intended to change system behavior). It is unclear whether this argument is warranted for browsers: most browser extensions do not supplant core functionality (unlike some telephony features they examined), but merely augment it.

2.8 Security monitors

Security monitors are necessary whenever a user (or a runtime system) does not trust another piece of code to run within some prescribed bounds. Monitors supervise the execution of untrusted code and intervene when necessary to maintain the desired *policy* of the system, accepting all program executions the policy deems good and rejecting the rest. Most practical policies are *properties*: they judge a program execution in isolation and not relative to other executions. There are two fundamental kinds of properties: *safety* properties that ensure "nothing bad ever happens", and *liveness* properties that ensure "something good eventually happens". Thus, never accessing `/etc/passwd` is a safety property, while always closing all open files is a liveness property. Obviously, these two properties can be conjoined to yield a non-safety, non-liveness property; surprisingly, *all* reasonable[9] properties can be written in this form [148]. All security monitors can thus be classified by whether they support safety, liveness, or a combination of both property types.

An idealized security monitor enforces a single policy over a given program, by observing and mediating any security-relevant actions taken by the program. The monitored program is oblivious to the presence of monitors (ignoring side channels such as timing discrepancies); in fact this obliviousness ensures that the monitor is not subvertible by the program. Policies are usually written by users or system administrators,

[9] Formally, a reasonable property is a decidable predicate over program executions that at minimum is true for the empty execution.

rather than the program's developers, and their granularity depends on the monitoring technique used. Multiple policies must be combined before the monitor can enforce the composite; the default is merely to intersect the policies and reject the program if any monitor rejects it, which means conflicts among policies may cause fewer programs than expected to be valid. Conflicts among policies are simply mismanaged user expectations; the monitor will run properly regardless of what cumulative policy it enforces.

The distinctions in security monitor implementations lie mainly in integration time: static access controls for specific policies can be enforced at build time (of either the mainline program or the policies) [18, 63, 113, 116, 199]; execution monitors can supervise the program's runtime; inlined reference monitors activate (at the latest) at load time. Among the dynamic approaches, few systems actually consider their extension model in any detail (e.g., [4, 52, 53]): how might multiple policies compose for a given target? For those that do, designs must consider whether policies apply only to the mainline program, or whether they layer over each other. (Consider combining a policy limiting memory usage with one limiting processor time: if for a tiny, fast program the CPU-usage policy took too much memory to enforce, should the memory-usage policy abort the program due to the CPU policy's expensive behavior, or accept the well-behaved mainline program?) Systems that ignore policy composition implicitly assume that policies inspect only the original program.

As was mentioned earlier, security monitors are one of AOP's strengths, describing a whole-program property that should be expressible in a concise and easily-understood way. Unsurprisingly, AOP is one of several typical implementation techniques for security monitors [52, 101, 201], though binary rewriting has been explored more thoroughly [19, 43, 70–73, 76] (among other reasons, because it can be used even when application sources are not available). Additionally, security monitors also appear as access control mechanisms in operating systems (e.g., [96, 150, 187]). These systems usually enforce only one policy at a time, obviating the need for policy composition.

2.8.1 Theoretical results

Before addressing the pragmatic concerns of how security policies might be specified, it is necessary to know what policies are expressible and enforceable. One line of research [7, 145, 146, 192, 217] precisely defined the complexity class of properties enforceable by security automata: if the only remedy available to a monitor is to terminate the offending program, then such a monitor can enforce precisely the safety policies; if a monitor can also forge or suppress behavior of the offending program, then a wider class of transaction-like policies are available. Understanding this class, and under what operations it remains closed, informs how policies may compose.

2.8.2 *Safety properties and beyond*

Evans [75], Evans and Twyman [76] focus on code safety properties, which ensure that no "bad thing" can occur in a program execution. Their implementation rewrites the monitored code during compilation to wrap all security-relevant functions with calls into the monitor, which in turn enforces the desired safety properties. Safety policies are straightforward to combine, since they are closed under conjunction: if two policies each prevent something bad from occurring, the conjunction of those policies naturally would prevent both bad things from occurring. When all policies are safety properties, then there can be no conflicts. Strict safety properties are of somewhat limited use, so their system supports slightly looser policies that wish to prohibit bad things *except* in special circumstances. These weakened policies provide a modicum of added flexibility without creating new conflict types.

Bauer et al. [19] provide a much richer environment for defining security policies. Most notably, they use a more expressive mechanism for defining policies, and they give the policy author much greater control over policy composition. Their formalism is based upon edit automata [145, 146, 148], which can delay, suppress or forge security-relevant actions taken by the monitored program. Like safety properties, edit automata are closed under composition, but permit a much broader class of enforceable transaction-like *renewal* policies. In essence, the monitor can delay the beginnings of a transaction (e.g., logging in to an ATM and asking for money) until a crucial commit action occurs (recording the withdrawal), at which point the monitor can replay the beginning of the transaction and insert the remaining events needed to complete it (logging out), in one atomic action. In this way, the observed behavior of the program jumps from one valid state to another, without the possibility of crashing in a visibly invalid state. To define the composition of multiple policies, their system permits *higher-order policies* that suggest modifications (intersection, or precedence, or outright modification) to earlier policies to fit them together.

Unfortunately, as noted by the authors [148, section 5], it is not always reasonable to assume the monitor can delay, suppress or forge arbitrary security-related actions. For instance, if a policy requires that programs close all opened files, it may not be possible to delay the open event until the matching close occurs: any intervening I/O operations would block or fail without a valid file handle. Dually, the monitor cannot forge events that require secrets or time-sensitive responses. These and other obstacles make the edit automata model a challenging fit for the current form of webapp programming, where authentication and time-sensitive network events are the norm.

Additionally, note that multiple policies cannot be written modularly in their system without someone eventually manually composing them. Their language does not expose any conflict

checking (i.e., if two policies both address the same situation)—though it is possible that the underlying edit automata may admit an intersection test answering exactly this question. Because the intended user of browser extensions is not equipped to write additional extensions, the edit-automata model cannot be used directly to resolve webapp extension conflicts.

2.9 Contrasting the web platform with related work

At first impression, the exokernel approach seems a poor fit for browser extensibility. Representing an extreme design point, an exokernel aims to let applications manage and define the semantics of their required resources: speed is a primary concern. But the resources available in a browser, such as incoming web content, profile data, or the user interface, are qualitatively different than those in an OS: these are structured, semantics-laden resources, and are the output of substantial processing effort. Skewing the system design to allow rapid access to slow resources does not address the performance bottlenecks of the system. Speed is not the primary concern, and browser extensions therefore need to be examined in a much more full-featured base environment.

On the other hand, if used correctly exokernels can provide robustness guarantees that current browsers cannot match. If the minimal kernel contains just enough power to draw the window containing content, and provided a narrow interface to talk to a renderer process, the result is fairly close to Chrome's architecture [182]. Alternatively, if the kernel provided most of the rendering logic, but provided a narrow interface for extensions to masquerade as HTTP content providers, the result is essentially the Xax approach [57], which (along with the similar Native Client [230]) provides practical mechanisms for safely running (nearly) unmodified third-party binary code in today's browsers. Pushing Chrome's approach to the extreme permits arbitrary content-renderers to be fitted into the browser's UI: there would be no fundamental distinction between rendering HTML and rendering Flash, for instance. This approach does make the browser dramatically more flexible, allowing increased experimentation with new scripting languages or web standards. To an extent, this form of flexibility is very useful (cf. C3 in Chapter 3). But it also strains the definition of the "web platform", since Xax or Native Client libraries are independent of the HTML/CSS/JS trio, and merely use the browser for its access to network-available information.

Firefox's traditional extension model is not ideal, but it is far more capable than any other current browser extension model. Firefox's glaring failing is in conflict detection. Because extensions by nature split into functionality and interface, it is natural that the techniques to resolve UI conflicts be different than those tailored for functionality.

Considering first UI conflicts, it seems reasonable for feature specification techniques to be

readily applicable. Indeed, viewing sequences of user interaction as features, feature specification readily checks that patterns of interaction remain feasible despite the addition of new features. (For instance, a feature could declare that "from any state, the Save functionality is available" and "once the Save menu item or Save toolbar button is selected, the page will eventually be saved". Extensions may then remove either the toolbar or the menu and not violate the existing feature, but removing both would result in conflict.) Ideally, conflict checking should be as efficient as possible (since users are notoriously impatient), which argues for modular verification of extensions. Unfortunately, modularity is in direct tension with the desire for minimal cooperation: while Li et al. [141, 142] enable modular checking of features compared to Felty and Namjoshi [79], it comes at the cost of explicitly specifying interaction points between features. Depending on how this is presented to the developer, this again may not be too onerous or restrictive a requirement. Additionally, Douence et al. [59] distinguish the notions of strong and weak independence. Adapting this notion to UI specification, the more extensions can be certified as strongly independent (i.e., never in conflict regardless of other extensions), the more modularly they can be verified for other feature interactions. In Chapter 5, I model XUL overlays as document transformers; weak and strong independence fall out naturally from those definitions (though exploiting such notions fully is left as future work).

Turning to address conflicts from code as well, it is apparent that the full flexibility of JS is a liability, and some more constrained language is necessary to prevent conflicts here. This was the approach taken by SPIN [23] and Singularity [55], which both abandoned C/C++ for a strongly typed extension language that permitted compile-time assurances of correctness. The danger with static type systems is they are specialized to enforce a particular safety property of the code; as I have shown, not all policies are safety policies, and policies may override each other when composed. However as Singularity has shown, a strong type system may be sufficient (but not necessary) to enforce an over-approximation of desired properties, provided one uses a narrow interface with just the right abstractions. The engineering challenge in adapting Singularity's approach will be to apply a typing discipline to the DOM [210] and extend it to a suitably constrained interface to the browser's runtime services. Several more dynamic attempts have been made to tame the control flow of JS [157] and the behavior of JS or DOM objects [209].

It is likely that many browser extensions cannot be considered "harmless", as they deliberately change behaviors of the browser. However, perhaps a weaker noninterference result holds: if the browser declares certain behaviors as extensible (à la open modules), harmless extensions merely must not modify anything else. If not: JS routinely employs event handlers that are dynamically added and removed; perhaps these patterns can be expressed using the dynamic pointcuts of

Douence et al. [59, 60], or even more powerfully using Felty and Namjoshi's intuitive specification language [79]? Then the absence of conflicts using these systems would imply extension compatibility.

Not all conflicts may be resolvable by load-time; sometimes users may wish to choose to dynamically prioritize one extension over another. Here security monitors' runtime techniques become relevant. Assuming the runtime system can present the user with a sufficient approximation of all conflicts between two extensions, the user can define a policy resolving the conflicts. When more than two extensions conflict, users may naturally need to compose policies. Moreover, these need not be simple security policies: it is too limiting to ensure only that no bad thing occurs (e.g., two extensions activate simultaneously); users may want to ensure that eventually something good occurs (e.g., both extensions run in a well-defined order). For appropriately expressive sets of "security-relevant" events, this is precisely Polymer's aim [19].

2.10 Summary

This chapter has examined the extensibility problem: how systems should be designed to support extensions, and how to reason about conflicts between them, particularly in the relatively new context of the extensible web browser, and I have explored extensibility efforts in aspect-oriented programming, operating systems, feature specification and security monitors in order to better understand the browser. I developed a classification scheme through which I have defined the extensibility models for each of these systems. Finally, I have examined several specific techniques from these domains, and explored how they might be used to design a better extensibility model for a web browser.

Chapter 3

BROWSER ARCHITECTURE CHOICES FOR EXTENSIBILITY[1]

3.1 Introduction

As I have described in previous chapters, browsers as they exist today all provide some measure of extensibility, but their approaches are ill-equipped to handle the intrinsic compatibility challenges that extensions cause. Further, because production-quality browsers are all monolithic, complex systems, it is particularly challenging to modify or experiment with the extension mechanisms themselves. Modifying Chrome's extension model, for example, would entail modifying their multi-process architecture, and modifying Firefox's extension mechanisms would require rewriting the entire Firefox UI!

The time has come to reconsider browser architectures with a focus on extensibility. This chapter presents C3: a reconfigurable, extensible implementation of HTML, CSS and JS designed for web client research and experimentation. C3 is written entirely in C# and takes advantage of .Net's libraries and type-safety. Similar to Firefox building atop Gecko, I have built a prototype browser atop C3, using only HTML, CSS and JS.

By *reconfigurable*, I mean that each of the modules in the browser—Document Object Model (DOM) implementation, HTML parser, JS engine, etc.—is loosely coupled by narrow, type-safe interfaces and can be replaced with alternate implementations compiled separately from C3 itself. By *extensible*, I mean that the default implementations of the modules support run-time extensions that can be systematically introduced to

1. extend the syntax and implementation of HTML with new tag names and DOM node types that are technically indistinguishable from the default C3-provided tags,
2. transform the DOM when being parsed from HTML by post-processing subtrees as the parser constructs them,
3. extend the UI of the running browser with new content,
4. extend the environment for executing JS with additional bindings to new native objects, and
5. modify running JS code by dynamically weaving advice into existing functions.

---

[1] This chapter is based on work initially published in the 2nd USENIX Conference on Web Application Development (WebApps '11) [138]

Compared to existing browsers, C3 introduces novel extension points (1) and (5), and generalizes existing extension points (2)–(4). These extension points are treated in order in Section 3.3. I discuss their functionality and their security implications with respect to the same-origin policy [188]. I also provide examples of various extensions that I and others have built.

3.1.1 Addressing a broader need

While my initial motivation for developing C3 grew from a need for a platform to develop novel extension mechanisms, a clean-slate reimplementation of the web platform is also of use to many other researchers who are experimenting with internal components of today's browsers:

1. XML3D [202] defines new HTML tags and renders them with a 3D ray-tracing engine—but neither HTML nor the layout algorithm are extensible.

2. Maverick [185] permits writing device drivers in JS and connecting the devices (e.g., webcams, USB thumb drives, GPUs, etc.) to web pages—but JS cannot send raw USB packets to the USB root hub.

3. RePriv [86] experiments with new ways to securely expose and interact with private browsing information (e.g., topics inferred from browsing history) via reference-monitored APIs—but neither plug-ins nor JS extensions can guarantee the integrity or security of the mined data as it flows through the browser.

4. Reis and Gribble [182] identify desired security and performance boundaries for webapps—but the existing abstractions (based on the same-origin policy) are only imperfect and inflexible proxies.

These projects incur development and maintenance costs well above the inherent complexity of their added functionality. Researchers often must modify the source code of the browsers—usually tightly-optimized, obscure, and sprawling C/C++ code—and this requirement of deep domain knowledge poses a high barrier to entry, correctness, and adoption of research results. Moreover, patching browser sources makes it difficult to update the projects for new versions of the browsers. This overhead obscures the fact that such research projects are essentially extensions to the web-browsing experience, and would be much simpler to realize on a flexible platform with more powerful extension mechanisms. Though existing extension points in mainstream browsers vary widely in both design and power, none can support the research projects described above.

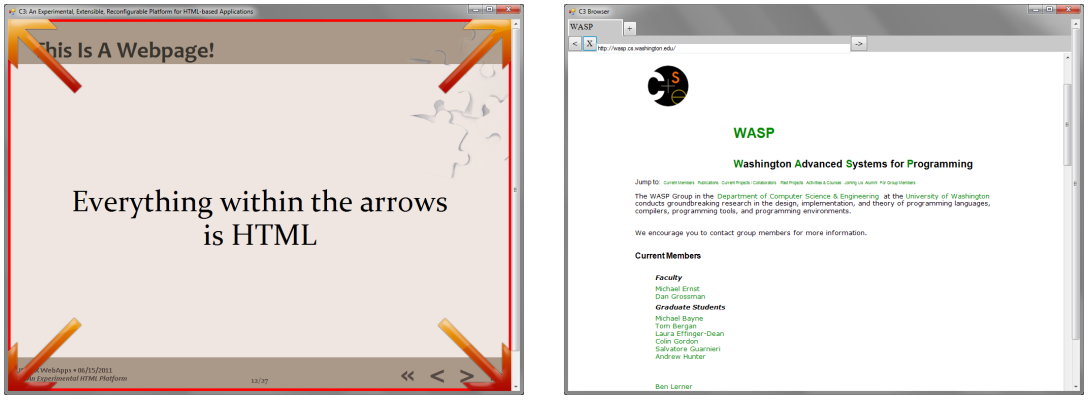


Figure 3.1: Screenshots of C3. Everything but the title bar is HTML (left), including the browser chrome (right).

3.1.2 *Contributions*

The rest of the chapter is structured as follows. Section 3.2 gives an overview of C3's architecture, highlighting the software engineering choices made to further the modularity and extensibility design goals. Section 3.3 presents the design rationale and implementation for the supported extension points. Section 3.4 evaluates the performance, expressiveness, and security implications of the extension points. Section 3.5 describes future work. Section 3.6 summarizes the chapter.

Developing a project as large as C3 would not have been possible without the collaboration and support of colleagues at Microsoft Research. Of specific note, Herman Venter designed and implemented the JS engine and the HTML and CSS parsers, and Nikolai Tillman and his interns implemented the Spur JIT engine used for JS execution. Wolfram Schulte led the project and prototyped the layout engine, which was enormously enhanced by Brian Burg during his internship. During my internships with the project, I designed the DOM implementation, the basic browser UI, and focused most of my attention on the extension mechanisms described in this and the following chapters.

3.2 *C3 architecture and design choices*

As a research platform, C3's explicit design goals are architectural modularity and flexibility where possible, instead of raw performance. Supporting the various extension mechanisms above requires hooks at many levels of the system. These goals are realized through careful design and implementation choices. Since many requirements of an HTML platform are standardized, aspects of the architecture are necessarily similar to other HTML implementations. C3 lacks some

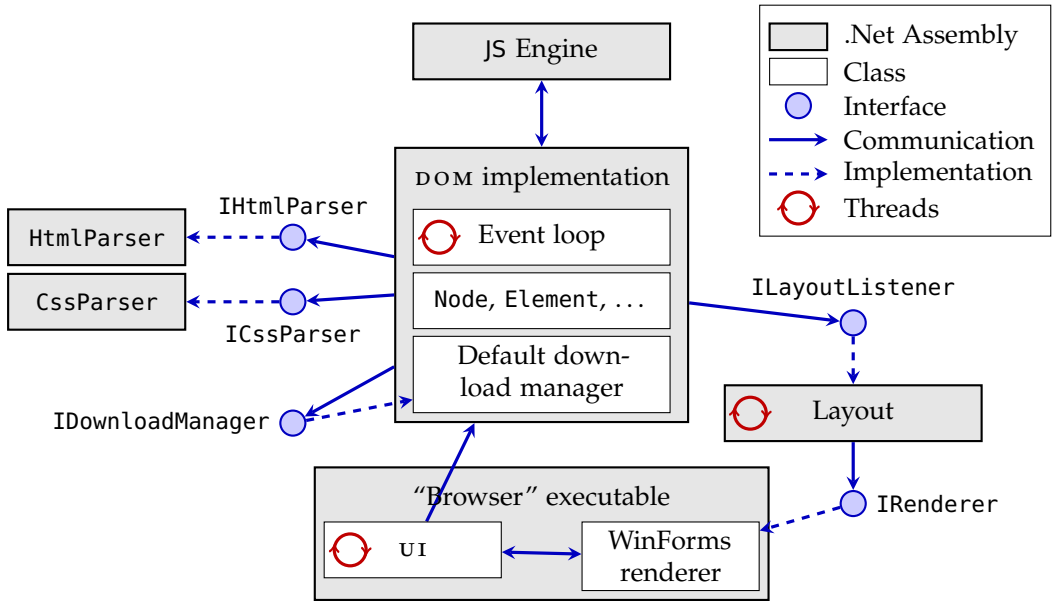


Figure 3.2: C3's modular architecture

of the features present in mature implementations, but contains all of the essential architectural details of an HTML platform.

C3's clean-slate implementation presented an opportunity to leverage modern software engineering tools and practices. Using a managed language such as C# sidesteps the headaches of memory management, buffer overruns, and many of the common vulnerabilities in production browsers. Using a higher-level language better preserves abstractions and simplifies many implementation details. Code Contracts [78] are used throughout C3 to ensure implementation-level invariants and safety properties—something that is not feasible in existing browsers.

Below, I sketch C3's module-level architecture, and elaborate on several core design choices and resulting customization opportunities. I also highlight features that enable the extension points examined in Section 3.3.

3.2.1 *Pieces of an HTML platform*

The primary task of any web platform is to parse, render, and display an HTML document. For interactivity, web applications additionally require the managing of events such as user input, network connections, and script evaluation. Many of these sub-tasks are independent; Fig. 3.2 shows C3's module-level decomposition of these tasks. The *HTML parser* converts a text stream into an object tree, while the *CSS parser* recognizes stylesheets. The *JS engine* dispatches and

executes event handlers. The DOM *implementation* implements the API of DOM nodes, and also defines bindings to expose these methods to JS scripts. The *download manager* handles actual network communication and interactions with any on-disk cache. The *layout engine* computes the visual structure and appearance of a DOM tree given current CSS styles. The *renderer* displays a computed layout. The browser's UI displays the output of the renderer on the screen, and routes user input to the DOM.

3.2.2 *Modularity*

Unlike many modern browsers, C3's design embraces loose coupling between browser components. For example, it is trivial to replace the HTML parser, renderer front end, or JS engine without modifying the DOM implementation or layout algorithm. To make such drop-in replacements feasible, C3 requires that all data structures shared between modules (e.g., the representation of objects between the JS engine, the DOM and the HTML parser) are hidden behind interfaces, and avoids sharing data structures between modules whenever possible (e.g., the DOM and the layout engine are heap-disjoint). This design decision also simplifies threading disciplines, and is further discussed in Section 3.2.9.

Simple implementation-agnostic interfaces describe the operations of the DOM implementation, HTML parser, CSS parser, JS engine, layout engine, and front-end renderer modules. Each module is implemented as a separate .Net assembly, which prevents modules from breaking abstractions and makes swapping implementations simple. Parsers could be replaced with parallel [119] or speculative [198] versions; layout might be replaced with a parallel [156] or incrementalizing version, and so on. The default module implementations are intended as straightforward, unoptimized reference implementations. This permits easy per-module evaluations of alternate implementation choices.

3.2.3 *Implementing JS objects*

The JS language is *prototype based* rather than class-based like C#. This impedance mismatch has several technical consequences for the implementation of the JS engine that, while not directly related to extensions, do bear on the modularity of the engine as it fits into C3. Specifically, the choices for implementing JS objects directly impact the implementation of DOM objects, which in turn affects the simplicity of the HTML extension point (discussed later). This section may be confusing, since it must discuss both the JS language and the C# implementation of that language.

The JS object model Each JS object is essentially a property bag that maps names to values. Additionally, each object internally contains a distinguished pointer to its *prototype* object; these pointers are not mutable, and each so-called *prototype chain* eventually terminates at one unique prototypical object that is the root of the prototype hierarchy. Prototype objects are crucial in property resolution: to determine the value of `fooObj.bar`, `fooObj`'s property bag is checked to see if it contains `bar`, and if so, the value is returned. If not, then `fooObj`'s prototype is recursively checked to see if *it* contains `bar`, and so on until the prototype chain ends:

```
var fooProto = { bar: 42 };
var fooObj = { baz: 43 };
// Assume fooObj.[[prototype]] == fooProto
fooObj.bar == (fooObj.[[prototype]]).bar == fooProto.bar == 42
fooObj.baz == 43
fooObj.notFound == (fooObj.[[prototype]]).notFound == fooProto.notFound
                == (fooProto.[[prototype]]).notFound == null.notFound
                == undefined
```

(The prototype pointers are not available in standard JS code, though some engines, including SpiderMonkey and V8, have exposed them via the property name `__proto__`. The notation used here conforms to the ECMAScript specification, and calls them `[[prototype]]`.)

Because prototype pointers are not directly manipulable through JS code, JS uses a concept of *constructor functions* to create new objects and populate their prototypes. Accordingly, constructor functions are JS functions—which means they are JS objects—that have a `prototype` property. This is a normal JS property that contains a normal JS object. When new JS objects are created from a constructor function, their internal `[[prototype]]` pointers are set to the current value of the constructor function's `prototype` property; additionally, the prototype object contains a field, named `constructor`, that points back to the constructor function:

```
function Foo() { this.baz = 43; };
Foo.prototype.bar = 42;
var fooProto = Foo.prototype;
// fooProto.constructor == Foo
var fooObj = new Foo();
// fooObj.[[prototype]] == fooProto
```

A diagram showing the relationships between `fooObj`, `fooProto`, and `Foo`, along with the built-in objects `Function` and `Object`, is shown in Fig. 3.3.

Implementing the JS object model The JS engine in C3 is part of a self-contained subproject called Spur [20], which is is a tracing-based JIT for MSIL. The JS engine is designed to work as a standalone shell or embedded in a web-platform or other "host environment", and is entirely implemented in C#. Consequently, the central design goal of the C# implementation of JS objects is that it simultaneously expose the correct semantics to JS and expose a natural API to .Net code. Exposing the correct semantics to JS is not trivial: where class-based OO languages have classes (that are not themselves objects) and instances (that are), JS has constructor functions, prototypes, and instances, all of which are objects. Accordingly, the C# implementation of JS must construct instances of three C# classes for each built-in JS object type:

1. A constructor class, e.g., `NumberConstructor`, that exposes properties equivalent to static constants or methods, e.g., `Number.MAX_VALUE`. Only one instance of this class is created.

2. A prototype class, e.g., `NumberProtoype`, that implements methods shared by all instances, e.g., `aNumber.toFixed()`. Only one instance of this class is created.

3. An instance class, e.g., `Number`, that implements the behavior of individual instances. Naturally, one instance of this class is created for each JS instance object.

The actual implementation of each object's "property bag" is an instance of another C# class called `TypeObject`: for example, the `TypeObject` in the `NumberPrototype` instance contains a mapping from the name `constructor` to the `NumberConstructor` instance.

However, having to work with three parallel classes for each object type does not yield a natural .Net API. Instead, the design follows a pattern where the constructor and prototype classes delegate their functionality to corresponding methods in the instance class. For example, calling `aNumber.toFixed()` resolves to the JS-callable method `toFixed` that is exposed on (the singleton C# instance of) `NumberPrototype`, which is implemented by a stub that marshals the arguments into strongly-typed C# values and then calls the `ToFixed` method of (the `aNumber` instance of) the `Number` class. While this indirection seems needlessly convoluted, the net effect is that C# code can *completely ignore* the constructor and prototype classes and pretend all functionality is defined on the instance class alone. (This is why the class is called `Number`, and not `NumberInstance`; C# code *only* deals with instances.) At the same time, the rest of the JS engine ensures that the prototype and constructor classes are instantiated as needed to make JS objects work, thereby fulfilling the main design goal at the outset of this section.

Note that `ObjectInstance` and `TypeObject` classes are public Spur APIs. This means that host environments such as the web platform can easily add their own built-in objects, as in Section 3.2.4.

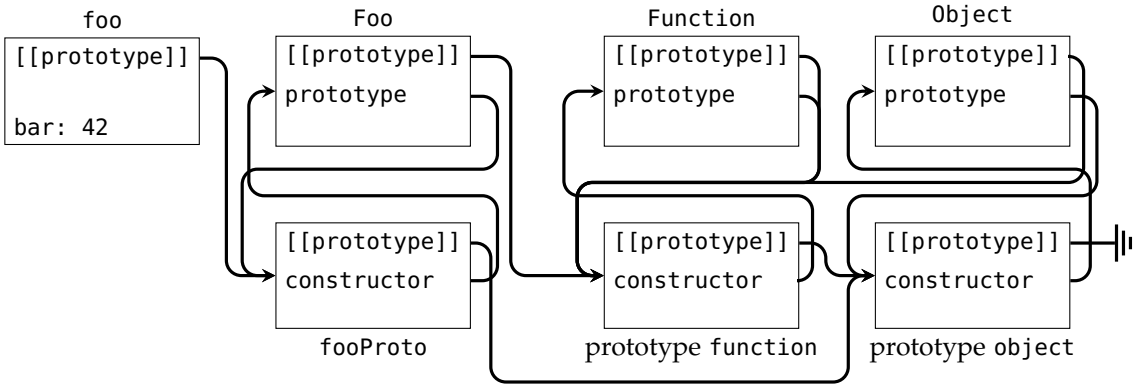


Figure 3.3: The structure of JS objects, including the built-in `Function` and `Object` functions

3.2.4 DOM *implementation*

The DOM API is a large set of interfaces, methods and properties for interacting with a document tree. I highlight two key design choices in the implementation: what the object graph for the tree looks like, and the bindings of these interfaces to C# classes. These choices aim to minimize overhead and "boilerplate" coding burdens for extension authors.

Object trees: The DOM APIs are used throughout the browser: by the HTML parser (Section 3.2.5) to construct the document tree, by JS scripts to manipulate that tree's structure and query its properties, and by the layout engine to traverse and render the tree efficiently. These clients use distinct but overlapping subsets of the APIs, which means they must be exposed both to JS and to C#, which in turn leads to the first design choice.

One natural choice is to maintain a tree of "implementation" objects in the C# heap separate from a set of "wrapper" objects in the JS heap. These "wrapper objects" would be C# objects deriving from `ObjectInstance` and visible to JS; the implementation objects would not. The implementations of these JS objects would contain pointers to their C# counterparts: the JS objects are a "view" of the underlying C# "model". The JS objects contain stubs for all the DOM APIs, while the C# objects contain implementations and additional helper routines. This design incurs the overheads of extra pointer dereferences (from the JS APIs to the C# helpers) and of keeping the wrappers synchronized with the implementation tree. However, it permits specializing both representations for their respective uses, and the extra indirection enables multiple views of the model: This is essentially the technical basis of Chrome extensions' "isolated worlds" [16], where the indirection is used to ensure security properties about extensions' JS access to the DOM. Firefox also uses the split to improve JS memory locality with "compartments" [220].

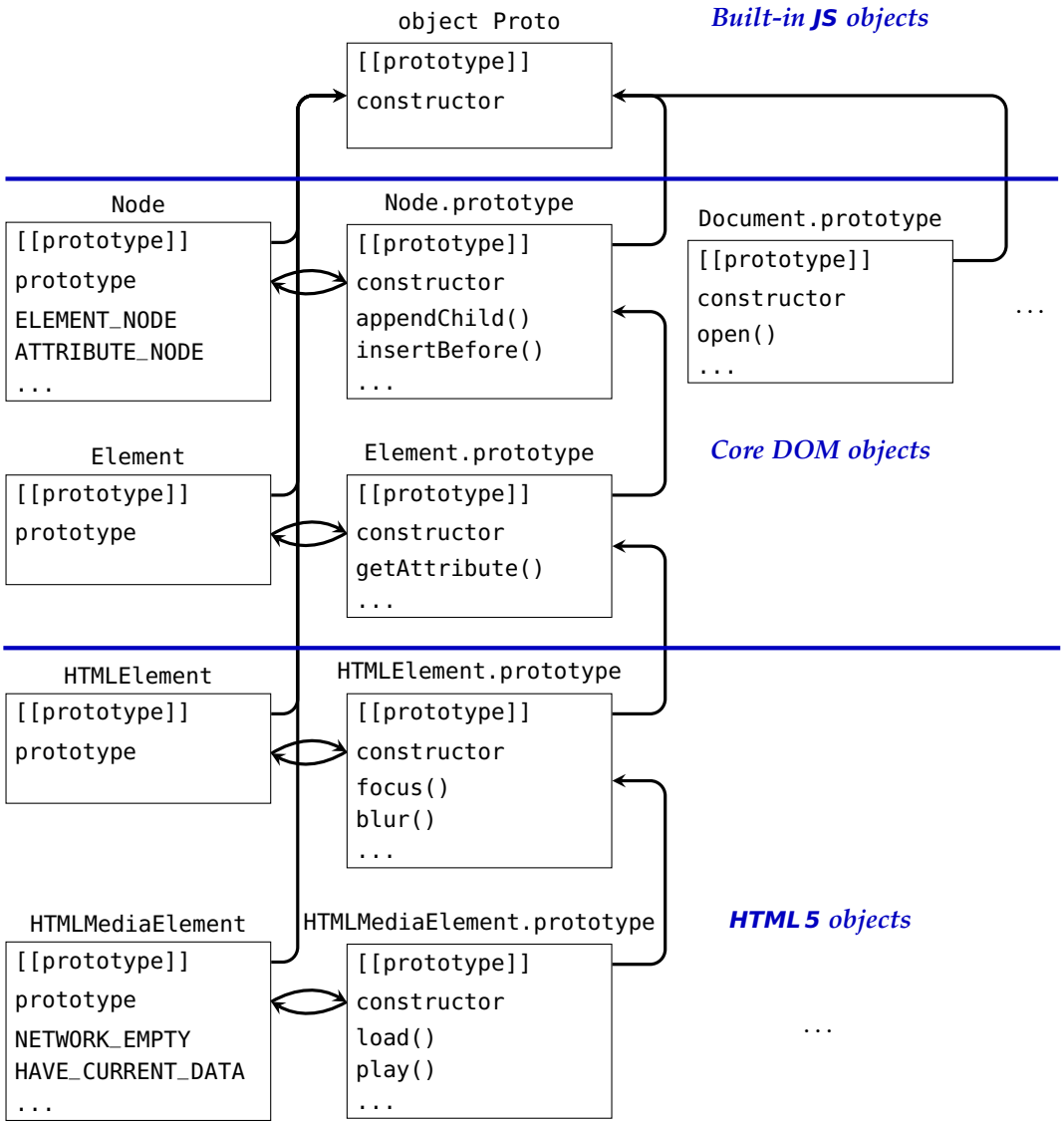


Figure 3.4: Part of the prototype and constructor hierarchy of the DOM. Prototype objects contain DOM methods, and constructor objects contain static constants. Not shown are the `Document` constructor or the rest of DOM core interfaces, or nearly all of HTML5.

By contrast, C3 instead uses a single tree of objects visible to both languages, with each DOM node being a C# subclass of an ordinary JS object, and each DOM API being a standard C# method that is exposed to JS. This design choice avoids both overheads mentioned above. Further, Spur [20], the tracing JS engine currently used by C3, can trace from JS into DOM code for better optimization opportunities. To date, no other DOM implementation/JS engine pair can support this optimization.

DOM language bindings: The second design choice stems from the first one: how to represent DOM objects such that their properties are callable from both C# and JS. Additionally, this representation must be open: extensions such as XML3D must be able to define new types of DOM nodes that are instantiable from the parser (see Section 3.3.1) and capable of supplying new DOM APIs to both languages as well. Therefore any new DOM classes must be able to subclass the C# DOM class hierarchy easily, and be able to use the same mechanisms as the built-in DOM classes. Given the discussion in the previous section on implementing JS objects, C3's chosen approach is mostly straightforward; a subset of the JS objects comprising the DOM is shown in Fig. 3.4, and an excerpted example is shown in Fig. 3.5 for the implementation of the `Element` IDL interface:

- The DOM class hierarchy derives from `DOMObjectInstance`, a subclass of `ObjectInstance`, and so DOM objects *are* JS objects, as needed. (Note that `DOMObjectInstance` is a nearly empty class, whose purpose is to ensure DOM C# methods receive only DOM C# objects.)

- Despite appearances, "constructors" such as `Node` or `HTMLAnchorElement` are not in fact callable functions, so unlike true constructor functions, DOM constructors' prototype pointers are the prototypical `object`, and not the prototypical `function`. (See Fig. 3.4.)

- As required by JS semantics, DOM methods, properties and constants are exposed on the respective instance, prototype and constructor objects. To expose a method on a JS object (e.g., `getAttribute`), the implementation simply adds a property to the `TypeObject` mapping the (JS) name to the (C#) stub function (see `ElementPrototype::InitTypeObject` in Fig. 3.5) that in turn calls the actual implementation function (`Element::GetAttribute`). Similar steps expose a property (e.g., `tagName`) on instance objects using C# getters and setters (e.g., `TagName`).

Note that just as Spur permits host environments to add their own bindings easily, C3 continues to provide that ability and so the DOM implementation is readily extensible by new node types.

```
// WebIDL sources
interface Element : Node {
  readonly attribute string tagName;
  string getAttribute(string id);
}
```

```
// Generated C# implementation
class ElementConstructor :  DOMObjectInstance {
  protected InitTypeObject(TypeObject ty) {
    ty.AddProperty("prototype", new ElementPrototype());
  }
}
class Element : Node { // and Node : DOMObjectInstance
  protected internal InitTypeObject(TypeObject ty) {
    Element.AddInstanceProperties(ty);
  }
  new static internal AddInstanceProperties(TypeObject ty) {
    // Add all new Element properties
    ty.AddProperty("tagName", getTagName);
    // Add all "inherited" Node properties
    Node.AddInstanceProperties(ty);
  }
  // Marshal the properties from JS to C#
  private void getTagName(ObjectInstance target,
               ref ValueWrapper result) {
    result.SetValue(((Element)target).TagName);
  }
  // Implement all properties and methods
  public string TagName { get { ... } }
  public string GetAttribute(string name) {
    ...
  }
}
class ElementPrototype : DOMObjectInstance {
  protected internal InitTypeObject(TypeObject ty) {
    // Add all Element methods
    AddProperty("getAttribute", getAttribute);
    // Note: all Node methods come from JS prototype chain
  }
  // Marshal the methods from JS to C#
  private void getAttribute(ObjectInstance target,
               string name, ref ValueWrapper result) {
    result.SetValue(((Element)target).GetAttribute(name));
  }
}
```

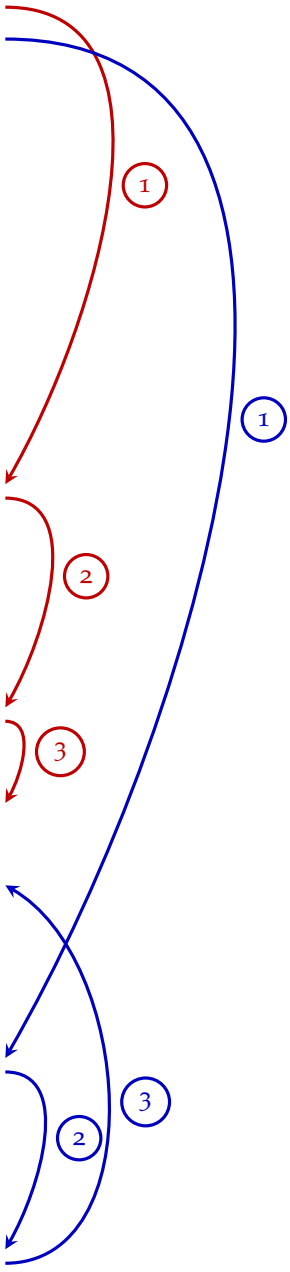


Figure 3.5: Abbreviated example of IDL definition and (slightly modified) C# implementation for the `Element` DOM interface. Each property in the IDL is 1) declared in the `TypeObject`, 2) marshaled from JS to C#, and 3) implemented just once, in C#

.

3.2.5 *The HTML parser*

The HTML parser is concerned with transforming HTML source into a DOM tree, just as a standard compiler's parser turns source into an AST. Extensible compilers' parsers can recognize supersets of their original language via extensions; similarly, C3's default HTML parser supports extensions that add new HTML tags (which are implemented by new C# DOM classes as described above; see also Section 3.3.1).

An extensible HTML parser has only two dependencies: a means for constructing a new node given a tag name, and a factory method for creating a new node and inserting it into a tree. This interface is far simpler than that of any DOM node, and so exists as the separate INode interface. The parser has no hard dependency on a specific DOM implementation, and a minimal implementation of the INode interface can be used to test the parser independently of the DOM implementation. The default parser implementation is given a DOM node factory that can construct INodes for the built-in HTML tag names. Extending the parser via this factory is discussed in Section 3.3.1.

3.2.6 *Computing visual structure*

C3 separates the interesting computation of layout from the mechanical steps of displaying it into two disjoint subsystems. The *layout engine* takes a document and its stylesheets, and produces as output a *layout tree*, an intermediate data structure that contains sufficient information to display a visual representation of the document. The *renderer* then consults the layout tree to draw the document in a platform- or toolkit-specific manner.

The layout engine's job of computing a layout tree requires three steps, described below: first, DOM nodes are attributed with style information according to any present stylesheets; second, the layout tree's structure is determined; and third, nodes of the layout tree are annotated with concrete styles (placement and sizing, fonts and colors, etc.) for the renderer to use. Each of these steps admits a naïve reference implementation, but both more efficient and more extensible algorithms are possible. I focus on the former here; layout extensibility is revisited in Section 3.3.3.

Assigning node styles The algorithm that decorates DOM nodes with CSS styles does not depend on any other parts of layout computation. Despite the top-down implementation suggested by the name "cascading style sheets", several efficient strategies exist, including recent and ongoing research in parallel approaches [156].

The default style "cascading" algorithm is self-contained, single-threaded and straightforward. It decorates each DOM node with an immutable calculated style object, which is then passed

to the related layout tree node during construction. This immutable style suffices thereafter in determining visual appearance.

Determining layout tree structure The layout tree is generated from the DOM tree in a single traversal. The two trees are approximately the same shape; the layout tree may omit nodes for invisible DOM elements (e.g., ⟨**script**/⟩), and may insert "synthetic" nodes for so-called generated content (e.g., the numbers in a numbered list) or to simplify later layout invariants. For consistency, this transformation must be serialized between DOM mutations, and so runs on the DOM thread (see Section 3.2.9). The layout tree must preserve a mapping between DOM elements and the layout nodes they engender, so that mouse movement (which occurs in the renderer's world of screen pixels and layout tree nodes) can be routed to the correct target node (i.e., a DOM element). A naïve pointer-based solution runs afoul of an important design decision: C3's architectural goals of modularity require that the layout and DOM trees share no pointers. Instead, all DOM nodes are given unique numeric ids, which are *preserved* by the DOM-to-layout tree transformation. Mouse targeting can now be defined in terms of these ids while preserving pointer-isolation of the DOM from layout: while numeric ids are conceptually similar to pointers, they cannot be used for thread-unsafe dereferences of the DOM nodes; see Section 3.2.9.

Solving layout constraints The essence of any layout algorithm is to solve constraints governing the placement and appearance of document elements. In HTML, these constraints are irregular and informally specified (if at all). Consequently the constraints are typically solved by a special-purpose, multi-pass algorithm over the layout tree, rather than a generic constraint-solver [28, 111, 156]. The manual algorithms found in production HTML platforms are often tightly optimized to eliminate some passes for efficiency.

C3's architecture admits such optimized approaches, too; the reference implementation keeps the steps separate for clarity and ease of experimentation. Indeed, because the layout tree interface does not assume a particular implementation strategy, several layout algorithm variants have been explored in C3 with minimal modifications to the layout algorithm or components dependent on the computed layout tree.

3.2.7 *The browser kernel and window proxies*

All security-relevant decisions are made by a *browser kernel* [45, 223] that mediates origin comparisons, network and file I/O requests, and (eventually) thread scheduling decisions. The DOM is based roughly on a capability model: if a script can reference a DOM node or object, it may call

any of its methods. Accordingly, access to DOM nodes in different documents is gated by accessing the `window` object of the foreign document: there is no other way to obtain a reference to a node in a foreign document than by first obtaining a reference to its `window` object and accessing its properties. Whether these property accesses succeed or throw a security exception depends on the security policy of the browser; in the same-origin policy [188], only accesses from documents in the same origin will succeed. Rather than scatter the relevant security checks throughout the browser kernel, `C3` contains two implementations of `window` objects: a `Window` will always access properties successfully, while a `WindowProxy` wraps an underlying `Window` and for nearly every property will always throw a security exception.

Window objects also have a handful of properties that themselves return `window` objects (e.g., the `opener` of a popup, or the child `frames` contained within a window), and these properties can be used to navigate the relationships between windows, regardless of origin.[2] Therefore, the implementations of these properties on both `Window` and `WindowProxy` always succeed, and return either the requested window or a proxied version, as determined by the browser kernel and the current executing script context. In this way, the implementation maintains the invariant that (nodes from) documents from the same origin can always access each other's unproxied `Window` objects, while (nodes from) documents from separate origins can only ever see `WindowProxy`s. This in turn maintains the same-origin policy (but see the discussion of extensions and security, below).

3.2.8 Accommodating privileged UI

Both Firefox and Chrome implement some (or all) of their user interface (e.g., address bar, tabs, etc.) in declarative markup, rather than hard-coded native controls. In both cases this gives the browsers increased flexibility; it also enables Firefox's extension ecosystem. The markup used by these browsers is *trusted*, and can access internal APIs not available to web content. To distinguish the two, trusted UI files are accessed via a different URL scheme: e.g., Firefox's main UI is loaded using `chrome://browser/content/browser.xul`.

I chose to implement the prototype browser's UI in `HTML` for two reasons. First, it is an experiment in writing sophisticated applications entirely within the web platform, and provides first-hand experience with likely challenges. Even in the prototype, such experience led to the two security-related changes described below. Second, having the UI in `HTML` enables the extensions described in Section 3.3; the entirety of a `C3`-based application is available for extension. Like

[2] Indeed, permitting these accesses while restricting all others is a tricky piece of engineering, particularly when re-architecting a browser for multiple-process support; see [181] for details.

Firefox, C3 supports a privileged URL scheme: launching C3 with a command-line argument of `chrome://browser/tabbrowser.html` will display the browser UI. Launching it with the URL of any website will display that site without any surrounding browser chrome. Currently, only HTML file resources bundled within the C3 assembly itself may be given privileged `chrome://` URLs.

Designing this prototype exposed deliberate limitations in HTML when examining the navigation history of child windows (popups or ⟨**iframe**/⟩s): the APIs restrict access to same-origin sites only, and are write-only. A parent window cannot see what site a child is on unless it is from the same origin as the parent, and can never see what sites a child has visited. A browser must avoid both of these restrictions so that it can implement the address bar.

Rather than change API visibility, C3 extends the DOM API in two ways. First, it gives privileged pages (i.e., from `chrome://` URLs) a new `childnavigated` notification when their children are navigated, just before the `unload` events that the children already receive. Second, it treats `chrome://` URLs as trusted origins that *always pass same-origin checks*. The trusted-origin mechanism and the custom navigation event suffice to implement the browser UI.

3.2.9 Threading architecture

One important point of flexibility is the mapping between threads and the HTML platform components described above. C3 does not impose any threading discipline beyond necessary serialization required by HTML and DOM standards. This is made possible by the decision to prevent data races by design: in this architecture, data is either immutable, or it is not shared among multiple components. Thus, it is possible to choose any threading discipline within a single component; a single thread could be shared among all components for debugging, or several threads could be used within each component to implement worker queues.

Below, I describe the default allocation of threads among components, as well as key concurrency concerns for each component.

The DOM/JS thread(s)

The DOM event dispatch loop and JS execution are single-threaded within a set of related web pages.[3] "Separate" pages that are unrelated[4] can run entirely parallel with each other. Thus, sessions with several tabs or windows open simultaneously use multiple DOM event dispatch loops.

---

[3] I ignore for now web-workers, which are an orthogonal concern.

[4] Defining when pages are actually separate is non-trivial, and is a refinement of the same-origin policy, which in turn has been the subject of considerable research [15, 114]

In C3, each distinct event loop consists of two threads: a *mutator* to run script and a *watchdog* to abort run-away scripts. The system maintains the invariant that all mutator threads are *heap-disjoint*: JS code executing in a task on one event loop can only access the DOM nodes of documents sharing that event loop. This invariant, combined with the single-threaded execution model of JS (from the script's point of view), means all DOM nodes and synchronous DOM operations can be lock-free. (Operations involving local storage are asynchronous and must be protected by the storage mutex.) When a window or ⟨**iframe**/⟩ is navigated, the relevant event loop may change. An event loop manager is responsible for maintaining the mappings between windows and event loops to preserve the disjoint-heap invariant.

Every DOM manipulation (node creation, deletion, insertion or removal; attribute creation or modification; etc.) notifies any registered DOM *listener* via a straightforward interface. One such listener is used to inform the layout engine of all document manipulations; others could be used for testing or various diagnostic purposes.

The layout thread(s)

Each top-level browser window is assigned a *layout* thread, responsible for resolving layout constraints as described in Section 3.2.6. Several browser windows might be simultaneously visible on screen, so their layout computations must proceed in parallel for each window to quickly reflect mutations to the underlying documents. Once the DOM thread computes a layout tree, it transfers ownership of the tree to the layout thread, and begins building a new tree. Any external resources necessary for layout or display (such as image data), are also passed to the layout thread as uninterpreted .Net streams. This isolates the DOM thread from any computational errors on the layout threads.

The UI thread

It is common for GUI toolkits to impose threading restrictions, such as only accessing UI widgets from their creating thread. These restrictions influence the platform insofar as *replaced elements* (such as buttons or text boxes) are implemented by toolkit widgets.

C3 is agnostic in choosing a particular toolkit, but rather exposes abstract interfaces for the few widget properties actually needed by layout. The prototype currently uses the .Net WinForms toolkit, which designates one thread as the "UI thread", to which all input events are dispatched and on which all widgets must be accessed. When the DOM encounters a replaced element, an actual WinForms widget must be constructed so that layout can in turn set style properties on that

widget. This requires synchronous calls from the DOM and layout threads to the UI thread. Note, however, that *responding* to events (such as mouse clicks or key presses) is *asynchronous*, due to the indirection introduced by numeric node ids: the UI thread simply adds a message to the DOM event loop with the relevant ids, which the DOM thread will eventually retrieve and dispatch to the relevant event handlers.

3.3 C3 Extension points

The extension mechanisms I introduce into C3 stem from a principled examination of the various semantics of HTML. When using a webapp, our interactions with them tacitly rely on manipulating HTML in two distinct ways: webapps rely on the platform to interpret it *operationally* via the DOM and JS code, and also to interpret it *visually* via CSS and its associated layout algorithms. Teasing these interpretations apart leads to the following two transformation pipelines:

- JS global object + HTML source[1,2]
  $\xrightarrow{\text{HTML } \textit{parsing}}$ [3] DOM subtrees[4]
  $\xrightarrow{\texttt{onload}}$ DOM document[5]
  $\xrightarrow{\text{JS } \textit{events}}$ DOM document ...
- DOM document + CSS source[6]
  $\xrightarrow{\text{CSS } \textit{parsing}}$ CSS content model[7]
  $\xrightarrow{\textit{layout}}$ CSS box model

The first pipeline distinguishes four phases of the document lifecycle, from textual sources through to the event-based running of JS: the initial `onload` event marks the transition point after which the document is asserted to be fully loaded; before this event fires, the page may be inconsistent as critical resources in the page may not yet have loaded, or scripts may still be writing into the document stream.

