Technical Report

UCAM-CL-TR-845
ISSN 1476-2986

Number 845

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Black-box composition of mismatched software components

Stephen Kell

December 2013

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2010 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Christ's College.

ISSN 1476-2986

Black-box composition of mismatched software components

Stephen Kell

Summary

Software is expensive to develop. Much of that expense can be blamed on difficulties in combining, integrating or re-using separate pieces of software, and in maintaining such compositions. Conventional development tools approach composition in an inherently narrow way. Specifically, they insist on modules that are *plug-compatible*, meaning that they must fit together down to a very fine level of detail, and that are *homogeneous*, meaning that they must be written according to the same conventions and (usually) in the same programming language. In summary, modules must have *matched interfaces* to compose. These inflexibilities, in turn, motivate more software creation and concomitant expense: they make programming approaches based on integration and re-use unduly expensive. This means that reimplementation from scratch is often chosen in preference to adaptation of existing implementations.

This dissertation presents several contributions towards lessening this problem. It centres on the design of a new special-purpose programming language, called Cake. This language is specialised to the task of describing how components having *mismatched* interfaces (i.e. not plug-compatible, and perhaps not homogeneous) may be *adapted* so that they compose as required. It is a language significantly more effective at capturing relationships between mismatched interfaces than general-purpose programming languages. Firstly, we outline the language's design, which centres on reconciling interface differences in the form high-level *correspondence rules* which *relate* different interfaces. Secondly, since Cake is designed to be a practical tool which can be a convenient and easily-integrated tool under existing development practices, we describe an implementation of Cake in detail and explain how it achieves this integration. Thirdly, we evaluate Cake on real tasks: by applying it to integration tasks which have already been performed under conventional approaches, we draw meaningful comparisons demonstrating a smaller (quantitative) size of required code and lesser (qualitative) complexity of the code that is required. Finally, Cake applies to a wide range of input components; we sketch extensions to Cake which render it capable of composing components that are *heterogeneous* with respect to a carefully identified set of *stylistic concerns* which we describe in detail.

Acknowledgments

Research is never finished. Rather than thanking people for anything they helped me complete, I'd like to thank a lot of people without whom I either couldn't have started or couldn't have continued.

Firstly, I thank my supervisor, David Greaves, for the keen interest he showed in my research goals, balanced with a quiet faith in letting me find out where they led. I am grateful for his patient feedback when it became time to put this dissertation together.

I thank Steven Hand for letting me through the door, finding funding, and always being available to offer honest and insightful feedback. Jon Crowcroft deserves special thanks, for his boundless positivity and ability to give sound feedback at a moment's notice. Alan Mycroft also provided generous feedback at several decisive points. I also thank Sandy Fraser for his generosity and encouragement during a valuable summer in Princeton.

Certain people deserve special mention. Michael Dales, Alastair Reid and Simon Ford gave me early encouragement which, unbeknownst to them, was essential in convincing me that I could and should pursue my research ambitions. Michael Hicks appeared at a critical time; his insight and positivity were vital in keeping me going. Somewhat later, Yvonne Coady rescued me from a trough out of which I might otherwise not have emerged.

My many hugely talented colleagues and friends in the Atlas Room have indulged my eccentricities on a daily basis over the years. In particular, I thank Chris Purcell, Henry Robinson, Steven Smith, Theodore Hong, Mark Williamson, Derek Murray, Nishanth Sastry, Periklis Akritidis, Chris Smowton, Malte Schwarzkopf and Malcolm Scott. I also thank all the other members of the Networks and Operating Systems group for their willingness to listen to my increasingly off-topic ideas, and the Cambridge Programming Research Group, especially Dominic Orchard, Robin Message, Max Bolingbroke and Boris Feigin, for letting me pretend to belong to their group when it suited me. Finally, a PhD student needs post-docs: Scott Owens, Kathy Gray, David Evans, David Eyers, Mike Dodds and Alan Lawrence all gave vital advice and encouragement. Vaughan Wittorff, while not a post-doc, belongs in this list too.

I thank the Computer Laboratory support staff for the department's efficiency and generous resourcing; EPSRC and Cambridge Philosophical Society for essential funding; Christ's College, ACM SIGPLAN, ACM SIGSOFT, EuroSys and the Royal Academy of Engineering for equally essential travel funding. I also thank various constituencies of Christ's College for enhancing my life in various ways, with special mentions to the Graduate Society, the porters and the catering staff. Finally, I thank my family and friends for their support, for providing distractions, and generally for being more important than the contents of this dissertation.

‘If we succeed in making an Intergalactic Network, then our main problem will be learning to communicate with Aliens.’