Explicitly highlighting these pipeline stages leads to designing extension points in a *principled* way: there is an extension point associated with the inputs accepted or the outputs produced by each stage, as long as the extensions preserve the pipeline structure, and always produce outputs that are acceptable inputs to the following stages. This is in contrast to the extension models of existing browsers, which support various extension points without relating them to

other possibilities or to the browser's behavior as a whole. The extension points engendered by the pipelines above are (as numbered):

1. Before beginning HTML parsing, extensions may provide *new tag names and* DOM*-node implementations* for the parser to support.

2. Before running any scripts, extensions may *modify the JS global scope* by adding or removing bindings.

3. Before inserting subtrees into the document, extensions may *preprocess* them using arbitrary C# code.

4. Before firing the `onload` event, extensions may declaratively inject new content into the nearly-complete tree using *overlays*.

5. Once the document is complete and events are running, extensions may modify existing event handlers using *aspects*.

6. Before beginning CSS parsing, extensions may provide *new CSS properties and values* for the parser to support.

7. Before computing layout, extensions may provide *new layout box types and implementations* to affect layout and rendering.

Some of these extension points are simpler than others due to regularities in the input language, others are more complicated, and others are as yet unimplemented. Points (1) and (5) are novel to C3. C3 does not yet implement points (6) or (7), though they are planned future work; they are also novel. I explain points (1), (3) and (4) in Section 3.3.1, points (2) and (5) in Section 3.3.2, and finally points (6) and (7) in Section 3.3.3. Points (5) and (4) are the subjects of Chapters 4 and 5, respectively.

3.3.1 HTML parsing/document construction

Point (1): New tags and DOM nodes The HTML parser recognizes concrete syntax resembling ⟨**tagName** attrName="val"/⟩ and constructs new DOM nodes for each tag. In most browsers, the choices of which tag names to recognize, and what corresponding objects to construct, are tightly

```
public interface IDOMTagFactory {
  IEnumerable<Element> TagTemplates { get; }
}
```

```
public class HelloWorldTag : Element {
  string TagName { get { return "HelloWorld"; } }
  ...
}
public class HelloWorldFactory : IDOMTagFactory {
  IEnumerable<Element> TagTemplates { get {
    yield return new HelloWorldTag();
  } }
}
```

Figure 3.6: Factory and simple extension defining new tags

coupled into the parser. In C3, however, both of these decisions are abstracted behind a factory, whose interface is shown in the top of Fig. 3.6.[5] Besides simplifying the code's internal structure, this approach permits extensions to contribute factories too.

The default implementation of this interface provides one "template" element for each of the standard HTML tag names; these templates inform the parser which tag names are recognized, and are then cloned as needed by the parser. Any unknown tag names fall back to returning an HTMLUnknownElement object, as defined by the HTML specification. However, if an extension contributes another factory that provides additional templates, the parser seamlessly can clone *those* instead of using the fallback: effectively, this extends the language recognized by the parser, as XML3D needed, for example. A trivial example that adds support for a ⟨**HelloWorld**/⟩ tag is shown in Fig. 3.6. A more realistic example is used by C3 to support overlays (see Fig. 3.8 and below).

The factory abstraction also provides the flexibility to support additional experiments: rather than adding *new* tags, a researcher might wish to *modify existing tags*. Therefore, factories are permitted to provide a new template for existing tag names—with the requirement that at most one extension does so per tag name. This permits extensions to easily subclass the C3 DOM implementation, e.g., to add instrumentation or auditing, or to modify existing functionality. Together, these extensions yield a parser that accepts a superset of the standard HTML tags and still produces a DOM tree as output. (Note that tag names cannot currently be *removed* from the parser; exten-

[5] Firefox seems not to use a factory; Chrome uses one, but the choice of factory is fixed at compile-time. C3 can load factories dynamically.

```
public interface IParserMutatorExtension {
  IEnumerable<string> TagNamesOfInterest { get; }
  void OnFinishedParsing(Element element);
}
```

Figure 3.7: The interface for HTML parser semantic actions

sion authors wishing to filter the node types present in a document can use the next extension point presented below.)

Point (3): Preprocessing subtrees The HTML5 parsing algorithm produces a document tree in a bottom-up manner: nodes are created and then attached to parent nodes, which eventually are attached to the root DOM node. Compiler-authors have long known that it is useful to support *semantic actions*, callbacks that examine or preprocess subtrees as they are constructed. Indeed, the HTML parsing algorithm itself specifies some behaviors that are essentially semantic actions, e.g., "when an ⟨**img**/⟩ is inserted into the document, download the referenced image file". Extensions might use this ability to collect statistics on the document, or to sanitize it during construction. These actions typically are local—they examine just the newly-inserted tree—and rarely mutate the surrounding document. (In HTML in particular, because inline scripts execute *during* the parsing phase, the document may change arbitrarily between two successive semantic-action callbacks, and so semantic actions will be challenging to write if they are not local.)

Extensions in C3 can define custom semantic actions using the interface shown in Fig. 3.7. The interface supplies a list of tag names, and a callback to be used when tags of those names are constructed.

Point (4): Document construction Firefox pioneered the ability to define both application UI and extensions to that UI using a single declarative markup language (XUL), an approach whose success is witnessed by the variety and popularity of Firefox's extensions. The basic construction is the *overlay*, which behaves like a "tree-shaped patch": the children of the ⟨**overlay**/⟩ select nodes in a target document and define content to be inserted into or modified within them, much as hunks within a patch select lines in a target text file. C3 adapts and generalizes this idea for HTML.

My implementation adds nine new tags to HTML, shown in Fig. 3.8 (and discussed more fully in Chapter 5), to define overlays and the various actions they can perform. As they are a language extension to HTML, I inform the parser of these new tags using the IDOMTagFactory described

Base constructions

⟨**overlay**/⟩**:** Root node of extension document

⟨**guard resource="***selector***" type="require | reject | first | last"**/⟩**:** Used within ⟨**overlay**/⟩, add guards that restrict when the overlay successfully applies

⟨**insert selector="***selector***" where="before | after"**/⟩**:** Insert new content adjacent to all nodes matched by CSS *selector*

⟨**replace selector="***selector***"**/⟩**:** Replace existing subtrees matching *selector* with new content

⟨**self attrName="value"...**/⟩**:** Used within ⟨**replace**/⟩, refers to node being replaced and permits modifying its attributes

⟨**contents**/⟩**:** Used within ⟨**replace**/⟩, refers to children of node being replaced

Syntactic sugar

⟨**before** ... /⟩ = ⟨**insert** where="before" ... /⟩

⟨**after** ... /⟩ = ⟨**insert** where="after" ... /⟩

⟨**modify** selector="*sel*" where="before"⟩
⟨**self** *new attributes*⟩
new content
⟨/**self**⟩
⟨/**modify**⟩

=

⟨**replace** selector="*sel*"⟩
⟨**self** *new attributes*⟩
new content
⟨**contents**/⟩
⟨/**self**⟩
⟨/**replace**⟩

Figure 3.8: The overlay language for document construction extensions. The bottom set of tags are syntactic sugar. A similar desugaring for ⟨**modify** where="after"/⟩ swaps the order of ⟨**contents**/⟩ with *new content*.

above.[6] Overlays can ⟨**insert**/⟩ or ⟨**replace**/⟩ elements, as matched by CSS selectors. To support *modifying* content, overlays have the ability to refer to the target node (⟨**self**/⟩) or its ⟨**contents**/⟩. Finally, the implementation defines syntactic sugar to make overlays easier to write.

Fig. 3.9 shows a simple but real example used during development of the system, to simulate bulleted lists while generated content support was not yet implemented. It appends a ⟨**style**/⟩ element to the end of the ⟨**head**/⟩ subtree (and fails if no ⟨**head**/⟩ element exists), and inserts a ⟨**span**/⟩ element at the beginning of each ⟨**li**/⟩.

The subtlety of defining the semantics of overlays lies in their interactions with scripts: when should overlays be applied to the target document? Clearly overlays must be applied after the document structure is present, so a strawman approach would apply overlays "when parsing finishes". This exposes a potential inconsistency, as scripts that run *during* parsing would see a partial, not-yet-overlaid document, with nodes *a* and *b* adjacent, while scripts that run after

[6] Technically, this is not sufficient to implement overlays, as they are currently inert. Actually *applying* the overlays requires just one general-purpose callback within the DOM code. This callback could be factored as a standalone, ad-hoc extension point, making overlays themselves truly an extension to C3.

```
⟨overlay⟩
  ⟨modify selector="head" where="after"⟩
    ⟨self⟩
      ⟨style⟩
        li > #bullet { color: blue; }
      ⟨/style⟩
    ⟨/self⟩
  ⟨/modify⟩
  ⟨before selector="li > *:first-child"⟩
    ⟨span class="bullet"⟩&bull;⟨/span⟩
  ⟨/before⟩
⟨/overlay⟩
```

Figure 3.9: Simulating list bullets (in language of Fig. 3.8)

parsing would see an overlaid document where *a* and *b* may no longer be adjacent. However, the HTML specification offers a way out: the DOM raises a particular event, `onload`, that indicates the document has finished loading and is ready to begin execution. Prior to that point, the document structure is in flux—and so I choose to apply overlays *as part of that flux*, immediately before the `onload` event is fired. This may break poorly-coded sites, but in practice has not been an issue with Firefox's extensions.

3.3.2 JS execution

Point (2): Runtime environment Extensions such as Maverick may wish to inject new properties into the JS global object. This object is an input to all scripts, and provides the initial set of functionality available to pages. As an input, it must be constructed before HTML parsing begins, as the constructed DOM nodes should be consistent with the properties available from the global object: e.g., `document.body` must be an instance of `window.HTMLBodyElement`. This point in the document's execution is stable—no scripts have executed, no nodes have been constructed—and extensions are permitted to manipulate the global object as they please. (This could lead to inconsistencies, e.g., if they modify `window.HTMLBodyElement` but do not replace the implementation of ⟨**body**/⟩ tags using the prior extension points. I ignore such buggy extensions for now.)

Point (5): Scripts themselves The extensions described so far modify discrete pieces of implementation, such as individual node types or the document structure, because there exist ways to name each of these resources *statically*: e.g., overlays can examine the HTML source of a page and

write CSS selectors to name parts of the structure. The analogous extension to script code needs to modify the sources of individual functions. Many JS idioms have been developed to achieve this, but they all suffer from JS's *dynamic* nature: function names do *not* exist statically, and scripts can create new functions or alias existing ones at runtime; no static inspection of the scripts' sources can precisely identify these names. Moreover, the common idioms used by extensions today are brittle and prone to silent failure.

C3 includes my prior work (see Chapter 4), which addresses this disparity by modifying the JS compiler to support a *dynamic weaving mechanism* adapting ideas from aspect-oriented programming to advise closures (rather than variables that point to them). Only a dynamic approach can detect runtime-evaluated functions, and this requires compiler support to advise all aliases to a function (rather than individual names). As a side benefit, aspects' integration with the compiler often improves the performance of the advice: examining the sources of twenty Firefox extensions, aspects could express nearly all observed idioms with shorter, clearer and often faster code.

3.3.3 CSS and layout

Discussion An extensible CSS engine permits incrementally adding new features to layout in a modular, clean way. The CSS 3 specifications themselves are a step in this direction, breaking the tightly-coupled CSS 2.1 specification into smaller pieces. A true test of the proposed extension points' expressiveness would be to implement new CSS 3 features, such as generated content or the flex-box model, as extensions. An even harder test would be to extricate older CSS 2 features, such as floats, and re-implement them as compositional extensions. The benefit to successfully implementing these extensions is clear: a stronger understanding of the semantics of (and particularly the interactions between) CSS features.

I discovered the possibility of these CSS extension points quite late, in exploring the consequences of making each stage of the layout pipeline extensible "in the same way" as the DOM/JS pipeline is. To my knowledge, implementing the extension points below has not been done before in any browser, and is planned future work.

Point (6): Parsing CSS values The CSS language can be extended in four ways: 1) by adding new property names and associated values, 2) by recognizing new values for existing properties, 3) by extending the set of selectors, or 4) by adding entirely new syntax outside of style declaration blocks. The latter two are beyond the scope of an extension, as they require more sweeping changes to both the parser and to layout, and are better suited to an alternate implementation of the CSS parser altogether (i.e., a different configuration of C3).

Supporting even just the first two extension points is nontrivial. Unlike HTML's uniform tag syntax, nearly every CSS attribute has its own idiosyncratic syntax:

```
font: italic bold 10pt/1.2em "Gentium", serif;
margin: 0 0 2em 3pt;
display: inline-block;
background-image: url(mypic.jpg);
...
```

However, a style declaration itself is very regular, as it is comprised of a semicolon-separated list of colon-separated name/value pairs. Moreover, the CSS parsing algorithm discards any unparsable attributes (up to the semicolon), and then parse the rest of the style declaration normally. This strategy for error recovery effectively splits each style declaration into individual substrings that can be parsed independently, and therefore suggests an implementation strategy.

Supporting the first extension point—new property names—requires making the parser table-driven and registering value-parsing routines for each known property name. Then, like HTML tag extensions, CSS property extensions can register new property names and callbacks to parse the values. (Those values must never contain semicolons, or else the underlying parsing algorithm would not be able to separate one attribute from another.)

Supporting the second extension point is subtler. Unlike the HTML parser's uniqueness constraint on tag names, here multiple extensions might contribute new values to an existing property; a conflict-detection algorithm must ensure that the syntaxes of such new values do not overlap, or else provide some ranking to choose among them.

Point (7): Composing layout The CSS layout algorithm describes how to transform the document tree (the *content model*) into a tree of boxes of varying types, appearances and positions. Some boxes represent lines of text, while others represent checkboxes, for example. This transformation is not obviously compositional: many CSS properties interact with each other in non-trivial ways to determine precisely which types of boxes to construct. As one example, ⟨**table**/⟩s create table-cell, table-row and (possibly) table-column boxes, unless the `display` property is used to override the defaults, in which case they might create text-line boxes instead... unless of course the `float` property is also present to specify that the whole subtree should be rendered outside the normal text flow. Floats also cause normal text to flow around them, which changes how the text is broken into line boxes, unless the `clear` property is used to specify that text wrapping should not occur. Rather than hard-code these (and many more) interactions, the layout transformation must become

table-driven as well. Then both types of extension above become easy: extensions can create new box subtypes, and patch entries in the transformation table to indicate when to create them.

3.4 Evaluation

The C3 platform has reached a semi-stable and somewhat functional state, and only a few extensions have yet been written. To evaluate the platform, I examine: the performance of the extension points, ensuring that the benefits are not outweighed by huge overheads; the expressiveness, both in the ease of "porting" existing extensions to this model and in comparison to other browsers' models; and the security implications of providing such pervasive customizations.

3.4.1 Performance

Any time spent running the extension manager or conflict analyses slows down the perceived performance of the browser. Fortunately, this process is very cheap: with one extension of each supported type, it costs roughly 100ms to run the extensions. This time includes: enumerating all extensions (27ms), loading all extensions (4ms), and detecting parser-tag conflicts (3ms), mutator conflicts (2ms), and overlay conflicts (72ms). All but the last of these tasks runs just once, at browser startup; overlay conflict detection must run per-page, since conflict detection cannot complete without knowing the structure of the page being overlaid. Enumerating all extensions currently reads a directory, and so scales linearly with the number of extensions. Parser and mutator conflict detection scale linearly with the number of extensions as well; overlay conflict detection is more expensive as each overlay provides more interacting constraints than other types of extensions do. If necessary, these costs can be amortized further by caching the results of conflict detection between browser executions.

3.4.2 Expressiveness

Fig. 3.10 lists several examples of extensions available for IE, Chrome, and Firefox, and the corresponding C3 extension points they would use if ported to C3. Many of these extensions simply overlay the browser's user interface and require no additional support from the browser. Some, such as Smooth Gestures or LastTab, add or revise UI functionality. As the UI is entirely script-driven, I support these via script extensions (point (5)). Others, such as the various Native Client libraries, are sandboxed programs that are then exposed through JS objects; I provide support for the JS objects (point (2)) and .Net provides the sandboxing.

Fig. 3.10 also shows some research projects that are not implementable as extensions in any other browser except C3. As described below, these projects extend the HTML language, CSS layout, and JS environment to achieve their functionality. Implementing these on C3 requires *no hacking of C3*, leading to a much lower learning curve and fewer implementation pitfalls than modifying existing browsers. I examine some examples, and how they might look in C3, in more detail here.

XML3D: Extending HTML, CSS and layout

XML3D [202] is a recent project aiming to provide 3D scenes and real-time ray-traced graphics for web pages, in a declarative form analogous to ⟨**svg**/⟩ for two-dimensional content. This work uses XML namespaces to define new scene-description tags and requires *modifying each browser* to recognize them and construct specialized DOM nodes accordingly. To style the scenes, this work must modify the CSS engine to recognize new style attributes. Scripting the scenes and making them interactive requires constructing JS objects that expose the customized properties of the new DOM nodes. It also entails informing the browser of a new scripting language (AnySL) tailored to animating 3D scenes.

Instead of modifying the browser to recognize new tag names, a C3 version would use extension point (1) to define them in an extension, and subclass the ⟨**script**/⟩ tag to recognize AnySL. Similarly, it could provide new CSS values and new box subclasses for layout to use. The full XML3D extension would consist of these four extension hooks and the ray-tracer engine.

Maverick: Extensions to the global scope

Maverick [185] aims to connect devices such as webcams or USB keys to web content, by writing device drivers in JS and connecting them to the devices via Native Client (NaCl) [230]. NaCl exposes a socket-like interface to web JS over which all interactions with native modules are multiplexed. To expose a more DOM-like API, Maverick injects an actual DOM ⟨**img**/⟩ node into the document, stashing state within it, and using JS properties on that object to communicate with NaCl. This object can then transliterate the image frames from the webcam into Base64-encoded `src` URLs on the the ⟨**img**/⟩, and so reuse the browser's image decoding libraries.

There are two main difficulties with Maverick's implementation that could be avoided in C3. Fist, NaCl isolates native modules in a strong sandbox that prevents direct communication with resources like devices; Maverick could not be implemented in NaCl without *modifying the sandbox* to expose a new system call and writing untrusted glue code to connect it to JS. Second, all these API marshaling functions are mutable JS properties; in particular, the internal state of the DOM

node and the convenience functions on it are all manipulable by web script, potentially breaking communication protocols the drivers expect. Ultimately, using a DOM node to expose a device is not the right abstraction: it is not a node in the document but rather a global JS object like XMLHttpRequest. And while using Base64-encoded URLs is a convenient implementation trick, it would be far more natural to call the image-decoding libraries directly, avoiding both overhead and potential transcoding errors.

RePriv: Extensions hosting extensions

RePriv [86] runs in the background of the browser and mines user browsing history to infer personal interests. It carefully guards the release of that information to websites, via APIs whose uses can be verified to avoid unwanted information leakage. At the same time, it offers its own extension points for site-specific "interest miners" to use to improve the quality of inferred information. These miners are all scheduled to run during an onload event handler registered by RePriv. Finally, extensions can be written to use the collected information to reorganize web pages at the client to match the user's interests.

While this functionality is largely implementable as a plug-in in other browsers, several factors make it much easier to implement in C3. First and foremost, RePriv's security guarantees rely on C3 being entirely managed code: the browser can be removed from RePriv's trusted computing base by isolating RePriv extensions in an AppDomain and leveraging .Net's memory safety guarantees. Obtaining such a strong security guarantee in other browsers is at best very challenging. Second, the document construction hook makes it trivial for RePriv to install the onload event handler. Third, AppDomains ensure the memory isolation of every miner from each other and from the DOM of the document, except as mediated by RePriv's own APIs; this makes proving the desired security properties much easier. Finally, RePriv uses Fine [100] for writing its interest miners; since C3, RePriv and Fine target .Net, RePriv can reuse .Net's assembly-loading mechanisms.

3.4.3 Other extension models

Shadow DOMs

An extension author who wishes to inject new content into structured positions within an existing document can use the overlay mechanism of C3 to do so. An extension author who wants to construct a whole new kind of HTML node can do so in C3 by extending the HTML parser and implemented the new node. However, these new node types are frequently *widgets*, composite controls that can be implemented using existing HTML/CSS/JS techniques but that ought to behave

as a single logical unit. Nearly all major JS libraries include an API for simulating widgets using HTML subtrees, but inter-framework incompatibilities hinder widget reuse [228].

Several techniques have been (or are being) proposed, with varying success and popularity, to simplify this widget-creation task [89, 90, 159, 218, 228]. Each of these proposals relies on a notion of "shadow DOMs" that imbue elements of a DOM tree with a hidden internal structure. These *bindings* are a declarative mix of XML, HTML, CSS and JS that is invisible to DOM manipulation methods (such as `Element.children`) but that is utilized for rendering and user-interaction. For example, one binding for a desktop-like ⟨**scrollbar**/⟩ tag might internally contain three ⟨**button**/⟩ elements in a ⟨**div**/⟩ to implement the endpoints, thumb, and track of a typical scrollbar. Any mouse clicks on these elements would appear as clicks on the scrollbar element itself. Another, more phone-like binding might forego the track and end buttons, and only include the thumb.

The primary advantage to shadow DOM-like schemes is that they enable self-hosting of HTML controls (like buttons or text boxes) in HTML. This eliminates many of the special cases needed to handle so-called "replaced content" in HTML. However, the major downside is that there is no agreed-upon semantics for how these bindings themselves may be composed or extended. In many regards, the range of proposed semantics for these bindings resembles the range of semantics between object-oriented inheritance and mixins.

Extensions to application UI

Internet Explorer 4.0 introduced two extension points permitting customized toolbars (Explorer Bars) and context-menu entries. These extensions were written in native C++ code, had full access to the browser's internal DOM representations, and could implement essentially any functionality they chose. Unsurprisingly, early extensions often compromised the browser's security and stability. IE 8 later introduced two new extension points that permitted self-updating bookmarks of web-page snippets (Web Slices) and context-menu items to speed access to repetitive tasks (Accelerators), providing safer implementations of common uses for Explorer Bars and context menus.

The majority of IE's interface is not modifiable by extensions. By contrast, Firefox explored the possibility that entire application interfaces could be implemented in a markup language, and that a declarative extension mechanism could *overlay* those UIs with new constructions. Research projects such as Perspectives change the way Firefox's SSL connection errors are presented, while others such as Xmarks or Weave synchronize bookmarks and user settings between multiple browsers. The UI for these extensions is written in precisely the same declarative way as Firefox's own UI, making it as simple to extend Firefox's browser UI as it is to design any website.

But the single most compelling feature of these extensions is also their greatest weakness: they permit implementing features that were never anticipated by the browser designers. End users can then install multiple such extensions, thereby losing any assurance that the composite browser is stable, or even that the extensions are compatible with each other. Indeed, Chrome's carefully curtailed extension model is largely a reaction to the instabilities often seen with Firefox extensions. Chrome permits extensions only minimal change to the browser's UI, and prevents interactions between extensions. For comparison, Chrome directly implements bookmarks and settings synchronization, and now permits extension context-menu actions, but the Perspectives behavior remains unimplementable by design.

The current design for overlays is based strongly on Firefox's declarative approach, but provides stronger semantics for overlays so that it can detect and either prevent or correct conflicts between multiple extensions. It also generalized several details of Firefox's overlay mechanism for greater convenience, without sacrificing its analyzability.

Extensions to scripts

Almost the entirety of Firefox's UI behaviors are driven by JS, and extensions can manipulate those scripts to customize those behaviors. A similar ability lets extensions modify or inject scripts within web pages. Extensions such as LastTab change the tab-switching order from cyclic to most-recently-used, while others such as Ghostery block so-called "web tracking bugs" from executing. Firefox exposes a huge API, opening basically the entire platform to extension scripts. This flexibility poses a problem: multiple extensions may attempt to modify the same scripts, often leading to broken or partially-modified scripts with unpredictable consequences.

Modern browser extension design, like Firefox's jetpacks or Chrome's extensions, are typically developed using HTML, JS, and CSS. While jetpacks are currently still fully-privileged, Chrome extensions run in sandboxed processes that cannot access privileged information and cannot crash or hang the browser. While these guarantees are vital to the stability of a commercial system protecting valuable user information, they also restrict the power of extensions.

The Fine project [100] is one attempt to curtail these scripts' mutual interactions within web pages. Instead of directly using JS, authors use a dependently-typed programming language to express the precise read- and write-sets of extension scripts, and a security policy constrains the information flow between them. Extensions that satisfy the security policy are provably non-conflicting. The Fine project targets C3 easily, either by compiling its scripts to .Net assemblies and loading them dynamically (by subclassing the ⟨**script**/⟩ tag), or by statically compiling its scripts to JS and

dynamically injecting them into web content (via the JS global-object hook). Guha et al. successfully ported twenty Chrome extensions into Fine to run on C3 with minimal developer effort.

As mentioned earlier, C3 includes my prior work on aspects for JS (Chapter 4), permitting extensions clearer language mechanisms to express how their modifications apply to existing code. Beyond the performance gains and clarity improvements, by eliminating the need for brittle mechanisms and exposing the intent of the extension, compatibility analyses between extensions become feasible.

3.4.4 Security considerations

Of the five implemented extension points, two are written in .Net and have full access to the DOM internals. In particular, new DOM nodes or new JS runtime objects that subclass the implementation may use protected DOM fields inappropriately and violate the same-origin policy. I view this flexibility as both an asset and a liability: it permits researchers to experiment with alternatives to the SOP, or to prototype enhancements to HTML and the DOM. At the same time, I do not advocate these extensions for web-scale use. The remaining extension points are either limited to safe, narrow .Net interfaces or are written in HTML and JS and inherently subject to the SOP. Sanitizing potentially unsafe .Net extensions to preserve the SOP is itself an interesting research problem. Possible approaches include using .Net AppDomains to segregate extensions from the main DOM, or static analyses to exclude unsafe accesses to DOM internals.

3.5 Future work

I have focused so far on the abilities extensions have within the C3 system. However, the more powerful extensions become, the more likely they are to conflict with one another. Indeed, part of the point of designing the extension mechanisms in C3 as I have is to *expose* this fact, and thereby make these conflicts amenable to detection and resolution. Certain extension points are easily amenable to conflict detection: for example, two parser tag extensions cannot both contribute the same new tag name, so conflict detection becomes trivial and so is not discussed further. Others are more challenging: as explained in Chapter 1, precisely defining conflicts between JS runtime extensions is a more challenging task; Chapter 4 discusses this briefly but leaves much of it to future work. Finally, some extension points admit a range of conflict detection schemes, depending on their precise expressive power: this will be the main focus of Chapter 5, which discusses variations on the overlay mechanism of C3.

Assuming a suitable notion of extension conflict exists for each extension type, it falls to the

extension loading mechanism to ensure that, whenever possible, conflicting extensions are not loaded. In some ways this is very similar to the job of a compile-time linker, ensuring that all modules are compatible before producing the executable image. Such load-time prevention gives users a much better experience than in current browsers, where problems never surface until runtime. However not all conflicts are detectable statically, and so some runtime mechanism is still needed to detect conflict, blame the offending extension, and prevent the conflict from recurring.

3.6 Summary

I have presented C3, a platform implementing of HTML, CSS and JS, and explored how its design was tuned for easy reconfiguration and runtime extension. I presented several motivating examples for each extension point, and confirmed that the design is at least as expressive as existing extension systems, supporting current extensions as well as new ones not previously possible.

Extensions	Available from	C3-equivalent extension points used
IE:		
Explorer bars		(4) overlay the main browser UI
Context menu items		(4) overlay the context menu in the browser UI
Accelerators		(4) overlay the context menu
WebSlices		(4) overlay browser UI
Chrome:		
Gmail checkers	`https://chrome.google.com/extensions/search?q=gmail`	(4) overlay browser UI, (5) script advice
Skype	`http://go.skype.com/dc/clicktocall`	(4) overlay browser UI, (2) new JS objects, (5) script advice
Smooth Gestures	`http://goo.gl/rN5Y`	(4) overlay browser UI, (5) script advice
Native Client libraries	`http://code.google.com/p/nativeclient/`	(2) new JS objects
Firefox:		
TreeStyleTab	`https://addons.mozilla.org/en-US/firefox/addon/5890/`	(4) overlay tabbar in browser UI, inject CSS
LastTab	`https://addons.mozilla.org/en-US/firefox/addon/112/`	(5) script advice
Perspectives	[227]	(5) script extensions, (4) overlay error UI
Firebug	`http://getfirebug.com/`	(4) overlays, (5) script extensions, (2) new JS objects
Research projects:		
XML3D	[202]	(1) new HTML tags, (6) new CSS values, (7) new layouts
Maverick	[185]	(2) new JS objects
Fine	[100]	(1) HTML ⟨**script**/⟩ tag replacement
RePriv	[86]	(2) new JS objects

Figure 3.10: Example extensions in IE, Firefox, and Chrome, as well as research projects best implemented in C3, and the C3 extension points that they might use

Chapter 4

JS ASPECTS[1]

4.1 Introduction

The software engineering challenges of building robust web applications are readily apparent: the design, development, and deployment model for client-side web code is, for better or worse, very different than the model for traditional desktop applications. Code is delivered in source form, at run-time, from multiple sources, and in a language (JS) that is highly dynamic and encourages run-time creation and evaluation of more code.

Consider just the JS code that runs when a user visits a typical "Web 2.0" site. The main page will use JS to provide an interactive experience, probably incorporating popular third-party JS libraries. Ads on the side will contain separately developed scripts. User-installed browser extensions or userscripts include yet more JS to affect browser functionality and how pages are displayed. In short, the code running for a page is a run-time conglomeration of scripts from multiple sources with multiple purposes.

Matters get worse: The entire purpose of some JS (e.g., in a browser extension) is to change the behavior of other JS (e.g., on a popular web page) in ways the affected code never expected. To do so, programmers (ab)use JS features such as the redefinition of top-level functions and the ability to retrieve a function's source code at run-time using its `toString()` method, rewrite it in some manner, and evaluate the result. While it is easy to dismiss such shenanigans as unprincipled hacks unworthy of serious programming-languages study, in this chapter I choose instead to accept this reality. Third-party modifications are extremely popular, with tens of thousands of extensions run by millions of users every day. It is unreasonable to expect web programmers to modify the behavior of web applications only in ways the application writers allow and prepare for.

4.1.1 Aspects for JavaScript

I propose adding to JS features that make third-party code modifications straightforward, with clear semantics that avoid the pitfalls and awkwardness of current practice. My approach adapts

[1] This chapter is based on an earlier work, originally published in OOPSLA'10 [137] and also published as: Supporting dynamic, third-party code customizations in JavaScript using aspects, in SIGPLAN Notices, {Vol. 45, 0362-1340, (October)} © ACM, 2010. http://doi.acm.org/10.1145/1932682.1869490

and extends mechanisms from aspect-oriented programming to the world of web applications and JS. The motivation here differs from the conventional motivation for aspects: rather than better modularizing code by separating cross-cutting concerns, aspects are used to specify code modifications by third parties concisely and accurately.

This approach is the first JS just-in-time compiler with support for aspects, taking advantage of the ideas behind recent advances in JIT compilation for JS [87]. Implementing aspects within the JIT offers better performance and completeness than prior approaches. Most aspect systems statically compile aspects into the code they modify. But the web domain has too much code arriving and being generated dynamically for this to make sense: aspect weaving at runtime permits advising of all JS code, regardless of its source. Moreover, aspects can be implemented more efficiently via runtime support than via source-to-source transformation.

JS web code is imperative and event-driven, with first-class functions and prototype-based object hierarchies. These features, combined with the unusual scoping and evaluation-order rules of JS, form a very different substrate on which to define aspects than more well-known efforts for object-oriented or functional languages. I also demonstrate that, despite efforts to the contrary, aspects cannot be properly provided as a JS library; support from the JS implementation is needed.

To evaluate this work, I have taken several popular third-party extensions and rewritten them using my aspects. The resulting code is shorter, simpler, and faster than existing idioms. To justify the features present in this language, I have examined twenty popular Firefox extensions, measuring the fraction of their code involved in aspect-like behavior and how frequently each type of advice occurs. The language presented here supports nearly every idiom I encountered, and every feature of the design is used significantly often.

Note that while this work required solving some technical issues peculiar to JS, many of the contributions transcend this language. Given that web applications will be changed and rewritten at run-time in unexpected ways by third-party scripts, linguistic mechanisms are needed to support such change in (relatively) robust ways.

4.1.2 Outline

Section 4.2 presents two examples of extensions—one userscript, one browser-level—and the idiomatic hooks they use to install their code. Section 4.3 introduces the key concepts of aspect-oriented programming and revises the examples to use aspects instead. Section 4.4 lays out the key properties that an dynamic weaving system for JS should support, then presents the aspect language in full. Section 4.5 describes the implementation. Section 4.6 evaluates effectiveness

```
var oldP = unsafeWindow.P;
unsafeWindow.P = function(iframe, data) {
  if (data[0] == "mb")
    data[1] = format(data[1]);
  return oldP.apply(iframe, arguments);
}
```

Figure 4.1: Central hook used to install a text formatter into Gmail. Note: The common DOMwindow object is available via the `unsafeWindow` alias.

and efficiency. Section 4.7 discusses related work. Section 4.8 discusses future work. Section 4.9 summarizes the contributions of this chapter.

4.2 Extensible Web-Programming Examples

To understand the nature of code that modifies web applications, consider two examples. The first modifies a web page running Gmail, Google's webmail client, to reformat the display of emails. The second modifies the Firefox browser to change the behavior of opening a new tab. Section 4.3 revisits these examples to show how aspects provide cleaner and more robust solutions.

4.2.1 Reformatting messages in Gmail

Many email clients detect `/italic/`, `_underlined_` and `*bold*` text using punctuation and format the text appropriately. This functionality is not present in Gmail, so a *userscript*[2] was written to add this feature, replacing, e.g., `/text/` with ⟨**i**⟩text⟨/**i**⟩. A userscript defines code and a set of pages on which it should be run. When a page finishes loading, all relevant userscripts are appended to the page and executed. For instance, this userscript runs for URLs matching `http://mail.google.com/mail/*`. Browsers may support userscript loading directly (as in Chrome) or via an extension (as in Greasemonkey for Firefox).

If all the email data were already present in the page's DOM, it would be simple to identify messages and format them. But like all modern webapps, Gmail fetches data lazily. All processing of that data begins with a function at global scope and so the userscript hooks into this function to preprocess the data. That crucial hook is shown in Fig. 4.1.

The basic idea is to replace Gmail's `unsafeWindow.P` function with a new function that `formats` the data when a message body (`"mb"`) arrives, and then proceeds to call the original code bound to `P` with the modified data. This technique is known as *wrapping*. The use of `JS`'s `apply` method

[2] `http://userscripts.org/scripts/show/8178`

```
SpeedDial.init = function () {
  ...
   eval("getBrowser().removeTab ="+
     getBrowser().removeTab.toString().replace('this.addTab("about:blank");',
      'if (SpeedDial.loadInLastTab) {'
     +'  this.addTab("chrome://speeddial/content/speeddial.xul"'
     +')} else { this.addTab("about:blank")}'
     ));
  ...
  if (SpeedDial.clearURLBarOnLoad) {
    if (!SpeedDial.isFirefox3) {
      ...
    } else {
      var newLocationChange =
         window.URLBarSetURI.toString().replace(/aURI.spec == \"about:blank\"/g,
           'aURI.spec == "about:blank" || '
                        +'(aURI.spec.indexOf("chrome://speeddial/content") == 0)');
      eval('window.URLBarSetURI = ' + newLocationChange + ';');
    }
  }
}
```

Figure 4.2: Central hooks used to modify the Firefox blank tab

is necessary so that the implicit `this` parameter of `P` is bound properly. This unintuitive idiom is common, but my approach using aspects makes it largely unnecessary.

4.2.2 SpeedDial: Customizing new tabs in Firefox

The Opera and Safari browsers offer a home page showing a grid of the user's most frequently visited sites. This feature is not built into Firefox, but two extensions have been written to add it. One in particular, SpeedDial, takes care to interact with other features of the browser: when a blank new tab appears, a page of thumbnails of frequently visited sites should appear instead, but the browser's address bar should optionally (depending on user preference) remain blank (as it does for normal blank tabs). Fig. 4.2 shows a simplified version of the main hook that installs this change; understanding the details of this code is tedious and unnecessary.

As in the Gmail example, the code is redefining a method. However, it is doing so by rewriting the source-code string of the original method—using regular expressions to find and replace text—and then calling `eval` on the resulting string. This technique is error-prone and brittle, and it produces code that is difficult to analyze or maintain. Yet this precise sequence—retrieve the source-code string, manipulate it, call `eval`—is so common that it has a name in web pro-

gramming: *monkey-patching*. My goal in this chapter is to design an aspect system expressive enough to obviate this idiom.

4.2.3 Discussion

The two examples above are typical representatives of extensions. They interact with the structure of the page or browser but need to patch the underlying scripts to enable their behaviors. Such patches are more the rule than the exception: this is "how things are done" on the web. There are nearly 40,000 userscripts and over 6,000 Firefox extensions available, with millions of daily users: this development model is successful and growing daily.

Sometimes, patches such as these can drive application revision. Thanks to the popularity of userscripts, Google has revised Gmail's code to provide some APIs for userscripts to use [94]. Similarly, some exceptionally popular extensions for Firefox have been merged into the application's core as built-in features [130, 131], and new APIs have been introduced to streamline ungainly workarounds used by many extensions [82]. But not all web applications will be so accommodating, nor will users wait for such support. In general, the underlying programs cannot be expected to plan for such diverse extensions, nor can extension authors be expected to know about and plan for all other extensions. Instead, this research tries to provide a more principled framework for writing and maintaining extensions.

Perhaps the most distasteful facet of monkey-patches is their rampant violation of function abstractions through the use of `replace` and `eval`. In an ideal world, such tricks would be unnecessary. However, when mainline programs do not expose a particular value through a convenient abstraction boundary (as with the `aURI.spec` variable in the SpeedDial example of Fig. 4.2), extension authors must resort to abstraction-violating patches to implement their features. My aim in this chapter is not to eliminate such hacks (which would be futile), but rather to recognize and give structure to commonly-used idioms, preserving abstraction boundaries where possible.

Wrapping and monkey-patching also make it difficult to analyze the impact extensions may have on each other or the mainline program: due to aliasing, code paths that were identical may now diverge, and of course all analyses are greatly complicated by the use of `eval`. By contrast, aspects can eliminate almost all uses of these idioms and so they may make such analyses feasible.

4.3 Using aspects for extensions

An extension is a self-contained unit of code that needs to insert itself into various places in the underlying application to implement its functionality. An extension is similar to an *aspect*,

though "extension-oriented" programming and aspect-oriented programming are very different in motivation. Aspects have traditionally been used for crosscutting concerns such as security monitors or loggers. I propose that the mechanisms of aspects are well-suited to the use case of extending web applications.

I briefly review the terms and concepts of aspect-oriented programming, then introduce my aspect language for JS by rewriting the examples in the previous section. The examples assume an informal description of the language constructs, full and complete definitions follow in Section 4.4.

4.3.1 Key aspect-oriented concepts

As described initially in Section 2.5, an *aspect* inserts new code into an existing *mainline* program to run at specific moments during that program's execution. The new code is called *advice*. Each specific moment is called a *joinpoint*, and sets of joinpoints are called *pointcuts*. There are several kinds of pointcuts: for example, one might be "when function `foo` is called", while another might be "when field `bar` is accessed on object `X`". Aspects also need to specify the type of advice: for instance, *before* or *after* or *around* the call to a function. Sometimes a pointcut might be too broad, describing too many moments during execution. A pointcut can therefore specify a *filter* to restrict the joinpoint to some lexical or dynamic scope: for instance, "all calls to function `foo`, when called by function `bar`". Integrating advice into mainline code is called *weaving*: it incorporates the advice into the mainline such that the advice is triggered at its specified pointcut. Multiple aspects may advise the same pointcut; the weaver uses precedence rules to order the aspects. Advice should rarely call the advised code directly, but rather use a mechanism, usually called `proceed`, to refer to the next installed advice or the underlying mainline, as determined by the weaver.

4.3.2 Advice surrounding functions

Examining the text-formatting example in Section 4.2.1, note that it takes the precise form of *before* advice. The new version of `unsafeWindow.P` inserts its preprocessing *before executing* the original function. This example can be rewritten more concisely as

```
at pointcut(callee(unsafeWindow.P)) before(iframe, data) {
  if (data[0] == "mb")
    data[1] = format(data[1]);
}
```

An aspect must define a pointcut and advice. Here, the pointcut is `callee(unsafeWindow.P)`, and the advice is the `before { }` block. The name "callee" emphasizes that the advice applies to the

code inside `unsafeWindow.P`, rather than inside its caller. Unlike in the userscript, which had to save a reference to and then manually call the original version of `P`, such explicit plumbing is unneeded. Instead, the advice examines and modifies the actual arguments to the function: essentially, the advice is *inlined* into the body of `P`. Traditionally, advice takes a parameter giving reflective access to the arguments of the advised code; this design avoids the indirection and specifies arguments explicitly: `iframe` and `data` are names the aspect binds to the callee's arguments.

4.3.3 Advice within functions

The SpeedDial extension in Section 4.2.2 inserted two hooks into Firefox code, so an equivalent version must define two aspects. Examining the first portion of the code, it is executing a particular statement only in certain conditions. There are several ways to translate this intent. Most literally, one might say:

```
at pointcut(statement_containing(this.addTab("about:blank"))
            && within(getBrowser().removeTab)) around(...) {
  if (SpeedDial.loadInLastTab)
    this.addTab("chrome://speeddial/content/speeddial.xul");
  else
    proceed;
}
```

This example shows a new type of pointcut, describing the nearest enclosing statement containing a particular subexpression. It surrounds that statement with advice that in some cases calls an entirely separate function, and in other cases `proceed`s with the original call. To avoid rewriting every statement containing the specified expression, a filter is used to constrain it `within` the lexical scope of the function `getBrowser().removeTab`. Note that this advice is still quite fragile, but is at least defined in terms of `JS` expressions rather than strings of concrete syntax.

Perhaps the intention was more general: advise *all* calls to `addTab` to open the SpeedDial page:

```
at pointcut(callee(this.addTab)) before(url) {
  if (url == "about:blank" && SpeedDial.loadInLastTab)
    url="chrome://speeddial/content/speeddial.xul";
}
```

(Where, as before, `url` is a name assigned to `addTab`'s parameter.) The extension code, as currently written, gives no indication which of these meanings (or others) were intended.

The second half of the SpeedDial code is actually broken: the mainline code no longer contains the expression `(aURI.spec == "about:blank")`. That means the `replace()` call becomes a no-op, and the monkey patch silently fails. However, the mainline code does set a variable `isBlank`, which likely is the original intent of this extension. This patch can be expressed as:

```
at pointcut(field(isBlank) && within(unsafeWindow.URLBarSetURI)) wrap {
  set { isBlank = (proceed ||
                   (uri.spec.indexOf("chrome://speeddial/content")==0)) }
}
```

This aspect uses the `field` pointcut to denote accessing variables or fields of objects, again filtered `within` the lexical scope of a particular function. The advice itself `wrap`s the variable, in this case only when it is being `set`. Depending on the situation, one might write `get` advice as well.

Note that the `statement_containing` and `field` advice are fragile: they depend on the syntax of the method being advised. If a subexpression is no longer used or a variable is renamed (which is what broke the SpeedDial code), the advice will fail to apply. I cannot prevent that brittleness—when extensions need to modify the internal logic of functions, there may be no simpler alternative. However, unlike monkey patching, using aspects will result in a weaving warning indicating that no joinpoints were found.

4.4 Aspects as a new JS primitive

In Section 4.4.1, I propose a more reasonable semantics for aspect weaving in JS than is possible with wrapping or monkey-patching. My approach ensures that functions may be advised regardless of when they are defined, and regardless of how they are referenced. Implementing this relies on two key features: weaving should occur at runtime rather than compile-time, and should apply to closures rather than variables. These two together give a third key feature, the ability to disable or re-enable installed advice dynamically. In Section 4.4.2, I show that neither wrapping, monkey-patching, nor any other idiom within the JS language can provide these guarantees. Section 4.4.3 describes the full language extensions to JS precisely.

4.4.1 Key features of an aspect primitive

Dynamic weaving Aspect weaving depends on naming functions as targets for advice. Weaving can happen at compile time or at runtime.[3] I argue that dynamic weaving is the only appropriate

[3] The expert JS programmer will note that "compile-time" is ambiguous, encompassing parsing time and function hoisting time as distinct phases before executing the top-level script statements; moreover this repeats for each ⟨**script**/⟩ on

method for JS: code frequently defines anonymous functions, or creates closures at runtime. A static weaver would be unable to advise either of these types of functions, leading to an artificial and unintuitive split between advisable and non-advisable functions. A dynamic weaver has no trouble with dynamically created functions: instead of being triggered based on the static name of the function, it is triggered by the *contents of variables at runtime*.

Weaving into closures Dynamic weaving exposes the distinction between variables and their values. Consider the following snippet:

```
var f = function(x) { return x*x; };
var g = f;
Install Advice: before executing f, print(x)
var h = f;
f(4); g(4); h(4);
```

Suppose the last four lines were in separate ⟨**script**/⟩ sources with an unknown loading order (e.g., as subfiles in a library, ads delivered to a page, or scripts in an application) following the first line. The intent is for `f`, `g`, and `h` to be aliases of one another. Reasonably, one would expect that all three calls in the last line *ought to trigger the advice*: clearly calling `f` should, and `g` and `h` are "the same function". The fact that `g` was defined before the advice was installed should be hidden from the extension author since the loading order is likely unknown. Hence advice should apply to the *underlying closure*, and not to a *variable* bound to the closure. In JS (or any higher-order language), functions may not have unique names, and indeed frequently do have more than one (for instance, by defining a function and subsequently installing it as an event handler in a page). Usually, the developer who uses aspects intends that such aliased functions be advised consistently. Similar conclusions have been reached by others [62].

The ability to advise closures marks the key departure from what is possible within JS: modifying closures cannot be expressed within JS. When this behavior is not desired (i.e., the intended behavior *does* depend on particular aliases), the wrapping idiom is appropriate and still available.

Dynamic disablement Disabling advice arises naturally in the extension setting as extensions provide multiple, mutually-exclusive features that can be selected at runtime by the user. Contrast this form of predicating the execution of advice with the filters mentioned earlier: filters restrict

---

the page. All of these suffer from the same inability to advise anonymous functions, so I consider them all "compile-time" here.

advice to particular lexical or dynamic scopes; disablement restricts advice based on arbitrary runtime decisions.

Disabling advice can be implemented manually by wrapping all advice code in `if`-tests, but this is tedious, and the examples presented earlier only partially implement it: the userscript makes no effort to do so, while the SpeedDial extension is inconsistent, inserting a guard around one piece of advice but not the other. Instead, in this design, aspects are *expressions* in the language that appear as objects, and are equipped with a mutable `disabled` field that selectively disables or re-enables individual advice. This field may be used easily and consistently to "turn off" woven advice.

4.4.2 Aspects cannot be implemented as a library

Both wrapping and monkey-patching rely on replacing the closure bound to a given variable; nothing else is expressible with variables within `JS`. Hence they are incorrect in the presence of aliasing. They also suffer additional, distinct problems.

Because wrapping eventually calls the underlying function, it is limited to adding code before and after the function—it cannot modify the internal control flow of the function. Extensions frequently require this ability, making wrapping unsuitable for their needs. (For instance, the SpeedDial code cannot be written as a wrapper, as it must modify code in the middle of the target function.)

Monkey patching fails in two other critical ways. The essential difference between wrapping and monkey-patching is that the former calls the original closure, while the latter discards it. Consequently, the new function does *not* close over the same environment as the original function—the `eval` happens in a different context. Consider this example:

```
function makeAdder(x) { return function(y) { return x + y; }; }
var addFive = makeAdder(5);
// addFive.toString() == "function(y) { return x + y; }"
```

The closure `addFive` makes use of closed-over variables, but `eval`ing its source code will lose the closure environment: if one were to monkey-patch `addFive`, the new function would use the global value for `x`, if it existed, or fail otherwise. It is impossible, using monkey patching, to determine if a function closed over local variables or not, and so this technique is fatally flawed.

While monkey patching can modify the middle of functions, it is challenging to write precisely the correct replacement operations solely in terms of their textual representation (as opposed to their more structured abstract syntax). The code invariably is obscure, needing fairly long pattern

$e \in$	Expr	$::=$	... | a
			| `retval` | `proceed`
$a \in$	AspectExp	$::=$	`at pointcut(`p`)` ad
$p \in$	FilteredPC	$::=$	b [`&&` f]* | p`||`p
$b \in$	BasePC	$::=$	`callee(`e`)` | `field(`e`)`
			| `statement_containing(`e`)`
$f \in$	Filter	$::=$	`stack(`sd`)` | `within(`e`)`
$sd \in$	StackDesc	$::=$	e[`,` sd] | `!`e[`,` sd]
$ad \in$	Advice	$::=$	`before(`$params$`) {`s`}`
			| `after(`$params$`) {`s`}`
			| `around(`$params$`) {`s`}`
			| `wrap {[get {`e`}][set {`e`}]}`
$params \in$	Params	$::=$	[$ident$[`,` $ident$]*]

Figure 4.3: Aspect syntax for JS

matches to find the right joinpoints, and inserting poorly-formatted and potentially malformed strings as replacement code.[4] Moreover, if the `replace` fails to match, it returns the original code unchanged, which means that monkey patches *fail silently*, making them devilishly hard to debug.

4.4.3 *Language semantics*

My language extensions to JS are shown in Fig. 4.3. I explain the language in stages, focusing on the semantics of each construct. Section 4.5 then shows how to implement these features efficiently, avoiding extra work or code blowup that a naïve implementation of the semantics might incur.

Advising functions: `at pointcut(callee(e))`

Operationally, the simplest form of advice applies to closures: in the grammar above, these are `at pointcut(callee(`e`))` ad `{` s `}` expressions, where ad is one of `before`, `after` or `around`. Such advice is *inlined* into the advised closure. When encountered, each pointcut evaluates e to a (reference to) a closure c, which I represent here as $\langle env, \lambda(x_1, \ldots, x_n)\{s\}\rangle$ where env is the environment when the closure was created. (If e does not evaluate to a closure, abort with a runtime error.) I first define weaving one aspect before explaining how full weaving is defined for all the aspects for a closure.

[4] The expert JS programmer will note that poor formatting is not merely aesthetically problematic: problems may arise due to the interaction of patch code containing newlines and the rules for semicolon insertion.