J.C.R. Licklider

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are among the most complex artificial systems in existence. This complexity is the root of the the notorious expense of software. One of the most economically significant costs of software, aside from the machines that run it, is the human cost of software construction and maintenance, measured in human programmer hours. Reducing this cost is the motivation of a great proportion of research in programming languages and software engineering, including the work presented in this dissertation.

Compositionality has long been recognised as a powerful approach to managing the complexity of software construction and maintenance. The idea of *software components* has been established for over forty years, following McIlroy's influential presentation [McIlroy 1969]. Unfortunately, the extent of composition practised in development reality continues to fall short of its apparent potential. Following several waves of optimistic research and other literature proclaiming the potential of *software re-use*, throughout the 1980s and 1990s, more pessimistic commentaries have emerged, claiming that re-use is only feasible for certain classes of component [Lampson 2004], or certain problem domains [Veldhuizen 2005] or under limited temporal variation in requirements [Glass 1998].

Considered with the benefit of forty years' hindsight, McIlroy's vision of a software component catalogue is optimistic in at least two senses. Specifically, the collected experience gathered over the intervening years has refuted McIlroy's suppositions concerning both *diversity* and *change*.

1.1 Diversity and change

Diversity in general McIlroy's envisaged reality assumes that whatever the developer's requirement might be, a precise match would be available in some large and well-known catalogue of pre-existing components. This component would be matched not only in desired performance characteristics and hardware compatibility, but also in interface details such as error-reporting conventions, robustness preconditions, the concrete form of input and output data structures, and so on. In other words, it envisages that sheer diversity in the available *provision* of components would be sufficient to satisfy diverse *requirements*. In particular, this diverse provision would satisfy diverse requirements not only of the functional and extra-functional kinds, but also of *interfacing* requirements: how components interact with the wider system.

Interface diversity specifically Concerning diversity, McIlroy lists several dimensions over which components may differ, other than their function. These dimensions can be grouped into two sets: performance characteristics (e.g. time–space complexity of a particular operation) and interface details (including error-reporting style, input and output data structures, input checking contracts, and so on). This is not a disjoint grouping, in that many interface details also have a systematic effect on performance. For example, adding input validation reduces speed, while choosing a different input data structure might allow more efficient processing. However, it is convenient to treat these classes of diversity as separate problems, for two reasons. Firstly, differing interface details rarely imply radical differences in the core of a component's implementation. By contrast, major performance differences usually result from at least moderately deep changes. Interfaces are superficial, and it is often possible to make one interface look like another without profound changes to component implementations (and indeed, several contemporary practices exploit this possibility, as we will see in §1.2). A second reason is that performance dimensions are relatively simply described: usually only a few dimensions of performance, such as processing time, throughput, memory requirements or power consumption, are of interest. It follows that the effective diversity in performance requirements is low. By contrast, interfaces model some abstract domain, and as illustrated by Kent [1989], the potential diversity among models of the same domain is huge. Therefore, interface diversity appears both especially significant and especially tractable: there is a lot of diversity to mitigate, and at the same time, a lot of that may be mitigated by relatively superficial changes to components.

Change in general As with diversity, McIlroy's vision includes no consideration of change. Changing requirements are commonplace in software [McConnell 1996]. Change in provision, among the available population of software components, is also a familiar phenomenon. As new components are developed and others cease to be maintained, a component which was previously a good choice may soon become a candidate for replacement. Unfortunately, such changes cannot always be anticipated. As a result, surrounding code embodies commitments to components selected earlier. Reversing these commitments is an expensive process, because they easily permeate large quantities of code.

Interface change specifically As with diversity, change in interfaces is both a more significant and a more tractable problem than change in functional or extrafunctional properties of software components. In general, reliability and performance of a given actively-maintained component will improve over time, as bug-fixes are incorporated and optimisations made. These changes are invariably welcomed by users of the component, and need not entail any changes to the user's code. Rather, users only need to change their code if *interface details*, rather than implementation internals, are changed. However, such changes *do* occur often—perhaps to accommodate those same bug-fixes or performance improvements, but perhaps simply to improve the interface's design for future users.

When summed over a prolonged stretch of a component's lifetime, this kind of ongoing change constitutes a very significant effort.

Integration effort Developers are used to diversity and change in interfaces. They are aware that available alternatives among candidate components will be limited, so rather than finding pre-existing components which exactly match requirements, some amount of *integration* effort will be necessary when using any preexisting component. Similarly, developers are familiar with the ongoing maintenance effort required to accommodate changing requirements and changing provision. Our goal in this dissertation is to reduce the human effort which must be expended on both of these kinds of task.

Interface mismatch By focussing on interface details and avoiding direct consideration of other extrafunctional concerns such as performance, we can treat both interface diversity and interface change as instances of the same underlying abstract problem: interface mismatch. Stated simply, this includes any problem where composition of components is required, but where their interfaces do not match. Conventional means of tackling such problems are laborious. This means that integration remains expensive, as described in the next section.

1.2 Existing practice

Practitioners have adopted a number of techniques for building software in a componentised fashion in the presence of (or risk of) interface mismatch. Unfortunately, these techniques have many shortcomings. We now briefly survey these techniques.

1.2.1 Library-based development

The concept of software libraries has existed for almost as long as the stored-program computer. McIlroy's vision can be understood as the realisation of a *complete* and *categorised* suite of libraries for any recurring programming task. By contrast, reality thus far has provided an incomplete and ill-organised collection of libraries. Programmers make use of this opportunistically rather than systematically.

Standard libraries In the last fifteen years, popular programming languages have offered ever-larger standard libraries. For example, the standard libraries provided by the Java [Arnold et al. 2005] or Python [Van Rossum and Drake Jr. 2003] languages are many times the size of those defined by older languages including C [Kernighan and Ritchie 1988], C++ [Stroustrup 1997] or Fortran. Correspondingly, programmers in these more modern languages generally write less utility code and concentrate more on application-domain concerns. Using standard libraries is convenient because they are well-known, slow to change, and unique with respect to their host language. In other words, having

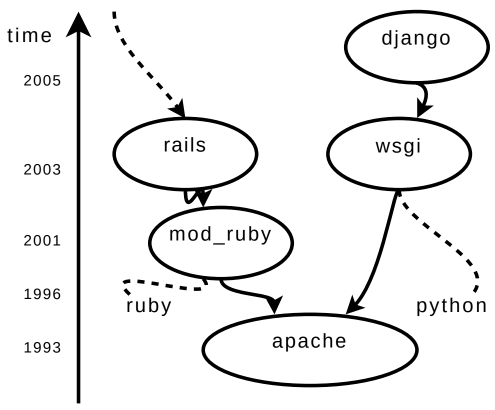


Figure 1.1: Software growing in vertically-dependent silos over time

adopted a particular programming language, using its library represents no greater commitment for the programmer, who may adopt it from the outset, be confident that it will remain well-supported for decades, and have little motivation to seek alternatives.

Application-domain libraries Libraries also exist in the application domain. In contrast with languages' standard libraries, application-domain libraries represent a population of changing components, in which many *alternative* choices are available. Committing to a particular choice of library at development time, and writing code against its application programming interfaces (APIs), means accepting that should a different library represent a better alternative later, considerable integration effort will be required to make the switch, since details of the original library's APIs will be scattered throughout the code.

Emergent behaviour An emergent consequence of library-based development is that software grows "upwards" to form dependency structures often called "stacks" or *silos.* "Upwards" here implies a vertical axis encoding both dependency and time, with higher-positioned components both *depending on* lower ones and being *developed after* them, as in Fig. 1.1. Within a silo, each piece of software is written "for" some specific pre-existing underlying piece—that is, targetting its interfaces directly. Frequently, users or developers wishing to add a particular feature to their chosen stack of code observe that code for the feature is already available in some other stack, but is effectively unusable because it is "trapped": the integration effort of linking it with the intended target codebase is infeasibly great. (This phenomenon is particularly easily observed when looking at how the availability of any given application-level software is commonly restricted to a narrow range of candidate infrastructure software—such as a particular operating system, windowing toolkit, desktop suite, text editor, programming language, database management system, and so on.)

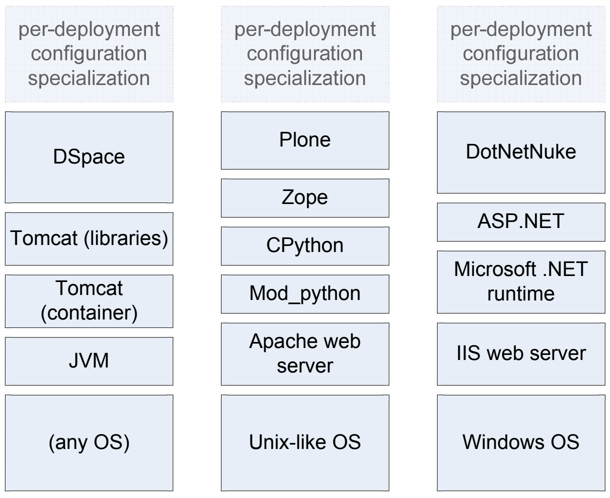


Figure 1.2: Some example silos: web-based content management systems

1.2.2 Abstraction layers

Software architects who foresee the need to support multiple or changing selections of libraries often incorporate an *abstraction layer* into their design. This is useful for accommodating both diversity and change. By writing most code against an abstract interface agnostic to a particular underlying API, that code is insulated from details of that API—from *changes* to those details and from *diversity* among candidate selections of that API. Fig. 1.3 illustrates this design.