Given a closure $\langle env, \lambda(y_1, \ldots, y_n)\{s_1\}\rangle$ and advice `before(`$x_1, \ldots, x_n$`) { `s_2` }`, define the new statement $s_2' = s_2[y_1/x_1, \ldots, y_n/x_n]$. To weave the advice I *mutate* the closure, replacing it with $\langle env, \lambda(y_1, \ldots, y_n)\{s_2'; s_1\}\rangle$. (`After` advice is analogous; `around` advice requires slightly more effort.) Notice that the updated closure uses the original environment, which avoids the rebinding-of-variables problem associated with monkey patching. Moreover, because the closure is updated in place, the aliasing problem is resolved: all references to that closure now contain the advice. This neatly includes recursive calls to e: if env contained an entry pointing to this closure, it will subsequently see the now-mutated version.

`Proceed and retval` Generalizing `before` and `after`, `around` advice surrounds the mainline code. To "call" the mainline code, `around` advice uses the `proceed` keyword. This is again essentially inlining: given `around` advice s_2 and mainline s_1 as above, define $s_2' = s_2[y_1/x_1, \ldots, y_n/x_n]$. The woven code is then $s_2'[s_1/\texttt{proceed}]$.

It is common for `after` or `around` advice to refer to the return value of the mainline code. In the body of such advice, I bind the return value to the name `retval`, which then can be read or modified as needed. (Outside advice, `retval` is not a reserved keyword.) Any `return` statements become jumps to subsequent advice; the "last" one sets the final return value, and the caller resumes control only after all advice have run.

Weaving order Overall, weaving for a particular closure c is defined in terms of *all* pointcuts for which the expression e evaluates to c. Take the *original, unadvised* closure $\langle env, \lambda(y_1, \ldots, y_n)\{s\}\rangle$ and ordered lists for before $(b_1, ..., b_i)$, after $(a_1, ..., a_j)$, and around $(r_1, ..., r_k)$ advice. (For simplicity, assume all aspects use the same parameter names as the mainline function; in general I perform the same substitution on parameter names as above.) Each list is ordered by when the pointcut was encountered during program execution. Next, apply each advice one at a time, starting with b_i in order, then a_i in reverse order, and then surrounding them by r_i in order, similar to AspectJ's weaving order [123]. The woven result is then

$$r_1\left[\cdots\left[r_k\left[\{b_1;\cdots;b_i;s;a_j;\cdots;a_1\}/\texttt{proceed}\right]\cdots\right]/\texttt{proceed}\right]$$

This ordering semantics means that weaving cannot be performed eagerly when dynamically evaluating each aspect expression. Instead, the JIT (see Section 4.5.3) stores the aspects for each closure with the closure and weaves while JITing when the closure is invoked. When the collection of aspects for a closure changes, weaving/JITing is redone.

The translation above deliberately does not deal explicitly with exceptional control flow. If any code (mainline or advice) throws an exception, all subsequent code is skipped. However, because advice is truly inlined into the function, `around` advice may surround calls to `proceed` with a `try-catch` statement, and control flow will work properly. (In AspectJ terms, all `after` advice is really `after returning`, and I do not support `after throwing` advice.) Similarly, if `before` advice `return`s early, no mainline code or subsequent advice runs, unless `around` advice uses a `try-finally` statement. This modifying of control flow by advice is surprisingly common behavior by real-world extensions (cf. Fig. 4.7 and the usages of `statement_containing` advice).

Runtime representation of aspects and dynamic disablement Aspects are expressions in this design, and when executed they evaluate to native objects (much like built-in objects, e.g., `Math` or `Array`): `aObj = at pointcut...{...};`. These objects have a mutable `disabled` field that "turns off" the advice. If the program sets `aObj.disabled = true`, then effectively `aObj` is removed from the installed-advice list in the closure, and the advice is rewoven. When the program resets `aObj.disabled = false`, `aObj` is restored to its original position in that list. (The implementation does not actually reweave or re-JIT, but still provides these semantics.)

Named parameters Note that in the example above, and in the abstract syntax for advice, I give names to the parameters of the function. Advice parameters are resolved by position, and do not have to match the parameter names defined by the mainline function. This has the benefit of letting aspects name parameters for native methods (such as `Math.sin`), which do not have explicit JS names for their parameters. By permitting the aspect author to name parameters, I enable a more natural coding style for advice.[5] Traditional aspect style would use a reflective parameters array; JS already includes this via the `arguments` array-like object. However, for technical reasons, *any* usage of the reflective `arguments` object prevents the JS engine from applying certain useful optimizations. Referencing the `arguments` object requires an additional object creation and initialization per function call, and requires an extra indirection when referencing all parameters in that function.

Stack Filters

Extensions frequently need to advise utility or library functions so as to change behaviors of the program. But perhaps not *every* call to those function needs advice; only ones with a particular

---

[5] This is similar to an idiomatic usage of AspectJ's `args` pointcut, though I do not support `args` as a pointcut for JS: all functions take arguments of dynamic types and arities, which defeats the intention of `args`.

call stack should be advised. To achieve this, the language lets developers specify *filters* to constrain which joinpoints match a pointcut: `callee(`e`) && stack(`s`)`. In this design, the `stack` filter generalizes AspectJ's `cflow` pointcut. It permits specifying multiple stack frames that must, or must not, be present when the advice is triggered.

For example, a filter `stack(a, !b, c, d)` states that functions `a`, `c` and `d` must be on the stack in that order, and that `b` must not be on the stack between `a` and `c`. Filters are designed to be greedy—`a` matches the deepest (i.e., oldest) `a` on the stack—and permit arbitrary intervening stack frames between specified frames. Thus, the pattern above will match the stack `main, a, a, c, b, d`, but will not match the stack `main, a, b, a, c, d`. This semantics fits well with extensions' requirements: supporting unspecified intervening frames lets stack patterns continue to match even if other extensions insert themselves into the call stack, while the greediness of negative assertions permits defensively avoiding specific other extensions.

To define the semantics for stack filters more precisely, let $S = e_1 :: \cdots :: e_m :: \top :: []$ be a stack, where e_1 is the deepest stack frame and $\top$ is added after e_m at the young end of the stack. Let `F = f`$_1$`::`$\cdots$`::f`$_n$`::`$\top$`::[]` be a stack filter (again $\top$ is implicitly added). Inductively, `F` matches S in the following cases:

1. `F` $= []$

2. `F = !f`$_1$`::`$\cdots$`::!f`$_n$`::h::F'`, $S = e_1 :: \cdots :: e_m :: g :: S'$, $e_i \neq$ `f`$_j$, $e_i \neq h$, $g =$ `h` and `F'` matches S'

3. `F = f::F'`, $S = e :: S'$, $e \neq$ `f` and `F` matches S'

4. `F = f::F'`, $S = e :: S'$, $e =$ `f` and `F'` matches S'

Note that negative assertions are not quite symmetric with positive assertions: they accumulate until a positive assertion discharges them. Case 2 therefore has to skip over (zero or more) stack frames e_i until finding the first one (g) that matches the next positive assertion (`h`), and check that none of e_i match any of the accumulated `f`$_j$.

Weaving with filters Naturally, the weaving definitions must change somewhat to accommodate filters. For a set of stack filters $f_1, \ldots, f_n$ guarding some aspect, and a current stack S, they must

check that *all* filters match the stack S. For `before` advice b or `after` advice a, define

$$\begin{aligned} b' &= \texttt{if } (match(S, f_1) \,\&\&\, \cdots \,\&\&\, match(S, f_n)) \; \{b\} \\ a' &= \texttt{if } (match(S, f_1) \,\&\&\, \cdots \,\&\&\, match(S, f_n)) \; \{a\} \end{aligned}$$

For `around` advice r, the woven code must be sure to call subsequent advice:

$$\begin{aligned} r' &= \texttt{if } (match(S, f_1) \,\&\&\, \cdots \,\&\&\, match(S, f_n)) \; \{r\} \\ &\qquad \texttt{else \{ proceed \}} \end{aligned}$$

Weaving multiple pieces of advice proceeds as before.

Advising multiple functions simultaneously

The construction `at pointcut(`$p_1||p_2$`)` *ad* is roughly syntactic sugar for `at pointcut(`p_1`)` *ad*`; at pointcut(`p_2`)` *ad*, and is evaluated in the obvious manner. This construct may nest arbitrarily: $p_1||p_2||\cdots||p_n$. The sole distinction between the parallel and desugared forms is that the former produces only *one* aspect object a while the latter produces several. Thus the parallel construction lets programs enable or disable the advice on all targets simultaneously. Each base pointcut p_i must be the same type of pointcut—all `callee`, all `field`, or all `statement_containing`.

Advising within function bodies

My survey of popular extensions' code emphasized that not all desired program modifications fall neatly at function boundaries. I therefore support two additional pointcuts, advising how variables are accessed within functions, and advising statements within function bodies.

Advising variables As in AspectJ, it is useful to advise getting and setting variables and fields. Extension code often modifies local variables within a function, to influence its behavior. Therefore, define the `field` pointcut and its corresponding `wrap` advice, which specifies a getter or setter (or both) to be used instead of the designated variable or field. Since variable names are commonly reused within a program, it is undesirable to advise all accesses to that name everywhere in the program. Therefore any aspects using the `field` pointcut also specify a `within` filter which (like the `callee` pointcut itself) accepts an expression that resolves to a closure at runtime; the advice is applied only to that closure.

The advice code must be an expression, just like the code it is replacing.[6] To accommodate weaving multiple advice onto the same variable or field, the keyword `proceed` is used to denote the next advice expression or the underlying expression as appropriate. Any expression of the form `a.b.c.d` can be used with the `field` pointcut, rather than just variables. For now, I do not support array indexing (e.g., `field(a[`e`].f)`), as the replacement advice may evaluate e at different (or potentially multiple) times, which may cause unintuitive side effects.

For example, the following code ensures that the `prefs` object never appears null during the `config` function:

```
at pointcut(field(prefs) && within(config))
wrap { get { prefs != null ? prefs : getPrefs() } }
```

while the following advice ensures that a variable containing a maximum value can never decrease:

```
at pointcut(field(maxVal) && within(computeStats))
wrap { set { maxVal = Math.max(maxVal, proceed) } }
```

Note that in JS, assignments are expressions, so the code above assigns the correct value to `maxVal`, and then returns it to any surrounding code.

The semantics of `field` advice are straightforward: given an aspect `at pointcut(field(x.y) && within(`f`)){ get { `g` } set { `s` } }`, evaluate f to some closure $\langle env, \lambda(y_1, \ldots, y_n)\{b\}\rangle$. Replace all assignments `x.y = `e in b with s[e/`proceed`]. Replace all other occurrences of `x.y` with g[`x.y/proceed`]. The extension to multiple advice for the same `x.y` is analogous to `callee` advice. Two subtleties arise: first, in the expression `a.b.c = 5`, if one aspect advises `a.b` and a later one advises `a.b.c`, the second one will *not apply*, because its target expression is no longer present. Second, if one aspect contains the expression `x.y` in its advice, and a later aspect advises `x.y`, again the second one will not apply, because later advice does not apply to code introduced by earlier advice.

Advising statements The remaining type of advice is a new way to insert code into the body of a function, inserting new statements before, after, or around existing statements within function code. Other than systems that expose explicit labels for joinpoints, I am unaware of any aspect system that allows this flexibility. The challenge is isolating a sufficiently expressive pointcut to identify individual statements. Define `statement_containing(`e`)` as the smallest statement containing expression e, as matched by abstract syntax. This has some subtleties: in the statement

```
while (x + 1 > 0) { if (x < 5 && x > 0) { x--; } }
```

[6] If desired, the programmer may wrap the advice in the `(function(){...})()` idiom to use statements and `return`.

the pointcut `statement_containing(x)` matches each of the `while` loop, the `if` statement, and the decrement statement, because each one contains an instance of `x` not contained in any smaller statement. Conversely, it matches the `if` statement only once, despite the repeated usages of `x`, so that advice is not woven multiple times at the same point.

Once the pointcut selects statements, it is easy to apply before, after, or around advice to them, replacing statements with statements. The `proceed` keyword executes the original statement or subsequent advice. This pointcut is particularly useful with extensions where the original code handled a certain set of behaviors and the extension adds a new, unexpected one. Suppose a mainline function assumes its argument is a member of some enumeration `{A,B,C,D}`, and throws an error otherwise. If an extension adds a new member `E` to that enumeration, it must also add code that handles `E` and avoids raising the error. Such code can be inserted before each `statement_containing` that argument.

Discussion

This section considers additional aspect constructs one might contemplate adding and discusses how the current aspects language may interact with new features added in ECMAScript 5 [64].

The current design treats aspects as new native objects. Only the installer of an aspect has a chance to store a reference to it; aspects are otherwise invisible. An alternate design could attach aspects as properties of the advised functions, but this might break JS code that enumerates the properties of those functions.

All pointcuts presented are constructed to select exactly one target via `callee` or `within`; the parallel $p||p$ construction permits specifying multiple pointcuts per advice. However, many aspect systems admit more free-ranging pointcuts, either with wildcards, catchalls, or arbitrary pointcut-designator functions, that select an arbitrary number of targets. In the web-extension context, this flexibility is unused, either because it is truly unneeded or because no extant JS idiom can express it. In the twenty examples I examined in depth (see Section 4.6.2), each wrapper was uniquely applied to a single function, as was nearly every monkey-patch. The few exceptions were either applied a fixed number of times—essentially $p||p$—or applied, one at a time, to an ad-hoc array of functions: it is unclear from the extensions' code what broader pointcut the developer might have intended that would select precisely those targets and no others.

Missing from the set of pointcuts is `caller`, which inlines code at call sites rather than within the callee. There are technical and pragmatic reasons for this. The technical reason stems from the dynamic nature of JS: one cannot know until runtime when a particular function is about to

be called—at which point, the caller has already been compiled, at which point it is too late to inline the advice at the call site. To overcome this, a weaver would have to insert conditional tests at every function call site, which would introduce significant overhead to the common, unadvised case. Pragmatically, I have not seen advice-like idioms that try to emulate a `caller` pointcut. In the code I have examined, those few usages that do pick out particular call sites (i.e., equivalent to advice that uses a `stack` filter) modify the caller's code only in ways that are better expressed using the `field` pointcut.

A larger survey of extensions might indicate further pointcuts and filters. Extensions may want to select only `if` statements, or only the n^{th} occurrence of some statement, or use wildcards to select any `statement_containing(_ > 0)`). Supporting these poses no inherent challenges: the abstract-syntax matcher would need to be enhanced, but no other portion of the language design would change.

The ECMAScript5 specification includes ways to define getter and setter functions for fields. Once these abilities are implemented in JS engines, advising a field's accessor functions using `callee` advice might seem to subsume `field` advice. However, accessor functions are neither required nor implicitly defined; retaining the explicit `field` pointcut lets users advise fields whether or not they have accessors.

JS is primarily hosted within the web setting, which may also inspire new constructs. With appropriate implementation hooks, the approach presented here is already capable of advising DOM methods (e.g., `Element.appendChild`). New pointcuts would be required to advise event dispatch or other moments in the lifecycle of a webpage [74]. New filters might be useful to restrict advice to some subtree of the current page. I leave such extensions to future work.

4.5 Implementation of advice weaving

I implemented the above extensions to JS in the Microsoft Research JScript compiler[20]. This compiler is a JIT that lazily compiles target function bodies at call time, specialized for the dynamic types of the arguments at that call-site. The compilation strategy leverages this behavior: when calling an advised function, the function body must be JITted anyway, so I weave the advice into the body just before JITing, which then compiles the woven result. Additionally, since weaving happens entirely at runtime, the weaver can produce special syntactic forms that are not available to concrete syntax: these special forms are crucial to code efficiency. Section 4.5.1 explains how the JIT compiles regular, unadvised code. Section 4.5.2 explains the changes needed to compile aspect expressions. Section 4.5.3 explains the weaving process itself.

4.5.1 Compiling unadvised code

In the unadvised case, when the JS engine encounters a JS function for the first time, it must generate intermediate code that it can then execute. Because the JIT is lazy, it instead creates a *code generator*, an object that encapsulates all the information eventually needed (e.g., the current environment) to compile the source JS. The runtime representation of a closure contains a pointer to its associated code generator.

When the JS engine encounters a function call expression $f(e)$, it must:

1. Evaluate f to a closure c_f and let $cg_f := c_f.codeGen$.

2. Ask cg_f to compile f, specialized to e's runtime type.

3. Jump to and execute the compiled code.

The specialization in step 2 is straightforward to do at runtime, and is key to efficiency. Since JITing is expensive, code generators memoize the compiled, specialized function bodies in a table $cache : \langle list\ of\ argument\ types \rangle \rightarrow CompiledCode$. Further, code generators are shared among all closures with the same source (e.g., as with higher-order functions): all closures sharing a code generator run identical code. The effect is to compile identical functions as few times as possible.

4.5.2 Compiling aspect expressions

Aspects rewrite their advised closures, which means they must erase the closures' code generators' memoized compiled code. Normal JIT mechanisms will then recompile the closure when needed, at which point I can weave the advice. To achieve this invalidation, I add a timestamp field to both closures and code generators: if a closure's timestamp is newer than its code generator's, then the memo table is out of date. To make code generators aware of aspects, I add a list of installed aspect expressions to each closure, which can then be used during weaving. In short, when a code generator compiles an aspect expression a, it generates code that:

4. Evaluates the pointcut p to a closure c_p.

5. Adds a pointer to a into c_p's list of aspects.

6. Updates c_p's timestamp.

In the advised case, when the JS engine encounters a function call $f(e)$, I change the cache lookups in step 2 above, since it now needs to account for the installed advice. The effect must be to change the memotable to be of type $\langle \textit{list of aspects} \rangle \to \langle \textit{list of argument types} \rangle \to \textit{CompiledCode}$. Additionally, the cache lookup must check timestamps: if the code generator is out of date, it must clear the cache and update the timestamp. But clearing this entire cache is too coarse: other closures may share the code generator, but may not share the same advice. The engine only needs to clear the part of the cache keyed by the current installed aspects. To do so, the implementation actually leaves the memotable as it was, and instead makes a *new* code generator for "the closure plus current advice". A naïve implementation of this would generate far too many code generators; just as multiple, identical, unadvised closures shared the same code generator, it would be good to preserve sharing when those multiple closures with identical code generators are advised by the same aspect. To do so, introduce a second table $\textit{aspectCache} : \textit{aspect} \to \textit{CodeGenerator}$. The remainder of the weaving algorithm uses this table, continuing after step 6:

6. Let $cg_p := c_p.\textit{codeGen}$.
7. If $cg_p.\textit{aspectCache}[a] \neq \textit{null}$, then set $c_p.\textit{codeGen} := cg_p.\textit{aspectCache}[a]$.
8. Otherwise, set $c_p.\textit{codeGen} := \textit{new CodeGenerator}$ and $cg_p.\textit{aspectCache}[a] := c_p.\textit{codeGen}$.

To see this in action, suppose three closures c_x, c_y and c_z share a code generator cg_1. When c_x is advised by aspect a, the JIT creates a new code generator cg_2 for it to use; cg_1 (with its cache) is still valid for c_y and c_z. If c_y is later advised by a, it finds $cg_2 = cg_1.\textit{aspectCache}[a]$, and so reuses cg_2 and its cache. Again, c_z continues using cg_1 and its cache.

This sequence of operations is minimally invasive on the critical path of function dispatch: in the common case of programs with no aspects, only a single branch (comparing timestamps) is added. Moreover, it is maximally sharing, creating the fewest number of distinct code generators. The worst case behavior involves repeatedly advising a function and calling it once, which completely defeats any caching behavior. Even in this unrealistic, pathological example, the compilation overhead is lower than that of other techniques (see Section 4.6.1).

4.5.3 Weaving advice

The weaving mechanism is essentially a function of type `ASTNode` $\times$ `Aspect List` $\to$ `ASTNode`, though I make use of `ASTNode` types that cannot be generated via concrete syntax. These synthetic nodes let the weaver temporarily alias function parameters to match advice parameter names or manipulate the return address of the function without strange syntactic contortions.

Weaving `callee` advice

Consider a function `f` with body `B`, and lists $\vec{b}$, $\vec{a}$ and $\vec{r}$ (of lengths l, m, and n) of installed before, after, and around advice. Assume for now that each advice came from a `callee(f)` pointcut with no filters. Fig. 4.4 shows a simplified resulting woven body `B'`. Before and around advice can be concatenated with the body (BA). Around advice is more complicated, as it may call `proceed` multiple times; I implement around advice by inlining the next around advice (R_{i+1}) into the current one. Labels L and L_{a_i} mark potential targets for `return` statements. The net result is the outermost R_1, along with some bookkeeping described below.

Named parameters Implementing the renaming of function parameters must ensure that any names introduced as parameters in one piece of advice must be scoped only to that advice. Rather than rewrite advice s explicitly to substitute names, the named parameters are temporarily introduced (i.e., within the advice body) as *aliases* for the parameters of the function. The advice is surrounded by a pair of directives `RenameParams` and `UnRenameParams` that have no runtime effect (in fact, they do not even appear in the compiled code), but temporarily change how the code generator compiles those identifiers: instead of local variables, they resolve to the appropriately positioned arguments on the stack. Assignments to these parameters via these aliases are visible to subsequent advice and to the mainline code. The current implementation requires that the arities of the function and advice must match; however, this is not a fundamental requirement.

Retval and exceptions Because advice is inlined into the callee, special care is needed for return values and targets. The calling conventions dictate where a function's return value is located; the AST node `retval` (used by `around` or `after` advice) provides a mutable way to describe that location. Advice can then change the return value by either assigning to `retval` or simply `returning` a new value. The calling conventions also dictate that `return` statements branch to the function epilogue before resuming the caller. To recapture control flow for `after` and `around` advice, the label that identifies the epilogue must be changed: the `InstallReturnLabel` directive does precisely this, diverting the return label from the epilogue to the next applicable advice. The original return label is reinstated by the `UninstallReturnLabels` directive.

`Proceed` It is possible that multiple `around` aspects that each call `proceed` multiple times could lead to exponential code blowup. In practice this is unlikely; none of the extensions I examined made use of such constructions. To avoid the blowup, I can compile `proceed` to a pair of jumps, similar to the construction for `retval`.

```
Let BA =
  For each Before advice b_i (1 ≤ i ≤ l) in install-order
          RenameParams(b_i.params)
          b_i.body
          UnRenameParams(b_i.params)
  InstallReturnLabel(L_{a_m})
  B
  For each After advice a_i (1 ≤ i ≤ m) in reverse order
  L_{a_i}:    InstallReturnLabel(L_{a_{i-1}})
          RenameParams(a_i.params)
          a_i.body
          UnRenameParams(a_i.params)
Let R_{n+1} = BA
For each Around advice r_i (1 ≤ i ≤ n) in install-order
Let R_i =
  RenameParams(r_i.params)
  r_i.body[Let L be a fresh label in
          InstallReturnLabel(L)
          UnRenameParams(r_i.params)
          R_{i+1}
          RenameParams(r_i.params)
          L
    /proceed]
  UnRenameParams(r_i.params)
Let B' =
  R_1
  UninstallReturnLabels()
  return retval
```

Figure 4.4: Weaving of `callee` aspects

Weaving `stack` filters

Implementing the design for `stack` filters is particularly efficient: though it appears an implementation must walk the stack when the advised function is called, it can instead achieve the same effect using only $O(1)$ work per function named by the filter.[7] A `stack` filter is a simple state machine whose state must be updated as each relevant function runs. To evaluate the filter in the aspect `at pointcut(callee(`e`) && stack(`$f_1, \ldots, f_m$`)) before {`s`}`, I need to add code to the entries and exits of closures f_i to update the state of the stack filter. Specifically, let st be a runtime representation of a stack filter. Then for each $1 \leq i \leq m$, evaluate `at pointcut(callee(`f_i`)) around { `$enter(st, f_i)$`; try { proceed; } finally { `$exit(st, f_i)$` } }`. Functions *enter* and *exit* update the

[7] I am not the first to recognize that stack inspection can be achieved more efficiently; Wallach et al. [222] use a similar technique for Java.

state of *st* to reflect how far the stack pattern matches the current call stack, based solely on the current stack frame f_i; using `finally` ensures that *exit* will run regardless of the mainline control flow.

Consider a filter `stack(a, !b, c)`. When `a` is called, the filter's state advances from "Start" to "Expecting `c`". If `c` is called (i.e., the stack is `a::···::c`), the state advances to "Success" and the enabled-filter counter is incremented. As `c` exits, the counter is decremented and the state reverts to "Expecting `c`". However, if `b` is called before `c` (i.e., `a::···::b::···::c`), the filter state is "Fail" until `b` exits.

The remaining details elided from Fig. 4.4 support `stack` filters. Since aspects can have multiple `stack` filters, I equip them with a counter of currently-enabled filters, maintained by the stack advice above. Advice is surrounded with an enablement check (shown here for `after` advice):

```
L_{a_i}: if (isEnabled(a_i)) {
           InstallReturnLabel(L_{a_{i-1}})
           RenameParams(a_i.params)
           a_i.body
           UnRenameParams(a_i.params)
         }
```

The `isEnabled` helper checks that the current count equals the installed count. If the test passes, the advice runs as before; otherwise, control falls through to the next installed advice. (Around advice cannot simply fall through; I instead generate an `else { proceed }` branch.)

Dynamic disablement To support programs that use the aspect object to dynamically disable an aspect (via `aObj.disabled = true`), the internal *isEnabled* function must also check that flag. The weaving remains unchanged.

Avoiding redundant overhead If an aspect has no `stack` filters, or if the aspect expression is used as a statement (and therefore the aspect object is ignored), I can simplify or eliminate the *isEnabled* tests, leading to more efficient woven code (see Section 4.6.1).

Weaving `wrap` and `statement_containing`

Compiling `wrap` or `statement_containing` advice is done by preprocessing the body before applying `callee` advice. During preprocessing, field advice is rewritten before statement advice. Recall that neither pointcut designates a function in which the rewriting should occur: this is specified by the mandatory `within` filter which, like the `callee` pointcut, evaluates its argument at runtime

to a closure, and installs the advice onto it. Such advice is therefore somewhat of a hybrid, as it is a syntactic (static) transformation on a runtime-specified (dynamic) closure.

The current implementation has the small limitation that it is impossible to advise global variables while in the global scope. This is a minor restriction on expressiveness, as there is relatively little code at global scope. In practice, this has not been a stumbling block. (Resolving this would change how I compile global code and penalize the performance of unadvised code; from my experiences examining extension code, there does not appear to be enough interesting code in global scope to warrant this change.)

4.6 Evaluation

The Microsoft Research JScript compiler is written entirely in C# and consists of a JS front-end and JIT code generators targeting either a specialized bytecode or the .Net Common Intermediate Language (MSIL); the MSIL in turn is compiled by the CLR JIT. The modifications needed were easily confined to the front end and the code generators; the CLR JIT was unchanged. Because the backends target .Net, and because the runtime environments are implemented in .Net, the CLR JIT can easily optimize JS code together with the runtime.

I evaluate the framework on performance (using the MSIL backend) and on expressiveness.

4.6.1 Performance

Recent work has seen enormous improvements in the performance of JS engines [87], so it is important that new constructs not undo this progress. The compiler is work in progress, so absolute performance numbers are preliminary. Instead, I measure the relative performance of 1) an unwoven base program for reference, 2) simple advice defined using aspects, along with equivalent monkeypatched, wrapped, and manually-woven versions, and 3) advice with a `stack` filter defined by aspects, and monkeypatched, wrapped and manually-woven versions. I pessimistically choose a trivial baseline function and minimal advice, to maximize the ratio of weaving overhead to useful runtime work: the aspect version of the test program is shown in Fig. 4.5. The seemingly-extraneous function surrounding the advice is included in the manual and monkeypatched versions too, to force them to execute the same number of closures while weaving in advice as when executing the equivalent wrapper idiom, thereby eliminating one (large) source of runtime differences. This hurts the non-wrapper versions equally, slowing them to the weaving speed of wrapping, without impacting the performance of the advised code. In practice, real code would not be written this way.

```
// Initial function, and caller (for stack-filtered advice)
square = function(v) { return v*v; };
callSquare = function(v) {
  var ret = square(v);
  return ret;
};

// Closure that installs the advice; other tests use monkeypatches or wrapping here
(function() {
  at pointcut(callee(square) && stack(callSquare) )
  before(v) { v++; };
})();

for (test = 0, incr = 1; test < N; test++, incr++)
  if (callSquare(test) != incr*incr)
    print("Test failed for test ", test);
```

Figure 4.5: Test microbenchmarks, without and with `stack` filters (boxed), written using advice. N is an integer literal that varies across the x-axis in Fig. 4.6.

To account for inter-run and system variability, I report the minimum time achieved for each technique: whereas system noise can slow down a test run, the minimum time represents the fastest time actually achieved by any approach. In this setting, measuring performance must distinguish two levels of JITing/caching: .Net JITing the compiler and the compiler JITing the JS program. Once the .Net cache warmed up (the first 10-20, out of 200 runs), variability was insignificant: the averages and minima were nearly equal. Every test iteration started from a cold cache for the compiler. The tests were run on a 2.8GHz Windows XP machine with 4GB of memory running .Net 3.5 SP1. Despite the confounding effects of caching and JITing, no test had a working set greater than 35MB of memory.

Results are shown in Fig. 4.6, presented as ratios of the total runtimes of each test versus that of the unadvised code. Solid lines show the tests without `stack` filters; dashed lines show the filtered ones. The x-axis counts how often the test function is called after being advised (the loop constant N in Fig. 4.5): when $N = 1$, the test function is called only once and the runtime effectively measures the overhead of the weaving process; when $N \gg 1$ the test function is called many times and the runtime asymptotically approaches the overhead of the woven code. For some techniques, the relative cost of weaving may be cheaper than that of the woven advice, leading to some curves *rising* for larger N.

In short, the weaving mechanism runs *as fast* as the unfiltered manually-woven version, and *5–27% faster* than the `stack`-filtered manually-woven version. Contrast this with wrapping (31–61%

slower than the filtered manual version), or with monkey-patching (1–63% slower). Wrapping's performance is badly hampered by the extra function calls within the advice (recall I already accounted for the extra function calls within the weaving process itself). Monkey patching generates code identical to the manual version, but pays a large initial cost for `eval`. The performance parity in the non-filtered case comes from simple optimizations that eliminate vacuous filter-enablement checks. The performance gains in the filtered case come from implementing the stack filter inside the runtime, rather than with state variables in JS itself. Runtime weaving as implemented here provides higher performance *and* stronger semantics than existing idioms—a win-win situation.

4.6.2 Expressiveness

I designed the aspect language to support extensions, but I deliberately did not examine a large selection of extensions in advance, to prevent over-fitting the language to awkward coding idioms or legacy code. Instead, after designing the language, I thoroughly examined the scripts of twenty other popular Firefox extensions. Note that because the engine is not embedded in Firefox, I have not yet implemented an aspect-enabled Firefox, but rather I show that extensions could easily be rephrased to use aspects instead. My twenty extensions were drawn from a snapshot of the top 50 add-ons in each category from Mozilla's `http://addons.mozilla.org`, as of October 2008; the versions of these twenty extensions tested here all support Firefox 3.0. The snapshot contains 350 distinct extensions, of which roughly 10% use monkey-patching; the twenty extensions were drawn from these. As such the sample may conservatively undercount the prevalence of wrapping in extension code.

The results are summarized in Fig. 4.7. The first three columns measure the amount of JS code in each extension, and the amount of code directly attributable to either monkeypatches or wrapping. I count as a monkeypatch any lines of code that call `eval` and `replace` on code, including the string arguments as they are the contents of the patch. It is more difficult to count the precise number of lines of code needed in wrapping, as sometimes a function is simply wholly replaced, rather than precisely fitting the wrapping idiom. In such cases, I choose to undercount and include only the single line that binds the new function to the old name. In total, these twenty extensions contain 98,700 lines of JS code (LOC), of which 2,713 LOC (2.8%) is a monkey-patch or wrapper.

The extensions range in size from 300–10,000 LOC, and subjectively range widely in their effects on the browser. The two are not correlated: Tree Style Tab requires far more lines of monkey-patching code ($894/6786 = 13\%$) than any other extension, even though it is less than half the size of TabMixPlus and even though subjectively it modifies the browser less than SplitBrowser.

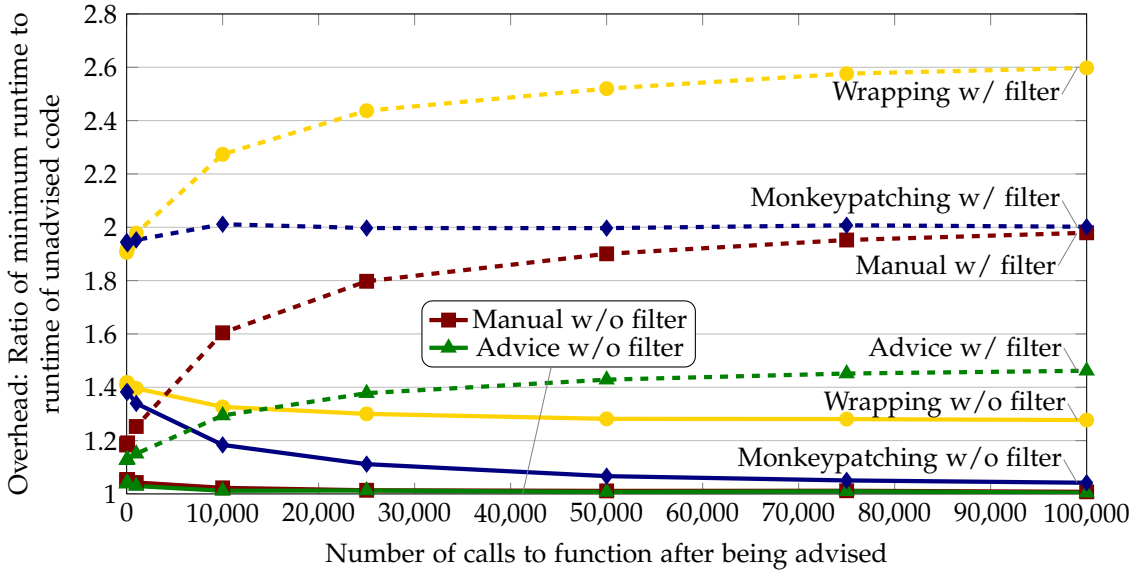


Figure 4.6: Overhead comparison for test in Fig. 4.5, with (dashed) and without (solid) stack filters. Lower is better.

Conversely, All-in-one Sidebar is nearly the same size overall, but has 5% the amount of patching code, while NoScript is twice as large and profoundly impacts the browsing experience, yet needs merely 11 lines of patches. Regardless of size, patches are key to extensions' behavior, and so simplifying them is helpful.

The remaining columns of the table count and categorize each hook (both monkeypatch and wrapping) as one or more of the supported advice types, or as "other" if I could not express it directly using the advice language. I was carefully literal-minded in these classifications: if a patch was not precisely expressible as an aspect, even if a semantically equivalent patch could be so expressed, I counted this patch as a failure. For example, Tree Style Tab includes seven patches that insert a new statement after the opening brace of an `if`-statement. None of the existing advice forms support this. However, in these cases the guards of the `if`s are pure, and so using `before statement_containing` advice that repeated the `if`-test would have the same semantic effect. Despite being so stringent, the language as presented can express 621 out of 636 (97.6%) observed hooks. Of the remaining 15: seven add a statement to the beginning of an `if` block; four change the condition of an `if` test; one changes a statement in the middle of an `if` body; one inserts a statement at the end of one case of a `switch` statement; one changes a `while` loop into a `for` loop; and one is unknown and appears broken. With slight revisions of the monkey-patches (e.g., by duplicating the condition of an `if`), aspects could express 12 of these 15.

Name	JS (LOC)	Monkeypatch (LOC)	Wrapping (LOC)	Function (count)	Field (count)	Stack (count)	Stmtcont (count)	Other (count)
Fission 1.0	367	5		5		2		
TabRenamizer 0.8.11	536	3					3	
Compact Menu 2-2.2.0	586	7		1	1	1		1
Multirow Bookmarks Toolbar 2.9	587	53		2		2	2	
Redirect Remover 2.5.5	926	20		2	2	2	1	
Multiple Tab Handler 0.3.2008101801	2048	83	5	10	2	3	1	1
Img Like Opera 0.6.17	2287	45		4	1	1	6	1
FireGestures 1.1.5.1	2455	6	4	2	1			
Gladder 2.0.3.1	2558	49		4	1			
All-in-one Gestures 0.19.1	4056	4					1	
Split Browser 0.5.2008101801	4519	333	269	71	48	20	14	
Session Manager 0.6.2.4	4697	6	3	3	2		1	
TabKit 0.4.3	5232	63	16	12				
SpeedDial 0.7.2.5	5641	20	22	3	2	1		
Tree Style Tab 0.7.2008101801	6786	894	5	45	32	13	26	7
All-in-one Sidebar 0.7.6	7079	38	23	22	9	3		
Gmarks 0.9.9	7700	3	1	2		1		
TorButton 1.2.0	10560	2	139	139				
NoScript 1.8.3.3	10809	10	1	4		3		
TabMixPlus 0.3.7.3	14278	551	30	90	21	51	23	5

Figure 4.7: Comparing 20 Firefox extensions, showing code size, patch size, and counts of how many patches by advice types.

Evident from the table is that all forms of advice are actually used: function advice is by far the most common, but statement advice accounts for over 10% of the advice. Of the 103 `stack` filters, while most filters were only one function deep, several instances used multiple patches to manually implement two- or three-function-deep filters; these patches could all be subsumed into a single `stack` filter.

4.7 Related work

Chapter 2 already discussed related work in greater detail. Here I reprise the discussion of language-design decisions for AOP languages, focusing in particular on where this design borrows or deviates from previous work.

4.7.1 Aspects for object-oriented languages

AspectJ [207, 208] is probably the most well-known aspect-oriented language. AspectJ was designed to support both static and load-time weaving. In Java all methods have statically known names. Compiling AspectJ efficiently poses several challenges, particularly for `cflow` [12]. However, since Java does not support first-class closures, aliasing issues simply do not arise.

AspectJ employs complicated heuristics to define how pointcuts match in the presence of inheritance, overridden methods, and interfaces. The rules are designed so that advising a method on some superclass will trigger that advice on all subclasses, regardless of whether they override the method. But JS is a prototype-based object-oriented language, making heuristics designed for class hierarchies unnatural. The design of advising closures reflects that distinction: by triggering advice through all aliases to a closure, I ensure that all objects of a given prototype share advice applied to the prototype. However, if a new object overrides a method from its prototype, it is a different kind of object, and advice does not implicitly attach to it.

Some of the compelling power of aspect-oriented programming derives from its freedom to tamper with almost any part of the code. Aldrich [6] proposed an explicit "open module" approach to curtailing that freedom. Many of these ideas may be directly applicable to aspects in JS: for instance, functions computing trusted values (encryption, cookies, passwords, etc.) might be sealed from advice. Indeed, the object-hardening proposals for `ECMAScript5` [64] effectively permit sealing functions from wrapping or monkey patching; the aspect system would need to support or integrate the same ability. Currently, this proposal focuses on expressiveness; *restricting* expressiveness is left to future work.

4.7.2 *Aspects for functional languages*

AspectML [50, 225] primarily focuses on the challenges in adapting an aspect system to a strongly-typed functional language. The authors choose not to permit advising anonymous functions or first-class functions. The design here permits advising these functions, as functions are frequently aliased to new variables or passed as arguments to functions. The treatment of `stack` filters is inspired by AspectML's stack patterns.

AspectScheme [62] contends with the challenges of aliasing in an aspect system for a higher-order functional language. Their aspects are fully first-class: a pointcut is simply a boolean Scheme function on joinpoints. Additionally, they explore both static and dynamic scoping constructs for advice; I focused solely on dynamic scoping. Their implementation depends heavily on Scheme's hygienic macros and continuation marks [40, 85], permitting a whole-program-transforming implementation (by redefining function application) within Scheme. While one could add continuation marks to a JS JIT [41], this approach exploits direct JIT integration instead.

The handling of replacing variables with new getters and setters bears some resemblance to the `map-closure` operation [197]: like that work, I provide a first-class mechanism for JS to open closures and reveal and revise their code without disturbing their closed-over environments. The authors note in their discussion that aspect systems and `map-closure` are related, though their scopes are different: aspects are applied globally, while `map-closure` can be applied to a dynamic scope. This distinction is pragmatically eliminated by dynamic filters such as `stack` filters.

4.7.3 *Aspects within JavaScript*

AOJS [226] is a recent prototype that implements weaving statically in a proxy server that modifies scripts before the browser sees them. This approach has several shortcomings, however, largely stemming from the choice to define weaving via preprocessing, rather than within JS. It supports two pointcuts—variable assignments and function calls, but not retrievals, `callee`, or filters. Pointcuts are implemented by replacing the variable or function name with calls to wrapper closures that in turn execute the advice and the original expressions. The approach relies explicitly on names and so does not avoid the aliasing problem, and it cannot handle anonymous or runtime-created functions.

AspectScript [211] is a recent and independent project designing aspects for JS. Like this work, they advise closures rather than variables, to avoid the aliasing problem. They also explore different scoping strategies (similar to the `within` filter). However, their implementation is fundamentally different: weaving is implemented at runtime by parsing and rewriting scripts from within

JS to wrap every potential joinpoint with a function. These "reifier" functions construct a context for the joinpoint and then call the weaver which in turn executes the relevant advice and the mainline code. All these rewritings and indirections come at cost: code is substantially larger and slower even when no aspects are deployed. For a browser whose interface is largely written in JS, and (despite many extensions) largely unmodified JS at that, such overhead is likely untenable. By contrast, the weaver causes zero code bloat and minimal runtime overhead. Additionally, introducing a second JS parser into a web browser is risky: it may not be bug-for-bug compatible with the underlying JS engine, leading to potentially incorrect results. Finally, AspectScript does not catch `eval`ed code or code that was loaded without being processed by their rewriter library: such code is not visible to their joinpoints and is not advisable. In this system, advice can use an arbitrary run-time expression to identify the closure to be advised and there are no restrictions on which closures are advisable.

4.7.4 Web extension in practice

There are currently over six thousand Firefox extensions used daily by over thirty million people [193]. These extensions customize nearly every facet of the browser, changing tab handling, mouse interactions, file downloading behavior, etc. Many of these extensions replace existing functionality with new code, using the wrapping idiom where appropriate. Other extensions merely modify existing code slightly, using monkey-patching. Some actually modify *each other* to resolve compatibility issues. This last usage motivated the `field` and `statement_containing` pointcuts: for example, one extension (SplitBrowser[8]) modified another (All-in-One Tabs[9]) by replacing all accesses to a local variable with a field on a globally visible DOM node, so that it could see that interim state later as needed. Moreover, a reference to the DOM node was introduced as a new local variable for brevity. Clearly, these contortions are *ad hoc* and difficult to reason about without a more structured approach.

The userscript mechanism is similarly popular: nearly 40,000 scripts exist to tweak individual applications such as Gmail, YouTube, or Facebook, remove ads on popular pages, etc. The top five userscripts have each been installed over eleven million times. Like browser extensions, userscripts frequently interact with the structure or style of the DOM of the page, in addition to modifying its script content. Analogously, those latter modifications are better expressed and reasoned about using aspects.

---

[8] `https://addons.mozilla.org/en-US/firefox/addon/4287`

[9] `https://addons.mozilla.org/en-US/firefox/addon/12`

4.8 Future work

JS aspects can also be used to good effect beyond extensions:

- Concurrent research by colleagues [155] examines using aspects to enforce security policies along the lines of Caja [209] or ADsafe [3]. Much of the challenge in freezing objects in Caja, for instance, requires interposing on any aliases to member functions in that object; the approach of advising the closure directly addresses this issue.

- Libraries such as Script.aculo.us or Prototype build upon existing JS objects to provide convenient, common functionality for websites. Some of that functionality is fairly aspect-like; Prototype, for instance, defines functions to extend objects with getters and setters, and explicitly thread through `proceed` functions to call the next installed code. Aspects can help simplify their development while improving their performance.

The declarative nature of aspects opens up the possibility of new analyses. For example, recent work has focused on staging information flow analyses in the face of dynamic composition of scripts [39]. Precision is lost whenever code is `eval`ed; declarative aspects present more structure than `eval` strings, which may improve precision. For another, web-application and browser extensions are notoriously prone to breaking when certain other extensions are simultaneously installed, due in large part to the fragility of the code-injection idioms. Not only do aspects subsume all the weaving complexity of extensions, but they permit identifying when multiple extensions advise the same code and potentially conflict, which could provide useful warnings.

4.9 Summary

Web applications and browsers are growing ever more complex. Their broad, ad-hoc customizations highlight the need for an expressive mechanism by which to program them. In this work, I have implemented the first JIT for JS that supports aspects. I identified key linguistic requirements in the web-application space and described how they differ from prior aspect systems. The aspect proposal for JS meets these requirements, and I have shown that it offers better performance and cleaner semantics, thereby improving the development of extensions.

Chapter 5
LAYOUT/MARKUP CONFLICTS

5.1 Introduction

5.1.1 An overview of overlays

We take for granted now that two different webpages need not look even remotely similar, yet both can be rendered by the same browser. Mozilla extended this notion to the platform itself, and in 1998 introduced *XUL* as a novel language for constructing application UIs.[1] XUL is an XML language with tags representing widgets such as buttons, scrollbars, and textboxes; Fig. 5.1a shows a "Hello world" example written in XUL. If launched with one XUL file, the rendering engine can generate Firefox's UI; if launched with another, it can generate Thunderbird's instead. This approach to declarative UI markup is popular, and has since been revisited multiple times, most notably in XAML, Microsoft's markup language for Silverlight, and HTML5 itself, the emerging standard for webapp development. Unless otherwise noted, the remainder of this chapter will use Firefox as the archetypal Mozilla application; discussions about Firefox extensions apply equally well to Thunderbird or other Mozilla applications.

XUL is currently unique among these languages with its support for *overlays*, a declarative mechanism to inject new XUL content into existing XUL content. Overlays (informally) consist of a *selection* of some element in the mainline document and a *content subtree* to be inserted into that element; they are rather like "tree-shaped patches". A simple overlay, and its composition with the base document, is shown in Figs. 5.1b and 5.1c: the content of the ⟨**vbox** id="msg"/⟩ in the ⟨**overlay**/⟩ is merged with the content of the tag with the same name and id in the base document. In this manner, UIs can be refactored into logical units that are composed at runtime: the overall structure of the UI (e.g., the relative positioning of menubars and content areas and statusbars) can be defined in the main Firefox UI file, while detailed pieces of UI (e.g., the actual menu items) can be separated into a smaller, more focused files. Firefox (and other Mozilla applications) make heavy use of this ability to aid code clarity.

The key enabling power of these overlays is that they are not restricted to just Firefox-authored files: *any developer* can write an overlay, and have it merge with existing content at runtime. This is

[1] http://www-archive.mozilla.org/xpfe/languageSpec.html

```
⟨window xmlns="http://www.mozilla.org/
   keymaster/gatekeeper/there.is.only.
   xul"⟩
  ⟨vbox id="msg"⟩
    ⟨spacer/⟩
    ⟨description⟩Hello, XUL⟨/description⟩
    ⟨spacer/⟩
  ⟨/vbox⟩
⟨/window⟩
```

(a) "Hello, XUL" base document

```
⟨overlay xmlns="http://www.mozilla.org/
   keymaster/gatekeeper/there.is.only.
   xul"⟩
  ⟨vbox id="msg"⟩
    ⟨button⟩Overlay!⟨/button⟩
    ⟨spacer/⟩

  ⟨/vbox⟩
⟨/overlay⟩
```

(b) Overlaying the ⟨**vbox**/⟩ in "Hello, XUL"

```
⟨window xmlns="http://www.mozilla.org/keymaster/
   gatekeeper/there.is.only.xul"⟩
  ⟨vbox id="msg"⟩
    ⟨spacer/⟩
    ⟨description⟩Hello, XUL⟨/description⟩
    ⟨spacer/⟩
    ⟨button⟩Overlay!⟨/button⟩
    ⟨spacer/⟩
  ⟨/vbox⟩
⟨/window⟩
```

(c) Composition of base with overlay

Figure 5.1: Simple example of XUL, overlay, and composite result

the foundation of extensions as found in Firefox. Nothing about overlays is inherently restricted to XUL, however: with some care, overlays are just as easily defined in HTML.

5.1.2 *Challenges of supporting multiple overlays*

As with other forms of composition, a working system must define carefully the intended forms of interactions among multiple overlays, and then filter out any interactions with problematic consequences. Doing so for overlays must distinguish two sorts of problems: those that arise from the semantics of the markup language itself, and those that stem from the precise design choices of overlay expressiveness.

Intuitively, extensions conflict due to language semantics if they behave in different *and unanticipated* ways when installed together as when installed separately, either in their overlay structure or in their styles. Not all differences are bad. For example, in Firefox, two extensions can declare menu items to be added to existing menus, possibly to the same menu. In the latter case, as long as both menu items appear on the menu, the extensions probably do not care about the precise order in which they appear, so these differences are minor and unimportant. However, extensions can also define hotkeys declaratively in XUL, possibly defining the same hotkey—but pressing the hotkey can trigger only one behavior. Here, the order matters: the behavior of the hotkey depends on which extension runs "first", while the "second" extension gets no hotkey support. But it isn't merely that "order matters for hotkeys": if two extensions define different hotkeys, then their installation order is irrelevant.

Additionally, extensions' CSS style definitions may (intentionally or not) apply to other extensions' overlays. These too may be benign or conflicting.

Beyond these semantic constraints, there are several independent choices for overlay design, summarized in Fig. 5.2, whose interactions influence the complexity of the conflict-detection problem. First, an overlay system may vary precisely what actions overlays may perform. Permitting only insertions of new content into a tree is the simplest action, as it ensures the tree grows monotonically: whatever existed before an overlay was applied will continue to exist afterward. Modifying existing content by changing attributes is often not more complicated, because those attributes typically do not participate in overlays' selection mechanism. The final ability, to remove existing content, substantially raises the complexity of overlays, but it is the most symmetric: any action taken by one overlay can be undone by another overlay. Said another way, overlays with all three abilities are *closed* under inverses, which may be very useful for the conflict-detection algorithm.

Second, an overlay system may vary the *expressiveness* of the selection mechanism. Firefox's

overlays are limited to selecting just a single element by its id and tag name,[2] but there is no fundamental reason why they must be so limited. A more general design would permit additional overlay idioms to be expressed declaratively that otherwise must be constructed imperatively using scripts. In particular, the system presented here will permit selectors to select elements using arbitrary CSS selectors; this generality obviates many scripted construction tasks. However, this flexibility significantly complicates the conflict-detection problem. As explained later, a necessary but insufficient component of such an analysis will be detecting potential *overlap* between different CSS selectors; a full solution will also require tracking the actual effects of CSS universal selectors and modeling how selectors should change in response to the presence of other extensions.

Third, an overlay system can vary the *composition time* when overlays are applied. While seemingly straightforward, subtleties of HTML semantics make this much murkier than it ought to be.

Fourth, an overlay system must choose whether overlays can *apply to each other*—i.e., to content produced by other overlays—or not. Such an ability yields more powerful extensions, but interactions with more expressive selectors badly complicate conflict detection: such a system now must consider dependencies between overlays, and unsatisfiable dependency chains lead to conflicts.

5.1.3 Detecting overlay conflicts

I present several variants of conflict detection algorithms, required by different choices for the options in Fig. 5.2, that build toward the most general case presented. All variants will ignore precisely choosing a composition time; I defer that discussion to Section 5.9.

Section 5.4, Node insertion only, by id only, base document only: This strawman scenario is weaker than Firefox's overlay mechanism, but introduces intra-overlay conflicts that are detectable regardless of other extensions. Conflicts here are defined to be duplicate ids or missing targets.

Section 5.5, Node insertion and attribute modification, by id only, overlay can see injected content: This scenario is much closer to Firefox's mechanism, and suffices for the sample of extensions I have analyzed. This analysis abstracts overlays as *document-state transformers* that can *require* or *define* targets (nodes with ids) in the document. This abstraction leads to a dependency analysis that detects many true errors in the analyzed extensions with few false positives. Conflicts from here through the end of the chapter also include resource-type specific conflicts (e.g., duplicate hotkeys), and incompatibilities between overlay-specified "guards".

---

[2] In the ideal case of well formed documents, the unique id suffices and the tagname is redundant; unfortunately not all documents are well formed, so using both provides a pragmatic "safety net".

Overlay abilities	
Node insertion only	Ensures the tree grows monotonically, yielding the simplest conflict detection
Node insertion and attribute modification	Equivalent to Firefox (in practice), this does not greatly affect the conflict-detection algorithm
Node insertion, attribute modification and node deletion	Equivalent to Firefox, this is the most general, and symmetric, case, but makes conflict detection very difficult
Selector expressiveness	
By id	Equivalent to Firefox, this choice yields the simplest conflict detection
`CSS` selectors using only descendant and sibling combinators	Assuming no node deletion, this yields a trickier but still tractable conflict-detection algorithm that can compute a clean loading order efficiently
Arbitrary `CSS` selectors	Even without node deletion, this yields a complicated conflict-detection algorithm
Composition time	
During parsing	This requires detailed coupling of the overlay mechanism to the parser, to check whether the selectors are satisfied
As subtrees are inserted	Precise details of the `HTML` parsing algorithm make this poorly defined without detailed parser support
When the parser completes	Inline scripts may have run, and seen inconsistent views of the document
Just before the `onReadyStateChange` event	A pragmatic compromise that sacrifices some consistency for a well-defined execution moment
During runtime execution as nodes are inserted	Not mutually-exclusive with the options above, this requires only a simple conflict-detection algorithm
Higher-order behavior	
Overlays see only the base document	Overlays have no effect on each other, leading to the simplest conflict-detection algorithm
Overlays can see overlay-injected content	Equivalent to Firefox, this is the most powerful case, and requires the most complicated conflict analysis

Figure 5.2: Key design choices in an overlay system

Section 5.7, Node insertion and attribute modification, selectors using only descendant and sibling combinators, overlay can see injected content: This scenario is more general than Firefox's mechanism, as it broadens the selectors used to pick targets. It can be handled by suitably generalizing the preceding algorithm, using a notion of the *intersection of two CSS selectors*.

Section 5.8, Arbitrary actions, arbitrary selectors, overlays can see injected content: I demonstrate by example why the preceding algorithm cannot suffice, and sketch future work that may handle this case. It treats overlays as *patches*, and develops an algebra describing how patches commute and conflict with one another.

5.1.4 Chapter overview

The remainder of this chapter is organized as follows. Section 5.2 defines the CSS selector language, then defines and proves properties of that language that are used in subsequent sections. Readers familiar with CSS can skim this section and refer back to it as needed. Section 5.3 defines the syntax of the overlay language as an extension to HTML, along with some examples of its use. Section 5.4 presents the strawman scenario above, and develops a conflict-detection algorithm that checks for unsatisfied dependencies in overlays. Section 5.5 presents the second, nearly-Firefox-equivalent scenario, and develops the machinery needed to analyze overlays with these capabilities. In a change of pace, Section 5.6 uses these results to analyze a sample of Firefox extensions in detail. Section 5.7 presents the first generalization of Firefox extensions, defines the intersection algorithm for CSS selectors, and shows how to use that algorithm in the preceding conflict-detection analysis to (partially) handle the new expressive power, positioning the remaining problems as future work. Section 5.8 explains why selectors were limited in the preceding case, and sketches an approach to handling these overlays as future work. Finally, Section 5.9 returns to the issue of precisely when overlays should be applied to a base document. Section 5.10 summarizes the contributions of this chapter. Proofs of the claims in this chapter are presented in Appendix A.

5.2 CSS selector language

5.2.1 CSS syntax and meaning

CSS 3 defines a grammar for its selector language,[3] which is paraphrased here in Fig. 5.3. (The paraphrasing ignores negation, functional pseudo-classes and namespaces, as they obscure the essential details without changing the results.) For typographical clarity, I write $a \sqcup b$ instead of

[3] http://www.w3.org/TR/css3-selectors/#grammar

Selector	::=	*SimpleSelector*[*Combinator SimpleSelector*]*
Combinator	::=	`+` | `>` | `~` | ␣
SimpleSelector	::=	[*TypeSel*|*UniversalSel*][*Id*|*Class*|*Attrib*|*Pseudo*]*
	|	[*Id*|*Class*|*Attrib*|*Pseudo*]+
Id	::=	`#`⟨*name*⟩
Class	::=	`.`⟨*name*⟩
Attrib	::=	`[`⟨*name*⟩ [[`~=` | `|=` | `^=` | `$=` | `*=` | `=`] `"`⟨*string*⟩`"`]`?` `]`
Pseudo	::=	`:`⟨*name*⟩
TypeSel	::=	⟨*element name*⟩
UniversalSel	::=	*

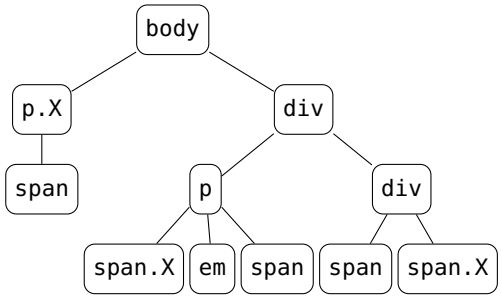
Figure 5.3: CSS Syntax paraphrased from the CSS 3 specification

a b to denote the descendant selector. In full CSS stylesheets, each style rule permits combining multiple selectors by commas, indicating the disjunction of the component selectors; this is a shorthand for duplicating the rule body once per disjunct, rather than a combinator in the selector language itself. I present the grammar first, describe its meaning informally, and then provide a formal denotational semantics for it in terms of *paths of elements within document trees*.

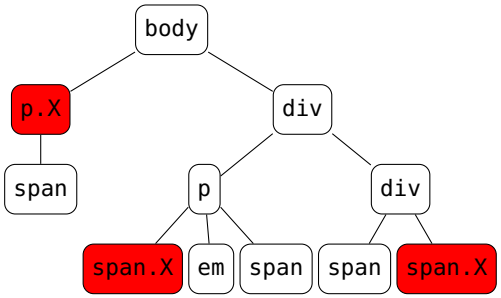
TypeSel picks elements by their tag name (e.g., ⟨**div**/⟩, ⟨**p**/⟩), while *UniversalSel* matches any element (i.e., ∗). Additionally, atomic selectors can be filtered by whether they have the correct id (*Id*), class (*Class*), attributes (*Attrib*, whose various operators describe string-matching requirements on the attribute), or positional attributes (*Pseudo*). This grammar adheres to the concrete syntax of CSS selectors, and includes a commonly-used shorthand: the second production of *SimpleSelector* implicitly assumes a leading *UniversalSel*. Additionally, the grammar specifies nothing about precedence or associativity among the combinators.

By design, *SimpleSelector*s match elements in isolation: while some selectors may match multiple elements in a document (e.g., `*.foo` matches all elements with class "`foo`"), they do so without examining any other elements. The remaining combinators connect a group of simple selectors, each applying to one element in isolation, into some path through a tree applying to multiple elements in combination. Ultimately, this path is the meaning assigned to the selector. Browsers then simplify this and treat CSS selector matching as a function that takes a tree and a selector and returns just the leaf-most element of each path in the tree matching the selector. Fig. 5.4 shows a tree and the results of matching several CSS selectors against it.

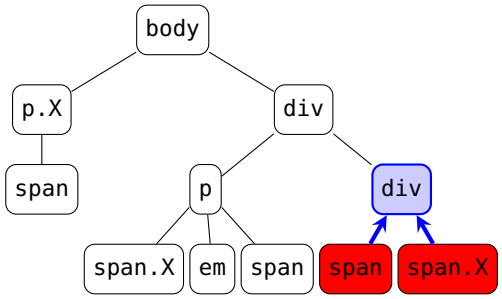
Formally, let *Tree* be the set of all valid HTML trees. A tree consists of a set of elements *e*, where



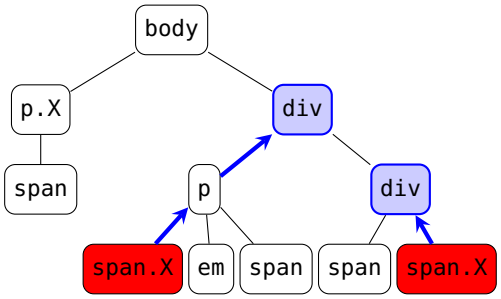
(a) Base HTML tree



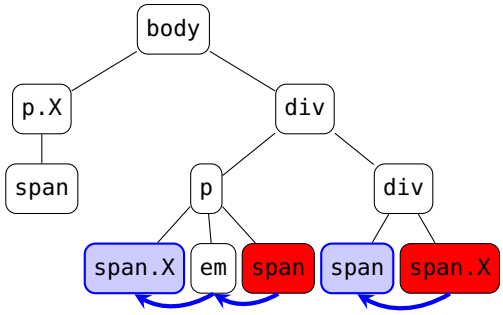
(b) Selecting `*.X` — node name does not matter



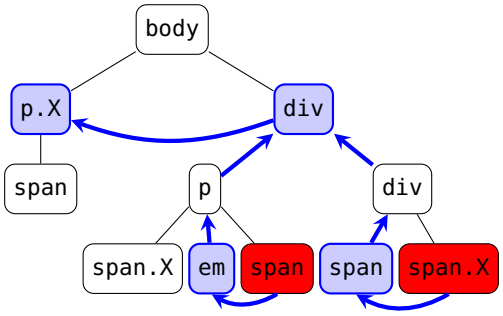
(c) Selecting `div>span` — only immediate children are selected



(d) Selecting `div␣*.X` — as shown, paths are greedily matched from leaf to root



(e) Selecting `span~span` — intervening siblings are ignored



(f) Selecting `*.X~div␣*+span`

Figure 5.4: Simple HTML tree, and nodes matched by various CSS selectors. Red nodes are the results. Blue edges are examined by the selectors during matching; blue nodes match simple-selector components of the selector.

every element has a pointer to its parent (*e.parent*) and to its previous sibling (*e.prevSibling*), when they exist, or null otherwise. (This formalism ignores the text nodes present in HTML trees, as they are not accessible via CSS selectors.) Elements also have pointers to their next sibling (*e.nextSibling*) and first child (*e.firstChild*), defined in the obvious way. Write $e_1.parent = e_2$ when e_2 is e_1's parent, and $e_1.parent^+ = e_2$ when e_2 is e_1's ancestor, and similarly for siblings. Elements have tag names *e.tagName*, identifiers *e.id*, classes *e.class*, and may have arbitrary attributes *e.attr* as well. Then, let *Path* be the set of all *paths* through the tree. A path p consists of a sequence of distinct elements $[p_1, \ldots p_k]$ such that p_{i+1} is reachable from p_i by an arbitrary, non-empty sequence of only parent or previous-sibling pointers: i.e., $p_{i+1} = p_i.parent^+$ or $p_{i+1} = p_i.prevSibling^+$. (Note that this definition explicitly lets paths skip nodes in the tree; paths intuitively include only "landmarks" along a route through the tree, rather than every single step.) By construction, paths are acyclic and finite, since they point "upward and leftward" in well-formed trees. Let $|p| = k$ be the *length* of the sequence. The first element, p_1, is the leaf-most element, and the one which browsers consider as having been matched by the selector. For notational convenience, let p_{last} always indicate the last element p_k of a sequence, whatever its length. Let $p +\!\!+ q$ be the concatenation of two sequences $[p_1, \ldots, p_i, q_1, \ldots q_j]$. Fig. 5.5 defines a browser's interpretations of a selector ("What nodes match selector *sel* in tree *t*?" and "Does element *elem* match selector *sel* in tree *t*?"), and the underlying path-based meaning of a selector.

5.2.2 *CSS syntax with operator precedence*

In the application that follows, I will want to specify parsing precedences among combinators. The grammar in Fig. 5.6 refactors the original CSS grammar (Fig. 5.3) to include precedence rankings among the combinators, and explicitly left-associates combinators of the same type. It is trivial to transform a selector from one form to the other.

Such a transformation is valid: the two grammars are semantically equivalent and so selectors may be freely re-associated with arbitrary precedence. In particular, if any selector *sel* as parsed by this second grammar has the same meaning as the corresponding left-associated selector parsed using the first grammar, the algorithms below can choose whichever associativity is needed. The associativity of CSS selectors falls out from the associativity of path concatenation as used in the path-based semantics. For example, in Fig. 5.4f, it makes no difference to select "`span` nodes that are adjacent to (any node descending from `div` nodes with at least one sibling with class `X`)" versus "(`span` nodes that are adjacent to any node) descending from (`div` nodes with at least one sibling with class `X`)" — both versions compute the blue paths highlighted in the figure.

Element-based meaning: $matches(\cdot)\cdot : Selector \to Tree \to 2^{Element}$

$$matches(sel)t \stackrel{\text{def}}{=} \{p_1 \mid p \in [\![sel]\!]t\}$$

Selector-based meaning: $\cdot.(\cdot) : Selector \to Tree \to Element \to Boolean$

$$sel_t(elem) \stackrel{\text{def}}{=} \exists p \in [\![sel]\!]t, elem = p_1$$

Path-based meaning: $[\![\cdot]\!]\cdot : Selector \to Tree \to 2^{Path}$

Simple selectors

$$[\![*]\!]t \stackrel{\text{def}}{=} \{[n] \mid n \in t\}$$

$$[\![\texttt{tag}]\!]t \stackrel{\text{def}}{=} \{[n] \mid n \in t, n.tagName = \texttt{tag}\}$$

$$[\![A\texttt{\#}\langle name\rangle]\!]t \stackrel{\text{def}}{=} \{p \mid p \in [\![A]\!]t, p_1.id = \langle name\rangle\}$$

$$[\![A\,\texttt{.}\,\langle name\rangle]\!]t \stackrel{\text{def}}{=} \{p \mid p \in [\![A]\!]t, p_1.class \text{ contains } \langle name\rangle\}$$

$$[\![A\,\texttt{[}\,\langle name\rangle\,\texttt{]}]\!]t \stackrel{\text{def}}{=} \{p \mid p \in [\![A]\!]t, p_1.\langle name\rangle \text{ exists}\}$$

$$[\![A\,\texttt{[}\,\langle name\rangle\texttt{=}val\,\texttt{]}]\!]t \stackrel{\text{def}}{=} \{p \mid p \in [\![A]\!]t, p_1.\langle name\rangle = val\}$$

$$[\![A\,\texttt{[}\,\langle name\rangle\texttt{\^{}=}val\,\texttt{]}]\!]t \stackrel{\text{def}}{=} \{p \mid p \in [\![A]\!]t, p_1.\langle name\rangle \text{ starts with } val\}$$

$$[\![A\,\texttt{[}\,\langle name\rangle\texttt{\$=}val\,\texttt{]}]\!]t \stackrel{\text{def}}{=} \{p \mid p \in [\![A]\!]t, p_1.\langle name\rangle \text{ ends with } val\}$$

$$[\![A\,\texttt{[}\,\langle name\rangle\,\texttt{|=}val\,\texttt{]}]\!]t \stackrel{\text{def}}{=} \{p \mid p \in [\![A]\!]t, p_1.\langle name\rangle = val \vee p_1.\langle name\rangle \text{ starts with } (val + \texttt{"-"})\}$$

$$[\![A\,\texttt{[}\,\langle name\rangle\texttt{\~{}=}val\,\texttt{]}]\!]t \stackrel{\text{def}}{=} \{p \mid p \in [\![A]\!]t, val \neq \texttt{""} \wedge val \text{ does not contain whitespace} \wedge (\texttt{" "} + p_1.\langle name\rangle + \texttt{" "}) \text{ contains } (\texttt{" "} + val + \texttt{" "})\}$$

Combinators

$$[\![A\,\texttt{+}\,B]\!]t \stackrel{\text{def}}{=} \{q +\!\!+ p \mid q \in [\![B]\!]t, p \in [\![A]\!]t, q_{last}.prevSibling = p_1\}$$

$$[\![A\,\texttt{\~{}}\,B]\!]t \stackrel{\text{def}}{=} \{q +\!\!+ p \mid q \in [\![B]\!]t, p \in [\![A]\!]t, q_{last}.prevSibling^{+} = p_1\}$$

$$[\![A\,\texttt{>}\,B]\!]t \stackrel{\text{def}}{=} \{q +\!\!+ p \mid q \in [\![B]\!]t, p \in [\![A]\!]t, q_{last}.parent = p_1\}$$

$$[\![A\;B]\!]t \stackrel{\text{def}}{=} \{q +\!\!+ p \mid q \in [\![B]\!]t, p \in [\![A]\!]t, q_{last}.parent^{+} = p_1\}$$

Figure 5.5: CSS selector semantics

$$
\begin{array}{rrcl}
a \in & \textsc{Atomic} & ::= & \mathit{TypeSel} \mid \mathit{UniversalSel} \\
\mathit{simple} \in & \textsc{SimpleSelector} & ::= & a \mid \mathit{simple}[\mathit{Attrib}|\mathit{Id}|\mathit{Class}|\mathit{Pseudo}] \\
\mathit{adj} \in & \textsc{AdjacentSibling} & ::= & \mathit{simple} \mid \mathit{adj} + \mathit{simple} \\
\mathit{sib} \in & \textsc{Sibling} & ::= & \mathit{adj} \mid \mathit{sib} \sim \mathit{adj} \\
\mathit{kid} \in & \textsc{Child} & ::= & \mathit{sib} \mid \mathit{kid} > \mathit{sib} \\
\mathit{desc} \in & \textsc{Descendant} & ::= & \mathit{kid} \mid \mathit{desc} \; \lrcorner \; \mathit{kid} \\
\mathit{sel} \in & \textsc{Selector} & ::= & \mathit{desc}
\end{array}
$$

Figure 5.6: CSS grammar with combinator precedence

Lemma 1. *Let S be a concrete-syntax CSS selector. Let sel_1 be the parsed representation of S according to the original grammar, and let sel_2 be the parsed representation of S according to the precedence-inducing grammar. Then $[\![sel_1]\!] = [\![sel_2]\!]$.*

5.3 *C3 Overlays*

I now present the overlay language and its semantics as developed for C3. The goal here is to define the overlay weaving algorithm concisely and crisply, to prepare for clean definitions of overlay conflicts in the following sections.

Fig. 5.7 presents C3's concrete overlay language. The syntax is familiar HTML, though the tag names are new—indeed, as was pointed out in Chapter 3, in fact I implement the parsing of overlays using a C3 HTML-parser extension (see Section 3.3.1). The abstract syntax is presented in Fig. 5.8; the translation between the two is straightforward. (The current implementation does not include concrete syntax for compositions; this is straightforward to add.) Later sections will consider sub-languages of this abstract syntax and develop appropriate conflict-detection algorithms to handle their various features.

Similar to XUL overlays (Fig. 5.1b), the language in Fig. 5.7 uses ⟨**overlay**/⟩ as the root element of an overlay. Its children define the individual actions to perform. There are two fundamental actions to take: an extension can ⟨**insert**/⟩ new content into the tree, either *before* or *after* nodes targeted by some CSS *selector*; or it can ⟨**replace**/⟩ some subtree targeted by a selector with new content. When replacing a node, it may need to reference the node itself so it can modify its attributes: the ⟨**self**/⟩ node provides such a reference. Within a ⟨**self**/⟩ node, an extension may also want to reference the existing ⟨**contents**/⟩. Together, these actions cover all the choices of overlay abilities in Fig. 5.2, with the benefit that each ability is syntactically distinct from the others.

Base constructions:

⟨**overlay**/⟩
- Description: Root node of extension document
- Children: ⟨**insert**/⟩, ⟨**replace**/⟩, ⟨**before**/⟩, ⟨**after**/⟩, ⟨**modify**/⟩, ⟨**guard**/⟩
- Attributes: None

⟨**guard**/⟩
- Description: Used within ⟨**overlay**/⟩, add guards that restrict when the overlay successfully applies
- Children: None
- Attributes:
  - resource="*res*": The resource being guarded
  - type="require | reject | first | last": The overlay requires the resource to exist, to not exist, to not yet be overlaid, or to never again be overlaid

⟨**insert**/⟩
- Description: Inserts new content (its children) into base document
- Children: Any HTML
- Attributes:
  - selector="*sel*": Selects target location(s) in the base document
  - where="before | after | start | end": Inserts content either before or after the selected targets, or at the start or end of the targets' contents
  - dynamic="true | false": If true, insertion will trigger for nodes dynamically created, as well as for nodes created during parsing

⟨**replace**/⟩
- Description: Replaces existing subtrees with new content
- Children: Any HTML, ⟨**self**/⟩
- Attributes: selector and dynamic, as in ⟨**insert**/⟩

⟨**self**/⟩
- Description: Used within ⟨**replace**/⟩, refers to node being replaced
- Children: Any HTML, ⟨**contents**/⟩
- Attributes: When present, attributes are added to (or replace) those on the targeted node

⟨**contents**/⟩
- Description: Used within ⟨**self**/⟩, refers to the existing contents of the targeted node
- Children: None
- Attributes: None

Syntactic sugar

$$\langle \textbf{before} \ \ldots / \rangle = \langle \textbf{insert} \text{ where="before"} \ldots / \rangle$$

$$\langle \textbf{after} \ \ldots / \rangle = \langle \textbf{insert} \text{ where="after"} \ldots / \rangle$$

$$\begin{array}{l} \langle \textbf{modify} \text{ selector="}\textit{sel}\text{"} \\ \quad \text{where="before"} \rangle \\ \quad \langle \textbf{self } \textit{new attributes} \rangle \\ \quad\quad \textit{new content} \\ \quad \langle / \textbf{self} \rangle \\ \langle / \textbf{modify} \rangle \end{array} = \begin{array}{l} \langle \textbf{replace} \text{ selector="}\textit{sel}\text{"} \rangle \\ \quad \langle \textbf{self } \textit{new attributes} \rangle \\ \quad\quad \textit{new content} \\ \quad\quad \langle \textbf{contents}/ \rangle \\ \quad \langle / \textbf{self} \rangle \\ \langle / \textbf{replace} \rangle \end{array}$$

Figure 5.7: The concrete overlay language for C3. The bottom set of tags are syntactic sugar. A similar desugaring for ⟨**modify** where="after"/⟩ swaps ⟨**contents**/⟩ with *new content*.

Given these abilities, it may seem that "insert content X after nodes matching Y" is equivalent to "replace content $Y.parent$ with itself, its children and new content X". However, there is no way to express $Y.parent$ in CSS and so these two actions are in fact distinct. In practice, these primitives are too low-level, so the overlay language defines convenient syntactic sugar to make inserting content ⟨**before**/⟩ and ⟨**after**/⟩ existing nodes easier, and similarly to ⟨**modify**/⟩ existing nodes' attributes and contents.

Example: While developing C3, the renderer reached a stage where it could support text positioning and layout, but not yet support bulleted lists. The following overlay simulates bulleted lists, by inserting a ⟨**span**/⟩ just before every list item, and by inserting a ⟨**style**/⟩ node into the head of the document to color the bullets:

```
⟨overlay⟩
  ⟨modify selector="head" where="after"⟩
    ⟨self⟩
      ⟨style⟩
        li > span { margin-right: 1em; color: blue; }
      ⟨/style⟩
    ⟨/self⟩
  ⟨/modify⟩
  ⟨modify selector="li" where="before"⟩
    ⟨self⟩⟨span⟩&bull;⟨/span⟩⟨/self⟩
  ⟨/modify⟩
⟨/overlay⟩
```

5.3.1 *Applying overlays to a base HTML document*

The full algorithm for overlay weaving is shown in Figs. 5.9 and 5.10. Note that procedure Apply *mutates* the target node p, and through it the document d. Consequently, the algorithm permits overlays to overlay one another: once overlay o_1 is applied to the document, a subsequent overlay o_2 cannot distinguish o_1-contributed nodes from ones that previously existed, and so o_2 may overlay those newly-added nodes as well. Moreover, the composite document that results depends on the *order* in which extensions were composed. This ability for overlays to apply to each other is an essential part of their power, but also makes dependency checking that much more important.

In the next section I first consider a trivial case where overlays *cannot* see overlay-injected content. This changes the weaving algorithm only slightly: instead of applying overlays in the innermost loops, the *insertionPoints* sets are saved, and overlays are all applied after the *insertionPoints* are fully computed.

$$
\begin{array}{rrll}
c \in & \textsc{Comp} & ::= & g \\
 & & | & c\,;c \qquad \textit{–sequential composition} \\
 & & | & c\,!\,c \qquad \textit{–exclusive composition} \\
 & & | & c? \qquad \textit{–optional composition} \\
g \in & \textsc{Guard} & ::= & o \\
 & & | & \mathit{Require}(\vec{r}, g) \qquad \textit{–g requires r be defined} \\
 & & | & \mathit{Reject}(\vec{r}, g) \qquad \textit{–g requires r be undefined} \\
 & & | & \mathit{First}(\vec{r}, g) \qquad \textit{–r must not be overlaid before g} \\
 & & | & \mathit{Last}(\vec{r}, g) \qquad \textit{–r must not be overlaid after g} \\
o \in & \textsc{Overlay} & ::= & \mathit{Overlay}(\vec{a}) \\
a \in & \textsc{Action} & ::= & \mathit{Insert}(s, w, d, h) \\
 & & | & \mathit{Replace}(s, d, \mathit{rep}) \\
r \in & \textsc{Resource} & = & \mathit{Selector}(s) \uplus \mathit{Id}(i) \uplus \mathit{Key}(k) \uplus \mathit{Selected}(h) \uplus \cdots \textit{–various resource types} \\
s \in & \textsc{Selector} & & \textit{–to be determined later} \\
w \in & \textsc{Where} & ::= & \text{before} \mid \text{after} \mid \text{start} \mid \text{end} \\
d \in & \textsc{Dynamic} & ::= & \text{true} \mid \text{false} \\
\mathit{rep} \in & \textsc{Replaced} & ::= & h \mid \mathit{Self}(\vec{t}, \mathit{con}) \\
\mathit{con} \in & \textsc{Contents} & ::= & h \mid \mathit{Contents} \\
h \in & \texttt{HTML} & & \\
i \in & \textsc{Ident} & & \\
k \in & \textsc{Key} & & \\
t \in & \textsc{Attribs} & ::= & (\texttt{name}, \texttt{value})
\end{array}
$$

Figure 5.8: The complete abstract syntax for the overlay language of C3. Following sections will consider sub-languages of this definition.

```
Before firing the readyStateChanged event on document d, WeaveDoc(d)
When a node n is inserted for the first time into a document, WeaveNode(Document n)
procedure WeaveDoc(d)
    for all e ← each extension to be loaded do
        for all o ← e.Overlays do
            for all a ← o.Actions do
                insertionPoints ← d.QuerySelectorAll(a.Selector)
                for all p ← insertionPoints do
                    Apply(a, p)          ▷ This mutates p, and therefore the containing document d
                end for
            end for
        end for
    end for
end procedure
procedure WeaveNode(Node n)
    for all e ← InstalledExtensions do
        for all o ← e.Overlays do
            for all a ← o.Actions do
                if a.Dynamic = true then
                    Apply(o, n)
                    insertionPoints ← n.QuerySelectorAll(a.Selector)
                    for all p ← insertionPoints do
                        Apply(a, p)
                    end for
                end if
            end for
        end for
    end for
end procedure
```

Figure 5.9: Static and dynamic weaving of overlays into HTML documents

```
procedure APPLY(Action a, Node n)
    if a.type = Insert then
        Create a document fragment f in n's document
        Deep-clone all children of a into f
        if a.where = Before then
            Insert f before n.
        else if a.where = After then
            Insert f before n's next sibling.
        end if
    else if a.type = Replace then
        if a contains a ⟨self/⟩ node s then
            Copy attributes from s to n
            Copy all children of s that precede a ⟨content/⟩ node before n's first child
            Copy all children of s that follow a ⟨content/⟩ node after n's last child
        else
            Create a document fragment f in n's document
            Deep-clone all children of a into f
            Remove all of n's children
            Insert f into n
        end if
    end if
end procedure
```

Figure 5.10: Weaving an overlay into a target node

$$
\begin{array}{rrcl}
o \in & \textsc{Overlay} & ::= & \mathit{Overlay}(a_1, \ldots, a_n) \\
a \in & \textsc{Action} & ::= & \mathit{Insert}(s, w, h) \\
s \in & \textsc{Selector} & ::= & \texttt{tagName\#id} \\
w \in & \textsc{Where} & ::= & \text{start} \mid \text{end} \\
h \in & & & \texttt{HTML}
\end{array}
$$

Figure 5.11: Abstract overlay language with only strawman abilities

5.4 *Overlay conflict detection: Naïve overlays*

This section examines the limited overlay language of Fig. 5.11, which limits overlays to using only the ⟨**insert**/⟩ tag (no modifications or removals) where the selector is always of the form *tagName#id*. Additionally, overlay composition is defined to target the base document only. In this limited case, inter-overlay interactions are nearly impossible, hence inter-overlay conflicts are similarly rare.

5.4.1 *Motivating examples*

Hello, world: Consider the simple HTML document "hello-world template" below:

⟨**p** id="greeting"⟩
  ⟨**span** id="opening"⟩Hello,⟨/**span**⟩
⟨/**p**⟩

Suppose two extension authors wanted to complete the greeting by supplying the subject:

- **OV1**: *Overlay*(*Insert*(p#greeting, end, ⟨**span** id="subject"⟩stranger.⟨/**span**⟩))
- **OV2**: *Overlay*(*Insert*(p#greeting, end, ⟨**span** id="subject"⟩friend.⟨/**span**⟩))

And suppose another extension author wanted to embellish the greeting:

- **OV3**: *Overlay*(*Insert*(p#greeting, end, ⟨**span** id="modifier"⟩and good day,⟨/**span**⟩))

If a user applied both OV1 and OV2, the result would be a non-sensical sentence—"Hello, stranger. friend." or "Hello, friend. stranger." [*sic*]—depending on the order of insertion. While this English-level conflict is certainly out of scope of overlay conflict detection, an automated system *can* detect that the composite HTML document is not well-formed, because two ⟨**span**/⟩ elements exist with the same id "subject".

On the other hand, if the user applied OV1 and OV3 (or OV2 and OV3), the result would be a well-formed HTML document, since no duplicate ids would be present. Note that the sentence might still be non-sensical English—"Hello, stranger. and good day," [*sic*]— due to insertion order. However, this is *not* a conflict between the extensions, since they merely claim to be inserted at the end of existing content; they make no mention of their order relative to other extensions. I will return to conflicts of this sort later.

Hotkey responses: The XUL hotkey scenario from Section 5.1.2 also occurs in HTML5, and can be expressed using the overlay language above. Application-wide hotkeys are defined in HTML5 using ⟨**command**/⟩ elements.[4] Assume the base document defines a tag ⟨**div** id="keys"/⟩, within which it chooses to place its ⟨**command**/⟩ elements. Then define three extensions:

- **OV4**: *Overlay*(*Insert*(`div#keys`, end, ⟨**command** accesskey="F" onclick="alert("1")"/⟩))
- **OV5**: *Overlay*(*Insert*(`div#keys`, end, ⟨**command** accesskey="F" onclick="alert("2")"/⟩))
- **OV6**: *Overlay*(*Insert*(`div#keys`, end, ⟨**command** accesskey="G" onclick="alert("3")"/⟩))

This presents a different challenge than the previous example: the composite document is well-formed, but the intended semantics of hotkeys are not satisfied. A conflict-detection system should detect that OV4 and OV5 are in conflict with each other, as they both are triggered by the hotkey "F", but that neither one conflicts with OV6.

5.4.2 Approach

For a single overlay to succeed, it must be the case that the nodes selected by each ⟨**insert**/⟩ action exist in the base document. Additionally, the well-formedness property of HTML requires that the overlay does not insert any new content with an id that is already defined in the base document. Formally, let defs(d) $\in$ *TagNames* $\times$ *Ids* be the set of pairs of tagnames and ids for all nodes with ids in the base document d. Let

$$\text{defs}(\mathit{Insert}(\texttt{tagName\#id}, \_, \_, h)) = \text{defs}(h)$$
$$\text{reqs}(\mathit{Insert}(\texttt{tagName\#id}, \_, \_, \_)) = \{(\mathit{tagName}, \mathit{id})\}$$

[4] There are several other ways to define "command concepts", and several other ways to define hotkeys. The scope and behavior of these alternate methods are not yet fully defined, so for now I focus solely on ⟨**command**/⟩ elements not contained within a ⟨**menu**/⟩; these seem intended to be globally-scoped.

Let $ids(s) = \{id \mid \exists t.(t, id) \in s\}$ project just the id component of tagname/id pairs. Finally for an overlay $o = Overlay(a_1, \ldots, a_n)$ let $\text{defs}(o) = \bigcup_i \text{defs}(a_i)$ and $\text{reqs}(o) = \bigcup_i \text{reqs}(a_i)$ be the union of the defs and reqs of their children actions, respectively. Assuming both o and d are well-formed, and each contain no duplicated ids, then o successfully applies to d exactly when

$$\text{reqs}(o) \subseteq \text{defs}(d) \wedge ids(\text{defs}(o)) \cap ids(\text{defs}(d)) = \emptyset. \tag{5.1}$$

For multiple overlays to succeed, all overlays must succeed individually. This simple statement relies on the restriction that overlays do not overlay each other: either the necessary targets are present in the base document or an overlay fails; no work is needed to ensure any dependencies between overlays are satisfied.

Moreover, no overlay can define an id that another overlay also defines:

$$ids(\text{defs}(o_1)) \cap ids(\text{defs}(o_2)) = \emptyset. \tag{5.2}$$

If there are other uniqueness requirements (as with hotkey support), they can be handled similarly to ids, ensuring that multiple overlays do not provide duplicate definitions.

These two conditions are sufficient for a conflict-detection algorithm for these limited overlays. If Eq. (5.1) is not satisfied by some overlay, then that overlay is faulty, and the ids causing the problem are readily identified. If Eq. (5.2) fails for some pair of distinct overlays, then they conflict with each other, and the ids causing the conflict are again readily identified. Otherwise, all the overlays are successfully applicable to the base document, and in any order.

5.4.3 Examples, revisited

Looking at the hello-world example, the earlier definitions will yield

$$\begin{aligned}
\text{defs}(\text{OV1}) &= \{(\text{span}, \text{subject})\} \\
\text{reqs}(\text{OV1}) &= \{(\text{p}, \text{greeting})\} \\
\text{defs}(\text{OV2}) &= \{(\text{span}, \text{subject})\} \\
\text{reqs}(\text{OV2}) &= \{(\text{p}, \text{greeting})\} \\
\text{defs}(\text{OV3}) &= \{(\text{span}, \text{modifier})\} \\
\text{reqs}(\text{OV3}) &= \{(\text{p}, \text{greeting})\} \\
\text{defs}(doc) &= \{(\text{p}, \text{greeting}), (\text{span}, \text{opening})\}
\end{aligned}$$

$$
\begin{array}{rrcl}
c \in & \textsc{Comp} & ::= & g \mid c\,;c \mid c\,!\,c \mid c? \\
g \in & \textsc{Guard} & ::= & o \\
 & & \mid & \mathit{Require}(\vec{r}, g) \\
 & & \mid & \mathit{Reject}(\vec{r}, g) \\
 & & \mid & \mathit{First}(\vec{r}, g) \\
 & & \mid & \mathit{Last}(\vec{r}, g) \\
o \in & \textsc{Overlay} & ::= & \mathit{Overlay}(\vec{a}) \\
a \in & \textsc{Action} & ::= & \mathit{Insert}(s, w, \vec{h}) \\
 & & \mid & \mathit{Modify}(s, \vec{t}) \\
r \in & \textsc{Resource} & = & \mathit{Selector} \uplus \mathit{Id} \uplus \mathit{Key} \uplus \mathit{Selected} \uplus \cdots \\
s \in & \textsc{Selector} & ::= & \texttt{tagName\#id} \\
w \in & \textsc{Where} & ::= & \text{start} \mid \text{end} \\
h \in & \textsc{HTML} & & \\
t \in & \textsc{Attribs} & ::= & (\texttt{name}, \texttt{value})
\end{array}
$$

Figure 5.12: Abstract overlay language with insertion, attribute modification and composition

Note that reqs(OV1) $\subseteq$ defs(doc), and that $\mathit{ids}(\text{defs}(\text{OV1})) \cap \mathit{ids}(\text{defs}(\mathit{doc})) = \emptyset$, and likewise for OV2 and OV3, so Eq. (5.1) is satisfied. However, Eq. (5.2) fails: $\mathit{ids}(\text{defs}(\text{OV1})) \cap \mathit{ids}(\text{defs}(\text{OV2})) \neq \emptyset$, which shows that OV1 and OV2 conflict with each other, as anticipated.

The hotkey example, however, does not present any conflicts under the algorithm as defined so far: OV4, OV5 and OV6 do not define any elements with ids, so they certainly do not define any elements with the same id. However, with the slight generalization alluded to above, defs can be expanded to include the hotkeys defined by $\langle$**command**/$\rangle$s, as follows:

$$\text{defs}(\text{OV4}) = \{(\text{command}, \mathit{Key}(\text{F}))\}$$
$$\text{defs}(\text{OV5}) = \{(\text{command}, \mathit{Key}(\text{F}))\}$$
$$\text{defs}(\text{OV6}) = \{(\text{command}, \mathit{Key}(\text{G}))\}$$

where Key(F) is a tagged value that is "not the same" as any regular ids. Then, as before, Eq. (5.2) fails: $\mathit{ids}(\text{defs}(\text{OV4})) \cap \mathit{ids}(\text{defs}(\text{OV5})) \neq \emptyset$, which gives the desired conflict.

5.5 *Overlay conflict detection: Firefox-like overlays*

This section examines the slightly generalized language in Fig. 5.12. It introduces a new action, *Modify*, that permits editing the attributes that are defined on an existing node. (For simplicity,

the overlay language presented here does not expose actions to remove existing attributes or to concatenate to existing values. These would be straightforward to add, but clutter the presentation needlessly.) Additionally, overlays are now permitted to "see" each other's injected content, which raises the possibility of overlays depending upon one another. Consequently, not only must the system detect conflicts among extensions, but it must also compute a feasible composition order if one exists.

5.5.1 Motivating examples

Load-order dependencies: It is common for Firefox extensions to add new submenus to the Firefox "Tools" menu. For example, the Session Manager extension adds a submenu with menu items to save currently open tabs as a "session", to load a previously-saved session, and to delete saved sessions:

- **OV7**: *Overlay*(*Insert*(`menu#tools-menu`, end,
  ⟨**submenu** id="sessionManagerMenu"⟩
  ⟨**menu**⟩Load session...⟨/**menu**⟩
  ⟨**menu**⟩Save session...⟨/**menu**⟩
  ⟨**menu**⟩Delete session...⟨/**menu**⟩
  ⟨/**submenu**⟩))

A common use-case for Session Manager is to use sessions as "temporary bookmarks", where the session is used once and then discarded. A second extension might make this available as its own menu item:

- **OV8**: *Overlay*(*Insert*(`submenu#sessionManagerMenu`, start,
  ⟨**menu**⟩Load and discard session...⟨/**menu**⟩))

These two extensions are not in conflict, but OV8 *depends upon* OV7 being loaded first: otherwise, OV8's target node does not yet exist.

Modifying attributes Consider an application with a several choices in an option list:

⟨**select** id="myOpts"⟩
⟨**option** id="optA"⟩A⟨/**option**⟩
⟨**option** id="optB" selected="selected"⟩B⟨/**option**⟩
⟨**option** id="optC"⟩C⟨/**option**⟩
⟨**option** id="optD"⟩D⟨/**option**⟩
⟨/**select**⟩

Suppose an extension tries to contribute a new option "E", and make that option selected by default. Not only will it need to *Insert* the new ⟨**option**/⟩, but it will have to *Modify* the currently selected one:

- **OV9**: *Overlay*(*Insert*(`select#myOpts`, end,
  ⟨**option** id="optE" selected="selected"⟩E⟨/**option**⟩),
  Modify(`option#optB`, (selected, "")))

Such modifications are useful only occasionally, but when they are, they are indispensable.

Remark: The portion of the language for attribute modifications does not include removing attributes or concatenating new values to existing ones. One natural use for concatenation might be to append new `CSS` classes to an element's class attribute, so as to enhance its style. However, the overlay's selector—which already found this element successfully, by its id—can just as easily be used by `CSS` style rules. In the language variants explored below, the selector language will be more flexible still, so again such usage is unneeded.

5.5.2 Guarded overlays and compositions

The preceding examples highlight several distinct types of conflicts that must all be properly accounted for by a conflict detection algorithm. The hotkey example requires knowing that each key must be unique within a document. The options list example must include the `HTML` semantics that at most one option be selected within a group. And the hello-world example shows that even if all uniqueness constraints are known and modeled, an analysis may potentially still miss higher-level conflicts that are not expressible within the markup language. To support all of these kinds of conflicts in a single system, in a uniform, declarative way, all future analyses will be defined in terms of an abstract set of *resources*, which may include pieces of the overlay itself (e.g., ids, keys, selectors), and *guards*, which may be automatically inferred or may be assertions manually provided by the overlay author to help further constrain when the overlay applies successfully.

The language of Fig. 5.12 includes two new constructions, *guards* and *compositions*, that permit expressing the constraints that were hard-coded in the previous section. I illustrate their behavior by revisiting the earlier examples.

"Hello, world", revisited: In the strawman language of Section 5.4, there was no way for OV3 to require that it be loaded before OV1 or OV2, because there was no way for one overlay to mention

content provided by another overlay. Now that overlays can "see" each other, it is possible for OV1 and OV2 to assert that they load after OV3 does, by attempting to overlay it but do nothing to it:

- **OV1***: *Overlay*(*Insert*(`p#greeting`, end, ⟨**span** id="subject"⟩stranger.⟨/**span**⟩),
  Modify(`span#modifier`))

Such an overlay has the desired effect of adding `span#modifier` to reqs(OV1*), but there are three fundamental problems with this approach. First, OV1 does not actually modify the ⟨**span**/⟩ produced by OV3, so OV1* should not claim to modify it. Second, this approach is non-local: it is OV3's responsibility to impose ordering constraints on itself, not OV1's and OV2's to accommodate OV3. Third, and most important, OV1* will not successfully apply if OV3 is not present, a behavior that differs from OV1.

Instead, a better approach would be to assert some extra *guards* in the definition of OV3, so that it can express the constraint that it must not apply when OV1 or OV2 is present:

- **OV3'**: *Reject*(*Id*(`subject`),
  Overlay(*Insert*(`p#greeting`, end,
  ⟨**span** id="modifier"⟩ and good day,⟨/**span**⟩))))

Read this as "OV3' applies successfully to documents in which the *Overlay* successfully applies, but not those documents that match the *Id*". The four guard types behave as follows:

- *Require*($\vec{r}$, g): When g succeeds and all resources in r are present in the document
- *Reject*($\vec{r}$, g): When g succeeds and all resources in r are not present in the document
- *First*($\vec{r}$, g): When g succeeds and no resources in r have been overlaid by some prior overlay
- *Last*($\vec{r}$, g): When g succeeds and no resources in r will be overlaid by some future overlay

Hotkey responses, revisited: Rather than hard-code knowledge of the markup language's semantics into the conflict-resolution algorithm, constraints can be encoded using additional guards. For example, uniqueness constraints can be expressed using *Last* guards: OV4 and OV5, which must be the last key to use the letter "F", can be written as

- **OV4'**: *Last*(*Key*(F),
  Overlay(*Insert*(`div#keys`, end, ⟨**command** accesskey="F" onclick="alert("1")"/⟩)))

- **OV5′**: *Last*(*Key*(F),
  Overlay(*Insert*(`div#keys`, end, ⟨**command** accesskey="F" onclick="alert("2")"/⟩)))

Now the presence of both OV4′ and OV5′ will trigger a conflict due to these general-purpose *Last* guards, rather than due to an algorithm finely tuned to `HTML5`'s idiosyncrasies.

Of course, it would be tedious to require that overlay authors write these *Last* guards every time they use a ⟨**command**/⟩ element: such guards are mechanically derivable from the markup. Fortunately, such idiosyncrasies can be encoded during the parsing of concrete-syntax overlays into the abstract overlay language. (Similarly, a parser for concrete-syntax `XUL` overlays could encode `XUL`-specific idiosyncrasies.) This leaves the underlying conflict-detection algorithm agnostic to the initial input language, but also relieves authors from the tedium of writing "obvious" guards.

Another representation of uniqueness

A slightly different approach might use *First* guards to encode uniqueness, and at first glance the two approaches seem nearly identical: after all, two overlays can't both be the first to overlay something, so only one of them can do so, and therefore also be the last to overlay it. But the converse is not actually true. Consider again the example at the start of this chapter, where Firefox UI is split into a base document and several overlays for purposes of code clarity. It may be reasonable for an extension to wish to be the only *extension* to overlay some resource. It would be overly brittle for the extension to be the *First* to overlay some resource, as such a constraint makes the extension incompatible when Firefox refactors that resource into an overlay. By contrast, the extension still can be the *Last* to access the resource; Firefox could refactor "beneath" the overlay and not affect its behavior.

Experimentally, it appears that Firefox handles keybindings using something akin to *Last*: if a key is defined by Firefox, it can be overridden by an overlay (i.e., the overlay takes precedence), but if two overlays try to define the same key, the behavior is undefined. Such behavior can be accommodated in the `XUL` overlay-to-guarded overlay translator, and if future versions of Firefox change their behavior, only this translator need be updated.

Composing overlays within one extension

So far, overlays are "all-or-nothing": they either apply successfully or not at all. Extension authors may reasonably want to control how their overlays are applied in a more fine-grained fashion. To support such intents, the overlay language includes three types of composition: sequencing, optional components, and one-of-several targeting.

Sequencing helps modularize overlay code: the extension author can separate overlays into logically-distinct pieces, and ensure that they are applied to the base document in the desired order. The composition $c_1 ; c_2$ succeeds for documents D whenever c_1 succeeds in D and c_2 succeeds in the document resulting from applying c_1 to D. Sequencing is associative: composing $c_1 ; (c_2 ; c_3)$ with D results in the same document as composing $(c_1 ; c_2) ; c_3$ with D.

One-of-several targeting is useful for the common case where extensions are written that may apply to multiple, "similar" applications. For example, an extension may add a command to the "Tools" menu of both Firefox and Thunderbird. In Firefox, that menu is ⟨**menu** id="tools-menu"/⟩, but in Thunderbird it is ⟨**menu** id="tasksMenu"/⟩. Currently, extension authors simply write

Overlay(*Insert*(`menu#tools-menu`, end, *new content*), *Insert*(`menu#tasksMenu`, end, *new content*))

This implicitly relies on a quirk of Gecko that individual overlays with missing targets are silently dropped. This often works in practice, but it is subtly wrong: if another extension happens to insert an ⟨**menu** id="tasksMenu"/⟩ element into Firefox, the result would be to insert *new content* twice, with potentially bizarre results. In the overlay language, the one-of-several composition allows extension authors to indicate *mutually exclusive overlays*. A composition $c_1 ! c_2$ succeeds in a document D if either of c_1 or c_2 can apply successfully to D, but ultimately only one will be applied. Like sequencing, one-of-many is associative: "one of (one of *a* or *b*) or *c*" means the same thing as "one of *a* or (one of *b* or *c*)".

Finally, extensions often deliberately include optional behaviors that are installed only when other extensions are installed as well. Once again Firefox extensions rely on Gecko's silent dropping of unmatched overlays to implement these optional features, and this works successfully. The unfortunate consequence, though, is that Gecko cannot distinguish optional components (to be ignored) from typos (to be surfaced as errors). In the overlay language, components must be marked explicitly as $c?$ for the composition and conflict-detection algorithms to consider them optional; everything else must succeed or fail as a unit.

5.5.3 Overlays as document transformers

The examples above should give a flavor of the analysis needed here: a precise conflict-detection algorithm must keep track of resources and must keep track of four kinds of constraints on those resources. Moreover, as OV7 and OV8 show, it must record how those constraints and available resources *change* as a consequence of sequentially applying extensions' compositions overlays, so that later compositions can see the effects of preceding ones. Ultimately, it must compute, given a base document d and a set of extensions $\vec{e}$ that define compositions $\vec{c}$, some permutation of

the extensions that yields a feasible loading order (i.e., the composition $d ; c_{\pi(1)} ; \cdots ; c_{\pi(n)}$, for some permutation π, is valid according to the analysis), or else demonstrate some subset of the overlays that are conflicting.

To model how resources change as a consequence of a single overlay, the analysis will view an overlay as a *document transformer* that takes an input document and produces a modified document as a result; the analysis is then concerned with tracking resources as they are manipulated within successive documents. Model the state of the document by a record $\{Def, Undef, Clean, Frozen\}$ of sets of resources describing the set of requirements that must be defined, that must not be defined, that have not yet been overlaid and that must not be overlaid in order for the overlay to succeed. These sets assert that resources exist in the input, or guarantee that they exist in the output; they convey no information about resources that are not mentioned. An empty input *Def* set does *not* assert that the input document is empty, but rather that nothing is known to be defined in it. For an input and output state pair, there are consequently eight sets: of these, the output *Def* and *Undef* sets are uniquely determined from the overlay itself and its input *Def* and *Undef* sets. Additionally, the input *Frozen* and the output *Clean* sets are redundant: anything that was *Clean* before some overlay applied, and that was unused by that overlay, will remain *Clean*; similarly, an overlay need only check the *output Frozen* set of the preceding overlay, and not vice versa. The remaining four sets can be specified or extended independently, using the four guard types above. The analysis can model the "interface" of an overlay, an abstract representation of its effects on a document, as this pair of states (S_i, S_o), which can be thought of as the weakest precondition and strongest postcondition of the overlay; an example is shown below. Computing this interface is purely structural, and does not encode any language-specific constraints (e.g., hotkey uniqueness).

Next, lift this notion of interface from overlays to guarded overlays, by folding into the input and output sets the resources mentioned by guards. This step incorporates the language-specific constraints introduced by translating from the concrete to the abstract overlay language. Finally, lift the interface notion once more, from guarded overlays to compositions. This step handles any explicit composition operators used by extension authors, and also lets the analysis treat any extension that defines multiple, independent overlays as one that instead defines a single sequential composition. Indeed, the algorithm uses essentially the same composition rules to combine multiple guarded overlays from a single extension as are needed to detect valid composition of guarded overlays from multiple extensions. The rest of this section illustrates the algorithm only through examples that convey the main ideas without distracting details; see Section 5.6 for evaluation, and Appendix B for more details.

The interface of "Hello, world": Recall the definitions of OV1 and OV3':

- **OV1**: *Overlay*(*Insert*(`p#greeting`, end, ⟨**span** id="subject"⟩stranger.⟨/**span**⟩))
- **OV3'**: *Reject*(*Id*(`subject`),
  Overlay(*Insert*(`p#greeting`, end,
  ⟨**span** id="modifier"⟩ and good day,⟨/**span**⟩))))

Including the constraints that ensure ids are unique, the meaning of OV1 in words is "for any document containing a node `p#greeting` and not containing nodes matching `span#subject` or any node with id `subject`, OV1 will produce a document that contains `p#greeting`, `span#subject` and a node with id `subject`". In symbols, this becomes the state pair (S_i^1, S_o^1):

$$S_i^1 = \left\{ \begin{array}{l} Def = \{Selector(\texttt{p\#greeting})\} \\ Undef = \{Selector(\texttt{span\#subject}), Id(\texttt{subject})\} \\ Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\}$$

$$S_o^1 = \left\{ \begin{array}{l} Def = \{Selector(\texttt{p\#greeting}), Selector(\texttt{span\#subject}), Id(\texttt{subject})\} \\ Undef = \emptyset \\ Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\}$$

The *Selector* resources are used to describe the structural changes to the document, so that compositions can detect if their targets exist. The *Id* resources are used for the unique-id analysis.

Similarly including the unique-id constraints along with the *Reject* constraint, the effect of OV3' in words is "for any document containing a node matching `p#greeting` and not containing nodes matching `span#modifier` or any node with id `modifier` or any node with id `subject`, OV3' will produce a document that contains `p#greeting`, `span#modifier` and a node with id `modifier` and still does not contain any node with id `subject`". In symbols, this is the state pair (S_i^3, S_o^3):

$$S_i^3 = \left\{ \begin{array}{l} Def = \{Selector(\texttt{p\#greeting})\} \\ Undef = \{Selector(\texttt{span\#modifier}), Id(\texttt{modifier}), Id(\texttt{subject})\} \\ Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\}$$

$$S_o^3 = \left\{ \begin{array}{l} \quad Def = \{Selector(\texttt{p\#greeting}), Selector(\texttt{span\#modifier}), Id(\texttt{modifier})\} \\ Undef = \{Id(\texttt{subject})\} \\ \quad Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\}$$

The *Reject* constraint is added to the input *Undef* set by the assertion that the specified resource not exist, and to the output *Undef* set because OV3′ does not itself cause that resource to be defined.

In summary, any concrete overlay can be represented as an abstract overlay, decorated with its language-specific attendant guards, and modeled as a document state-pair. These state pairs are used to define the composition of multiple (concrete) overlays, and thereby determine a feasible composition order.

5.5.4 Determining overlay composition order: the conflict graph

Consider just the two overlays OV3′ and OV1. Can OV3′ be composed with OV1, and if so in what order? Suppose the system tried OV1 ; OV3′, applying OV1 first. Then the *input* state of OV3′ must be compatible with the *output* state of OV1. However, it is clear that $S_o^1.Def \cap S_i^3.Undef = \{Id(subject)\} \neq \emptyset$. In words, something which OV3′ requires to be undefined is guaranteed to be defined by OV1. This one contradiction suffices to prohibit the ordering OV1 ; OV3′.

On the other hand, suppose the system tried the other order, OV3′ ; OV1. This time, the intersection test above succeeds, as well as several others. In general, for OV3′ to precede OV1 the system must check:

$$S_o^3.Def \cap S_i^1.Undef = \emptyset \tag{5.3}$$

$$S_o^3.Def \cap \text{defs}(\text{OV1}) = \emptyset \tag{5.4}$$

$$S_o^3.Frozen \cap \text{used}(\text{OV1}) = \emptyset \tag{5.5}$$

$$\text{reqs}(\text{OV3}') \cap S_i^1.Clean = \emptyset \tag{5.6}$$

These equations assert that OV3′ must not define anything OV1 requires as undefined, nor anything that OV1 itself defines; it also must not freeze anything that OV1 later uses, or overlay anything that OV1 asserts to be clean. These four equations indeed hold for OV3′ and OV1, and hence OV3′ ; OV1 is a feasible composition order for the two overlays.

The resulting composition *itself* can be represented by a state-pair interface $(S_i^{3,1}, S_o^{3,1})$ that is

computable from S_i^1, S_i^3, S_o^1 and S_o^3. The interface should mean "in a document satisfying the combined requirements of OV3′ and OV1, as described by $S_i^{3,1}$, the composition OV3′;OV1 will result in a document satisfying their combined guarantees, as described by $S_o^{3,1}$". Consequently, the input state $S_i^{3,1}$ must be a combination of the *Def* and *Undef* sets of S_i^3 with those of S_i^1. To do otherwise would effectively enforce that OV3′ define everything needed by OV1, which is not the intended semantics:

$$S_i^{3,1} = \left\{ \begin{array}{rl} \mathit{Def} = & \{\mathit{Selector}(\texttt{p\#greeting})\} \\ \mathit{Undef} = & \{\mathit{Selector}(\texttt{span\#modifier}), \mathit{Id}(\texttt{modifier}), \\ & \quad \mathit{Selector}(\texttt{span\#subject}), \mathit{Id}(\texttt{subject})\} \\ \mathit{Clean} = & \emptyset \\ \mathit{Frozen} = & \emptyset \end{array} \right\}$$

$$S_o^{3,1} = \left\{ \begin{array}{rl} \mathit{Def} = & \{\mathit{Selector}(\texttt{p\#greeting}), \\ & \quad \mathit{Selector}(\texttt{span\#subject}), \mathit{Id}(\texttt{subject}), \\ & \quad \mathit{Selector}(\texttt{span\#modifier}), \mathit{Id}(\texttt{modifier})\} \\ \mathit{Undef} = & \emptyset \\ \mathit{Clean} = & \emptyset \\ \mathit{Frozen} = & \emptyset \end{array} \right\}$$

In words, this says that "for any document containing a node matching `p#greeting` and not containing nodes matching `span#modifier` or `span#subject` or any nodes with id `subject` or `modifier`, the composition (OV3′;OV1) will produce a document containing `p#greeting`, `span#modifier`, `span#subject`, and nodes with id `subject` or `modifier`".

Unfortunately, this pairwise approach will not scale beyond a small number of overlays: because the system needs to track the effects of *all* preceding overlays to determine if the next overlay will succeed, it would need to test all $n!$ permutations of n overlays, which is exponentially too much work to be feasible.

The approach Eqs. (5.3) to (5.6) above define when one overlay may feasibly follow another. If, however, *any* equation is unsatisfied (as when applying OV1 before OV3′), the analysis knows such an ordering is infeasible: in the example above, OV3′ *must not follow* OV1. The analysis records such observations in a directed graph, whose nodes are overlays, and whose edges are the pairwise must-not-follow relation. For diagnostic purposes later, it annotates the edges with *which* of the four equations above were unsatisfied. This is the *conflict graph* for a set of extensions.

Note that it is possible that extension X must-not-follow extension Y according to Eqs. (5.3) to (5.6), and Y must-not-follow extension Z, but that X may feasibly follow Z:

- X: *Overlay*(*Insert*(`p#w`, ⟨**p** id="x"/⟩))
- Y: *Overlay*(*Insert*(`p#x`, ⟨**p** id="y"/⟩))
- Z: *Overlay*(*Insert*(`p#y`, ⟨**p** id="z"/⟩))

X must-not-follow Y because Y uses something that X defines; similarly for Y and Z. But X and Z are independent of each other, and can be applied in either order. However, when all three extensions are loaded together, the conflict graph shows that Z transitively must-not-follow X because of a *path* of pairwise must-not-follow edges.

The edges in the conflict graph record ordering dependencies between pairs of extensions. Just as informative, though, are the *missing* edges: if two overlays in the conflict graph are not connected by a directed path, then they can be applied to base document in *either order*—they commute with one another, at least with respect to all the constraints they define. In this way, the must-not-follow analysis defined here is a commutativity analysis in disguise.

Recall that, given some base document d and a set of extensions $\vec{e}$, the analysis must compute either a feasible composition order, if one exists, or else a set of conflicting extensions. It can resolve both questions simply by computing whether the conflict graph is *acyclic*. If the graph contains a cycle, then there exist extensions $e_1, \ldots e_n$ such that e_i must not follow e_{i+1}, and e_n must not follow e_1. Transitively, each extension must not follow itself, which is problematic: there cannot exist a loading order for which all of these constraints are satisfied, and so extensions $e_1, \ldots, e_n$ are in conflict. Moreover, the annotations on each edge of the cycle permit informing the user precisely *why* the extensions conflict. In the current example, this analysis can inform the user that OV1 defines something, namely *Id*(subject), that OV3′ requires as undefined, and therefore cannot precede OV3′.

However, if the conflict graph is acyclic, then any topological sort of the graph will yield a *potentially* valid loading order that respects all extension dependencies. It remains to check whether the base document in fact satisfies the input requirements of the resulting composition: that is, does the actual target document D satisfy $S_i^{3,1}$? Additionally, the analysis must check that the composition does not define anything which the document already defines. In fact, it must check precisely the same four conditions as it did for sequencing of two compositions.

To resolve these remaining questions, in practice the analysis actually represents the base document itself by its interface, as if it too were an overlay. The interface for a base document begins with the empty input state, and produces an output state containing everything defined in the base document. Adding this interface as a node to the conflict graph obtains the needed sequencing checks "for free". The final algorithm for computing conflicts and loading order is:

1. For each extension E_i and its corresponding composition c_i, compute its interface I_i.

2. For the base document D, compute its interface I_D.

3. Construct the conflict graph, with nodes I_i and I_D, and edges $I \rightarrow J$ if and only if I must-not-follow J using the four conflict rules above.

4. If the graph is acyclic, and I_D is traversed first in a topological sort of the graph, then the extensions E_i are compatible with each other. Further,

    1. If the ultimate input state is compatible with the empty world, i.e., $S_i.Def = \emptyset$, then the extensions are compatible with the base document D, and may be loaded in the topologically-sorted order.
    2. Otherwise, the extensions are incompatible with the base document, and some required resources are missing, which are then reported to the user.

5. If the graph is acyclic, but I_D is not traversed first, then somehow an extension defines something that D relies upon. This cannot happen with the current overlay language (but see Section 5.7 where it may occur).

6. Else the graph is cyclic, and so report the extensions contributing to a cycle as conflicting.

This algorithm assumes that all semantic constraints have been encoded appropriately as guards on the relevant overlays. If any constraints are not so encoded, some topologically-sorted orderings may be "bad" while others are "good", though nothing in the conflict graph distinguishes them. In the limit, this may cause false negatives: the algorithm will claim the conflict graph is acyclic and hence some extensions are compatible, but when all semantic constraints are properly modeled, the conflict graph becomes cyclic and the extensions in fact are conflicting. The modeling will automatically handle several sorts of semantic constraints (i.e., the *Key*, *Id*, and *Selected* constraints seen before), but any other constraints are beyond the scope of this analysis,

and require user annotations. Note that the default behavior in Firefox, for example, is to include *no* guards, and implicitly permit *any* ordering among extensions.

5.5.5 Heuristics for determining optional composition order

The conflict-graph algorithm above takes guarded overlays as input. These have the simple property that they either successfully apply or fail. However, extension authors define *compositions*, not merely guarded overlays, and compositions may contain optional (or one-of-several) components that can "successfully apply" by silently doing nothing. As we've seen, optional and one-of-several components are useful for expressing higher-level structural design constraints on the guarded overlays within an extension. Supporting such constructions, however, complicates the definition of when two guarded overlays conflict: an optional component can *always succeed* by doing nothing!

Extension authors would not write an optional component and expect it always to do nothing; the intent is for it to apply whenever possible, but not to fail if it cannot do so. The goal must therefore be to compute a maximal set of optional components from source extensions that, when combined with the extensions' non-optional components and when treated as non-optional themselves, succeed in finding a compatible loading order. Unfortunately, this task is substantially harder than before: with n optional components, there are 2^n subsets to try, to see whether they can be loaded compatibly.

In fact, it is straightforward to prove that selecting such a subset is at least NP-hard. In particular, rephrase it as a decision problem:

Theorem 1. *Given a set of compositions $\{c_1, \ldots, c_n\}$ that may contain optional (?) or one-of-several (!) clauses, determining whether there exists a compatible loading order $c_{\pi(1)}, \ldots, c_{\pi(n)}$ (for some permutation π) is NP-hard.*

Proof sketch. It is reasonably straightforward encode a 3-CNF-SAT instance as a set of optional compositions and guarded overlays. One composition is non-optional, and encodes the 3-CNF-SAT formula to be solved. It will only successfully apply if some subset of the optional compositions can be loaded compatibly; this subset induces a solution to the original 3-CNF-SAT problem.

Since an exact solution is inefficient in the general case, any effective heuristic algorithm is appropriate instead. (Note that current data indicates that the average user installs five extensions, with non-negligible numbers of users installing ten or more [194]. Even for ten extensions, testing a worst case of over one thousand subsets is unreasonably slow.) The current approach uses a

greedy algorithm, starting with all optional components and removing one arbitrary, conflicted component at a time until a compatible loading order is found. If it removes all optional components and still has not found a valid loading order, it reports this as an error. (Note that this may be a false positive, as in the 3-CNF-SAT reduction above. But this is a pathological case not likely to occur in practice.)

5.6 Case study: Firefox extension conflicts

I applied the analysis defined in the previous section to a corpus of 350 Firefox extensions. As noted above, the analysis must support several idiosyncrasies of XUL and Firefox's implementation to model Firefox's behavior faithfully. Not handling these quirks lead to unacceptably many false positives in the analysis: too many extensions were claimed to conflict when in fact they were compatible. Note that there may be false negatives in the analysis (i.e., extensions are claimed to be compatible when in fact they conflict) only if it neglects to model some semantic constraint on resources. For example, the algorithm guarantees no false negatives with regard to *Key* and *Id* constraints. I first describe briefly the structure of a Firefox extension, then summarize the results, and finally explain two of these quirks in detail, as they illustrate the benefits of a cleanly-specified overlay language.

5.6.1 Firefox extension structure

A Firefox extension consists of

- An *install file* containing metadata about the extension, such as Firefox version compatibility
- A subdirectory containing the overlays, scripts, styles, and other resources used by the extension
- A *manifest* describing which overlays apply to which target Firefox files, with optional versioning information.

All extensions in my case study claimed compatibility with Firefox 3.0, so could all conceivably be installed simultaneously. Many extensions also claimed compatibility with other Mozilla products, and so included overlays that were not intended for Firefox. Interpreting the manifest correctly is therefore required for a precise model of Firefox extensions.

Firefox itself is packaged similarly, with its own manifests and overlays, and this too must be modeled precisely, or else the analysis may have incomplete information on what is truly defined by Firefox and available to extension.

5.6.2 Results

Detected conflicts: My analysis proceeds by processing each Firefox source file independently, examining all the overlays that are declared to target it. For example, of the 350 extensions I examined, 261 of them declare 331 overlays targeting `chrome://browser/content/browser.xul`, the main Firefox browser window. If any conflicts appear in this analysis, I report them. I analyze the target files independently because Firefox gives no guarantee that extensions are loaded atomically; instead, overlays are applied on demand, when the target files are parsed and displayed to the user.

Of the 350 extensions, with 5360 individual manifest entries, 1121 entries define overlay targets, of which I found 18 entries that purport to overlay some target file with a non-existent overlay, or that overlay a non-existing target file. Because extensions may target multiple Mozilla applications, I scanned the other applications' manifests (Thunderbird, SeaMonkey, Sunbird and Songbird) to eliminate false positives. Any remaining conflicts are true positives: even when accounting for other applications, these extensions still have missing targets. Note that by design, no individual Mozilla application can detect these conflicts (because no Mozilla application is bundled with information about the structure of all the other Mozilla applications).

Once mistaken manifest entries were accommodated, and other quirks accounted for, I found several problematic extensions:

- For `chrome://browser/content/browser.xul` (the main browser UI):
  - Sage and Sage Too, two versions of the same extension, conflict.
  - CaptureIt 1.0 and CaptureIt 2.5, two versions of the same extension, conflict.
  - ForecastFox 0.9.7.7 and ForecastFox_l10n 0.7, two versions of the same extension, conflict.
  - TabPopup and TabPreview conflict with each other over a tooltip.
  - All-in-one Sidebar defines two elements with the same id.
  - Locator overlaid a non-existent tagname: instead of the correctly-named ⟨**popup**/⟩ tag, it targeted ⟨**popupid** id="contentAreaContextMenu"/⟩.
  - MrTech Toolkit and the Nightly Tester Tools conflict over a toolbar button that enables nightly tester-tools functionality.

  - TabClicking Options overlays a resource that is not present in Firefox, though it used to be present in earlier versions of other Mozilla products.
  - Footiefox mistakenly overlaid a non-existent tag ⟨**popup** id="mainPopupSet"/⟩ instead of the correctly-named ⟨**popupset**/⟩ tag.
  - Firefox, MiniMap Sidebar and Toolbar Buttons participate in one pairwise conflict and one three-way cycle. Both Minimap Sidebar and Toolbar Buttons define an element with id `google`, placing them directly in conflict. Additionally, Toolbar Buttons defines a ⟨**menuitem**/⟩ with id `bookmarksShowAll` that Firefox itself defines, placing it in conflict with Firefox. Finally, Firefox must load before Minimap Sidebar, because the latter overlays content defined by Firefox. (Note that the redundant definition of `bookmarksShowAll` is not as easily handled by the "idiosyncrasies" described later, so I count this as a true conflict.)

- For `chrome://mozapps/content/extensions/extensions.xul` (the extension-manager window UI):
  - MrTech Toolkit and the Nightly Tester Tools conflict over nightly tester-tools functionality for enabling/disabling apps, and localized strings common to both.
  - DownloadSort and DownloadStatusbar conflict over creating a donations spacer and hotkey to remove the donations bar.

Four points immediately jump out from these results. First, the algorithm correctly detects the three pairs of differently-versioned extensions (Sage/Sage Too, CaptureIt 1.0 and 2.5, and ForecastFox 0.9.7.7 and ForecastFox_l10n 0.7) as conflicting with each other. This is a valuable sanity check that the algorithm is properly computing intersections.

Second, there are only twelve conflicting pairs of extensions, out of nearly 35,000 possible conflicting pairs. This may reveal a heavy selection bias in my sample: I examined the *top 350 most popular* extensions; it is quite possible that, had these extensions been conflicting or buggy, they would not have become the most popular. Unfortunately, it is not easy to obtain a fair sampling of the heavy tail of the infrequently-used extensions.

Third, the algorithm can detect simple typos that would otherwise be silently ignored by Firefox (as in the case of Footiefox), or targets that are no longer defined (as in TabClicking Options). These are ignored by design, because they *might* be correct for other Mozilla applications or other versions, but this design choice prevents extension authors from detecting simple but subtle bugs

in their code. To be sure, warnings about typos could likely be detected by a much simpler system; the analysis presented here incorporates them while providing additional benefit.

Finally, and most importantly, the algorithm must model the effects of previous extensions, because extensions do extend each other. This is crucial in both directions: There exist true conflicts due to cycles of length greater than two (as in the case of Firefox, MiniMap Sidebar and Toolbar Buttons) that would not be detected by any pairwise analysis. Likewise the algorithm must correctly ignore true negatives: if asked to analyze a set of extensions including both Firebug and FireCookie, no warning should be raised. But if Firebug were missing, the algorithm must properly warn that FireCookie has missing dependencies.

The potential for false positives is highlighted by a trio of non-conflicting extensions. For `chrome://firebug/content/firebugOverlay.xul` (the main Firebug panel UI), both YSlow and FireCookie extend Firebug's main toolbar with an additional item. However, due to a quirk of Firefox's overlay loader (described further in Appendix B, particularly Appendix B.2), the precise syntax they use for their overlays makes it appear (to an analysis that strictly enforces `XUL` overlay semantics) that both YSlow and FireCookie actually define a node (with id `"fbToolbarInner"`) that Firebug itself defines, when in fact Firefox treats that node as an overlay target. Further, because of how Firebug factored its code, it would appear (to that same strict analysis) that `"fbToolbarInner"` is not even defined! Without a sufficiently precise accounting of all three extensions, and a sufficient encoding of Firefox's quirks, the analysis might detect two distinct problems, neither of which in fact exist.

Performance: The stress test of the algorithm runs on all 264 overlays applicable to the main Firefox UI file; this is by far the largest target document and an order of magnitude more overlays than the typical user will have installed. The analysis completed in under four minutes, with 1.2GB of peak memory usage. While these numbers are too large to be practical for end-users casually installing extensions, several facts and simple optimizations make this much more plausible:

- The implementation has not been highly optimized yet; in particular, many expensive set-intersection tests can be memoized or optimized away.

- The algorithm is inherently cubic: for all pairs of extensions, it computes a potential edge in the conflict graph by examining each overlay's interface. A 10-fold reduction in the number of installed extensions (from 264 to 26) will result in roughly a 1000-fold speedup, bringing the runtime down to a few seconds and a few dozen megabytes of peak memory usage.

- The results of the compatibility analysis need not be repeated unless the set of installed extensions changes; they can be cached easily and compactly the rest of the time. Therefore, very few times the browser is launched will result in an expensive compatibility check.

- It is highly likely that most users download their extensions from `addons.mozilla.org`. The compatibility check can be done on demand by Mozilla, and the results cached for all users, amortizing the cost to practically nothing.

- A four-minute analysis is well within the realm of plausibility for *extension developers* to use regularly as a tool to ensure their work is compatible with many common extensions.

In short, the performance of the analysis is sufficiently decent for a prototype, and several obvious optimizations should improve it further.

5.6.3 Handling XUL idiosyncrasies

The intuitive understanding of XUL overlays is easily representable in the overlay language. A concrete XUL overlay

⟨**overlay**⟩
⟨**tag1** id="id1" *attrs*⟩
content1
⟨/**tag1**⟩
more actions
⟨/**overlay**⟩

might be modeled by the abstract overlay

Overlay(*Insert*(`tag1#id1`, end, *content1*), *Modify*(`tag1#id1`, attrs), *more actions*)

Unfortunately, the actual implementation of XUL overlays does not match this simple intuition. To ensure that the analysis detects as many feasible targets as possible, but no false positives, I manually pre-processed the extensions' XUL overlays to remove some behaviors that are challenging to model. First, I eliminated "self-overlays" (where nodes in an overlay apply to other nodes in that same overlay) and "recursive overlays" (that apply to other overlays rather than a base document), two emergent properties of overlays as they happen to be loaded by Firefox that are not implied by the documentation for overlays [166]. The first causes many false-positive reports among every extension that defines a keybinding, while the second causes false positives claiming that targets are missing. Second, because the implementation uses CSS selectors instead of

simple (*tagName*, *id*) pairs (in anticipation of the generalized analysis in Section 5.7), an additional pre-processing step is needed for all nodes lacking an id to eliminate false positives.

Self-overlays:

Firefox permits defining *self-overlays*, which must be handled to avoid false positives. One example is found in Update Notifier (`DownloadManagers/33-update_notifier`). In `XUL`, ⟨**key**/⟩ elements define hotkeys similar to `HTML 5`'s ⟨**command**/⟩ elements, and must be grouped within ⟨**keyset**/⟩ elements. Additionally, ⟨**key**/⟩ elements are permitted to *omit* the "key" attribute; in this case the ⟨**key**/⟩ represents a "catch-all" that responds to *any* keypress. Therefore, the intent of the following overlay (with seemingly invalid `XUL`):

⟨**keyset** id="mainKeyset"⟩
  ⟨**key** id="stylish-open-manage"/⟩
⟨/**keyset**⟩
⟨**key** id="stylish-open-manage" *more attrs*... /⟩

is to produce

⟨**keyset** id="mainKeyset"⟩
  ⟨**key** id="stylish-open-manage" *more attrs*... /⟩
⟨/**keyset**⟩

as opposed to two separate key handlers, one of which is a catch-all. Were the analysis to ignore this self-overlaying behavior, it would report the catch-all ⟨**key**/⟩ as conflicting with every other ⟨**key**/⟩ defined by every other extension, which clearly does not match Firefox's observed behavior.

If the analysis follows the "intuitive" translation given above, it actually produces a seemingly correct abstract overlay:

Overlay(*Insert*(`keyset#mainKeyset`, end, ⟨**key** id="stylish-open-manage"/⟩),
  Modify(`keyset#mainKeyset`), *Insert*(`key#stylish-open-manage`, end),
  Modify(`key#stylish-open-manage`, *more attrs*))

However, this overlay produces an incorrect interface, according to the rules for abstracting overlays to their interfaces. The interface requires `key#stylish-open-manage` to be defined (for the latter two actions to succeed), but itself defines `key#stylish-open-manage` (in the first action).

One attempt to resolve this modeling mistake might try representing `XUL` overlays as *sequential compositions* of overlays:

Overlay(*Insert*(`keyset#mainKeyset`, end, ⟨**key** id="stylish-open-manage"/⟩));

Overlay(*Modify*(`keyset#mainKeyset`));

Overlay(*Insert*(`key#stylish-open-manage`, end));

Overlay(*Modify*(`key#stylish-open-manage`, *more attrs*))

However, this does not actually match Firefox's behavior: the order of elements within a XUL overlay is irrelevant, even when there are dependencies among them. (From inspection of the Firefox source code, overlay merging appears to be a fixpoint-style computation, where all overlay actions are placed in a pool, and applied in any order they can be until no further actions apply. No ordering guarantees are given; the contents of a single ⟨**overlay**/⟩ might not even be applied consecutively.) Therefore translating a XUL overlay to a sequential composition may introduce the *wrong sequencing*. Instead, a better approach to modeling this self-overlay process is to compute the following

for all child $c = \langle$***tag*** id="*id*"/$\rangle$ of the root ⟨**overlay**/⟩ tag **do**
  for all descendants d of the ⟨**overlay**/⟩ tag where $d \neq c$, $d.tagName = tag$ and $d.id = id$ **do**
    Append a deep clone of c's children to d, and copy all of c's content attributes to d.
    Remove c from the ⟨**overlay**/⟩.
  end for
end for

This effectively "inlines" any self-overlays, and transforms the original example into its net effect, which normalizes the overlay for use with the subsequent algorithms.

Recursive overlay weaving

Firefox appears to support "recursive" overlay semantics, where nodes deep in the overlay are also considered as overlay actions rather than node definitions. While the intended behavior is to use only the *children* of the ⟨**overlay**/⟩ as actions:

for all children c of the root ⟨**overlay**/⟩ tag **do**
  Let $es \leftarrow \{e \mid e \text{ in the base document}, e.tagName = c.tagName, e.id = c.id\}$
  Assert $|es| = 1$, and let e be the unique member of es.
  Append a clone of the children of c to e.
end for

the actual behavior appears to involve descendants, looking for *their* ids in the base document:

for all children c of the root ⟨**overlay**/⟩ tag **do**
  Let $es \leftarrow \{e \mid e \text{ in the base document}, e.tagName = c.tagName, e.id = c.id\}$
  Assert $|es| = 1$, and let e be the unique member of es.

for all children d of c **do**
if a child f of e exists where $f.tagName = d.tagName$ and $f.id = d.id$ **then**
"Merge" a clone of d with f
else
Append a clone of d to e
end if
end for
end for

It is unclear from examining the Firefox code whether the "merge" step is in fact a full, deeply-recursive overlay operation or whether it is a single, shallow merge. (In either case, the observed behavior does not match the documentation.)

This behavior makes the detection of conflicts prone to false positives. For example, the Speed-Dial extension (`Appearance/06-speeddial`) contains the following excerpt, which deals with new popup menu items:

⟨**overlay**⟩
⟨**popupset** id="mainPopupSet"⟩
⟨**popup** id="contentAreaContextMenu"⟩
...actual overlay code...
⟨/**popup**⟩
⟨**popup** id="speedDialButtonMenu"...⟩
...new code...
⟨/**popup**⟩
⟨/**popupset**⟩
⟨/**overlay**⟩

In the base Firefox document, the ⟨**popupset** id="mainPopupSet"/⟩ element already contains a ⟨**popup** id="contentAreaContextMenu"/⟩ child, and the intent is clearly to overlay that popup with additional menu items. However, the following ⟨**popup** id="speedDialButtonMenu"/⟩ is a *new* popup menu, and is intended to overlay the ⟨**popupset** id="mainPopupSet"/⟩. The desired outcome ought to determine that ⟨**popup** id="speedDialButtonMenu"/⟩ does not conflict with anything else, so an abstraction of this overlay should ensure `popup#speedDialButtonMenu` is undefined. But syntactically there is no way to distinguish this from `popup#contentAreaContextMenu`, which in fact *should* exist and so should have different behavior. The "correct" overlays to achieve these two behaviors should in fact be

⟨**overlay**⟩

⟨**popup** id="contentAreaContextMenu"⟩
...actual overlay code...
⟨/**popup**⟩
⟨**popupset** id="mainPopupSet"⟩
⟨**popup** id="speedDialButtonMenu"...⟩
...new code...
⟨/**popup**⟩
⟨/**popupset**⟩
⟨/**overlay**⟩

where now it is always the case that children of the node-to-be-overlaid must not previously exist. Note that repairing this error will introduce new constraints into the dependency graph (the analysis now models each child element separately with its own constraint, rather than producing just one constraint for the erroneous parent element), but these are desired constraints which previously had been masked.

In my dataset, I see this pattern repeatedly. Twenty-seven extensions made this mistake 40 times, detectable merely by looking at the overlays themselves in the absence of a base document. If I include the base document in the analysis, another 21 mistakes appear.

CookieSafe (`Webdev/38-cookiesafe`) is slightly different, including two overlays

⟨**overlay**⟩
⟨**window** id="main-window"⟩
⟨**popupset** id="mainPopupSet"/⟩
⟨/**window**⟩
⟨**window** id="messengerWindow"⟩
⟨**popupset** id="mainPopupSet"/⟩
⟨/**window**⟩
⟨/**overlay**⟩

designed to ensure that the popupset *exists* before proceeding to overlay that popupset in the next piece of the overlay. The two versions are intended to apply equally well to either Firefox or SeaMonkey. This code is better served by writing it as an explicit one-of-many composition, rather than with an apparently required and redundant overlay. NoScript (`WebDev/01-noscript`) includes the same redundant overlay, with an explicit comment indicating it is used for SeaMonkey compatibility.

It is difficult to assign blame when multiple extensions have the same false positive-inducing error. But to estimate the severity of the problem, note that removing the last instance of the

$$
\begin{array}{rrcl}
c \in & \textsc{Comp} & ::= & g \mid c\,;c \mid c\,!\,c \mid c? \\
g \in & \textsc{Guard} & ::= & o \mid \mathit{Require}(\vec{r}, g) \mid \mathit{Reject}(\vec{r}, g) \mid \mathit{First}(\vec{r}, g) \mid \mathit{Last}(\vec{r}, g) \\
o \in & \textsc{Overlay} & ::= & \mathit{Overlay}(\vec{a}) \\
a \in & \textsc{Action} & ::= & \mathit{Insert}(s, w, \vec{h}) \\
 & & \mid & \mathit{Modify}(s, \vec{t}) \\
r \in & \textsc{Resource} & = & \mathit{Selector} \uplus \mathit{Id} \uplus \mathit{Key} \uplus \mathit{Selected} \uplus \cdots \\
s \in & \textsc{Selector} & ::= & \mathit{SimpleSelector} \mid s \mathbin{␣} s \mid s \sim s \\
w \in & \textsc{Where} & ::= & \text{start} \mid \text{end} \\
h \in & \textsc{HTML} & & \\
t \in & \textsc{Attribs} & ::= & (\texttt{name}, \texttt{value})
\end{array}
$$

Figure 5.13: Extending the overlay language in Fig. 5.12 with CSS simple selectors and some (but not all) combinators

⟨**popupset** id="mainPopupSet"/⟩ error removed 34 reported extension conflicts, properly handling self-overlaid nodes removed 70 reported conflicts, and the total number of conflicts dropped from 304 to 12 — precisely the twelve conflicts reported earlier.

Elements without IDs

Firefox overlays only match based on element names and identifiers. If an element has no identifier, then it cannot participate in the overlay process, and should not contribute to any conflicts. For example, the presence of a simple ⟨**menuitem**/⟩ with no identifier should not conflict with any contributed ⟨**menuitem** id="someId"/⟩ that does have an identifier. The naïve approach to translating nodes into selectors, however, would render the former as `somePath > menuitem` and the latter as `somePath > menuitem#someId`, and these two selectors might indeed refer to the same element.

To handle this, the translation of the base documents must assign a nonce identifier to all nodes that lack them, so that they cannot be confused with any other nodes. This step is automatic and straightforward, and eliminates many hundreds of false positives. (Again, the precise number is difficult to measure, because these errors compound with the false positives in the previous section to yield even more problems.)

5.7 *Overlay conflict detection: Generalizing selectors*

This section presents the first variant of the overlay language, shown in Fig. 5.13, that goes beyond the expressive power of Firefox extensions. The additional expressiveness of the selector language

lets overlays apply to *multiple targets*, and is motivated by two examples, one simplified from real-world extension idioms and one that highlights why such flexibility is distinctly different from the versions of this language examined above.

The ability for selectors to match multiple nodes introduces a new and difficult problem: nodes can match several different selectors, rather than just the sole selector determined by the tagname and id of the node. Said another way, it is possible for two distinct selectors to *intersect*, such that there exist nodes matching them both. Consequently, it is possible for two overlays with different selectors to in fact apply to the same targets, and hence potentially conflict with one another. Therefore, after motivating the benefits of these expressive selectors, I define an *intersection algorithm* that computes a description of precisely which nodes may match two selectors simultaneously.

With this new tool, the conflict-graph analysis defined above can be adapted to provide *partial* support for these new selectors. The use of the intersection algorithm complicates the maintenance of the *Undef* sets of the analysis; additionally, the `CSS` universal selector must be handled differently than others. I explain the adaptation of the analysis, and then explain these two caveats. Fully adapting the analysis to remove these problems remains as future work.

5.7.1 Motivating examples

Refactoring a single extension: Consider an extension (simplified from real examples) that adds a submenu of actions to the "Tools" menu of some application, and adds an identical submenu to the application's context menu. One way to write the extension might be to duplicate the submenu's contents in two overlays (written here using `XUL`):

- **OV7***: *Overlay*(*Insert*(`menu#tools-menu`, end, ⟨**submenu** id="aMenu"⟩⟨**menuitem**⟩Hi⟨/**menuitem**⟩⟨/**submenu**⟩))
- **OV8***: *Overlay*(*Insert*(`menu#context-menu`, end, ⟨**submenu** id="aMenu"⟩⟨**menuitem**⟩Hi⟨/**menuitem**⟩⟨/**submenu**⟩))

However, such copy-and-paste duplication easily leads to divergences between the two versions. An alternate approach might simply be to permit extensions to specify a set of selectors:

Overlay(*Insert*({`menu#tools-menu`, `menu#context-menu`}, ...))

Such an overlay would apply to all nodes matching any element in that set. However, this is a fairly limited improvement, as the extension author must enumerate all targets explicitly, which may not always be possible, as the next example will show. A better approach (and, indeed, one taken by some Firefox extensions), is to refactor the common code (in this case, the ⟨**menuitem**/⟩) into a *third* overlay:

- **OV7****: *Overlay*(*Insert*(`menu#tools-menu`, end, ⟨**submenu** id="myMenu"/⟩))
- **OV8****: *Overlay*(*Insert*(`menu#context-menu`, end, ⟨**submenu** id="myMenu"/⟩))
- **OV9***: *Overlay*(*Insert*(`submenu#myMenu`, end, ⟨**menuitem**⟩Hi⟨/**menuitem**⟩))

Here, overlays OV7** and OV8** create *stubs* that can be filled in by OV9*. As written, however, these overlays violate the well-formedness property of the combined document, since two ⟨**submenu**/⟩ items with the same id will be created. The best approach, then, is to use a property other than ids for OV9*:

- **OV7**: *Overlay*(*Insert*(`menu#tools-menu`, end, ⟨**submenu** class="myMenu"/⟩))
- **OV8**: *Overlay*(*Insert*(`menu#context-menu`, end, ⟨**submenu** class="myMenu"/⟩))
- **OV9**: *Overlay*(*Insert*(`submenu.myMenu`, end, ⟨**menuitem**⟩Hello⟨/**menuitem**⟩))

Such an approach requires more flexible selectors than we've permitted so far. This seemingly-small improvement—after all, this is still merely a simple selector—provides a large expressive jump, as such selectors can match *multiple targets*.

Supporting relationships between nodes: As mentioned in Section 5.3, the development of `C3` reached a stage where it could render the correct positioning and layout of unordered lists, but could not support the list-item bullets (more generally known as "generated content"). That generated content could be *simulated* by an overlay

Overlay(*Insert*(`ul␣li`, start, ⟨**span**⟩•⟨/**span**⟩))

that inserts a ⟨**span**/⟩ containing the bullet character before every list item—an *a priori* unbounded number of targets, and therefore not explicitly enumerable—within an unordered list. This overlay uses a non-simple selector (thanks to the descendant combinator), and so marks a large expressive jump from the languages in the previous sections.

A similar, though more contrived example might choose to render lists as set notation, by surrounding them with braces and inserting commas between list items. Such an overlay might be written using

Overlay(*Insert*(`ul`, start, ⟨**span**⟩{⟨/**span**⟩),
Insert(`ul li ~ li`, before, ⟨**span**⟩, ⟨/**span**⟩),
Insert(`ul`, end, ⟨**span**⟩}⟨/**span**⟩))

The first and last actions insert the list braces, while the middle action inserts commas only before those list items with at least one preceding sibling, i.e., every list item except the first.[5] Again, such an overlay could not be written with the simpler selector languages considered earlier.

(In practice, the author of this list-formatting overlay might want to apply the overlay whenever lists are present, but gracefully do nothing when they are not. In other words, this overlay should be optional. Such concerns are orthogonal to the effect an overlay has on the document. Recall that the composition language includes the ? combinator for exactly this purpose, and that extension authors actually provide compositions of guarded overlays, and not merely bare overlays. The remainder of this discussion focuses on the overlays and elides any composition operators.)

Note that both of these examples are broken in the presence of nested lists: they will insert bullets or commas before list items within *numbered* lists that are themselves nested within an unordered list. Correctly handling this case would need to use *children* selectors instead of descendant selectors; however, these will cause problems that I defer addressing until Section 5.8.

5.7.2 CSS selector intersection

As noted above, nodes can be matched by several distinct selectors. The dual *CSS intersection problem* asks, for a given pair of selectors, whether there could exist a tree with elements matching both selectors simultaneously. Such language intersection problems are common (cf. regular expression intersections [9, 10, 34, 206], XQuery and XPath intersection [21, 22, 38], etc.), and for arbitrary (context-free or larger) languages these problems are undecidable [103, 173]. Fortunately, CSS is at heart a regular language, and so the intersection problem is decidable. The CSS intersection problem, in particular, is of more than just academic interest; web developers encounter this problem in practice, and though partial answers have been posed for specific cases,[6] to my knowledge no systematic, complete, or provably-correct solution has been previously published.

For a conflict analysis that uses this intersection algorithm to be of diagnostic use, it needs more than just a Boolean response: it should *demonstrate* to the user or developer what the intersection

---

[5] The HTML 5-expert reader will notice that these three insertions produce a technically malformed ⟨**ul**/⟩, as the only permissible children for lists are ⟨**li**/⟩ elements. However, since scripts can already create malformed content at runtime, I do not view this as a cause for concern. The CSS 3-expert reader will notice that these three insertions can be mimicked by `:before` and `:after` generated content. This approach has the advantage of not cluttering the DOM of the page, but the inserted content is not selectable and does not behave as regular text.

[6] `http://stackoverflow.com/questions/4764712/grouping-css-selectors`

contains, to aid in debugging or eliminating unwanted conflicts. Ideally, it needs a function $Sel \times Sel \rightarrow Sel$ that takes a pair of CSS selectors and returns a selector that matches the intersection of the two inputs. However in general, two CSS selectors do not intersect in a single selector, but rather in a set of them. For example, the selectors `b␣a` and `c␣a` match any `a` node that has both `b` and `c` ancestors, in any order. So both `b␣c␣a` and `c␣b␣a` are in their intersection, but no single selector can describe both of those paths. Instead, it must expect the intersection algorithm to return a *set* of selectors.

Let $Inter : Sel \times Sel \rightarrow 2^{Sel}$ be a function such that for all $s, t \in Sel$ and for all $T \in Tree$, *Inter* is

$$\begin{aligned}
\text{Correct} &: \forall sel \in Inter(s,t), \forall elem \in T, sel_T(elem) \rightarrow s_T(elem) \wedge t_T(elem) \\
\text{Total} &: \forall elem \in T, s_T(elem) \wedge t_T(elem) \rightarrow \exists sel \in Inter(s,t), sel_T(elem) \\
\text{Sufficient} &: (\exists elem \in T, s_T(elem) \wedge t_T(elem)) \rightarrow Inter(s,t) \neq \emptyset
\end{aligned}$$

These conditions ensure that *Inter* produces selectors that only match elements matched by both s and t (Correct), that every element matched by both s and t is matched by some selector from *Inter* (Total), and that *Inter* will be non-empty as long as *some* element is matched by s and t (Sufficient). Obviously, Total implies Sufficient, but not vice versa. While the conflict analyses in this chapter only require the intersection algorithm to be sufficient, the correctness proofs for that algorithm will be easier if it is total as well.

Fig. 5.14 defines *Inter*, the CSS selector intersection algorithm. It requires that its inputs be in precedence-associated form, which by Lemma 1 (cf. Section 5.2.2) is semantically valid. In a slight abuse of notation, it lifts CSS combinators pointwise from selectors to sets of selectors, so that e.g., $s \oplus Inter(a,b) = \{s \oplus x \mid x \in Inter(a,b)\}$ and $Inter(a,b) \oplus Inter(c,d) = \{x \oplus y \mid x \in Inter(a,b), y \in Inter(c,d)\}$, for $\oplus$ a CSS combinator. It relies on three helper routines:

- *Canonical* takes two *SimpleSelector*s and returns either the empty set (when both selectors cannot match the same node, e.g., if the tag names differ), or a singleton set (with one selector matching all the properties of both inputs). The definition is straightforward and omitted.

- *Interleavings* takes either two descendant selectors or two sibling selectors, and returns all possible ways to interleave and combine their components to produce selectors in the intersection of the two original inputs. This function is defined in Fig. 5.15.

- *Pairings* computes a similar intersection, for either a pair of descendant and child selectors or a pair of sibling and adjacent sibling selectors. The essential difference between this and

the previous routine is that children and adjacent siblings are *tight* constraints, so fewer interleavings are present in the intersection. This function is also defined in Fig. 5.15.

The details of *Interleavings* can be understood fairly intuitively by looking at Fig. 5.16. The simple case is shown in Figs. 5.16a to 5.16d. Here, the base tree contains all interleaved orderings of `a` and `b` with `x` and `y`, ending in a node `d`. All the `d` nodes match both `a␣b␣d` and `x␣y␣d`. However, no single selector can describe all six orderings of `a`, `b`, `x` and `y`. Instead *Interleavings* must enumerate all orderings as separate selectors. Examining the six "strands" of these trees in order shows a recursive structure: all interleavings of `a␣b` with `x␣y` either

- begin with `a` and continue with all interleavings of `b` and `x␣y`, or
- begin with `x` and continue with all interleavings of `y` and `a␣b`.

This leads to the first half of the definition for *Interleavings*.

The remaining clause of the definition is illustrated by Figs. 5.16e to 5.16h. In these trees, the left two "strands" are similar to the previous tree (indeed, the other four orders could be present as well; they are elided for space). The right three strands, however, are structurally different: unlike in the preceding example, here it is possible for a single node (e.g., `x.b`) to match clauses from *both* input selectors (here, `*.b` and `x`), and in fact it is sometimes necessary. Thus in the definition of *Interleavings*, the final clause picks a single clause from each input selector and intersects them. This splits both selectors into two pieces, which must in turn be interleaved. As the choice of which clauses to choose was arbitrary, the definition in fact unions all such choices together.

The details of *Pairings* are similar: here, the central observation is that a child *is* a descendant (or an adjacent sibling *is* a sibling), but not vice versa. Therefore, given selectors $d_1 \sqcup d_2 \sqcup \cdots \sqcup d_M$ $c_1 > c_2 > \cdots > c_N$, the algorithm may pair off and intersect clauses $d_M, d_{M-1}, \ldots$ with $c_{i_1}, c_{i_2}, \ldots$ for $N \geq i_1 > i_2 > \cdots \geq 1$, being careful to preserve both the adjacency of the c_i clauses as well as the relative order of the d_j clauses. This accounts for the first clauses of *PairOff*; the remainder simply prepend the remaining d_j clauses before c_1.

Theorem 2. *Inter is Correct and Total.*

Proof sketch. The crux of the proof relies on how the different `CSS` combinators affect the path of a composite selector. Intuitively, (~) and (+) describe *horizontal* segments of the path, by defining some requirements on the leftward siblings of the target node. Therefore intersecting selectors using only these combinators will produce results that also use only these combinators.

$$
\begin{array}{lll}
Inter(simple_1, & simple_2 &) \stackrel{\text{def}}{=} Canonical(simple_1, simple_2) \\
\\
Inter(adj_1 + simple_1, & simple_2 &) \stackrel{\text{def}}{=} \{adj_1 + s \mid s \in Inter(simple_1, simple_2)\} \\
Inter(adj_1 + simple_1, & adj_2 + simple_2 &) \stackrel{\text{def}}{=} \{a + s \mid a \in Inter(adj_1, adj_2), s \in Inter(simple_1, simple_2)\} \\
\\
Inter(sib_1 \sim adj_1, & simple_2 &) \stackrel{\text{def}}{=} \{sib_1 \sim s \mid s \in Inter(adj_1, simple_2)\} \\
Inter(sib_1 \sim adj_1, & adj_2 + simple_2 &) \stackrel{\text{def}}{=} Pairings_{(\sim),(+)}(sib_1 \sim adj_1, adj_2 + simple_2) \\
Inter(sib_1 \sim adj_1, & sib_2 \sim adj_2 &) \stackrel{\text{def}}{=} Interleavings_{(\sim)}(sib_1 \sim adj_1, sib_2 \sim adj_2) \\
\\
Inter(kid_1 > sib_1, & simple_2 &) \stackrel{\text{def}}{=} \{kid_1 > s \mid s \in Inter(sib_1, simple_2)\} \\
Inter(kid_1 > sib_1, & adj_2 + simple_2 &) \stackrel{\text{def}}{=} \{kid_1 > adj_2 + s \mid s \in Inter(sib_1, simple_2)\} \\
Inter(kid_1 > sib_1, & sib_2 \sim adj_2 &) \stackrel{\text{def}}{=} \{kid_1 > sib_2 \sim a \mid a \in Inter(sib_1, adj_2)\} \\
Inter(kid_1 > sib_1, & kid_2 > sib_2 &) \stackrel{\text{def}}{=} \{k > s \mid k \in Inter(kid_1, kid_2), s \in Inter(sib_1, sib_2)\} \\
\\
Inter(desc_1 \, \text{␣} \, kid_1, & simple_2 &) \stackrel{\text{def}}{=} \{desc_1 \, \text{␣} \, k \mid k \in Inter(kid_1, simple_2)\} \\
Inter(desc_1 \, \text{␣} \, kid_1, & adj_2 + simple_2 &) \stackrel{\text{def}}{=} \{desc_1 \, \text{␣} \, adj_2 + k \mid k \in Inter(kid_1, simple_2)\} \\
Inter(desc_1 \, \text{␣} \, kid_1, & sib_2 \sim adj_2 &) \stackrel{\text{def}}{=} \{desc_1 \, \text{␣} \, sib_2 \sim k \mid k \in Inter(kid_1, adj_2)\} \\
Inter(desc_1 \, \text{␣} \, kid_1, & kid_2 > sib_2 &) \stackrel{\text{def}}{=} Pairings_{(\text{␣}),(>)}(desc_1 \, \text{␣} \, kid_1, kid_2 > sib_2) \\
Inter(desc_1 \, \text{␣} \, kid_1, & desc_2 \, \text{␣} \, kid_2 &) \stackrel{\text{def}}{=} Interleavings_{(\text{␣})}(desc_1 \, \text{␣} \, kid_1, desc_2 \, \text{␣} \, kid_2) \\
\\
Inter(x, & y &) \stackrel{\text{def}}{=} Inter(y, x) \text{ for all other combinations}
\end{array}
$$

Figure 5.14: CSS selector intersection algorithm

$$
\begin{aligned}
\mathit{Interleavings}_{\oplus}(s_1 \oplus \cdots \oplus s_N, t_1 \oplus \cdots \oplus t_M) &\stackrel{\text{def}}{=} s_1 \oplus \mathit{Interleavings}_{\oplus}(s_2 \oplus \cdots \oplus s_N, t_1 \oplus \cdots \oplus t_M) \\
&\quad \cup\ t_1 \oplus \mathit{Interleavings}_{\oplus}(s_1 \oplus \cdots \oplus s_N, t_2 \oplus \cdots \oplus t_M) \\
&\quad \cup\ \{x \oplus y \mid \\
&\qquad 1 \le i < N, 1 \le j < M, \\
&\qquad x \in \mathit{Interleavings}_{\oplus}(s_1 \oplus \cdots \oplus s_i, t_1 \oplus \cdots \oplus t_j), \\
&\qquad y \in \mathit{Interleavings}_{\oplus}(s_{i+1} \oplus \cdots \oplus s_N, t_{j+1} \oplus \cdots \oplus t_M)\} \\
\mathit{Interleavings}_{\oplus}(s_1 \oplus \cdots \oplus s_N, t) &\stackrel{\text{def}}{=} \mathit{Inter}(s_1 \oplus \cdots \oplus s_N, t) \\
\mathit{Interleavings}_{\oplus}(s, t_1 \oplus \cdots \oplus t_M) &\stackrel{\text{def}}{=} \mathit{Inter}(s, t_1 \oplus \cdots \oplus t_M)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Pairings}_{\oplus,\otimes}(s_1 \oplus \cdots \oplus s_N, t_1 \otimes \cdots \otimes t_M) &\stackrel{\text{def}}{=} (\mathit{PairOff}_{\oplus,\otimes}(s_1 \oplus \cdots \oplus s_{N-1}, t_1 \otimes \cdots \otimes t_{M-1}) \\
&\qquad \otimes \mathit{Inter}(s_N, t_M)) \\
\mathit{PairOff}_{\oplus,\otimes}(s_1 \oplus \cdots \oplus s_N, t_1 \otimes \cdots \otimes t_M) &\stackrel{\text{def}}{=} \mathit{PairOff}_{\oplus,\otimes}(s_1 \oplus \cdots \oplus s_N, t_1 \otimes \cdots \otimes t_{M-1}) \otimes t_M \\
&\quad \cup\ (\mathit{PairOff}_{\oplus,\otimes}(s_1 \oplus \cdots \oplus s_{N-1}, t_1 \otimes \cdots \otimes t_{M-1}) \\
&\qquad \otimes \mathit{Inter}(s_N, t_M)) \\
\mathit{PairOff}_{\oplus,\otimes}(s_1 \oplus \cdots \oplus s_N, t) &\stackrel{\text{def}}{=} \{s_1 \oplus \cdots \oplus s_N \oplus t\} \\
&\quad \cup\ s_1 \oplus \cdots \oplus s_{N-1} \oplus \mathit{Inter}(s_N, t)
\end{aligned}
$$

Figure 5.15: *Interleavings* combines either two descendant or two sibling selectors, and *Pairings* combines either descendants with children or adjacent siblings with siblings.

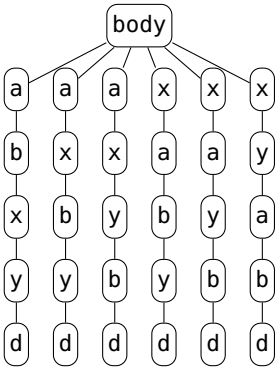
Similarly, $(\llcorner)$ and $(>)$ describe *vertical* segments of the path, by defining some requirements on the ancestors of the target node. Therefore intersecting selectors using only these combinators will again produce results that also use only these combinators.

The subtlety in the overall algorithm lies in the interaction of these two types of combinators—but these interactions can be tightly controlled by Lemma 1, which permits the re-association of the input selectors. The input selectors now consist of "vertically-combined" sets of "horizontal-only" selectors, for which the intersections can be computed independently.

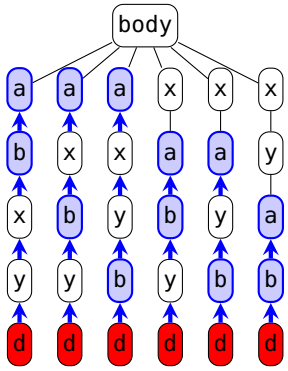
5.7.3 Runtime analysis

The performance of *Inter* on arbitrary selectors can be measured in three different ways: how *large* are each of the computed results, how *many* are there in total, and how *fast* can it compute whether the result set is empty or not.

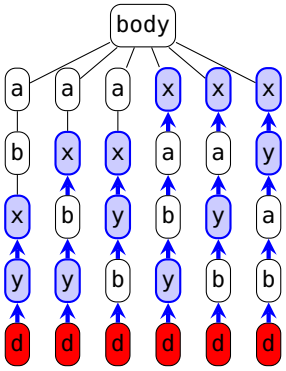
Theorem 3. *Let $s, t \in$ Selector be arbitrary, precedence-associated selectors. Then*



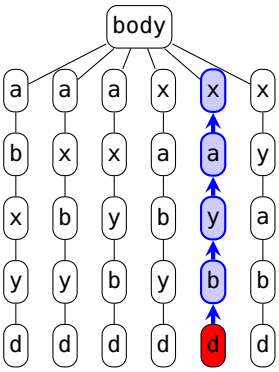
(a) Base HTML tree, showing all interleavings of a and b with x and y



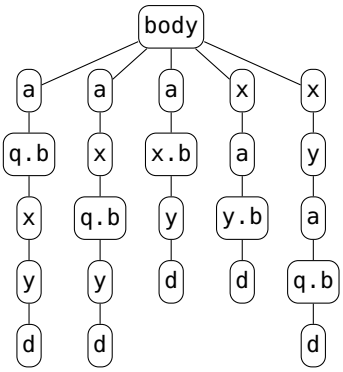
(b) Matching a ␣ b ␣ d: all d nodes match, regardless of intervening x and y nodes



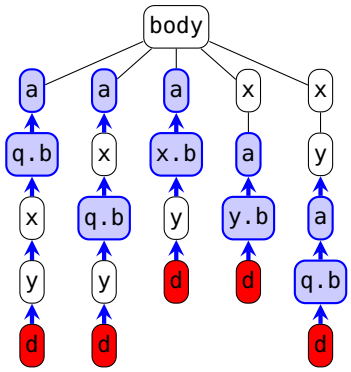
(c) Matching x ␣ y ␣ d: all d nodes match, regardless of intervening a and b nodes



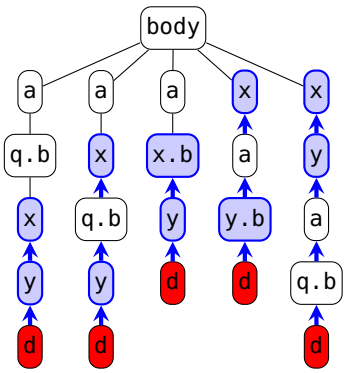
(d) Matching x ␣ a ␣ y ␣ b ␣ d: any one selector will match only one ordering



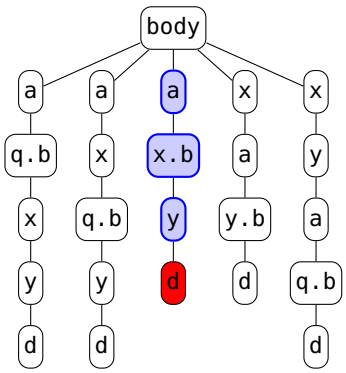
(e) Base HTML tree, to illustrate intersection of nodes along paths



(f) Matching a ␣ ∗.b ␣ d



(g) Matching x ␣ y ␣ d



(h) a ␣ x.b ␣ y ␣ d ∈ *Inter*(a ␣ ∗.b ␣ d, x ␣ y ␣ d): This case involves merging x with ∗.b

Figure 5.16: Demonstrating the interleaving algorithm

- *Length:* $\forall x \in \mathit{Inter}(s,t), |x| \leq |s| + |t|$.
- *Count:* $|\mathit{Inter}(s,t)| \in O(3^{|s|+|t|})$.
- *Speed: Deciding whether* $\mathit{Inter}(s,t) = \emptyset$ *can be computed in time* $O(|s| + |t|)$.

Proof sketch. Intuitively, the *Length* claim simply shows that if two selectors denote paths p and q that examine m and n nodes respectively, *Inter* shouldn't have to examine more than $m + n$ nodes to match both selectors.

The *Speed* claim asserts that computing *just one* element of $\mathit{Inter}(s,t)$ can be done quickly: *Inter* defers to *Pairings* and *Interleavings*, and at any point in their definitions where it "chooses" some splitting of the input, it can always choose one such split easily.

The only challenging claim is of *Count*. Here, it is easy to see that $\mathit{Inter}(s,t) \in \Omega(2^{|s|+|t|})$: for two sibling selectors of lengths m and n, the number of interleavings is the number of ways to choose n positions among $n + m$, which is $\binom{n+m}{m} \leq 2^m$. A more detailed accounting proves that *Inter* exceeds this bound, and the true bound has a base of 3 rather than 2.

5.7.4 *Using descendant and sibling selectors*

The ideas from Section 5.5 are still useful in analyzing the compatibility of the overlays in this section, particularly the machinery of the conflict graph from Section 5.5.4. The essential change is relaxing the intuition of "sets of resources, compared by equality" to accommodate intersections between selectors, as defined above.

In more detail, consider a base document consisting solely of the empty node ⟨**a**/⟩, and the following three overlays

- **OV14**: *Overlay*(*Insert*(`a`, end, ⟨**b** class="x y"/⟩), *Insert*(`a`, end, ⟨**c** class="y z" id="idC"/⟩), *Reject*(*Selector*(`d.foo`)))
- **OV15**: *Overlay*(*Insert*(`b.x`, ...))
- **OV16**: *Overlay*(*Insert*(`*.foo`, ...))

OV14's interface should provide as much detail as possible; in this case it will include[7]

[7] Because selectors are restricted to descendant and sibling only, they do not capture the child and adjacent-sibling relationships between the nodes here.

$$S_i^{14}.Def = \{Selector(\texttt{a})\} \qquad S_i^{14}.Undef = \{Id(\texttt{idC}), Selector(\texttt{d.foo})\}$$

$$S_o^{14}.Def = \{Selector(\texttt{a}), \qquad S_o^{14}.Undef = \{Selector(\texttt{d.foo})\}$$
$$Selector(\texttt{a␣b.x.y}),$$
$$Selector(\texttt{a␣b.x.y~c.y.z\#idC}),$$
$$Id(idC)\}$$

This interface is the *best* interface that can be specified, as it defines a maximally-specific selector for every node in the document. Many other selectors are possible: some might elide `b.x.y` from the last *Selector*, and assert `c` as a child of `a` directly; others might elide `a` altogether and simply assert that three simple selectors exist with no structural relationships between them; still others might elide class information, or id information, or both. Indeed, there are far too many variants to list explicitly (particularly for larger overlays or documents), but it is also *unnecessary* to do so. Every one of these variants provides less specific information about the document than the selectors shown above.

Turning to OV15, note that it tries to overlay a node matching `b.x`. Suppose the algorithm examines if OV15 can follow OV14. It simply asks whether there exists $d \in S_o^{14}.Def$ such that $Inter(d, \texttt{b.x}) \neq \emptyset$. Sure enough, $Inter(\texttt{a␣b.x.y}, \texttt{b.x}) = \{\texttt{a␣b.x.y}\} \neq \emptyset$, so it knows such a node exists in the document—despite not explicitly including $Selector(\texttt{b.x}) \in S_o^{14}.Def$.

The key observation is that when checking sequencing of two overlays (Eqs. (5.3) to (5.6)), the algorithm cannot directly use set operations that check implicitly for pairwise element *equality*. Instead, they must use *Inter* to check explicitly for pairwise element *intersection*: for two overlays OV1 and OV2, and their interfaces S^1 and S^2,

$$\forall d_2 \in S_o^2.Def, \forall u_1 \in S_i^1.Undef, Inter(d_2, u_1) = \emptyset \tag{5.7}$$

$$\forall d_2 \in S_o^2.Def, \forall d_1 \in \text{defs}(\text{OV1}), Inter(d_2, d_1) = \emptyset \tag{5.8}$$

$$\forall f_2 \in S_o^2.Frozen, \forall u_1 \in \text{used}(\text{OV1}), Inter(f_2, u_1) = \emptyset \tag{5.9}$$

$$\forall r_2 \in \text{reqs}(\text{OV2}), \forall c_1 \in S_i^1.Clean, Inter(r_2, c_1) = \emptyset \tag{5.10}$$

These equations strictly generalize Eqs. (5.3) to (5.6) (because $s_1 = s_2 \implies Inter(s_1, s_2) \neq \emptyset$, taking the contrapositive shows that Eq. (5.7) implies Eq. (5.3), and likewise for the other equations).

However, they introduce the potential for *aliasing*: CSS selectors now may produce unexpectedly non-empty intersections.

Caveat: Universal selectors: For example, overlay OV16 applies to nodes matching `*.foo`. Its interface includes

$$S_i^{16}.Def \supseteq \{Selector(\texttt{*.foo})\}$$
$$S_o^{16}.Def \supseteq \{Selector(\texttt{*.foo})\}$$

(The input *Def* set must contain the selector being overlaid, and the output *Def* set should contain at least as much as was previously defined.) Suppose the analysis tried to compose OV14;OV16. Then Eq. (5.7) requires that it check $S_i^{14}.Undef$ against $S_o^{16}.Def$. By examination, $Selector(\texttt{d.foo}) \in S_o^{14}.Undef$, and $Inter(\texttt{d.foo}, \texttt{*.foo}) = \{\texttt{d.foo}\}$, which implies that Eq. (5.7) fails for OV14;OV16. But this clearly is wrong: OV16 might apply to *any* element with class `foo`, not just `d` elements. So as long as *some* element existed with tagname not equal to `d` and with class `foo`, OV14 would be satisfied and subsequently so would OV16. The problem arises because using $*$ in both S_i and S_o of an interface is misleading. In S_i it means "match all possible tag names"; in S_o it *should* mean "there is some tag name that was matched". However, using it directly in calls to *Inter* implicitly uses the former meaning only. This all/any dichotomy is essentially DeMorgan's law applied to quantifiers. Properly handling these two distinct meanings of $*$ would require universal and existential selectors to the language, and is an intriguing avenue for future work. For now, I ignore such issues and accept the consequent loss of precision.

Caveat: *Undef* sets: Aliasing arises at another point of the analysis, where it determines what belongs in the *output Undef* set for a sequence of overlays. Recall that for a composition OV1;OV2, the set $S_o.Undef$ is computed from $S_o^1.Undef$ and $S_o^2.Undef$. Intuitively, the output *Undef* set should contain everything that is still known to be undefined by $S_o^2.Undef$, and anything still known to be undefined by $S_o^1.Undef$ *except* for anything defined by OV2. In standard set notation, this would be $S_o^2.Undef \cup (S_o^1.Undef \setminus \text{defs}(\text{OV2}))$. But suppose the two overlays OV1 and OV2 were such that $S_o^1.Undef = \{Selector(\texttt{a.foo}), Selector(\texttt{b.foo.bar})\}$, $S_o^2.Undef = \emptyset$, and OV2 defines $Selector(\texttt{a\#id})$ and $Selector(\texttt{b.foo.bar.baz})$. How should the intuitive definition for $S_o.Undef$ be interpreted? Using standard set operations, where elements are compared by equality, is inconsistent with the choices above to use the selector-intersection algorithm. But if instead the intersection algorithm is used, note that

$$Inter(\texttt{a.foo}, \texttt{a\#id}) = \{\texttt{a.foo\#id}\} \notin S_o^1.Undef$$

and yet

$$\mathit{Inter}(\texttt{b.foo.bar}, \texttt{b.foo.bar.baz}) = \{\texttt{b.foo.bar.baz}\} \notin S_o^1.\mathit{Undef}$$

For a elements, a proper analysis *should* preserve a.foo, because OV2 does not define an element matching a.foo, despite the intersection algorithm claiming it is possible. For b elements, a proper analysis *should not* preserve b.foo.bar, because OV2 defines an element matching that, despite the intersection algorithm producing a more-specific result.

In the current implementation, I accept the inconsistency, and *use standard set operations*, with pairwise element equality, to define the output *Undef* set for sequences of overlays. Specifically, in the set-difference operation $(S_o^1.\mathit{Undef} \setminus \mathrm{defs}(\mathrm{OV2}))$, I discard only those items in $S_o^1.\mathit{Undef}$ that are exactly equal to elements in $\mathrm{defs}(\mathrm{OV2})$. In practice this works well; it remains for future work to provide a cleaner accounting for why this is the appropriate choice.

5.8 Overlay conflict detection: Fully-general overlays

This section considers, at last, the full language originally presented in Fig. 5.8. Again I present motivating examples to show what additional expressiveness gains come from permitting arbitrary *Replace* actions in overlays, and from broadening the selector language to full CSS selectors. Both of these expressiveness gains cause the conflict-graph algorithm to break down, so I sketch a potential, new analysis based on the theory of *text-based patches* in version control systems, which might successfully model and analyze these last new components of the overlay language.

5.8.1 Motivating examples:

***Replace* actions:** All previous sections of this chapter considered only insertions of new content and modification of node attributes. A fully-general system would need to consider removing content as well, or more conveniently, *replacing* existing content with new content. (Removal then simply replaces existing content with nothing.)

Removing existing content occurs frequently. A simple ad-blocker, for instance, might define

Overlay(*Replace*(*[href~="some.adserver.com"]))

to remove scripts, images, objects, or anything else served from some.adserver.com. Similarly, FlashBlock might be implemented simply by

Overlay(*Replace*(object[src$=".swf"]), *Replace*(embed[src$=".swf"]))

Restructuring existing content might also be useful. For example, visual editors such as PowerPoint or Photoshop frequently add "handles" to the corners and edges of objects so they can be

resized and rotated. These handles are hidden until an object is selected. Emulating this behavior via an overlay is straightforward: to add handles to all images, for example, an overlay might use

Overlay(*Replace*(`img`,
⟨**span** class="hidden editable"⟩
⟨**span** class="inactive handle NW"/⟩
⟨**span** class="inactive handle NE"/⟩
⟨**self**/⟩
⟨**span** class="inactive handle SW"/⟩
⟨**span** class="inactive handle SE"/⟩
⟨/**span**⟩))

This uses the ⟨**self**/⟩ tag to re-parent the ⟨**img**/⟩ within the ⟨**span** class="hidden editable"/⟩. With appropriate styling rules, the handles can be positioned appropriately at the corners of the ⟨**span**/⟩ (which derives its size from the ⟨**img**/⟩ inside it), and hidden until selected. Similarly, the handles can be inactive until individually selected.

(Note that this is not a foolproof approach: style rules that applied to the ⟨**img**/⟩ may no longer apply, since its parent element has changed. An extension using this approach might need to manually manage styles to preserve the original appearance. Alternatively, a far more complicated overlay mechanism might make the inserted outer ⟨**span**/⟩ "invisible" to the `CSS` rule-matching process; this is similar in spirit to `XBL` [218]. Doing so, however, is beyond the scope of this work.)

The primary problem with *Replace* actions, however, is that they are challenging to describe modularly using the interfaces of the conflict graph-based algorithm. Consider a base document

⟨**html**⟩
⟨**body**⟩
⟨**p** class="greeting para"⟩Hello ⟨**span** id="s"⟩World⟨/**span**⟩⟨/**p**⟩
⟨/**body**⟩
⟨/**html**⟩

The state-pair for this document might summarize it by saying that its *Def* set is

$$Def = \{Selector(\texttt{html}), Selector(\texttt{html > body}), Selector(\texttt{html > body > p.greeting.para}),$$
$$Selector(\texttt{html > body > p.greeting.para > span\#s})\}$$

Suppose an overlay tried to replace the paragraph with something else:

Overlay(*Replace*(`p.para`, ⟨**a** href="#"⟩A link⟨/**a**⟩))

How should an analysis define the effect this overlay has on the document? Clearly, it requires

Selector(`p.para`) to be defined in the document, and places no restrictions on what is undefined, clean, or frozen. But after it applies, that paragraph is no longer in the document, so the analysis must somehow indicate that it is now undefined. This immediately poses a problem: the resource *Selector*(`p.para`) *is not explicitly part of the document's Def set.* One resolution might consider removing every selector in *Def* that intersects `p.para`, but in general the aliasing issues inherent in `CSS` make this a bad choice. Additionally, a full solution must somehow know to remove all references to `span#s` as well, because it was removed along with its parent. Finally, while these questions are solvable for this particular case, because the *Def* set includes as much information about the structure of the document as is possible (i.e., every node in the paths from each node to the root is included in each selector), in general when the *Def* set includes "local" information obtained from other overlays (i.e., the path information may not be complete), there may not be any good solution.

More broadly, the inclusion of `Replace` in the overlay language permits changes to the document that, in terms of resources, are *no longer monotonic*. Because the conflict graph-based dependency-resolution algorithm assumed that the *Def* and *Frozen* sets grew monotonically, while the *Undef* and *Clean* sets shrank monotonically, violating these assumptions breaks the algorithm.

Child and adjacent-sibling selectors: As seen earlier, using only descendant or sibling selectors leads to a clumsy loss of precision, and limits the utility of overlays because they cannot define their targets sufficiently precisely. However, supporting child and adjacent-sibling selectors causes the conflict-detection algorithm to break, and potentially claim to produce a valid extension loading order that actually causes one extension to fail.

Suppose the base document is a simple tree

⟨**a** id="a1"⟩
  ⟨**b**/⟩
  ⟨**c**/⟩
⟨/**a**⟩

and three overlays are defined:

- **OV11:** *Overlay*(*Insert*(`a#a1 > c`, after, ⟨**d** id="d1"/⟩))
- **OV12:** *Overlay*(*Insert*(`a#a1 > c`, after, ⟨**e** id="e1"/⟩))
- **OV13:** *Overlay*(*Insert*(`c + d`, after, ⟨**f** id="f1"/⟩))

Examining all six possible loading orders, note that the three orders where OV13 precedes OV11 will fail due to lack of a ⟨**d**/⟩ element. Indeed, after constructing the conflict graph for these three extensions, the only dependency edge is that OV11 must not follow OV13. This implies that the order (OV11;OV12;OV13) is a valid loading order, as it is a valid topological ordering of

the conflict graph. But stepping through the application of each overlay reveals a problem: after OV11 applies, the merged document is

⟨**a** id="a1"⟩
  ⟨**b**/⟩
  ⟨**c**/⟩
  ⟨**d** id="d1"/⟩
⟨/**a**⟩

Both OV12 and OV13 could successfully apply to this document. However, once OV12 runs, the document becomes

⟨**a** id="a1"⟩
  ⟨**b**/⟩
  ⟨**c**/⟩
  ⟨**e** id="e1"/⟩
  ⟨**d** id="d1"/⟩
⟨/**a**⟩

OV13 can no longer apply in this document, because ⟨**c**/⟩ and ⟨**d**/⟩ are *no longer adjacent*. A similar problem holds when considering child selectors and *Replace* actions, which can analogously change the parent-child relationship between nodes. Note that without child or adjacent-sibling combinators, this problem vanishes, because nothing in the overlay language permits changing the sibling or ancestor-descendant relationships.

Again, the general overlay language permits changes to the document that, in terms of resources, are no longer monotonic. And just as with the analysis of *Replace* actions, this non-monotonicity can cause the conflict-detection algorithm to break down.

5.8.2 Approach: future work

Fundamentally, the implicit mistake in the example above was that while it correctly accounted for how the *document* changed as a consequence of applying overlays, it failed to account for how the *targets of overlays* should change as well. Recalling the analogy from the beginning of the chapter, overlays are "tree-shaped patches"—and experience with version control systems, which routinely handle textual patches of flat files, shows that describing the effects of one patch depends heavily on which other patches have been applied.

I take particular inspiration from the Darcs version control system [46, 115, 151], which elevates patches to the fundamental object of manipulation. In particular, Darcs defines a "theory of

patches", where patches are a formal algebra equipped with a commutation relation that describes how to commute patches past one another. Said another way, if two patches p and q produce some combined result $(p\,;q)$, their theory defines how to compute two related patches q' and p' such that q' "does the same thing as" q, and p' "does the same thing as" p, such that the composition $(q'\,;p')$ produces the same result as $(p\,;q)$. This commutation relation is essential for computing the *merge* of multiple patches onto a single base document.

The analogy between text-based patches and tree-based overlays is fairly tight, but there is one essential difference that makes the overlay case harder. Just as overlays consist of a set of actions that apply at certain selectors, patches consist of a set of *hunks* that replace text at certain *locations*. A location is a pair of character offsets $(start, end)$, describing what region of the text to excise. A hunk then consists of a location and a pair of strings (old, new). The intent is to replace the contents of the document between *start* and *end*—which must be *old*—with the new content *new*. Unlike general-purpose CSS selectors, which can match *multiple* nodes, locations match *exactly one* position in the document. But I have argued that the ability to target multiple nodes is a large gain in expressive power; I would like not to give up that flexibility if at all possible.

Because the theory of patches relies solely on an abstract type *Patch* that satisfies certain algebraic axioms, it may be possible to adapt Darcs patch theory for describing overlays. One essential requirement for successfully computing the commutation relation is that every patch must have an *inverse* that can undo the effects of a patch. For text-based patches $(start, end, old, new)$, the inverse patch must replace *new* with *old*, and must fix the start and end locations appropriately: $(start, start + |new|, new, old)$. For overlays, the current language does not quite suffice for defining an inverse for every operation. Certainly every *Insert* modification can be undone by a *Replace* operation that removes the new content. But modifying attributes or parentage is not so easy, as the *Replace* operation does not preserve enough information to describe the inverse.

With a suitable revision of the language to accommodate inverses, this patch-based approach would need to consider how overlays commute past each other. In particular, it must explore how to transform CSS selectors to include the effects of other overlays. Continuing the example above, suppose it was known that (OV11 ; OV13 ; OV12) was a valid sequence, and the analysis tried to commute OV13 past OV12. This would yield two new overlays OV12′ and OV13′, such that OV13′ injects the same content *given* $(OV11\,;OV12')$ as OV13 would have *given only OV11*. The analysis must therefore modify OV13's selector to account for OV12's actions, so it might change `c + d` into `c + * + d`, to account for the newly-inserted $\langle \mathbf{e}/\rangle$. But this does not quite preserve the meaning of OV13, because there might exist other $\langle \mathbf{d}/\rangle$ nodes in the document, one of which may yet be adjacent to a $\langle \mathbf{c}/\rangle$ node. Consider the following, slightly more complicated document:

⟨**a** id="a1"⟩
  ⟨**b**/⟩
  ⟨**c**/⟩
  ⟨**a** id="a2"⟩
    ⟨**c**/⟩
    ⟨**d** id="d2"/⟩
  ⟨/**a**⟩
⟨/**a**⟩

(Ignore for the moment that OV13 no longer depends on OV11, since the base document matches `c + d` even before being overlaid.) Supposing OV11 were applied to this document, the result is

⟨**a** id="a1"⟩
  ⟨**b**/⟩
  ⟨**c**/⟩
  ⟨**d** id="d1"/⟩
  ⟨**a** id="a2"⟩
    ⟨**c**/⟩
    ⟨**d** id="d2"/⟩
  ⟨/**a**⟩
⟨/**a**⟩

At this point, OV13 applies to `c + d`, that is, to *both* ⟨**d** id="d1"/⟩ and ⟨**d** id="d2"/⟩. If OV12 were then applied to this document, the resulting composite is

⟨**a** id="a1"⟩
  ⟨**b**/⟩
  ⟨**c**/⟩
  ⟨**e** i="e1"/⟩
  ⟨**d** id="d1"/⟩
  ⟨**a** id="a2"⟩
    ⟨**c**/⟩
    ⟨**d** id="d2"/⟩
  ⟨/**a**⟩
⟨/**a**⟩

If OV13's selector is left unchanged, it now will match *only* ⟨**d** id="d2"/⟩. But if OV13's selector is modified as described above, to `c + * + d`, it now will match only ⟨**d** id="d1"/⟩. Neither of these options preserves the meaning of OV13 when applied after OV11, which means neither of them

describe OV13′. Instead, the "correct" modification is a set of selectors that pick out *both* `c+d` and `c+*+d`. Computing that these are the correct selectors for OV13′ is nontrivial. Even more complicated is the reverse: how to determine that commuting OV12′ past OV13′ results in an OV13″ such that OV13″ = OV13?

It is unclear whether overlays will actually conform to the algebraic structure required by Darcs patch theory. However, *if* it does, then the theory will provide a solid foundation for understanding the behavior of overlays in a more principled fashion.

5.9 Runtime behavior of overlays

The analyses in the preceding sections tacitly assumed that pages were completely static: no dynamically generated content, no scripts running during page loading, etc. Even the Firefox case study assumed `chrome://browser/content/tabbrowser.xul` was static. However, real pages are not so well-behaved, and this introduces complications in defining precisely *when* extensions are applied to the base system. Both Firefox and Chrome contend with similar issues: Firefox makes no guarantees about precisely when extensions' overlays are applied, while Chrome permits extension authors to specify one of three moments when extension scripts are evaluated [92, `run_at` attribute].

In the case of overlays, such complications arise in defining when overlay weaving occurs. Ultimately the definitions of overlays must choose whether they are conceptually "part" of the page and visible to script, or not. The overlays shown here are patterned upon Mozilla's `XUL`, and so overlay-provided nodes are visible to script; by contrast, `XBL` [218] hides its dynamically-inserted content from script. Neither choice is inherently superior; however, my overlay design does introduce consistency challenges in relation to scripts: it becomes tricky to avoid situations where one script runs and sees the original page, while a later script runs and sees an overlaid version. There are three reasonable possibilities for when to apply overlays, ordered by time of occurrence:

1. Overlays might be applied at the instant the document is created. Clearly overlays cannot be applied at least until parsing is under way, as the document structure has not yet been created, so this choice requires coupling the weaving of overlays tightly into the parser.

2. Overlays might be applied as subtrees are inserted into the document; unfortunately, the `HTML` parsing algorithm makes this tricky, as some scenarios involve inserting subtrees in one place, only to move them again immediately (the so-called "adoption-agency" algorithm [112,

section 8.2.5.4.7]). It is technically difficult to distinguish the initial from final positions without close communication between the DOM and parser.

3. Finally, overlays might be applied all at once, when the parser finishes parsing. However, HTML permits inline scripts to run during page parsing, which means the desired consistency goal is violated: inline scripts will see the un-overlaid document structure, while later event-triggered scripts will see the overlaid structure.

These possibilities all have drawbacks: either they compromise the architectural goals for C3's modularity (the HTML parser should not be tightly coupled to the overlay mechanism, if at all possible), or violate consistency. However, the HTML specification provides a particular event, `readyStateChange`, that signifies when a document (or other resource) has "nothing [to execute] that delays the load event" [112, section 8.2.6]: all inline scripts have run, all critical resources are loaded, and the document is quiescent. Said another way, *prior* to this event being fired, the document is *expected* to be in an inconsistent state, and web developers understand this form of inconsistency. Moreover, I expect that well-written webapps will eschew complicated inline scripts in favor of external and event-triggered scripts: it is relatively poor practice to use them, as they prevent speculative parsing and parallelization of resource downloads, and so slow down the app's perceived performance. Therefore I adopt *the moment just prior to firing the* `readyStateChange` *event* to apply overlays.

Once the document has transitioned to the event-driven phase of its lifecycle, another opportunity arises to apply overlays. Consider highly-dynamic but fairly-regular documents: the messages in a auto-updating Twitter feed, or the tabs in a browser's tab bar. Extension authors reasonably might want to overlay that repetitive structure, but clearly the at-load-time moment defined above will not suffice. Therefore overlays can also be applied to dynamically-constructed nodes *the first time they are inserted into the document tree.* Clearly this is brittle in the face of bizarre DOM manipulations, so overlay authors are required to mark their overlays explicitly: only actions with a *dynamic* attribute will be applied at node-insertion time as well as at document-parsing time.

(It is worth comparing these three identified moments in the lifecycle of a page to those identified by Chrome: in fact, they are nearly identical. Because Chrome is dealing with scripts rather than overlays, they are examining milestones in the JS heap rather than the DOM tree: `document_start` is item 1, modified to accommodate other Chrome extension details (such as injecting stylesheets as well as scripts); `document_end` is essentially item 3 and occurs after parsing but before external resources may have completed loading; and `document_idle` occurs at some point

between then and just after the `onload` event. The differences between these and the three above are solely due to which properties scripts can observe. Because scripts do not trigger based on DOM subtree insertion, item 2 is irrelevant, and item 1 becomes feasible. The caveat about external resources in `document_end` is necessary because the state of such resources is observable by scripts.)

5.10 Summary

This chapter has examined the problem of extending HTML markup in a declarative and analyzable manner. Based on Firefox's overlay approach, I have developed a newer, more general and more systematic overlay language with clearly defined semantics. Those semantics were then used to define a series of conflict-detection algorithms that prevent real-world problems seen in a user study. The conflict-detection algorithms for the generalized forms of the overlay language required formalizing a notion of the intersection of two CSS selectors, and this led to identifying a new notion of aliasing in selectors and overlays. I identified a challenging failure mode when applying the algorithms in this chapter to the full overlay language: the ability to remove or rearrange existing content makes existing child or adjacent-sibling selectors no longer apply, and this non-monotonicity breaks the compositional approach presented here. Finally, I have identified some potentially promising avenues to address this in future work.

Chapter 6

CONCLUSION

We are in the middle of a transition as client-side applications and web browsing converge to a new and still imperfectly understood "webapps" model. The preceding chapters have elaborated several of the strengths of both approaches to engineering programs that should be preserved by future web platforms, focusing heavily on extensibility of the platform. This dissertation has begun to address several of the weaknesses of current extension mechanisms, but much more remains. The following section discusses several possible lines of future work. The remaining section summarizes and concludes this dissertation.

6.1 Future work

Each of the projects presented in this dissertation both solve existing problems and expose new ones for consideration. This section discusses future platform-level work, new analyses enabled by aspects for JS, and potential approaches for resolving overlay compatibility. It then looks at more holistic properties of extensions, including their security and cross-language conflicts.

6.1.1 Platform-level future work

As discussed in Chapter 3, C3 was designed for easy experimentation with major components of the web platform, as well as for extensive support for extensions atop the platform. That chapter presented seven systematic extension points at key points of the execution of HTML, CSS and JS, of which only five points have so far been implemented. The remaining two points, namely extending the CSS language with new properties and values, and extending the layout engine with new box types and layouts, remain for future work.

New CSS properties and values: Implementing support for new CSS properties and values is relatively straightforward, thanks to the "large-scale" regularity of the CSS grammar, where each style declaration is a semicolon-delimited list of colon-separated name/value pairs. The primary challenge is engineering-related, rather than conceptual: Currently the parser uses a fixed enumeration to encode property names, and uses direct function calls to property-specific parsing

functions once the property name is recognized. An extensible version would need to add a level of indirection, so extensions could add support for new property names and their values. This requires changing the encoding from enumerations to something else, as C# enumerations are not dynamically extensible—but this is not an insurmountable challenge.

Adding new property names and values via extensions, like adding new HTML tag names, requires that the platform enforce compatibility among the extensions. This amounts to a uniqueness check on each property name, which is readily handled by the same conflict-detection algorithm used for tag name uniqueness. Checking that new values (note: the concrete syntax for the value, not its semantics) do not conflict is slightly harder, as it is a check that two different parsers cannot both recognize the same string. Since CSS properties are essentially matched by regular expressions, and the intersection test for regular languages is decidable, this conflict check is different only in degree, not in kind.

New layout algorithms: New layout algorithms are harder to support as extensions, because the existing layout algorithms are very tightly-coupled. Some prior work has looked at constraint-based layout [28, 111, 156]; while these approaches do not currently support the full breadth of existing HTML and CSS rendering foibles, if such a milestone were achieved then extending it with new constraints would be comparatively straightforward. It remains to be seen whether constraint-based approaches can actually handle all the details of existing HTML layout, and if not, whether the unsupportable corner cases are of low-enough importance that they are not relevant.

Other extension mechanisms: While C3 admits seven extension points in its current configuration, certainly other more *ad hoc* extension points are possible. For example, it was mentioned that as currently implemented, overlays are not truly an extension to C3, but rather require one hard-coded hook in the page-loading sequence to trigger their weaving: If this hook were made explicit, overlays could be entirely decoupled from C3. There is no *a priori* reason to have this hook—and there are likely several such hooks waiting to be inserted.

The Gecko platform has exposed many hooks in its API, for functionality as diverse as customizing the script loader (used by extensions such as NoScript and Firebug), enumerating addons (used by MrTech Toolkit), adding new URL protocol handlers, or even hooking into the chrome-loading mechanism itself [171]. These hooks are interspersed with platform APIs like opening disk files, constructing special-purpose JS objects, or accessing platform preferences. It is a tedious task to disentangle the hooks from the APIs, and promote a clean split between the platform and

webapp levels, but there may be a commonality to the actual hooks which may inform another systematic extension mechanism.

6.1.2 *Aspects: Future work*

Chapter 4 explored the semantics of aspects and aspect weaving in JS, and indicated that using aspects obviates the need for most instances of the monkey-patch idiom. This in turn improves the precision of practically *any* analyses done over JS code, particularly pointer analyses [98], since nearly all non-trivial analyses must approximate the side-effects of `eval`. Additionally, by moving more code out of `eval`'ed strings and into directly-parsable source, more code is available in the direct control flow of the program for analysis.

Aspects additionally create the opportunity for a novel analysis, which may serve as the backbone for a conflict analysis for JS code: much as the analyses in Chapter 5 determined whether two overlays must be composed in some strict order, or whether they could *commute* and still produced the same side effects, extensions' aspect code could be analyzed for commutativity as well to determine whether their weaving order did not matter. Such an analysis relies on a pointer analysis to determine the potential joinpoints of each aspect, which in the design here are explicit JS expressions. Ideally it will be statically provable that most extensions do not advise the same functions or, if they do, that their advice commutes. Moreover, it is probable that precisely those extensions whose code does *not* commute are likely to have similar functionality, and hence truly do conflict.

It is possible for two extensions to advise disjoint sets of functions, yet still interact poorly through side effects on the shared state of the webapp. Handling this scenario requires an information flow analysis over the script of the program. Such analyses have been tried before [13, 39, 56, 97, 132], though they either require dynamic checks, rewriting the code, or sacrificing soundness. Nearly all have trouble with the problematic `eval` and `with` constructs of the language—but aspects largely eliminate `eval` as used by the extensions I have examined, and most extension code does not use `with` in any significant way. Further, recent efforts to formalize the semantics of JS attempt to desugar `with` into more primitive constructions, eliminating it as well [99]. In short, aspects may greatly improve the precision of existing analyses, and permit additional specialization where heuristics are still required.

6.1.3 *Overlays: Future work*

As described in Section 5.8, the current conflict-graph analysis fails on fully-general overlays due to both aliasing in CSS selectors and the confounding effects of non-monotonic operations like

removing or modifying existing content. That chapter sketched a potential avenue for future improvement, based upon *patch theory* [46, 115, 151]. The essential idea is to model overlays as mathematically invertible operations that therefore admit clean reasoning principles, rather than to duplicate and reason about the structure of Mozilla's overlays. More broadly, such work has implications on how to compare tree-shaped data efficiently, which is worth pursuing beyond the application to UI extension used here.

The description of overlays in Chapter 5 sits at an intermediate level of expressiveness between textual patches on the one hand, and XSLT (a declarative language for transforming input XML trees into modified XML or other output forms [219]) on the other. Overlays do not contain any explicit looping constructions, for example. It is unclear whether such power is actually necessary here, and if it is, how to model such behavior for purposes of conflict and commutativity analysis.

6.1.4 Security: Future work

Through most of this dissertation I deliberately ignored security issues pertaining to extensions, and assumed that the user trusted the extensions, which in turn were written by benevolent but perhaps inept programmers who wanted to write correct and compatible code. However, in the real web this may not be the case. There are three types of adversaries I consider here: individually malicious extensions, colluding extensions, and confused-deputy extensions.

Proof-of-concept malicious Firefox extensions have been written that hide a keylogger within the browser, scraping all form information and exfiltrating the data to a potentially malicious site [80]. With minimal extra effort, such extensions could make themselves all but invisible, hiding themselves from the list of installed extensions and having no UI. Only a determined search through the JS heap or through the list of installed chrome files would reveal them.

Colluding extensions may be even more challenging to detect, as no one extension of the group may be actively malicious. One extension may routinely "call home" with some packet of seemingly innocuous information. A second may overwrite one of the data sources with secret data, which the first will happily expose. The data flows between the two extensions may be arbitrarily convoluted, involving side channels through the DOM of the browser or web pages, timing channels, custom event sequences, or many others. Crucially, none of these data flows occur unless all the colluding extensions are present.

Finally, confused-deputy extensions may individually be benign, but leave some entrance point in their code sufficiently unguarded that they can be called from untrusted code. This leaves them open to confused-deputy attacks where malicious web sites or other extensions can extract

information from the victim extension or use it to execute some privileged action on their behalf. Here the detection challenge is not so much identifying malicious data flows as it is detecting whether all publicly visible functionality validates and sanitizes its inputs.

Addressing these types of extension requires a different mindset than the preceding conflict-detection algorithms, and an advance over prior approaches. Information flows must be tracked precisely through script *and* the DOM of the browser, client web sites, and all the extensions. Some heuristics can be used, perhaps assuming that the mainline browser functionality is immune to confused-deputy attacks, so that attention can be focused just on those functions that are modified by advice. Naturally, truly malicious extensions will not be so obliging as to use the more easily analyzable aspects presented here, opting instead for the analysis-defeating effects of `eval`, but assuming they do use aspects gives a limit study of the effectiveness of such security analyses in the browser—and later webapp—setting.

6.2 Conclusions

This dissertation has examined the role of *extensibility* in current web browsers, and argued that as web browsers morph into web *platforms*, the case for extensions is no less compelling. My thesis is that:

- A powerful extension mechanism for the web platform is justified and desirable: users overwhelmingly like having extensions for their browser, and as browsers and webapps merge ever closer, the ability to customize webapps is equally attractive.

- The existing implementations of extensions on current web platforms is insufficient: extensions are either too crippled in expressive power or too powerful to be stable, and can arbitrarily break the browser or each other. Moreover, these implementations are not suitable for research and experimentation, as they couple the browser with its extension mechanisms too tightly.

- Programming languages research is an appropriate tool to help. Improving the languages in which extensions are written will provide increased leverage in analyzing extensions for compatibility with each other and with the underlying platform.

To that end, in my thesis I have examined the forms extensibility has taken in several other domains, and positioned web-browser extensibility among them. I then presented three projects which address these claims. I first presented *C3*, a novel web platform architecture designed to

support extensibility at several levels, including coarse-grained and pervasive reconfigurability, and fine-grained extensibility of the HTML and JS components of the platform. Next I presented an *aspect-oriented extension for JS* that addresses the semantic failings of common extension idioms, and showed that this new primitive is both more efficient and more correct than existing extensions, and have claimed that as a building block of extensions it enables higher precision in analyzing the JS of extensions. Finally, I presented an *overlay mechanism for HTML*, generalizing ideas from Firefox's overlays, which provides a well defined semantics for how overlays apply, and which supports conflict-detection algorithms to ensure extensions are compatible with each other. Each of these contributions improve the stability and robustness of the extensible web platform, but many opportunities remain for future work and improvement.

Appendix A

PROOFS

Proof of Theorem 1. Given a set of compositions $\{c_1, \ldots, c_n\}$ that may contain optional (?) or one-of-several (!) clauses, determining whether there exists a compatible loading order $c_{\pi(1)}, \ldots, c_{\pi(n)}$ (for some permutation π) is NP-hard.

Proof. I show this by reduction from 3-CNF-SAT. Construct two sets of compositions that encode an arbitrary 3-CNF-SAT instance; a compatible loading order for the compositions will imply a satisfying assignment for the 3-CNF-SAT problem. The first set will use only optional components; the second will use only one-of-several components.

Consider a 3-CNF-SAT formula $\bigwedge_i C_i$, where $C_i = (x_a \vee x_b \vee x_c)$ is a standard clause, and x_a are literals (or their complements). First, construct a base document

$\langle$**formula** id="sat"$\rangle\langle$**clause** id="C_i"/$\rangle \cdots \langle$**clause** id="C_n"/$\rangle\langle$/**formula**$\rangle$

Second, for each literal (and complement) x_i, construct the composition

$$c_{x_i} = \Big(\mathit{Last}(\mathit{Selector}(\texttt{lit\#}C_1\overline{x_i}), \ldots, \mathit{Selector}(\texttt{lit\#}C_n\overline{x_i}), \\ \mathit{Overlay}(\mathit{Insert}(\mathit{Selector}(\texttt{clause\#}C_p), \langle\textbf{lit}\ \text{id="}C_p x_i\text{"}/\rangle)\ , \cdots))\Big)?\,; \\ \Big(\mathit{Require}(\mathit{Selector}(\texttt{lit\#}C_p x_i), \mathit{Overlay}(\mathit{Insert}(\mathit{Selector}(\texttt{clause\#}C_p, \langle\textbf{done}\ \text{id="d-}C_p\text{"}/\rangle)))))?\,; \\ \cdots\Big)$$

where the C_p range over the various clauses covered by x_i. (As is necessary, note that c_{x_i} is linear in the number of literals and the number of clauses, i.e., polynomial in the size of the input.) If the first component of c_{x_i} applies successfully to a document, it overlays all clauses where x_i is true, and (using *Last*) ensures that $c_{\overline{x_i}}$ cannot successfully apply. The remaining sequence of optional components indicates that all covered clauses C_p are satisfied by the variable x_i: the *Require* assertion enforces that x_i is true (by the first component of c_{x_i}), and defines $\langle$**done** id="d-C_p"/$\rangle$ to mark C_p as covered. The success of the first component is necessary for the second component to succeed, though not sufficient: multiple variables could satisfy the same clause C_p, but defining the $\langle$**done** id="d-C_p"/$\rangle$ tag multiple times is invalid. Therefore these parts of the composition are optional so that each can individually succeed once

and fail later times without failing the entire c_{x_i} composition. Last, construct the composition $c_{SAT} = \mathit{Require}(\mathit{Selector}(\texttt{done\#d-}C_1), \ldots, \mathit{Selector}(\texttt{done\#d-}C_n), \mathit{Overlay}())$, which is also linear in the number of clauses. Crucially, it is not optional, and it clearly depends on nodes defined by the optional overlays. The only way for it to successfully apply to a document is for some subset of the optional c_{x_i} overlays to successfully apply, such that the original formula is satisfied, and for the remainder of the optional overlays to fail to apply. Since this is equivalent to determining the variable assignment, I have reduced 3-CNF-SAT to the overlay ordering problem and thus it is NP-hard.

A similar reduction applies with one-of-several compositions. Again consider clauses C_i and literals x_i and complements $\overline{x_i}$. For each variable x_i, construct the composition

$$c_{x_i} = \mathit{Overlay}(\mathit{Insert}(\texttt{clause\#}C_p, \langle \textbf{lit id=“}C_p x_i\text{”}/\rangle), \ldots)\,!$$

$$\mathit{Overlay}(\mathit{Insert}(\texttt{clause\#}C_q, \langle \textbf{lit id=“}C_q \overline{x_i}\text{”}/\rangle), \cdots)$$

where C_p range over all clauses containing x_i, and C_q range over all clauses containing $\overline{x_i}$. The one-of-several operator ensures that I overlay either nodes corresponding to clauses containing x_i or nodes corresponding to clauses containing $\overline{x_i}$, but not both. Note that the total length of the c_{x_i} is linear in the length of the original 3-CNF-SAT formula.

Next, for each clause $C_p = (x_a \vee x_b \vee x_c)$, define the composition

$$c_{C_p} = \mathit{Require}((C_p x_a, \ldots), \mathit{Overlay}(\mathit{Insert}(\texttt{clause\#}C_p, \langle \textbf{done id=“d-}C_p\text{”}/\rangle)))\,!$$

$$\mathit{Require}((C_p x_b, \ldots), \mathit{Overlay}(\mathit{Insert}(\texttt{clause\#}C_p, \langle \textbf{done id=“d-}C_p\text{”}/\rangle)))\,!$$

$$\mathit{Require}((C_p x_c, \ldots), \mathit{Overlay}(\mathit{Insert}(\texttt{clause\#}C_p, \langle \textbf{done id=“d-}C_p\text{”}/\rangle)))$$

Now c_{C_p} can only succeed if at least one of c_{x_a}, c_{x_c} or c_{x_b} succeeded. Again, the total length of the c_{C_p} compositions is linear in the length of the original formula.

Finally, I again define $c_{SAT} = \mathit{Require}(\mathit{Selector}(\texttt{done\#d-}C_1), \ldots, \mathit{Selector}(\texttt{done\#d-}C_n), \mathit{Overlay}())$, which asserts that all of c_{C_p} succeed. Because c_{SAT} is not optional, the entire set of compositions will have a compatible loading order iff there is a satisfying assignment to the 3-CNF-SAT problem. Once again, the length of c_{SAT} is linear in the length of the original formula. So the size of the constructed set of compositions is polynomial in the size of the original problem, and the solutions are equivalent, so I have reduced 3-CNF-SAT to the composition ordering problem using only one-of-several operators. □

Proof of Lemma 1. Let S be a concrete-syntax CSS selector. Let sel_1 be the parsed representation of S according to the original grammar, and let sel_2 be the parsed representation of S according to the precedence-inducing grammar. Then $[\![sel_1]\!] = [\![sel_2]\!]$.

Proof. I prove this by strong induction on the number of simple selectors of sel_1 (or sel_2):

Case 1: $|sel_1| = |sel_2| = 1$: Both selectors are the same simple selector, so there is only one way to parse S according to both grammars and so $sel_1 = sel_2$, and the result follows.

Case 2: $|sel_1| = |sel_2| = 2$: $sel_1 = sel_2 = s \oplus t$, where $s, t \in \mathit{SimpleSelector}$, $\oplus$ is some combinator, and again there is only one way to parse S according to either grammar, so again $sel_1 = sel_2$ and the result follows.

Case 3: $|sel_1| = |sel_2| > 2$: Then $S = A \oplus_1 B \oplus_2 C$, for some selectors A, B and C, and combinators $\oplus_1$ and $\oplus_2$. I show the stronger result, that for any such A, B, C, $\oplus_1$ and $\oplus_2$,

$$[\![(A \oplus_1 B) \oplus_2 C]\!] = [\![A \oplus_1 (B \oplus_2 C)]\!]$$

By definition,

$$[\![A \oplus_1 B]\!] = \{q +\!\!+ p \mid p \in [\![A]\!], q \in [\![B]\!], f_{\oplus_1}(q_{last}, p_1)\}$$
$$\text{where } f_{\oplus_1} \text{ depends on } \oplus_1 \text{ by the definition of } [\![\cdot]\!]$$

and

$$[\![B \oplus_2 C]\!] = \{r +\!\!+ q \mid q \in [\![B]\!], r \in [\![C]\!], f_{\oplus_2}(r_{last}, q_1)\}$$
$$\text{where } f_{\oplus_2} \text{ depends on } \oplus_2 \text{ by the definition of } [\![\cdot]\!]$$

In particular, the $f_\oplus$ functions are:

$$f_{(+)}(q,p) \stackrel{\text{def}}{=} q_{last}.\mathit{prevSibling} = p_1 \qquad f_{(\sim)}(q,p) \stackrel{\text{def}}{=} q_{last}.\mathit{prevSibling}^+ = p_1$$
$$f_{(>)}(q,p) \stackrel{\text{def}}{=} q_{last}.\mathit{parent} = p_1 \qquad f_{(\_)}(q,p) \stackrel{\text{def}}{=} q_{last}.\mathit{parent}^+ = p_1$$

Therefore, by direct calculation,

$$\begin{aligned}
[\![(A \oplus_1 B) \oplus_2 C]\!] &= \{r +\!\!+ (q +\!\!+ p) \mid r \in [\![C]\!], (q +\!\!+ p) \in [\![A \oplus_1 B]\!], f_{\oplus_2}(r_{last}, (q +\!\!+ p)_1)\} \\
&= \{r +\!\!+ (q +\!\!+ p) \mid r \in [\![C]\!], q \in [\![B]\!], p \in [\![A]\!], f_{\oplus_1}(q_{last}, p_1), f_{\oplus_2}(r_{last}, (q +\!\!+ p)_1)\} \\
&= \{\ r +\!\!+ q +\!\!+ p \ \mid r \in [\![C]\!], q \in [\![B]\!], p \in [\![A]\!], f_{\oplus_1}(q_{last}, p_1), f_{\oplus_2}(r_{last}, q_1)\} \\
&= \{(r +\!\!+ q) +\!\!+ p \mid r \in [\![C]\!], q \in [\![B]\!], p \in [\![A]\!], f_{\oplus_1}((r +\!\!+ q)_{last}, p_1), f_{\oplus_2}(r_{last}, q_1)\} \\
&= \{(r +\!\!+ q) +\!\!+ p \mid (r +\!\!+ q) \in [\![B \oplus_2 C]\!], p \in [\![A]\!], f_{\oplus_1}((r +\!\!+ q)_{last}, p_1)\} \\
&= [\![A \oplus_1 (B \oplus_2 C)]\!]
\end{aligned}$$

where I use obvious properties about first and last elements of paths (namely, $(q +\!\!+ p)_1 = q_1$ and $(r +\!\!+ q)_{last} = q_{last}$), and the associativity of path concatenation. (The precise predicates defined by $f_{\oplus_1}$ and $f_{\oplus_2}$ are not important; the proof only manipulates their arguments and not their definitions.)

This result implies the original goal, that $[\![sel_1]\!] = [\![sel_2]\!]$, by strong induction. Given the precedence-associated parse sel_2 of S, I can transform it by repeated application of the result above:

> **Case 3.a:** $sel_2 = A \oplus B$ for some selectors A and B and combinator $\oplus$, and $|B| = 1$. Then since $|A| < |sel_2|$, by induction I can reassociate A to A' such that A' is left-associated and $[\![A]\!] = [\![A']\!]$. Then $[\![sel_2]\!] = [\![A' \oplus B]\!]$, and $A' \oplus B$ is left-associated, so $A' \oplus B = sel_1$
>
> **Case 3.b:** $sel_2 = A \oplus B$ for some selectors A and B, and combinator $\oplus$, and $|B| = n > 1$. Since $|A| < |sel_2|$, again I can reassociate A to A' such that A' is left-associated and $[\![A]\!] = [\![A']\!]$. Similarly, since $|B| < |sel_2|$, by induction I can reassociate B to B' such that B' is left-associated and $[\![B]\!] = [\![B']\!]$. I therefore have $[\![A \oplus B]\!] = [\![A' \oplus B']\!]$. By construction, I now know $B' = (\cdots (B_1 \oplus_1 B_2) \cdots \oplus_n B_n)$. By n applications of the result above, I obtain $[\![A' \oplus (\cdots (B_1 \oplus_1 B_2) \cdots \oplus_n B_n)]\!] = [\![(\cdots ((A \oplus B_1) \oplus_1 B_2) \cdots \oplus_n B_n)]\!]$, which is left-associated, and therefore equal to $[\![sel_1]\!]$ as desired. □

Proof of Theorem 2. *Inter* is Correct and Total.

Proof. Assume without proof that *Canonical* is Correct and Total for simple selectors. Correctness and totality are proven by induction on $\max(|s|, |t|)$, by cases that introduce one combinator at a time. Assume that the inputs are in precedence-associated form, and assume that all outputs are implicitly reassociated to that form as well.

$$
\begin{aligned}
\mathit{Interleavings}_{\oplus}(s_1 \oplus \cdots \oplus s_N, t_1 \oplus \cdots \oplus t_M) &\stackrel{\text{def}}{=} s_1 \oplus \mathit{Interleavings}_{\oplus}(s_2 \oplus \cdots \oplus s_N, t_1 \oplus \cdots \oplus t_M) \\
&\quad \cup\ t_1 \oplus \mathit{Interleavings}_{\oplus}(s_1 \oplus \cdots \oplus s_N, t_2 \oplus \cdots \oplus t_M) \\
&\quad \cup\ \{x \oplus y \mid \\
&\qquad 1 \le i < N, 1 \le j < M, \\
&\qquad x \in \mathit{Interleavings}_{\oplus}(s_1 \oplus \cdots \oplus s_i, t_1 \oplus \cdots \oplus t_j), \\
&\qquad y \in \mathit{Interleave}_{\oplus}(s_{i+1} \oplus \cdots \oplus s_N, t_{j+1} \oplus \cdots \oplus t_N)\} \\
\mathit{Interleavings}_{\oplus}(s_1 \oplus \cdots \oplus s_N, t) &\stackrel{\text{def}}{=} \mathit{Inter}(s_1 \oplus \cdots \oplus s_N, t) \\
\mathit{Interleavings}_{\oplus}(s, t_1 \oplus \cdots \oplus t_M) &\stackrel{\text{def}}{=} \mathit{Inter}(s, t_1 \oplus \cdots \oplus t_M)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Interleave}_{\oplus}(s_1 \oplus \cdots \oplus s_N, t_1 \oplus \cdots \oplus t_M) &\stackrel{\text{def}}{=} s_1 \oplus \mathit{Interleave}_{\oplus}(s_2 \oplus \cdots \oplus s_N, t_1 \oplus \cdots \oplus t_M) \\
&\quad \cup\ t_1 \oplus \mathit{Interleave}_{\oplus}(s_1 \oplus \cdots \oplus s_N, t_2 \oplus \cdots \oplus t_M) \\
\mathit{Interleave}_{\oplus}(s_1 \oplus \cdots \oplus s_N, t) &\stackrel{\text{def}}{=} \mathit{Inter}(s_1 \oplus \cdots \oplus s_N, t) \\
\mathit{Interleave}_{\oplus}(s, t_1 \oplus \cdots \oplus t_M) &\stackrel{\text{def}}{=} \mathit{Inter}(s, t_1 \oplus \cdots \oplus t_M)
\end{aligned}
$$

Figure A.1: An optimized form of *Interleavings* that computes all elements in the result set with fewer redundancies

Case 1: $|s| < |t|$**:** The definitions of Correct and Total are symmetric in s and t. Every equation (except the last) in Fig. 5.14 recursively calls *Inter* on some smaller portion of its inputs, and assumes asymmetrically that $|s| \ge |t|$. The last equation prevents getting stuck when $|s| < |t|$, and so I get that if *Inter* is correct and total when $|s| \ge |t|$, then it is too when $|s| < |t|$. For the remaining cases of the proof, assume that $|s| \ge |t|$ and so no equations get stuck.

Case 2: s **and** t **contain no combinators:** Then by the first equation in Fig. 5.14,

$$\mathit{Inter}(s, t) = \mathit{Canonical}(s, t)$$

and by assumption I am done.

Case 3: s **and** t **contain only** (**+**)**:** Examine the next two equations in Fig. 5.14. Let $s = s_1 + \cdots + s_N$, where s_i are simple selectors.

Case 3.a: Any elements satisfying both s and t, when t is a simple selector, must satisfy both s and t, and their adjacent left siblings must satisfy $s_1 + \cdots + s_{N-1}$. By construction, s_N and t

are both simple selectors (so $Inter(s_N, t)$ is correct and total by Case 2 above). Therefore

$$s_1 + \cdots + s_{N-1} + Inter(s_N, t)$$

describes exactly those nodes satisfying both s and t, and so is correct and total as well.

Case 3.b: Similarly, elements satisfying both s and $t = t_1 + \cdots + t_M$ (where t_i are simple selectors) must satisfy both s_N and t_M, and so must satisfy some element of $Inter(s_N, t_M)$. Additionally, their adjacent left siblings must satisfy $s_1 + \cdots + s_{N-1}$ and $t_1 + \cdots + t_{M-1}$. By induction, I know that $Inter(s_1 + \cdots + s_{N-1}, t_1 + \cdots + t_{M-1})$ is correct and total, and by construction and Case 2 above $Inter(s_N, t_M)$ is correct and total. Therefore

$$Inter(s_1 + \cdots + s_{N-1}, t_1 + \cdots + t_{M-1}) + Inter(s_N, t_M)$$

describes exactly those nodes satisfying both s and t, and so is correct and total as well.

Case 4: s and t contain only $(+)$ or $(\sim)$: Examine the next three equations in Fig. 5.14. Let $s = s_1 \sim \cdots \sim s_N$, where s_i contain only $(+)$.

Case 4.a: The case where t is a simple selector is trivial, similar to Case 3.a.

Case 4.b: Elements satisfying both s and $t = t_1 + \cdots + t_M$ (where t_i are simple selectors) must satisfy both s_N and t_M, and so must satisfy some element of $Inter(s_N, t_M)$. Additionally, they must have *adjacent* siblings satisfying $t_1 + \cdots + t_{N-1}$ and *some* siblings satisfying $s_1 \sim \cdots \sim s_{N-1}$. Each of the t_j (counting down from M) may match a sibling that is also matched by one of the s_i (counting *sequentially* down from N), or the t_j may match siblings distinct from those matched by s_i. There are no other possibilities.

(For example, suppose $N = 4$ and $M = 3$. I know s_4 and t_3 must intersect (because they are the last items in their respective selectors). The following are then valid pairings:

$$\begin{aligned}
s_1 \sim s_2 \sim s_3 \sim t_1 \quad &+ t_2 \quad &+ Inter(s_4, t_3) \\
s_1 \sim s_2 \sim Inter(t_1, s_3) \quad &+ t_2 \quad &+ Inter(s_4, t_3) \\
s_1 \sim s_2 \sim t_1 \quad &+ Inter(s_3, t_2) \quad &+ Inter(s_4, t_3) \\
s_1 \sim Inter(s_2, t_1) \quad &+ Inter(s_3, t_2) \quad &+ Inter(s_4, t_3)
\end{aligned}$$

as they all preserve the relative orderings and adjacencies of s_i and t_j, but the following is not:

$$s_1 \sim \mathit{Inter}(s_2, t_1) + s_3 + t_2 + \mathit{Inter}(s_4, t_3)$$

because t_1 is no longer adjacent to t_2.)

Examining the definition of *Pairings*, note that it directly computes $\mathit{Inter}(s_N, t_M)$ as required, and then calls *PairOff* with the remainder of s and t. Suppose $|t| > 1$, then examining the first equation for *PairOff*, note that it computes results both where t_M intersects with s_N and where it is left alone, and ensures that t_N remains adjacent to t_{N-1}. When $|t| = 1$, the second equation for *PairOff* shows the same intersection behavior.

There are no other possible pairings of the s_i with the t_j that would yield selectors that match elements that match both s and t, so the construction is total. Moreover, it is correct by construction, since it preserves the ordering and adjacency requirements of its input arguments, and uses *Inter* to produce results that are inductively correct and total. Therefore the construction is correct and total as desired.

Case 4.c: The remaining case is more challenging. Elements satisfying s and $t = t_1 \sim \cdots \sim t_M$ (where t_i contain only $(+)$) must satisfy some element of $\mathit{Inter}(s_N, t_M)$, and must have some sibling satisfying $s_1 \sim \cdots \sim s_{N-1}$ and some sibling satisfying $t_1 \sim \cdots \sim t_{M-1}$—but nothing is known about the relative ordering of any s_i with any t_i: for some elements, they may match two distinct siblings (in either order), and for others they may in fact match the same sibling. This description is correct and complete: there are no other ways for a selector to be in the intersection of s and t. Therefore I need only compute correct and total sets for each of these three options, and I will have shown correctness and totality for the full intersection, as needed.

Looking at the definition of *Interleavings* in Fig. 5.15, there are precisely these three options. The base cases ensure that I intersect s_N and t_M, and by induction I know that such cases are correct and total. In the recursive case, the first two recursive calls permit s_1 and t_1 to appear in either order in the final result. again, these two calls are inductively correct and total, so they compute the correct and total intersection when s_1 (or t_1) comes first.

The remaining component splits s and t into two parts:

$$s = \big((s_1 \sim \cdots \sim s_{i-1}) \sim s_i\big) \sim \big((s_{i+1} \sim \cdots \sim s_{N-1}) \sim s_N\big)$$
$$t = \big((t_1 \sim \cdots \sim t_{j-1}) \sim t_j\big) \sim \big((t_{j+1} \sim \cdots \sim t_{M-1}) \sim t_M\big)$$

By induction, I know that the intersections $\mathit{Inter}(((s_1 \sim \cdots \sim s_{i-1}) \sim s_i), ((t_1 \sim \cdots \sim t_{j-1}) \sim t_j))$ and

$Inter(((s_{i+1} \sim \cdots \sim s_{N-1}) \sim s_N), ((t_{j+1} \sim \cdots \sim t_{M-1}) \sim t_M))$ are both correct and total. And since they ensure that their results intersect the last elements of their arguments, I get the intersection of s_i with t_j, as asserted. Therefore combining them via (~) gives the correct and total intersection of s and t *for elements with a sibling matching the intersection of* s_i *and* t_i. Taking their union for all $1 \leq i < N$ and $1 \leq j < M$ yields the correct and total intersection of s and t *for elements with at least one sibling matching the intersection of some* s_i *and* t_j.

Since I have computed correct and total results for all possible forms of the intersection, I have shown *Inter* is correct and total for this case, as needed.

Case 5: s **and** t **contain only** (+)**,** (~) **or** (>)**:** Examine the next four cases of Fig. 5.14. Let $s = s_1 > \cdots > s_N$ where s_i contain only (+) or (~).

Case 5.a: The case where t is a simple selector is again trivial.

Case 5.b: Elements satisfying both s and $t = t_1 + \cdots + t_M$ (where t_i are simple selectors) must satisfy both s_N and t_M. Additionally, they must have an adjacent sibling satisfying $t_1 + \cdots + t_{M-1}$, and a parent satisfying $s_1 > \cdots > s_{N-1}$: there is no way for these selectors to describe the same nodes in the tree, so I do not need to intersect them. Instead, the only possible results are $(s_1 > \cdots > s_{N-1}) > (t_1 + \cdots + t_{M-1}) + Inter(s_N, t_M)$. By induction I know $Inter(s_N, t_M)$ is correct and total, and therefore so is the combined result.

Case 5.c: The case where $t = t_1 \sim \cdots \sim t_M$ follows identical reasoning.

Case 5.d: Elements satisfying both s and $t = t_1 > \cdots > t_M$ (where t_i contain only (+) or (~)) must satisfy both s_N and t_M. Additionally, their parents must satisfy both $s_1 > \cdots > s_{N-1}$ and $t_1 > \cdots > t_{M-1}$. By induction, $Inter(s_1 > \cdots > s_{N-1}, t_1 > \cdots > t_{M-1})$ is correct and total. Therefore

$$Inter(s_1 > \cdots > s_{N-1}, t_1 > \cdots > t_{M-1}) > Inter(s_N, t_M)$$

describes exactly those nodes satisfying both s and t and so is correct and total as well.

Case 6: s **and** t **contain any combinators:** Examine the remaining five cases of Fig. 5.14. Let $s = s_1 \sqcup \cdots \sqcup s_N$, where s_I contain only (+), (~) or (>).

Case 6.a: The case where t is a simple selector is again trivial.

Case 6.b: The cases where $t = t_1 + \cdots + t_M$ or $t = t_1 \sim \cdots \sim t_M$ follow reasoning identical to the corresponding cases for $s = s_1 > \cdots > s_N$.

Case 6.c: The cases where $t = t_1 > \cdots > t_M$ (where t_i contain only (+) or (~)) follows reasoning identical to the case for $s = s_1 \sim \cdots \sim s_N$, $t = t_1 + \cdots + t_M$.

Case 6.d: The case where $t = t_1 \text{␣} \cdots \text{␣} t_M$ (where t_i contain only (+), (~) or (>)) follows reasoning identical to the case for $s = s_1 \sim \cdots \sim s_N$, $t = t_1 \sim \cdots \sim t_M$.

□

Proof of Theorem 3. Let $s, t \in \mathit{Selector}$ be arbitrary, precedence-associated selectors. Then

- Length: $\forall x \in \mathit{Inter}(s,t), |x| \leq |s| + |t|$.
- Count: $|\mathit{Inter}(s,t)| \in O(3^{|s|+|t|})$.
- Speed: Deciding whether $\mathit{Inter}(s,t) = \emptyset$ can be computed in time $O(|s| + |t|)$.

Proof sketch. I give an informal overview of proofs for each of these claims. As written, the *Interleavings* function is hard to analyze, so I define a more complicated but optimized version that computes the result set with fewer redundancies. The optimized verion is defined in Fig. A.1. (Ironically, it is possible to fully optimize the function so it computes every result exactly once, but that function is again hard to analyze for its precise running time.)

Length: It is intuitive that for all $x \in \mathit{Inter}(s,t)$, $|x| \leq |s| + |t|$: looking at the definitions of *Inter*, note that the output either includes every simple selector in s and t, or combines some pairs of them into a single simple selector, but never creates new simple selectors from nothing.

Count: Certainly the potential size of $\mathit{Inter}(s,t)$ itself is exponential in the size of its arguments, due to *Interleavings* computing at least $2^{(|s|+|t|)}$ possibilities: the number of ways to interleave l items between m others is the number of ways to choose l slots out of $l+m$ positions, or $\binom{l+m}{l}$. When $l = m$, this simplifies to $\binom{2l}{l} \in O(2^{2l})$. To show that the performance is at least the claimed $O(3^{|s|+|t|})$, in the worst cases:

Case 1: $s, t \in \mathit{SimpleSelector}$: Since two simple selectors either can describe the same element or cannot, $|\mathit{Inter}(s,t)| \leq 1$.

Case 2: $s, t \in \mathit{AdjacentSibling}$: Since there are no degrees of freedom to reorder the simple selectors, I must match up and intersect each corresponding simple selector, yielding $|\mathit{Inter}(s,t)| \leq 1$.

Case 3: $s \in \mathit{SiblingSelector}$, $t \in \mathit{AdjacentSibling}$: Let $s = s_1 \sim \cdots \sim s_l$ and $t = t_1 + \cdots + t_m$, and let $n = l + m$. Let $T_{AP}(l,m) = |\mathit{Pairings}(s,t)|$ be the size of the result set of *Pairings*, and $T_{PO}(l,m) = |\mathit{PairOff}(s,t)|$ similarly be the size of the result of *PairOff*. Examining the definition of *Pairings* note that T_{AP} and T_{PO} must obey the recurrences

$$\begin{aligned} T_{AP}(l,m) &= T_{PO}(l-1,m-1)|\mathit{Inter}(s_l,t_m)| \\ T_{PO}(l,m) &= T_{PO}(l,m-1) + T_{PO}(l-1,m-1)|\mathit{Inter}(s_l,t_m)| \\ T_{PO}(l,1) &= 1 + |\mathit{Inter}(s_l,t)| \end{aligned}$$

By construction, I know that t_m (or t in the last equation) is a simple selector, so by the previous cases, $|\mathit{Inter}(s_l,t_m)| \leq 1$. Simplifying yields

$$\begin{aligned} T_{AP}(l,m) &\leq T_{PO}(l-1,m-1) \\ T_{PO}(l,m) &\leq T_{PO}(l,m-1) + T_{PO}(l-1,m-1) \\ T_{PO}(l,1) &\leq 2 \end{aligned}$$

Since m decreases by 1 in every recursive call to T_{PO} and there are two distinct calls to T_{PO} (they have unequal arguments), this has the simple solution $T_{PO} \in O(2^m)$. I therefore deduce $|\mathit{Inter}(s,t)| = T_{AP}(|s|,|t|) \in O(2^{|t|}) \in O(2^{(|s|+|t|)})$.

Case 4: $s,t \in \mathit{SiblingSelector}$: Let $s = s_1 \sim \cdots \sim s_l$ and $t = t_1 \sim \cdots \sim t_m$. Again define a function $T_{AI}(l,m) = |\mathit{Interleavings}_{(\sim)}(s,t)|$ to be the size of the result set of $\mathit{Interleavings}_{(\sim)}$, and $T_I(l,m) = |\mathit{Interleave}_{(\sim)}(s,t)|$ to be the size of the result set of $\mathit{Interleave}_{(\sim)}$. Examining its definition, note that T_{AI} and T_I must obey the recurrences

$$\begin{aligned} T_{AI}(l,m) &= T_{AI}(l-1,m) + T_{AI}(l,m-1) + \sum_{i=1}^{l-1}\sum_{j=1}^{m-1} T_{AI}(i,j)T_I(l-i,m-j) \\ T_{AI}(l,1) &= |\mathit{Inter}(s,t)| \\ T_{AI}(1,m) &= |\mathit{Inter}(s,t)| \\ T_I(l,m) &= T_I(l-1,m) + T_I(l,m-1) \\ T_I(l,1) &= |\mathit{Inter}(s,t)| \\ T_I(1,m) &= |\mathit{Inter}(s,t)| \end{aligned}$$

By construction, I know that each call *Inter* is being called with at least one adjacent-sibling selector,

so I can simplify the base cases to

$$T_{AI}(l,1) \in O(2^{|t|})$$
$$T_{AI}(1,m) \in O(2^{|s|})$$
$$T_I(l,1) \leq 1$$
$$T_I(1,m) \leq 1$$

I solve the recurrence for T_I to get $T_I(l,m) \in \Theta(\binom{l+m-2}{l-1})$ using the substitution method: For the inductive case I must show that $T_I(l,m) = \binom{l+m-2}{l-1}$, and substituting into the recurrence yields

$$\begin{aligned}
T_I(l,m) &\leq \binom{l+m-3}{l-2} + \binom{l+m-3}{m-2} \\
&= \frac{(l+m-3)!}{(l-2)!(m-1)!} + \frac{(l+m-3)!}{(l-1)!(m-2)!} \\
&= \frac{l-1}{l-1}\frac{(l+m-3)!}{(l-2)!(m-1)!} + \frac{m-1}{m-1}\frac{(l+m-3)!}{(l-1)!(m-2)!} \\
&= \frac{(l-1)(l+m-3)!}{(l-1)!(m-1)!} + \frac{(m-1)(l+m-3)!}{(l-1)!(m-1)!} \\
&= \frac{(l+m-2)(l+m-3)!}{(l-1)!(m-1)!} \\
&= \binom{l+m-2}{l-1}
\end{aligned}$$

For the base cases, consider when $|s| = |t| = 1$. Then $T_I(1,1) = |Inter(s,t)| = 1 = \binom{1+1-2}{1-1}$ as desired.

An aside: for $X \neq 2$,

$$\sum_{i=1}^{n-1} \left(\frac{2}{X}\right)^{n-i} = \sum_{i=1}^{n-1} \left(\frac{2}{X}\right)^{i} = \frac{2}{X-2}\left[1 - \left(\frac{2}{X}\right)^{n-1}\right] \tag{A.1}$$

which can be verified by induction:

$$\begin{aligned}
\sum_{i=1}^{n} \left(\frac{2}{X}\right)^{i} &= \sum_{i=1}^{n-1} \left(\frac{2}{X}\right)^{i} + \left(\frac{2}{X}\right)^{n} \\
&\quad \textit{\{by induction\}} \\
&= \frac{2}{X-2}\left[1 - \left(\frac{2}{X}\right)^{n-1}\right] + \left(\frac{2}{X}\right)^{n}
\end{aligned}$$

$$
\begin{aligned}
&= \frac{2}{X-2}\left[1-\left(\frac{2}{X}\right)^{n-1}\right] + \left(\frac{2}{X}\right)^{n}\left(\frac{X-2}{2}\right)\left(\frac{2}{X-2}\right) \\
&= \frac{2}{X-2}\left[1-\left(\frac{2}{X}\right)^{n-1} + \left(\frac{2}{X}\right)^{n}\left(\frac{X-2}{2}\right)\right] \\
&\quad \textit{\{expanding } \tfrac{X-2}{2} \textit{ and simplifying\}} \\
&= \frac{2}{X-2}\left[1-\left(\frac{2}{X}\right)^{n-1} + \left(\frac{2}{X}\right)^{n-1} - \left(\frac{2}{X}\right)^{n}\right] \\
&= \frac{2}{X-2}\left[1-\left(\frac{2}{X}\right)^{n}\right]
\end{aligned}
$$

Checking the base cases, $n = 1$ is vacuously true, and case $n = 2$ simplifies to

$$\sum_{i=1}^{2-1}\left(\frac{2}{X}\right)^{i} = \left(\frac{2}{X}\right)^{1} = \left(\frac{2}{X}\right)\left(\frac{2}{X-2}\right)\left(\frac{X-2}{2}\right) = \frac{2}{X-2}\left[\frac{2}{X}\frac{X-2}{2}\right] = \frac{2}{X-2}\left[1-\frac{2}{X}\right]$$

I use this formula below.

I also know that

$$\binom{n}{m} \leq \binom{n}{n/2} \leq 3(2^n(n+1)^{-1/2}) \tag{A.2}$$

(In fact, the factor of 3 is an over-approximation; the formula is asymptotically tight with a constant of $\sqrt{2}$. I only need the looser result here.)

With this, I can check that the hypothesis $T(l,m) \in O(3^{l+m})$ satisfies the recurrence, again using the substitution method:

$$
\begin{aligned}
T_{AI}(l,m) &= T_{AI}(l-1,m) + T_{AI}(l,m-1) + \sum_{i=1}^{l-1}\sum_{j=1}^{m-1} T_{AI}(i,j)T_I(l-i,m-j) \\
&\leq c3^{l-1+m} + c3^{l+m-1} + \sum_{i=1}^{l-1}\sum_{j=1}^{m-1} c3^{i+j}\binom{l-i+m-j-2}{l-i-1} \\
&= 2c3^{l+m-1} + \sum_{i=1}^{l-1}\sum_{j=1}^{m-1} c3^{i+j}\frac{(l-i+m-j-2)!}{(l-i-1)!(m-j-1)!}
\end{aligned}
$$

By Eq. (A.2)

$$\leq 2c3^{l+m-1} + \sum_{i=1}^{l-1}\sum_{j=1}^{m-1} c3^{i+j+1}\frac{2^{l-i+m-j-2}}{\sqrt{1+(l-i+m-j-2)/2}}$$

Changing $i \to l - i$ and $j \to m - j$,

$$
\begin{aligned}
&= 2c3^{l+m-1} + \sum_{i=1}^{l-1}\sum_{j=1}^{m-1} c3^{l-i+m-j+1}\frac{2^{i+j-2}}{\sqrt{1+(i+j-2)/2}}\left(\frac{3}{2}\right)^{i+j-2}\left(\frac{2}{3}\right)^{i+j-2} \\
&= 2c3^{l+m-1} + \sum_{i=1}^{l-1}\sum_{j=1}^{m-1} \frac{c}{4}3^{l+m-1}\frac{1}{\sqrt{1+(i+j-2)/2}}\left(\frac{2}{3}\right)^{i+j} \\
&= 2c3^{l+m-1} + \frac{c}{4}3^{l+m-1}\sum_{i=1}^{l-1}\sum_{j=1}^{m-1} \frac{1}{\sqrt{1+(i+j-2)/2}}\left(\frac{2}{3}\right)^{i+j} \\
&\leq 2c3^{l+m-1} + \frac{c}{4}3^{l+m-1}\sum_{i=1}^{l-1}\sum_{j=1}^{m-1} \left(\frac{2}{3}\right)^{i+j} \\
&= 2c3^{l+m-1} + \frac{c}{4}3^{l+m-1}\left(\sum_{i=1}^{l-1}\left(\frac{2}{3}\right)^{i}\right)\left(\sum_{j=1}^{m-1}\left(\frac{2}{3}\right)^{j}\right)
\end{aligned}
$$

Using Eq. (A.1), with $X = 3$,

$$
= 2c3^{l+m-1} + \frac{c}{4}3^{l+m-1}\left[\frac{2}{3-2}\left(1-\left(\frac{2}{3}\right)^{l-1}\right)\right]\left[\frac{2}{3-2}\left(1-\left(\frac{2}{3}\right)^{m-1}\right)\right]
$$

The parenthetical terms stay between 0 and 1, so the bracketed terms are each less than 2, so

$$
\begin{aligned}
&\leq 2c3^{l+m-1} + 4\frac{c}{4}3^{l+m-1} \\
&\leq 2c3^{l+m-1} + c3^{l+m-1} \\
&= c3^{l+m}
\end{aligned}
$$

The base cases follow, since $O(2^n) \in O(3^n)$ for any positive n. So $|Inter(s,t)| \in O(3^{|s|+|t|})$, as desired.

Case 5: $s, t \in$ *ChildSelector*: Since there are no degrees of freedom to reorder the sibling selectors, I must match up and intersect each corresponding sibling selector, yielding $|Inter(s,t)| \in O(3^{|s|+|t|})$ by the previous case.

Case 6: $s, t \in$ *DescendantSelector*: The same analysis as for sibling selectors applies, only this time, I have $O(3^n)$ ways to match up each child selector (with sibling selectors inside). However, the presence of sibling selectors cuts down on the number of descendant selectors available at a given total

length. The worst case behavior occurs when $s = s_1 \sqcup \cdots \sqcup s_{m/2} \sqcup s_{m/2+1} \sim \cdots \sim s_m$ and t similarly. Both portions behave as $O(3^{n/2})$, with a product that remains $O(3^n)$. So $|Inter(s,t)| \in O(3^{|s|+|t|})$.

The worst case behavior, therefore is $|Inter(s,t)| \in O(3^{|s|+|t|})$.

Speed: In practice, I only need to know whether $Inter(s,t) \neq \emptyset$. Here the performance is much faster: in fact, it is $O(|s| + |t|)$. I must show two things: first, I need only demonstrate a single element in the intersection to know that it is non-empty, and that can be done quickly. Second, I must be able to return the empty set without the exponential blowup incurred in some cases of the algorithm.

For quickly demonstrating a single element when one exists, at each point in the algorithm where there is a choice (e.g., calls to *Interleavings*), I simply pick the easiest choice available (e.g., the non-interleaved $sib_1 \sim adj_2 + Inter(adj_1, simple_2)$). Every clause for *Inter* makes a recursive call to a smaller-sized input, and the base case returns at most one argument. So every call is constant-time, and there are at most as many calls as there are combinators in the inputs, yielding the claimed runtime.

For quickly demonstrating emptiness, note that the easiest choice mentioned above is also the least-constrained, in that it avoids any unnecessary calls to *Inter*: the only recursive calls left are those in the child, adjacent-sibling, and simple-selector cases which are hard constraints on the intersection. Said another way, if *any* choice of interleavings or pairings would lead to a feasible selector in the intersection, these easiest choices will find *some* feasible selector, and if they fail, then the intersection truly is empty. Because the easiest choice suffices, I can ignore all calls to *Interleavings* and *Pairings*, yielding a simple, linear algorithm as claimed. □

Appendix B

OVERLAY DETAILS

B.1 Firefox-like overlays: algorithmic details

Defining the conflict graph-algorithm formally requires defining the abstraction process from concrete overlays through to compositions and guarded overlays and then to state-pair interfaces. The conflict-graph algorithm then falls out naturally from the rules for compositions.

Concrete overlays to guarded overlays: The first step is to compile a concrete overlay into a guarded overlay. (Modelling Firefox-level extensions will not need to construct any compositions explicitly; so the rules for compositions are deferred for now.) This process is shown in Fig. B.3. It uses four helper functions, reqs*, rejs*, first* and last*, that collect any explicit $\langle$**guard**/$\rangle$s into abstract guards and use them to surround an *Overlay*. The *Overlay* in turn contains *Insert* and *Modify* actions for each $\langle$**insert**/$\rangle$ or $\langle$**modify**/$\rangle$ tag in the source $\langle$**overlay**/$\rangle$. One additional helper function, defs*, is used that encompasses the HTML-specific (or XUL-specific) knowledge the analysis needs. Here, it detects and defines *Key*, *Id*, and *Selected* resources to describe additional semantic constraints that should be enforced on HTML overlays.

The remainder of Fig. B.3 defines two "structural" helper functions, reqs and defs, that are used to bootstrap the translation from abstract guarded overlays to state-pair interfaces as shown in Fig. B.2. These functions record only the generic structure of the content being inserted by the overlay, and are then lifted from overlays to guarded overlays. Rule Overlay-Interface asserts that "if everything required by o is present, and everything defined is o be absent in the input, then o guarantees that everything it uses is defined in the output".

Guarded overlays to state-pair interfaces: The next step moves up one layer in the language, to define state-pair interfaces for guarded overlays, and this computation is performed by the judgment $[\![S_i]\!]\ g\ [\![S_o]\!]$. Note the few asymmetries: if you *Require* a particular requirement be defined as a precondition, it remains defined as a postcondition, but if you *Reject* a particular requirement as defined in the precondition, it may become defined in the postcondition. These asymmetries result from the postcondition's *Def* and *Undef* sets being determined by the overlay and the precondition, as mentioned earlier. Additionally, freezing a particular requirement in the

$$
\begin{aligned}
&\mathit{ElemSel} : \mathit{Element} \to \mathit{Bool} \to \mathit{Resource} \\
&\mathit{ElemSel}(e, \mathit{useNonce}) \stackrel{\text{def}}{=} \begin{cases} \mathit{Selector}(e.\mathit{tagName}\#e.\mathit{id}\,.\,e.\mathit{class}) & e.\mathit{id} \neq \texttt{""} \\ \mathit{Selector}(e.\mathit{tagName}\#\mathit{nonce}\,.\,e.\mathit{class}) & e.\mathit{id} = \texttt{""} \wedge \mathit{useNonce} \\ \mathit{Selector}(e.\mathit{tagName}\,.\,e.\mathit{class}) & e.\mathit{id} = \texttt{""} \wedge \neg \mathit{useNonce} \end{cases} \\
&\mathit{CompleteSubtreeSels} : \mathit{Element} \to \mathit{Bool} \to \mathit{Resource} \\
&\mathit{CompleteSubtreeeSels}(e) \stackrel{\text{def}}{=} \mathit{CompleteSubtreeSels}(e, \mathit{ElemSel}(e, \mathit{true})) \\
&\mathit{CompleteSubtreeSels}(e, \mathit{sel}) \stackrel{\text{def}}{=} \{\mathit{Selector}(\mathit{sel})\} \\
&\qquad \cup\ \mathit{CompleteSubtreeSels}(e.\mathit{firstChild}, \mathit{sel} > \mathit{ElemSel}(e.\mathit{firstChild}, \mathit{true})) \\
&\qquad \cup\ \mathit{CompleteSubtreeSels}(e.\mathit{nextSibling}, \mathit{sel} + \mathit{ElemSel}(e.\mathit{nextSibling}, \mathit{true})) \\
&\mathit{PartialSubtreesSels} : \mathit{Element} \to \mathit{Bool} \to \mathit{Resource} \\
&\mathit{PartialSubtreeeSels}(e) \stackrel{\text{def}}{=} \mathit{CompleteSubtreeSels}(e, \mathit{ElemSel}(e)) \\
&\mathit{PartialSubtreeSels}(e, \mathit{sel}) \stackrel{\text{def}}{=} \{\mathit{Selector}(\mathit{sel})\} \\
&\qquad \cup\ \bigcup \{\mathit{PartialSubtreeSels}(c, \mathit{sel} > \mathit{ElemSel}(c, \mathit{false})) \mid \\
&\qquad\qquad c.\mathit{parent} = e, c.\mathit{id} \neq \texttt{""}\} \\
&\qquad \cup\ \bigcup \{\mathit{PartialSubtreeSels}(c, \mathit{sel} \lrcorner \mathit{ElemSel}(c, \mathit{false})) \mid \\
&\qquad\qquad c.\mathit{parent}^{+} = e, c.\mathit{id} = \texttt{""}, \\
&\qquad\qquad (\neg \exists p. c.\mathit{parent}^{+} = p \wedge p.\mathit{parent}^{+} = e \wedge p.\mathit{id} \neq \texttt{""})\}
\end{aligned}
$$

Figure B.1: Helper routines for extracting selectors from trees

pre- or post-condition does not require it frozen on the other side; an overlay may wish to be the first or last to overlay a requirement, but not necessarily both first and last. Note too that the grammar permits self-inconsistent overlays, where $\mathit{Def} \cap \mathit{Undef} \neq \emptyset$.

Compositions to state-pair interfaces: At the top level of the language are compositions, in which multiple guarded overlays can be composed together in various ways. One of the essential parts of the semantics of overlays is that elements whose identifiers do not match anything defined in the document will silently fail to overlay. This is modelled with an optional ? constructor, indicating that part of the composition may fail without failing the entire composition. However, this implies that to compute the state-transformation effects of a composition, one must know what is defined in the document. Therefore the judgment will look like $S \vdash [\![S_i]\!]\ c\ [\![S_o]\!]$, indicating that when the document is in state S, composition c requires state S_i and produces state S_o.

The rules here are somewhat subtle. S-Sequence deals with sequencing two compositions. It assumes that both succeed (and relegates dealing with failed optional compositions to another

$$
\begin{array}{lll}
S \in \textsc{State} ::= \{\mathit{Def} = \vec{r}, & & \textit{–currently defined in document} \\
\quad \mathit{Undef} = \vec{r}, & & \textit{–currently undefined in document} \\
\quad \mathit{Clean} = \vec{r}, & & \textit{–has not yet been overlaid} \\
\quad \mathit{Frozen} = \vec{r}\,\} & & \textit{–must never again be overlaid}
\end{array}
$$

Requirements and definitions for guarded overlays: $\text{reqs}, \text{defs}, \text{used} : \mathit{Guard} \to 2^{\mathit{Resource}}$

$$\text{reqs}(\mathit{Require}(\vec{r}, g)) \stackrel{\text{def}}{=} \text{reqs}(g) \qquad \text{defs}(\mathit{Require}(\vec{r}, g)) \stackrel{\text{def}}{=} \text{defs}(g)$$

$$\text{reqs}(\mathit{Reject}(\vec{r}, g)) \stackrel{\text{def}}{=} \text{reqs}(g) \qquad \text{defs}(\mathit{Reject}(\vec{r}, g)) \stackrel{\text{def}}{=} \text{defs}(g)$$

$$\text{reqs}(\mathit{First}(\vec{r}, g)) \stackrel{\text{def}}{=} \text{reqs}(g) \qquad \text{defs}(\mathit{First}(\vec{r}, g)) \stackrel{\text{def}}{=} \text{defs}(g)$$

$$\text{reqs}(\mathit{Last}(\vec{r}, g)) \stackrel{\text{def}}{=} \text{reqs}(g) \qquad \text{defs}(\mathit{Last}(\vec{r}, g)) \stackrel{\text{def}}{=} \text{defs}(g)$$

$$\text{used}(g) \stackrel{\text{def}}{=} \text{reqs}(g) \cup \text{defs}(g)$$

$$\textsc{Overlay-Interface}\quad \frac{S_i = \left\{ \begin{array}{r} \mathit{Def} = \text{reqs}(o) \\ \mathit{Undef} = \text{defs}(o) \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = \emptyset \end{array} \right\} \qquad S_o = \left\{ \begin{array}{r} \mathit{Def} = \text{used}(o) \\ \mathit{Undef} = \emptyset \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = \emptyset \end{array} \right\}}{[\![o]\!] = (S_i, S_o)}$$

Figure B.2: Abstracting overlays into document-state interfaces

rule), and requires that the input requirements of the second composition (c_2) are compatible with the output guarantees of the first (c_1). Specifically, c_2 cannot require as undefined anything guaranteed to be defined by c_1, nor can it define anything already defined by c_1. Additionally, c_2 must respect the *Frozen* demands of c_1, and c_1 must preserve c_2's *Clean* requirements. Propagating the correct values to S_i and S_o is routine; the only surprising point is that no effort is made to compute $S_i.\mathit{Frozen}$ or $S_o.\mathit{Clean}$, since these are never needed.

The rules for optional compositions are trickier. The only components of the document state that are needed are *Def* and *Frozen* — the former summarizes what is known to be defined at this point in the document, while the latter summarizes any effects prior compositions have had on freezing portions of the document. S-Opt-Succeed therefore checks that its argument is valid (i.e., any internal compositions succeed with respect to each other), that its *Def* and *Undef* requirements are compatible with the document, and that it doesn't affect anything already *Frozen*. S-Opt-Fail, on the other hand, checks that its argument could succeed in some hypothetical document state,

HTML-to-guarded overlay: $[\![\cdot]\!] : \mathsf{HTML} \to \mathit{Guard}$

$$
\begin{aligned}
[\![\langle \textbf{overlay} \rangle \mathit{acts} \langle / \textbf{overlay} \rangle]\!] &= \mathit{Require}(\bigcup\{\text{reqs}^*(a) \mid a \in \mathit{acts}\}, \\
&\qquad \mathit{Reject}(\bigcup\{\text{defs}^*(a) \cup \text{rejs}^*(a) \mid a \in \mathit{acts}\}, \\
&\qquad\quad \mathit{First}(\bigcup\{\text{first}^*(a) \mid a \in \mathit{acts}\}, \\
&\qquad\qquad \mathit{Last}(\bigcup\{\text{last}^*(a) \mid a \in \mathit{acts}\}, \\
&\qquad\qquad\quad \mathit{Overlay}(\{[\![a]\!] \mid a \in \mathit{acts}\}))))) \\
[\![\langle \textbf{insert} \rangle \mathit{ins} \langle / \textbf{insert} \rangle]\!] &= \mathit{Insert}(a.\mathit{selector}, a.\mathit{where}, a.\mathit{kids}) \\
[\![\langle \textbf{modify} \rangle \langle \textbf{self } \mathit{attrs} / \rangle \langle / \textbf{modify} \rangle]\!] &= \mathit{Modify}(a.\mathit{selector}, \mathit{attrs}) \\
\text{defs}^*(h) &= \{\mathit{Key}(k) \mid \mathit{matches}(\texttt{command[accesskey=}k\texttt{]})h_i \neq \emptyset, \\
&\qquad h_i.\mathit{parent}^+ = h\} \\
&\quad \cup \{\mathit{Id}(\mathit{id}) \mid \mathit{matches}(\texttt{*[id=}\mathit{id}\texttt{]})h_i \neq \emptyset, h_i.\mathit{parent}^+ = h\} \\
&\quad \cup \{\mathit{Selected}(s) \mid \\
&\qquad \mathit{matches}(\texttt{select > option[selected=selected]})h_i \neq \emptyset, \\
&\qquad h_i.\mathit{parent}^+ = h\} \\
\text{reqs}^*(\langle \textbf{guard type=“require”} / \rangle) &= \{g.\mathit{resource}\} \\
\text{rejs}^*(\langle \textbf{guard type=“reject”} / \rangle) &= \{g.\mathit{resource}\} \\
\text{first}^*(\langle \textbf{guard type=“require”} / \rangle) &= \{g.\mathit{resource}\} \\
\text{last}^*(\langle \textbf{guard type=“require”} / \rangle) &= \{g.\mathit{resource}\}
\end{aligned}
$$

Requirements and definitions for abstract overlays: $\text{reqs}, \text{defs}, \text{used} : \mathit{Overlay} \to 2^{\mathit{Resource}}$

$$
\begin{aligned}
\text{reqs}(\mathit{Overlay}(\vec{a})) &\stackrel{\text{def}}{=} \bigcup\{\text{reqs}(a_i) \mid a_i \in \vec{a}\} \\
\text{reqs}(\mathit{Insert}(s, \_, \_)) &= \{\mathit{Selector}(s)\} \\
\text{reqs}(\mathit{Modify}(s, \_)) &= \{\mathit{Selector}(s)\} \\
\text{defs}(\mathit{Overlay}(\vec{a})) &\stackrel{\text{def}}{=} \bigcup\{\text{defs}(a_i) \mid a_i \in \vec{a}\} \cup \text{defs}^*(\langle \textbf{overlay} \rangle a_i \langle / \textbf{overlay} \rangle) \\
\text{defs}(\mathit{Insert}(s, \_, \vec{h})) &= \bigcup\{\mathit{CompleteSubtreeSels}(h_i, s) \mid h_i \in \vec{h}\} \\
\text{defs}(\mathit{Modify}(\_, \_)) &= \emptyset \\
\text{used}(o) &\stackrel{\text{def}}{=} \text{reqs}(o) \cup \text{defs}(o)
\end{aligned}
$$

Figure B.3: Compiling HTML to guarded overlays

$\boxed{[\![S_i]\!]\ g\ [\![S_o]\!]}$

$$\textsc{G-Overlay}\ \frac{[\![o]\!] = (S_i, S_o)}{[\![S_i]\!]\ o\ [\![S_o]\!]}$$

$$\textsc{G-Require}\ \frac{[\![S_i]\!]\ g\ [\![S_o]\!] \quad S_i' = S_i \cup \{Def = \vec{r}\} \quad S_o' = S_o \cup \{Def = \vec{r}\}}{[\![S_i']\!]\ Require(\vec{r}, g)\ [\![S_o']\!]}$$

$$\textsc{G-Reject}\ \frac{[\![S_i]\!]\ g\ [\![S_o]\!] \quad S_i' = S_i \cup \{Undef = \vec{r}\} \quad S_o' = S_o \cup \{Undef = \vec{r} \setminus S_o.Def\}}{[\![S_i']\!]\ Reject(\vec{r}, g)\ [\![S_o']\!]}$$

$$\textsc{G-First}\ \frac{[\![S_i]\!]\ g\ [\![S_o]\!] \quad S_i' = S_i \cup \{Clean = \vec{r}\}}{[\![S_i']\!]\ First(\vec{r}, g)\ [\![S_o]\!]}$$

$$\textsc{G-Last}\ \frac{[\![S_i]\!]\ g\ [\![S_o]\!] \quad S_o' = S_o \cup \{Frozen = \vec{r}\}}{[\![S_i]\!]\ Last(\vec{r}, g)\ [\![S_o']\!]}$$

Figure B.4: Semantics of guarded overlays

but that any such state is incompatible with the current one. If so, the failed composition has no requirements or effects.

These two rules together conveniently give the idempotence property that $c??$ will have the same effects as $c?$. (Specifically, if $S \vdash [\![S_i]\!]\ c?\ [\![S_o]\!]$ is derivable using S-Opt-Fail, then $S \vdash [\![S_i]\!]\ c??\ [\![S_o]\!]$ is derivable using either S-Opt-Succeed or S-Opt-Fail, but all three derivations will return the same empty effects for S_i and S_o.) Additionally, to make the sequencing of optional compositions more precise, in S-Sequence c_2 is checked under document state $S' = \{S.Def \cup S_o^1.Def, \emptyset, \emptyset, S_o^1.Frozen\}$, rather than just S — this ensures that if c_1 defines something which c_2 requires, $c_1\,;(c_2?)$ can succeed. A sequence of compositions successfully applies *in order* to a document if the sequence is self-consistent, and the cumulative preconditions are satisfied by the document. The inference rules presented so far take the convention that a base document merely guarantees a set of resources, but does not prohibit anything from being futher overlaid. (In particular, to accommodate an idiosyncracy of Firefox, this convention ensures that by choosing to *not* include any declared keybindings from the guaranteed set, those keybindings are free for future overlays, so extensions get a clean slate to begin with.)

B.1.1 Motivating examples, revisited

Recall the example "Hello, world" overlays, with requirements added. I write them here using concrete syntax, to illustrate the entire pipeline:

$$\mathrm{reqs}(S, g) \stackrel{\text{def}}{=} \mathrm{reqs}(g) \qquad \mathrm{defs}(S, g) \stackrel{\text{def}}{=} \mathrm{defs}(g) \qquad \mathrm{reqs}(S, c_1\,;c_2) \stackrel{\text{def}}{=} \mathrm{reqs}(S, c_1) \cup \mathrm{reqs}(S, c_2)$$

$$\mathrm{defs}(S, c_1\,;c_2) \stackrel{\text{def}}{=} \mathrm{defs}(S, c_1) \cup \mathrm{defs}(S, c_2) \qquad \frac{S \vdash [\![S_i]\!]\ c?\ [\![S_o]\!]}{\mathrm{reqs}(S, c?) \stackrel{\text{def}}{=} \mathrm{reqs}(S, c) \cap S_i.\mathit{Def}}$$

$$\frac{S \vdash [\![S_i]\!]\ c?\ [\![S_o]\!]}{\mathrm{defs}(S, c?) \stackrel{\text{def}}{=} \mathrm{defs}(S, c) \cap S_o.\mathit{Def}} \qquad \frac{S \vdash [\![S_i]\!]\ c_1\,!\,c_2\ [\![S_o]\!]}{\mathrm{defs}(S, c_1\,!\,c_2) \stackrel{\text{def}}{=} (\mathrm{defs}(S, c_1) \cup \mathrm{defs}(S, c_2)) \cap S_o.\mathit{Def}}$$

$$\frac{S \vdash [\![S_i]\!]\ c_1\,!\,c_2\ [\![S_o]\!]}{\mathrm{reqs}(S, c_1\,!\,c_2) \stackrel{\text{def}}{=} (\mathrm{reqs}(S, c_1) \cup \mathrm{reqs}(S, c_2)) \cap S_i.\mathit{Def}} \qquad \mathrm{used}(S, c) \stackrel{\text{def}}{=} \mathrm{reqs}(S, c) \cup \mathrm{defs}(S, c)$$

$$\boxed{S \vdash [\![S_i]\!]\ c\ [\![S_o]\!]}$$

S-Guard
$$\frac{[\![S_i]\!]\ g\ [\![S_o]\!]}{S \vdash [\![S_i]\!]\ g\ [\![S_o]\!]}$$

S-Unique-1
$$\frac{S \vdash [\![S_i]\!]\ p_1\ [\![S_o]\!]}{S \vdash [\![S_i]\!]\ p_1\,!\,p_2\ [\![S_o]\!]}$$

S-Unique-2
$$\frac{S \vdash [\![S_i]\!]\ p_2\ [\![S_o]\!]}{S \vdash [\![S_i]\!]\ p_1\,!\,p_2\ [\![S_o]\!]}$$

S-Opt-Succeed
$$\frac{\begin{array}{c} S \vdash [\![S_i]\!]\ c\ [\![S_o]\!] \\ S_i.\mathit{Def} \subseteq S.\mathit{Def} \\ S_i.\mathit{Undef} \cap S.\mathit{Def} = \emptyset \\ \mathrm{used}(S, c) \cap S.\mathit{Frozen} = \emptyset \end{array}}{S \vdash [\![S_i]\!]\ c?\ [\![S_o]\!]}$$

S-Opt-Fail
$$\frac{\begin{array}{c} S'' \vdash [\![S_i]\!]\ c?\ [\![S_o]\!] \\ (S_i.\mathit{Def} \setminus S.\mathit{Def} \neq \emptyset) \vee \\ (S_i.\mathit{Undef} \cap S.\mathit{Def} \neq \emptyset) \vee \\ (\mathrm{used}(S'', c?) \cap S.\mathit{Frozen} \neq \emptyset) \\ S' = \{\emptyset, \emptyset, \emptyset, \emptyset\} \end{array}}{S \vdash [\![S']\!]\ c?\ [\![S']\!]}$$

S-Sequence
$$\frac{\begin{array}{c} S \vdash [\![S_i^1]\!]\ c_1\ [\![S_o^1]\!] \qquad S' = \left\{ \begin{array}{r} \mathit{Def} = S.\mathit{Def} \cup S_o^1.\mathit{Def} \\ \mathit{Undef} = \emptyset \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = S_o^1.\mathit{Frozen} \end{array} \right\} \qquad S' \vdash [\![S_i^2]\!]\ c_2\ [\![S_o^2]\!] \\ S'.\mathit{Def} \cap S_i^2.\mathit{Undef} = \emptyset \qquad S'.\mathit{Def} \cap \mathrm{defs}(S', c_2) = \emptyset \qquad S'.\mathit{Frozen} \cap \mathrm{used}(S', c_2) = \emptyset \\ S_i^2.\mathit{Clean} \cap \mathrm{reqs}(S, c_1) = \emptyset \qquad S_i = \left\{ \begin{array}{r} \mathit{Def} = S_i^1.\mathit{Def} \cup (S_i^2.\mathit{Def} \setminus \mathrm{defs}(S, c_1)) \\ \mathit{Undef} = S_i^1.\mathit{Undef} \cup S_i^2.\mathit{Undef} \\ \mathit{Clean} = S_i^1.\mathit{Clean} \cup S_i^2.\mathit{Clean} \\ \mathit{Frozen} = \emptyset \end{array} \right\} \\ S_o = \left\{ \begin{array}{r} \mathit{Def} = S_o^1.\mathit{Def} \cup S_o^2.\mathit{Def} \\ \mathit{Undef} = S_o^2.\mathit{Undef} \cup (S_o^1.\mathit{Undef} \setminus \mathrm{defs}(S_o^1, c_2)) \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = S_o^1.\mathit{Frozen} \cup S_o^2.\mathit{Frozen} \end{array} \right\} \end{array}}{S \vdash [\![S_i]\!]\ c_1\,;c_2\ [\![S_o]\!]}$$

Figure B.5: Semantics of sequencing

$D \in \textsc{Doc} ::= h \mid D[c]$ *–overlay sequence*

$\boxed{\vdash d : [\![S_o]\!]}$

D-BASE

$$\frac{S_o = \left\{ \begin{array}{rl} \mathit{Def} & = \mathrm{defs}^*(h) \\ & \cup\ \mathit{CompleteSubtreeSels}(h) \\ \mathit{Undef} & = \emptyset \\ \mathit{Clean} & = \mathrm{defs}^*(h) \\ & \cup\ \mathit{CompleteSubtreeSels}(h) \\ \mathit{Frozen} & = \emptyset \end{array} \right\}}{\vdash h : [\![S_o]\!]}$$

D-COMPOSE

$$\frac{\begin{array}{c} \vdash d : [\![S_d]\!] \qquad S_d \vdash [\![S_i^s]\!]\ s\ [\![S_o^s]\!] \\ S_d.\mathit{Def} \supseteq S_i^s.\mathit{Def} \\ S_d.\mathit{Def} \cap S_i^s.\mathit{Undef} = \emptyset \\ S_d.\mathit{Clean} \supseteq S_i^s.\mathit{Clean} \\ S_d.\mathit{Frozen} \cap \mathrm{used}(S_d, s) = \emptyset \end{array} \quad S_o = \left\{ \begin{array}{rl} \mathit{Def} & = S_d.\mathit{Def} \cup S_o^s.\mathit{Def} \\ \mathit{Undef} & = \emptyset \\ \mathit{Clean} & = (S_d.\mathit{Clean} \setminus \mathrm{reqs}(S_d, s)) \\ & \cup\ \mathrm{defs}(S_d, s) \\ \mathit{Frozen} & = S_d.\mathit{Frozen} \cup S_o^s.\mathit{Frozen} \end{array} \right\}}{\vdash d[s] : [\![S_o]\!]}$$

$\boxed{\vdash D : \mathrm{ok}}$

D-OK

$$\frac{\pi : \{1, \ldots, n\} \to \{1, \ldots, n\} \qquad \vdash h[c_{\pi(1)}] \cdots [c_{\pi(n)}] : [\![S_o]\!]}{\vdash h[c_1] \cdots [c_n] : \mathrm{ok}}$$

Figure B.6: Semantics of Overlays

⟨**overlay** id="OV1"⟩
  ⟨**insert** id="i_1" selector="`p#greeting`" where="end"⟩
    ⟨**span** id="subject"⟩stranger.⟨/**span**⟩
  ⟨/**insert**⟩
⟨/**overlay**⟩
⟨**overlay** id="OV3'"⟩
  ⟨**guard** id="g" type="reject" resource="*Id*(`subject`)"/⟩
  ⟨**insert** id="i_3" selector="`p#greeting`" where="end"⟩
    ⟨**span** id="modifier"⟩ and good day,⟨/**span**⟩
  ⟨/**insert**⟩
⟨/**overlay**⟩

The first step is to compile OV1 to an abstract guarded overlay, as in Fig. B.3:

$$g_1 = [\![\mathrm{OV1}]\!] = \mathit{Require}(\mathrm{reqs}^*(i_1), \mathit{Reject}(\mathrm{defs}^*(i_1) \cup \mathrm{rejs}^*(i_1), \mathit{First}(\mathrm{first}^*(i_1), \mathit{Last}(\mathrm{last}^*(i_1), \mathit{Overlay}([\![i_1]\!])))))$$

$$\begin{aligned}
\text{reqs}^*(i_1) &= \emptyset \\
\text{defs}^*(i_1) &= \{Id(\texttt{subject})\} \\
\text{rejs}^*(i_1) &= \emptyset \\
\text{first}^*(i_1) &= \emptyset \\
\text{last}^*(i_1) &= \emptyset \\
ov_1 = [\![i_1]\!] &= \mathit{Insert}(\texttt{p\#greeting}, \mathit{end}, \langle \textbf{span} \text{ id=“subject”}\rangle\text{stranger.}\langle/\textbf{span}\rangle)
\end{aligned}$$

Substituting in, and leaving out empty sets, yields:

$$\begin{aligned}
g_1 = [\![\text{OV1}]\!] = \mathit{Reject}(&\{Id(\texttt{subject})\}, \\
&\mathit{Overlay}(\mathit{Insert}(\texttt{p\#greeting}, \text{end}, \\
&\qquad \langle \textbf{span} \text{ id=“subject”}\rangle\text{stranger.}\langle/\textbf{span}\rangle)))
\end{aligned}$$

This is nearly identical to the hand-written OV1; the only difference is that defs* has successfully extracted the unique-id constraint without having to hard-code it into the conflict-detection algorithm and without the overlay author having to specify it.

Turning to OV3′, the derivation continues by compiling OV3′ to an abstract guarded overlay, following Fig. B.3:

$$\begin{aligned}
g_2 = [\![\text{OV3'}]\!] = \mathit{Require}(&\text{reqs}^*(i_3) \cup \text{reqs}^*(g), \\
&\mathit{Reject}(\text{defs}^*(i_3) \cup \text{rejs}^*(i_3) \cup \text{defs}^*(g) \cup \text{rejs}^*(g), \\
&\quad \mathit{First}(\text{first}^*(i_1) \cup \text{first}^*(g), \mathit{Last}(\text{last}^*(i_1) \cup \text{last}^*(g), \\
&\qquad \mathit{Overlay}([\![i_3]\!] \cup [\![g]\!])))))
\end{aligned}$$

$$\begin{aligned}
\text{reqs}^*(i_3) &= \emptyset \\
\text{defs}^*(i_3) &= \{Id(\texttt{modifier})\} \\
\text{rejs}^*(i_3) &= \emptyset \\
\text{first}^*(i_3) &= \emptyset \\
\text{last}^*(i_3) &= \emptyset \\
\text{reqs}^*(g) &= \emptyset \\
\text{defs}^*(g) &= \emptyset \\
\text{rejs}^*(g) &= \{Id(\texttt{subject})\} \\
\text{first}^*(g) &= \emptyset
\end{aligned}$$

$$\begin{aligned}
\text{last}^*(g) &= \emptyset \\
ov_3 = [\![i_3]\!] &= \{\mathit{Insert}(\texttt{p\#greeting}, \mathit{end}, \langle \textbf{span} \text{ id=“modifier”}\rangle \text{ and good day,} \langle/\textbf{span}\rangle)\} \\
[\![g]\!] &= \emptyset
\end{aligned}$$

Substituting in, and leaving out empty sets, yields:

$$\begin{aligned}
g_3 = [\![\text{OV3'}]\!] = \mathit{Reject}(&\{\mathit{Id}(\texttt{modifier}), \mathit{Selector}(\texttt{span\#subject})\}, \\
&\mathit{Overlay}(\mathit{Insert}(\texttt{p\#greeting}, \text{end}, \\
&\qquad \langle \textbf{span} \text{ id=“modifier”}\rangle \text{ and good day,} \langle/\textbf{span}\rangle))))
\end{aligned}$$

Again, this is nearly identical to OV3', differing only in the automatic addition of the unique-id constraint on *Id*(`modifier`).

Next, these results are abstracted into state-pair interfaces, starting by collecting the state information for OV1:

$$\begin{aligned}
\text{reqs}(ov_1) &= \{\mathit{Selector}(\texttt{p\#greeting})\} \\
\text{defs}(ov_1) &= \{\mathit{Selector}(\texttt{p\#greeting > span\#subject}), \mathit{Id}(\texttt{subject})\} \\
\text{used}(ov_1) &= \{\mathit{Selector}(\texttt{p\#greeting}), \mathit{Selector}(\texttt{p\#greeting > span\#subject}), \mathit{Id}(\texttt{subject})\}
\end{aligned}$$

By Overlay-Interface,

$$S'_i = \left\{ \begin{array}{r} \mathit{Def} = \text{reqs}(ov_1) \\ \mathit{Undef} = \text{defs}(ov_1) \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = \emptyset \end{array} \right\} = \left\{ \begin{array}{r} \mathit{Def} = \{\mathit{Selector}(\texttt{p\#greeting})\} \\ \mathit{Undef} = \{\mathit{Selector}(\texttt{p\#greeting > span\#subject}), \\ \qquad \mathit{Id}(\texttt{subject})\} \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = \emptyset \end{array} \right\}$$

$$S'_o = \left\{ \begin{array}{r} \mathit{Def} = \text{used}(ov_1) \\ \mathit{Undef} = \emptyset \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = \emptyset \end{array} \right\} = \left\{ \begin{array}{r} \mathit{Def} = \{\mathit{Selector}(\texttt{p\#greeting}), \\ \qquad \mathit{Selector}(\texttt{p\#greeting > span\#subject}), \\ \qquad \mathit{Id}(\texttt{subject})\} \\ \mathit{Undef} = \emptyset \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = \emptyset \end{array} \right\}$$

By G-Overlay,

$$[\![ov_1]\!] = (S_i', S_o')$$

By G-Reject,

$$\begin{aligned}
S_i^1 &= S_i' \cup \{\mathit{Undef} = \{\mathit{Id}(\texttt{subject})\}\} \\
&= \left\{ \begin{array}{r} \mathit{Def} = \{\mathit{Selector}(\texttt{p\#greeting})\} \\ \mathit{Undef} = \{\mathit{Selector}(\texttt{p\#greeting > span\#subject}), \mathit{Id}(\texttt{subject})\} \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = \emptyset \end{array} \right\} \\
S_o^1 &= S_o' \cup \Big\{ \mathit{Undef} = \big(\{\mathit{Id}(\texttt{subject})\} \setminus \\
&\qquad \{\mathit{Selector}(\texttt{p\#subject}), \mathit{Selector}(\texttt{p\#greeting > span\#subject}), \\
&\qquad \mathit{Id}(\texttt{subject})\}\big) \Big\} \\
&= \left\{ \begin{array}{r} \mathit{Def} = \{\mathit{Selector}(\texttt{p\#greeting}), \mathit{Id}(\texttt{subject}), \\ \qquad \mathit{Selector}(\texttt{p\#greeting > span\#subject})\} \\ \mathit{Undef} = \emptyset \\ \mathit{Clean} = \emptyset \\ \mathit{Frozen} = \emptyset \end{array} \right\}
\end{aligned}$$

In words, the effect of OV1 is "for any document containing `p#greeting` and not containing `p#greeting > span#subject` or any node with id `subject`, OV1 will produce a document that contains `p#greeting`, `p#greeting > span#subject` and a node with id `subject`". Notice that the *Id*(`subject`) resources have been added correctly and entirely automatically, thanks to defs* encoding of HTML-specific constraints.

Repeating this process with OV3′ results in a very similar derivation:

$$\begin{aligned}
\text{reqs}(ov_3) &= \{\mathit{Selector}(\texttt{p\#greeting})\} \\
\text{defs}(ov_3) &= \{\mathit{Selector}(\texttt{p\#greeting > span\#modifier}), \mathit{Id}(\texttt{modifier})\} \\
\text{used}(ov_3) &= \{\mathit{Selector}(\texttt{p\#greeting}), \mathit{Selector}(\texttt{p\#greeting > span\#modifier}), \mathit{Id}(\texttt{modifier})\}
\end{aligned}$$

By Overlay-Interface,

$$S_i' = \left\{ \begin{array}{r} Def = \text{reqs}(ov_3) \\ Undef = \text{defs}(ov_3) \\ Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\} = \left\{ \begin{array}{r} Def = \{Selector(\texttt{p\#greeting})\} \\ Undef = \{Selector(\texttt{p\#greeting > span\#modifier}), \\ \qquad Id(\texttt{modifier})\} \\ Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\}$$

$$S_o' = \left\{ \begin{array}{r} Def = \text{used}(ov_3) \\ Undef = \emptyset \\ Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\} = \left\{ \begin{array}{r} Def = \{Selector(\texttt{p\#greeting}), \\ \qquad Selector(\texttt{p\#greeting > span\#modifier}), \\ \qquad Id(\texttt{modifier})\} \\ Undef = \emptyset \\ Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\}$$

By G-Overlay,

$$[\![ov_3]\!] = (S_i', S_o')$$

By G-Reject,

$$S_i^3 = S_i' \cup \{Undef = \{Id(\texttt{modifier}, Id(\texttt{subject}))\}\}$$

$$= \left\{ \begin{array}{r} Def = \{Selector(\texttt{p\#greeting})\} \\ Undef = \{Selector(\texttt{p\#greeting > span\#modifier}), Id(\texttt{modifier}), Id(\texttt{subject})\} \\ Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\}$$

$$S_o^3 = S_o' \cup \big\{Undef = \big(\{Id(\texttt{modifier}), Id(subject)\} \setminus \{Selector(\texttt{p\#modifier}), Selector(\texttt{p\#greeting > span\#modifier}), Id(\texttt{modifier})\}\big)\big\}$$

$$= \left\{ \begin{array}{r} Def = \{Selector(\texttt{p\#greeting}), Id(\texttt{modifier}), \\ \qquad Selector(\texttt{p\#greeting > span\#modifier})\} \\ Undef = \{Id(\texttt{subject})\} \\ Clean = \emptyset \\ Frozen = \emptyset \end{array} \right\}$$

Comparing this to the prior versions obtained informally in Section 5.5.3, these are more informative in giving the full selectors for all nodes, e.g., `p#greeting > span#modifier` rather than just the `span#modifier` piece.

To compose (S_i^1, S_o^1) sequentially with (S_i^3, S_o^3), use rule S-GUARD to lift these state pairs into the judgement on compositions, then use S-SEQUENCE to compose them. Looking at the premises of the latter, observe that S' is unneeded (since S-GUARD does not use it). The remaining four guards of S-SEQUENCE are precisely Eqs. (5.3) to (5.6). The conclusions from before carry over, and therefore $S'.Def \cap S_i^3.Undef = S_i^3.Def \cap S_i^3.Undef = \{Id(\texttt{subject})\} \neq \emptyset$, so OV1 and OV3′ cannot be composed in that order. In the reverse order, the four tests succeed, thereby yielding the same, correct $(S_i^{3,1}, S_o^{3,1})$ as before.

(The attentive reader might notice that no conflict is computed between the overlays over the resource *Selector*(`p#greeting > span#subject`), even though this resource goes hand-in-hand with *Id*(`subject`). The system described here does not include "resource inference", which constructs "similar" resources for the ones given. Such heuristics are largely unnecessary, work poorly in the face of `CSS` selector intersection, and can be added if experience shows they are truly needed in some circumstances.)

B.2 Manually Resolved Overlay False-positives

Each of the following selectors `a#b > c#d` represents a `XUL` overlay excerpt of the form

⟨**a** id="b"⟩
  ...new code overlaying `a#b`*...*
  ⟨**c** id="d"⟩
    ...new code overlaying `c#d`*...*
  ⟨/**c**⟩
⟨/**a**⟩

that must be rewritten to the more explicit and more correct form

⟨**a** id="b"⟩
  ...new code overlaying `a#b`*...*
⟨/**a**⟩
⟨**c** id="d"⟩
  ...new code overlaying `c#d`*...*
⟨/**c**⟩

The following list contains all mistakes of this form that can be detected without even looking at the base Firefox document:

Appearance/06-speeddial: popupset#mainPopupSet > popup#contentAreaContextMenu

Appearance/29-tab_popup: window#main-window > popupset#mainPopupSet

Bookmarks/20-taboo: toolbox#navigator-toolbox > toolbarpalette#BrowserToolbarPalette

Bookmarks/43-wired_marker: hbox#browser > vbox#appContent

Bookmarks/44-google_bookmarks_button: window#main-window > stringbundleset#stringbundleset

DownloadManagers/07-download_statusbar: window#main-window > stringbundleset#stringbundleset,
window#main-window > keyset#mainKeyset, window#main-window > vbox#browser-bottombox,
window#main-window > vbox#browser-bottombox > statusbar#status-bar,
window#extensionsManager > stringbundleset#extensionsSet,
window#extensionsManager > keyset#extensionsKeys

DownloadManagers/21-mr_tech_toolkit: window#extensionsManager > keyset#extensionsKeys,
commandset#mainCommandSet > command#Tools:Sanitize

DownloadManagers/32-custom_download_manager: window#main-window > commandset#mainCommandSet,
window#main-window > broadcasterset#mainBroadcasterSet

DownloadManagers/35-download_sort: window#extensionsManager > keyset#extensionsKeys

Feeds/09-blogrovr: window#main-window > broadcasterset#mainBroadcasterSet

Feeds/16-shareaholic: toolbox#navigator-toolbox > toolbarpalette#BrowserToolbarPalette

Feeds/20-sage: toolbox#navigator-toolbox > toolbarpalette#BrowserToolbarPalette

Feeds/41-digg_toolbar: window#main-window > stringbundleset#stringbundleset,
window#main-window > toolbox#navigtor-toolbox

LanguageSupport/05-translator:
toolbox#navigator-toolbox > toolbarpalette#BrowserToolbarPalette

Other/45-wmlbrowser: toolbox#navigator-toolbox > toolbarpalette#BrowserToolbarPalette

Photos/08-fire.fm: window#main-window > stringbundleset#stringbundleset,
window#main-window > popupset#mainPopupSet,
window#main-window > popupset#mainPopupSet > popup#contentAreaContextMenu,
window#main-window > toolbox#navigator-toolbox

Privacy/33-procon_latte: popupset#mainPopupSet > popup#contentAreaContextMenu

Search/10-facebook_toolbar: toolbox#navigator-toolbox > toolbarpalette#BrowserToolbarPalette

Tabs/24-tab_sidebar: hbox#browser > vbox#appContent

Tabs/28-tab_preview: window#main-window > popupset#mainPopupSet

Tabs/33-tab_splitter: hbox#browser > vbox#appContent,
popupset#mainPopupSet > popup#contentAreaContextMenu,
window#main-window > toolbox#navigator-toolbox, window#main-window > hbox#browser,
window#main-window > vbox#browser-bottombox

Webdev/24-colorzilla: statusbar#status-bar > keyset#mainKeyset

Webdev/26-extended_statusbar: `window#main-window > vbox#browser-bottombox`

Webdev/33-picnik: `menu#tools_menu > menupopup#menu_ToolsPopup`

Webdev/42-redirect_remover: `popupset#mainPopupSet > popup#contentAreaContextMenu`

Appearance/39-compact-menu: Note that in this extension, the `menupopup#bookmarksMenuPopup` is being re-parented, as well as overlaid. `menu#bookmarksMenu > menupopup#bookmarksMenuPopup`, `menupopup#bookmarksMenuPopup > menu#bookmarksToolbarFolderMenu` `menu#bookmarksToolbarFolderMenu > menupopup#bookmarksToolbarFolderPopup`

Bookmarks/17-toolbar_buttons:

`toolbarbutton#bookmarks-menu-buton > menupopup#menu_BookmarksPopup` (Note that the `menupopup` is declared as ⟨**menupopup** d="menu_BookmarksPopup"/⟩ *[sic]*, with a typo on the id attribute.)

The following list contains all mistakes of this form that are noticeable only when comparing against the base document:

DownloadManagers/34-pdfescape: `toolbox#navigator-toolbox > toolbar#nav-bar`

Bookmarks/43-wired-marker: `menubar#main-menubar > menu#tools-menu`, `menubar#main-menubar > menu#tools-menu > menupopup#menu_ToolsPopup`

DownloadManagers/06-all-in-one-sidebar: `popupset#mainPopupSet > popup#toolbar-context-menu`, `hbox#browser > vbox#sidebox-box`, `broadcasterset#mainBroadcasterSet > broadcaster#viewBookmarksSidebar`, `broadcasterset#mainBroadcasterSet > broadcaster#viewHistorySidebar`, `broadcasterset#mainBroadcasterSet > broadcaster#viewWebPanelsSidebar`, `toolbarpalette#BrowserToolbarPalette > toolbarbutton#bookmarks-button`, `toolbarpalette#BrowserToolbarPalette > toolbarbutton#history-button`, `toolbarpalette#BrowserToolbarPalette > toolbarbutton#downloads-button`

Appearance/46-no_squint: `commandset#mainCommandSet > command#cmd_fullZoomEnlarge`, `commandset#mainCommandSet > command#cmd_fullZoomReduce`, `commandset#mainCommandSet > command#cmd_fullZoomReset`

DownloadManagers/32-custom_download_manager:

`cinnabdset#mainCommandSet > command#Tools:Downloads`

Tabs/02-tab_mix_plus: `menu#historyUndoMenu > menupopup#historyUndoPopup`

Webdev/26-extended_statusbar: `statusbar#status-bar > statusbarpanel#statusbar-display`

Privacy/16-glubble_family_edition: `commandset#mainCommandSet > command#cmd_close`, `commandset#mainCommandSet > command#cmd_closeWindow`, `toolbarpalette#BrowserToolbarPalette!>!toolbaritem#urlbar-container`,

Feeds/04-cooliris_previews: `window#main-window > popupset#mainPopupSet`

BIBLIOGRAPHY

[1] M. Abadi and L. Lamport. Open systems in TLA. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):507–535, 1995.

[3] ADsafe. Retrieved Nov. 2009. `http://www.adsafe.org/`.

[4] I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. *Electronic Notes in Theoretical Computer Science*, 197(1):45–58, 2008. Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007).

[5] M. Al-Mansari, S. Hanenberg, and R. Unland. On to formal semantics for path expression pointcuts. In *ACM Symposium on Applied Computing (SAC)*, 2008.

[6] J. Aldrich. Open modules: Modular reasoning about advice. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, 2005.

[7] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3): 117–126, Sept. 1987.

[8] T. Anderson. The case for application-specific operating systems. In *IEEE Workshop on Workstation Operating Systems*, 1992.

[9] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155:291–319, Mar. 1996.

[10] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. *Theoretical Computer Science*, 143(1):51 – 72, 1995.

[11] E. Artiaga, A. Serra, and M. Gil. Porting multithreading libraries to an exokernel system. In *ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, 2000.

[12] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[13] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, Aug. 2010.

[14] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, 3362:49–69, 2005.

[15] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *USENIX Security Symposium*, Aug. 2009.

[16] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[17] L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. In M. Okada, B. C. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *International Symposium on Software Security*, volume 2609 of *Lecture Notes in Computer Science*, 2002.

[18] L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. *Software Security — Theories and Systems*, 2609:253–264, 2003.

[19] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[20] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2010.

[21] M. Benedikt and C. Koch. XPath leashed. *ACM Computing Surveys (CSUR)*, 41:3:1–3:54, Jan. 2009.

[22] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM (JACM)*, 55(2):1–79, 2008.

[23] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. J. Eggers. Extensibility safety and performance in the SPIN operating system. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[24] K. Bierhoff and C. Hawblitzel. Checking the hardware-software interface in Spec#. In *ACM SIGOPS Workshop on Programming Languages and Operating Systems (PLOS)*, 2007.

[25] J. Bisbal and B. H. C. Cheng. Resource-based approach to feature interaction in adaptive software. In *ACM SIGSOFT Workshop on Self-Managed Systems*, 2004.

[26] L. Blair, T. Jones, and S. Reiff-Marganiec. A feature manager approach to the analysis of component-interactions. In *IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 2002.

[27] A. Bolour. Notes on the eclipse plug-in architecture. Written July 2003. `http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html`.

[28] A. Borning, R. Lin, and K. Marriott. Constraints for the web. In *ACM International Conference on Multimedia*, 1997.

[29] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin. The feature interaction problem in telecommunications systems. In *International Conference on Software Engineering for Telecommunication Switching Systems (SETSS)*, July 1989.

[30] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010.

[31] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code bubbles: a working set-based interface for code understanding and maintenance. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 2010.

[32] C. Breuel and F. Reverbel. Join point selectors. In *ACM Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2007.

[33] G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *International Conference on Concurrency Theory (CONCUR)*, 2000.

[34] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11:481–494, Oct. 1964.

[35] M. Cain. Managing run-time interactions between call-processing features [intelligent networks]. *IEEE Communications Magazine*, 30(2):44–50, Feb. 1992.

[36] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115 – 141, 2003.

[37] M. Chechik, S. Easterbrook, and B. Devereux. Model checking with multi-valued temporal logics. *IEEE International Symposium on Multiple-Valued Logic (ISMVL)*, 0:187–192, 2001.

[38] J. Cheney. Satisfiability algorithms for conjunctive queries over trees. In *International Conference on Database Theory (ICDT)*, 2011.

[39] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[40] J. Clements. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University, 2005.

[41] J. Clements, A. Sundaram, and D. Herman. Implementing continuation marks in JavaScript. In *Scheme and Functional Programming Workshop*, 2008.

[42] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Workshop on Foundations of Aspect Languages (FOAL)*, 2002.

[43] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2000.

[44] P. Costanza. Dynamically scoped functions as the essence of AOP. *SIGPLAN Notices*, 38(8): 29–36, 2003.

[45] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *IEEE Symposium on Security and Privacy (Oakland)*, 2006.

[46] J. Dagit. Type-correct changes — a safe approach to version control implementation. Master's thesis, Oregon State University, June 2009. `http://blog.codersbase.com/2009/03/type-correct-changes-safe-approach-to.html`.

[47] K. Dangoor, I. Awad, A. Berlin, A. Breitkreuz, D. Friesen, W. Garland, K. Kowal, D. Landolt, P. Michaux, G. Moschovitis, M. O'Brien, T. Robinson, H. Wallnoefer, M. Wilson, O. Zara, C. Zumbrunn, and K. Zyp. CommonJS. Retrieved June 2011. `http://www.commonjs.org/`.

[48] D. S. Dantas and D. Walker. Harmless advice. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan. 11–13 2006.

[49] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. PolyAML: a polymorphic aspect-oriented functional programming language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Sept. 2005.

[50] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):1–60, 2008.

[51] B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2008.

[52] A. S. de Oliveira. Rewriting-based access control policies. *Electronic Notes in Theoretical Computer Science*, 171(4):59–72, 2007. Workshop on Security and Rewriting Techniques (SecReT 2006).

[53] A. S. de Oliveira, E. K. Wang, C. Kirchner, and H. Kirchner. Weaving rewrite-based access control policies. In *ACM Workshop on Formal Methods in Security Engineering*, 2007.

[54] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Computing Surveys (CSUR)*, 34(4):450–468, 2002.

[55] J. DeTreville. Making system configuration more declarative. In *Hot Topics in Operating Systems (HotOS)*, 2005.

[56] M. Dhawan and V. Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Annual Computer Security Applications Conference (ACSAC)*, 2009.

[57] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In R. Draves and R. van Renesse, editors, *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[58] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION)*, 2001.

[59] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *ACM SIGPLAN-SIGSOFT Conference on Generative Programming and Component Engineering*, 2002.

[60] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2004.

[61] L. du Bousquet, F. Ouabdesselam, J. L. Richier, and N. Zuanon. Feature interaction detection using a synchronous approach and testing. *Computer Networks*, 32(4):419 – 431, 2000.

[62] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, 2006.

[63] R. Echahed and F. Prost. Security policy in a declarative style. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, 2005.

[64] ECMA International. ECMAScript language specification, 5th edition. Written June 2011. `http://www.ecmascript.org/`.

[65] A. Edwards and G. Heiser. Components + security = OS extensibility. In *Australasian Conference on Computer Systems Architecture (ACSAC)*, 2001.

[66] B. Eich, C. Jones, M. Shaver, and A. Gal. B2g. Retrieved Aug. 3, 2011. `https://wiki.mozilla.org/B2G`.

[67] E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[68] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Hot Topics in Operating Systems (HotOS)*, 1995.

[69] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an operating system architecture for application-level resource management. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[70] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004. `http://portal.acm.org/citation.cfm?id=997617`.

[71] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *New Security Paradigms Workshop (NSPW)*, 2000.

[72] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy (Oakland)*, 2000.

[73] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[74] Ú. Erlingsson, B. Livshits, and Y. Xie. End-to-end web application security. In *Hot Topics in Operating Systems (HotOS)*, 2007.

[75] D. Evans. *Policy-Drected Code Safety*. PhD thesis, Massachusetts Institute of Technology, 1999. `http://www.cs.virginia.edu/~evans/phd-thesis/abstract.html`.

[76] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy (Oakland)*, 1999.

[77] A. Faaborg. Introducing operator. Written Dec. 2006. `http://labs.mozilla.com/2006/12/introducing-operator/`.

[78] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *ACM Symposium on Applied Computing (SAC)*, 2010.

[79] A. P. Felty and K. S. Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(1):3–27, 2003.

[80] FFsniFF (Firefox sniffer). Written 2008. `http://azurit.elbiahosting.sk/ffsniff/`.

[81] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, Research Institude for Advanced Computer Science, 2000.

[82] M. Finkle, B. King, N. Ponomarev, J. Resig, P. Ringnalda, and R. Sayre. FUEL. Retrieved July 2011. `https://wiki.mozilla.org/FUEL`.

[83] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. `patch (1)` considered harmful. In *Hot Topics in Operating Systems (HotOS)*, 2005.

[84] M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (or revenge of the son of the lisp machine). In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 1999.

[85] M. Flatt, G. Yu, R. B. Findler, and M. Felleisen. Adding delimited and composable control to a production programming environment. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2007.

[86] M. Fredrikson and B. Livshits. RePriv: Re-envisioning in-browser privacy. Technical report, Microsoft Research, Aug. 2010. http://research.microsoft.com/apps/pubs/default.aspx?id=137038.

[87] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[88] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Transactions on Computer Systems (TOCS)*, 20(1):49–83, Feb. 2002.

[89] D. Glazkov. Component model: Landing experimental shadow DOM API in WebKit. Written June 2011. http://lists.w3.org/Archives/Public/public-webapps/2011AprJun/1345.html.

[90] D. Glazkov. What the heck is Shadow DOM? Written Jan. 2011. http://glazkov.com/2011/01/14/what-the-heck-is-shadow-dom/.

[91] D. Glazman. Search – webchunks. Retrieved Nov. 2009. http://www.glazman.org/weblog/dotclear/index.php?q=webchunks.

[92] Google. Content scripts. Retrieved July 6, 2011. http://code.google.com/chrome/extensions/content_scripts.html.

[93] Google. PPAPI: Pepper plugin API. Retrieved July 8, 2001. http://code.google.com/p/ppapi/.

[94] Google. GmailGreasemonkey10API: API reference for version 1.0 of the experimental Gmail Greasemonkey API. Written Feb. 2010. http://code.google.com/p/gmail-greasemonkey/wiki/GmailGreasemonkey10API.

[95] Google. Chrome web store: Extensions. Retrieved June 29, 2011. https://chrome.google.com/webstore?category=ext.

[96] R. Grimm and B. N. Bershad. Separating access control policy, enforcement, and functionality in extensible systems. *ACM Transactions on Computer Systems (TOCS)*, 19(1):36–70, 2001.

[97] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, Aug. 2009.

[98] S. Guarnieri and B. Livshits. GULFSTREAM: staged static analysis for streaming javascript applications. In *USENIX Conference on Web Application Development (WebApps)*, 2010.

[99] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010.

[100] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy (Oakland)*, May 2011.

[101] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2008.

[102] P. B. Hansen. The nucleus of a multiprogramming system. *Communications of the ACM (CACM)*, 13(4):238–241, 1970.

[103] M. A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1978.

[104] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2007.

[105] J. D. Hay and J. M. Atlee. Composing features and resolving interactions. *SIGSOFT Software Engineering Notes*, 25(6):110–119, 2000.

[106] C. Hofer and K. Ostermann. On the relation of aspects and monads. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2007.

[107] P. Hui and J. Riely. Typing for a minimal aspect language: preliminary report. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2007.

[108] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, B. Steensgaard, D. Tarditi, and T. Wobber. Sealing OS processes to improve dependability and safety. In *ACM SIGOPS-EuroSys European Conference on Computer Systems (EuroSys)*, 2007.

[109] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review (OSR)*, 41(2):37–49, 2007.

[110] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad new OS research: challenges and opportunities. In *Hot Topics in Operating Systems (HotOS)*, 2005.

[111] N. Hurst, K. Marriott, and P. Moulder. Cobweb: a constraint-based WEB browser. In *Australasian Computer Science Conference (ACSC)*, 2003.

[112] E. Ian Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML. Retrieved July 6, 2011. `http://dev.w3.org/html5/spec/Overview.html`.

[113] A. Igarashi and N. Kobayashi. Resource usage analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[114] C. Jackson and A. Barth. Beware of finer-grained origins. In *In Web 2.0 Security and Privacy (W2SP 2008)*, 2008.

[115] J. Jacobson. A formalization of darcs patch theory using inverse semigroups. Technical Report (09-83), UCLA Computational and Applied Mathematics, Oct. 2009. `http://www.math.ucla.edu/~jjacobson/patch-theory/`.

[116] F. Jacquemard, M. Rusinowitch, and L. Vigneron. Compiling and verifying security protocols. *Logic for Programming and Automated Reasoning*, 1955:535–554, 2000.

[117] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[118] R. Jagadeesan, C. Pitcher, and J. Riely. Open bisimulation for aspects. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2007.

[119] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodík. Parallelizing the Web Browser. In *Hot Topics in Parallelism (HotPar)*, Mar. 2009.

[120] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[121] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2008.

[122] D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, 1998.

[123] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[124] J. Kim. Activities and WebSlices in Internet Explorer 8. Written Mar. 2008. `http://blogs.msdn.com/ie/archive/2008/03/06/activities-and-webslices-in-internet-explorer-8.aspx`.

[125] D. E. Knuth. *The TeXbook*. Addison-Wesley, 1984.

[126] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981.

[127] S. Kojarski and D. H. Lorenz. Pluggable AOP: designing aspect mechanisms for third-party composition. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[128] S. Kojarski and D. H. Lorenz. Identifying feature interactions in multi-language aspect-oriented frameworks. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2007.

[129] S. Kojarski, D. H. Lorenz, S. Kojarski, and D. H. Lorenz. Awesome: an aspect co-weaving system for composing multiple aspect-oriented extensions. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007.

[130] M. Labs. Personas. Retrieved July 2011. `http://mozillalabs.com/personas/`.

[131] M. Labs. Firefox Sync. Written Mar. 2011. `https://addons.mozilla.org/en-US/firefox/addon/firefox-sync/`.

[132] M. S. Lam, M. Martin, B. Livshits, and J. Whaley. Securing web applications with static and dynamic information flow tracking. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 2008.

[133] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[134] L. Lamport. *LaTeX: A document preparation system, user's guide and manual*. Addison-Wesley Professional, 2nd edition, 1994.

[135] B. Lampson. On reliable and extendible operating systems. In *NATO Conference on Techniques in Software Engineering*, 1971.

[136] B. S. Lerner and D. Grossman. Language support for extensible web browsers. In *ACM Analysis and Programming Languages for Web Applications and Cloud Applications (APLWACA)*, 2010.

[137] B. S. Lerner, H. Venter, and D. Grossman. Supporting dynamic, third-party code customizations in JavaScript using aspects. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2010.

[138] B. S. Lerner, B. Burg, H. Venter, and W. Schulte. C3: An experimental, extensible, reconfigurable platform for html-based applications. In *USENIX Conference on Web Application Development (WebApps)*, June 2011.

[139] T. Leschke. Achieving speed and flexibility by separating management from protection: embracing the exokernel operating system. *ACM SIGOPS Operating Systems Review (OSR)*, 38(4):5–19, 2004.

[140] N. Lesiecki. AOP@Work: Enhance design patterns with AspectJ, part 1. Written May 2005. `http://www.ibm.com/developerworks/java/library/j-aopwork5/index.html`.

[141] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002.

[142] H. C. Li, S. Krishnamurthi, and K. Fisler. Modular verification of open features using three-valued model checking. *Automated Software Engineering (ASE)*, 12(3):349–382, 2005.

[143] J. Liedtke. On micro-kernel construction. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[144] J. Liedtke. Toward real microkernels. *Communications of the ACM (CACM)*, 39(9):70–77, 1996.

[145] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, Feb. 2005.

[146] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *European Symposium on Research in Computer Security (ESORICS)*, volume 3679 of *Lecture Notes in Computer Science*, Sept. 2005.

[147] J. Ligatti, D. Walker, and S. Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 63(3):240–266, 2006.

[148] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):1–41, 2009.

[149] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat. Configurable memory protection by aspects. In *ACM SIGOPS Workshop on Programming Languages and Operating Systems (PLOS)*, 2007.

[150] P. A. Loscocco and S. D. Smalley. Meeting critical security objectives with security-enhanced linux. In *Ottawa Linux Symposium*, 2001.

[151] I. Lynagh. Darcs patch theory (more or less). Originally posted to darcs-users mailing list, Sept. 2008. `http://lists.osuosl.org/pipermail/darcs-users/2008-August/013040.html`.

[152] G. Maone. Dear AdBlock Plus and NoScript users, dear Mozilla community. Written May 2009. `http://hackademix.net/2009/05/04/dear-adblock-plus-and-noscript-users-dear-mozilla-community/`.

[153] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Sept. 2005.

[154] A. Metzger. Feature interactions in embedded control systems. *Computer Networks*, 45(5): 625–644, 2004.

[155] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[156] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *International Conference on the World Wide Web (WWW)*, 2010.

[157] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[158] Microsoft. Add-ons gallery. Retrieved 2009. `http://www.ieaddons.com/en/`.

[159] Microsoft. Trees in WPF. Retrieved July 2011. `http://msdn.microsoft.com/en-us/library/ms753391.aspx`.

[160] Microsoft. Developing Visual Studio extensions. Written 2011. http://msdn.microsoft.com/en-us/library/dd885119.aspx.

[161] Microsoft. VSPackage essentials. Written 2011. http://msdn.microsoft.com/en-us/library/bb165754.aspx.

[162] Microsoft Developer Network. Browser extensions. Retrieved Mar. 2009. http://msdn.microsoft.com/en-us/library/aa753587(VS.85).aspx.

[163] F. Mittelbach, M. Goossens, J. Braams, D. Carlisle, and C. Rowley. *The LaTeX Companion*. Addison-Wesley Professional, 2nd edition, 2004.

[164] A. Mohta. Get IE8 accelerator in Firefox: Select n go. Written Feb. 2009. http://www.technospot.net/blogs/get-ie-8-accelerators-in-firefox/.

[165] Mozilla. Add-on SDK: Add-on development made easy. Retrieved July 2011. https://addons.mozilla.org/en-US/developers/docs/sdk/1.0/.

[166] Mozilla. XUL overlays. Written Jan. 2010. https://developer.mozilla.org/en/XUL_Overlays.

[167] Mozilla. Mozilla Firefox: Add-ons. Retrieved June 29, 2011. https://addons.mozilla.org/en-US/firefox.

[168] Mozilla. Mozilla Firefox: Extensions. Retrieved June 29, 2011. https://addons.mozilla.org/en-US/firefox/extensions/.

[169] Mozilla. Mozilla Labs: Chromeless browser. Retrieved June 29, 2011. https://mozillalabs.com/chromeless/.

[170] Mozilla. Mozilla Firefox: Extensions. Retrieved June 29, 2011. http://prism.mozillalabs.com/.

[171] mozillaZine. Dev : Extending the chrome protocol. Retrieved July 2011. http://kb.mozillazine.org/Dev_:_Extending_the_Chrome_Protocol.

[172] G. Muller, Y. Padioleau, J. L. Lawall, and R. R. Hansen. Semantic patches considered helpful. *ACM SIGOPS Operating Systems Review (OSR)*, 40(3):90–92, 2006.

[173] M.-J. Nederhof and G. Satta. The language intersection problem for non-recursive context-free grammars. *Information and Computation*, 192(2):172 – 184, 2004.

[174] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding open modules to AspectJ. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2006.

[175] W. Palant. Attention NoScript users. Written May 2009. `http://adblockplus.org/blog/attention-noscript-users`.

[176] M. Plath and M. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1):53–84, 2001.

[177] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1989.

[178] A. Raskin. My dream way to write a Firefox extension. Written Mar. 2009. `http://www.azarask.in/blog/post/my-dream-way-to-write-a-firefox-extension/`.

[179] A. Raskin and the Jetpack development team. Announcing the Jetpack SDK: First milestone release. Written Mar. 2010. `http://mozillalabs.com/jetpack/2010/03/09/announcing-the-jetpack-sdk/`.

[180] A. Raskin, A. Varma, N. Nguyen, and the Jetpack development team. Introducing Jetpack, call for participation. Written May 2009. `http://mozillalabs.com/blog/2009/05/introducing-jetpack-call-for-participation/`.

[181] C. Reis. *Web Browsers as Operating Systems: Supporting Robust and Secure Web Programs*. PhD thesis, University of Washington, 2009. `http://www.charlesreis.com/research/publications/creis-thesis.pdf`.

[182] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *ACM SIGOPS-EuroSys European Conference on Computer Systems (EuroSys)*, 2009.

[183] C. Reis, S. D. Gribble, and H. M. Levy. Architectural principles for safe web programs. In *Hot Topics in Networks (HotNets)*, Nov. 2007.

[184] A. Reynolds, M. E. Fiuczynski, and R. Grimm. On the feasibility of an AOSD approach to linux kernel extensions. In *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2008.

[185] D. Richardson and S. D. Gribble. Maverick: Providing web applications with safe and flexible access to local devices. In *USENIX Conference on Web Application Development (WebApps)*, June 2011.

[186] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. *SIGSOFT Software Engineering Notes*, 29(6):147–158, 2004.

[187] C. Rippert. Protection in flexible operating system architectures. *ACM SIGOPS Operating Systems Review (OSR)*, 37(4):8–18, 2003.

[188] J. Ruderman. Same origin policy for JavaScript. Written Oct. 2010. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`.

[189] Y. Saito and B. Bershad. A transactional memory service in an extensible operating system. In *USENIX Annual Technical Conference (USENIX ATC)*, 1998.

[190] S. B. Sanjabi and C.-H. L. Ong. Fully abstract semantics of additive aspects by translation. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2007.

[191] S. Savage and B. N. Bershad. Issues in the design of an extensible operating system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[192] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[193] J. Scott. How many Firefox users use add-ons? Written Aug. 2009. `http://blog.mozilla.com/addons/2009/08/11/how-many-firefox-users-use-add-ons/`.

[194] J. Scott. How many Firefox users have add-ons installed? 85%! Written June 2011. `http://blog.mozilla.com/addons/2011/06/21/firefox-4-add-on-users/`.

[195] D. Sereni and O. de Moor. Static analysis of aspects. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2003.

[196] S. Siddiqi and J. M. Atlee. A hybrid model for specifying features and detecting interactions. *Computer Networks*, 32(4):471–485, 2000.

[197] J. M. Siskind and B. A. Pearlmutter. First-class nonstandard interpretations by opening closures. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Jan. 17–19 2006.

[198] H. Sivonen. Speculative HTML5 parsing landed. Written Nov. 2009. `http://hsivonen.iki.fi/speculative-html5-parsing/`.

[199] C. Skalka and S. Smith. Static enforcement of security with types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[200] C. Small and M. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference (USENIX ATC)*, 1996.

[201] D. R. Smith. Requirement enforcement by transformation automata. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2007.

[202] K. Sons, F. Klein, D. Rubinstein, S. Byelozyorov, and P. Slusallek. XML3D: interactive 3d graphics for the web. In *International Conference on Web 3D Technology*, 2010.

[203] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the starting problem: device drivers as self-describing artifacts. In *ACM SIGOPS-EuroSys European Conference on Computer Systems (EuroSys)*, 2006.

[204] R. M. Stallman. EMACS the extensible, customizable self-documenting display editor. In *ACM SIGPLAN SIGOA Symposium on Text Manipulation*, 1981.

[205] StumbleUpon. Stumbleupon. Written Mar. 2010. `http://www.stumbleupon.com/sublog/su_chrome_extension/`.

[206] M. Sulzmann. Playing with regular expressions: Intersection. Written Nov. 2008. `http://sulzmann.blogspot.com/2008/11/playing-with-regular-expressions.html`.

[207] The AspectJ Team. The AspectJ programming guide. Written 2003. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

[208] The AspectJ Team. The AspectJ 5 development kit developer's notebook. Written 2005. `http://www.eclipse.org/aspectj/doc/next/adk15notebook/`.

[209] The Caja Team. Caja. Written Nov. 2009. `http://code.google.com/p/google-caja/`.

[210] P. Thiemann. A type safe DOM API. *Database Programming Languages*, 3774:169–183, 2005.

[211] R. Toledo, P. Leger, and E. Tanter. AspectScript: expressive aspects for the web. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2010.

[212] D. Townsend. Why do Firefox updates break add-ons? Written June 2011. `http://www.oxymoronical.com/blog/2011/06/Why-do-Firefox-updates-break-add-ons`.

[213] D. Townsend. Bootstrapped extensions. Retrieved July 8, 2011. `https://developer.mozilla.org/en/Extensions/Bootstrapped_extensions`.

[214] D. Townsend. Unloading JS modules. Written July 2011. http://www.oxymoronical.com/blog/2011/07/Unloading-JS-modules.

[215] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *International Conference on Aspect-Oriented Software Development (AOSD)*, 2003.

[216] M. Vardi. Sometimes and not never re-revisited: on branching versus linear time. *Concurrency Theory (CONCUR)*, 1466:1–17, 1998.

[217] M. Viswanathan. *Foundations for the run-time analysis of software systems*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 2000.

[218] W3C. XML binding language (XBL) 2.0: Candidate recommendation. Written Mar. 2007. http://www.w3.org/TR/xbl/.

[219] W3C. XSL transformations (XSLT) version 2.0. Written Jan. 2007. http://www.w3.org/TR/xslt20/.

[220] G. Wagner, A. Gal, C. Wimmer, B. Eich, and M. Franz. Compartmental memory management in a modern web browser. In *ACM International Symposium on Memory Management (ISMM)*, June 2011.

[221] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2003.

[222] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: a security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4): 341–378, 2000.

[223] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the gazelle web browser. In *USENIX Security Symposium*, Aug. 2009.

[224] M. Wang, K. Chen, and S.-C. Khoo. Type-directed weaving of aspects for higher-order functional languages. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 2006.

[225] G. Washburn and S. Weirich. Good advice for type-directed programming: aspect-oriented programming and extensible generic functions. In *ACM SIGPLAN Workshop on Generic Programming*, 2006.

[226] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: Aspect-oriented JavaScript programming framework for web development. In *Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2009.

[227] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference (USENIX ATC)*, June 2008.

[228] WHATWG. Component model use cases. Retrieved July 13, 2011. `http://wiki.whatwg.org/wiki/Component_Model_Use_Cases`.

[229] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: the kernel of a multiprocessor operating system. *Communications of the ACM (CACM)*, 17(6): 337–345, 1974.

[230] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (Oakland)*, May 2009.

VITA

Benjamin Lerner was born and raised in New York City. He completed his undergraduate studies at Yale University, where in 2004 he received a Bachelor of Science in Computer Science and Mathematics. After working for one year at Microsoft on Windows Vista, he began his graduate work at the University of Washington, focusing on programming languages, web development, and systems. After a fruitful collaboration with Microsoft Research, he completed his Ph.D. in 2011.