Unfortunately, abstraction layers are often an incomplete solution, for two reasons. Firstly, abstraction layers require anticipation. It may only become apparent *later* that multiple or changing selections of libraries need to be accommodated—in which case it is too late to incorporate an abstraction layer. Secondly, abstraction layers represent an up-front design overhead which expedience often overrules, even if the possibility of requiring one in future is anticipated. Programmers are notoriously prone to optimism and wishful thinking [Beck and Andres 2004; McConnell 1996]. Abstraction layers cannot be incorporated in an evolutionary fashion: since their purpose is to isolate specific API details from leaking into a wider codebase, they rely on disciplined adoption from the start. Where this discipline is not applied, laborious programming effort may be required later.

"Plug-in" systems found in many applications (such as most popular web browsers) are an extension of the abstraction layer design: an application presenting its abstraction layer interface through a level of run-time indirection (e.g. a dispatch table) can then load "plug-in" implementations of this interface dynamically into the running program. For our purposes, this dynamic loading does not affect the problem of interface mismatch.

1.2.3 Reimplementation

When integration of existing code is not a practical option, reimplementation of the desired functionality from scratch is often the chosen fallback. This brings obvious expense. In

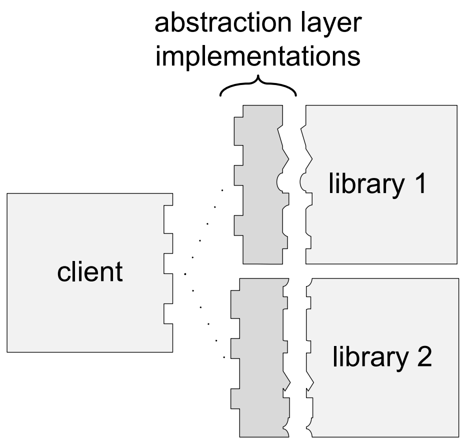


Figure 1.3: An abstraction layer design

cases of simple components or large degrees of mismatch, it may nevertheless be the cheapest option. This dissertation's contributions will effectively shift this calculation somewhat in favour of integration, by reducing the associated cost.

We mention reimplementation because it is unquestionably a reality of software development and a source of much avoidable expense. However, it is very difficult to quantify the extent of "avoidable" reimplementation. Programmers rarely document their motivation, and invariably no reimplementation is identical to pre-existing code feature-for-feature, nor exactly matched in its performance characteristics and other extrafunctional properties. The availability of a few competing implementations of the same functionality is often a beneficial force, whether for marketplace competition between commercial software houses or as a source of competitive incentive among open-source coders.

Anecdotally, however, avoidable reimplementation appears common. It can be observed in the same way as the silo phenomenon (§1.2.1): parallel implementations of similar functionality are often evidenced in the contexts of different supporting libraries. Fig. 1.2 illustrates this with three example silos, each implementing a web-based content management system headed by popular software packages (DSpace[1], Plone[2] and DotNetNuke[3]). The key observation is that portability between silos is extremely limited: with a few exceptions, pieces of any silo cannot be transposed into either of the other silos without substantial development effort.

1.2.4 Non-implementation

Non-implementation is another common fallback when integration effort is prohibitive. Developers simply avoid providing some desired functionality hence saving implementation effort. Again, non-implementation is easier to observe by anecdote than by rigorous measurement. Among open-source projects, despite free availability of code, we nevertheless frequently observe features present in one software package which are not found in a

[1]http://www.dspace.org/
[2]http://plone.org/
[3]http://www.dotnetnuke.com/

competitor. Moreover, we often see requests for the missing features in the relevant bug tracker. Many such requests go years without being satisfied, despite the availability of an implementation of the feature in the other project's codebase. This is an indication of the significant integration effort involved.

1.2.5 Standardisation

One approach to coping with diversity and change is to rule them out. This is the essence of standardisation [Kuhn 1990]. There are many well-known successful examples of standardisation in software interfaces: programming languages' standard libraries, standard operating system interfaces such as POSIX [IEEE POSIX, 1988], long-lived network protocols and client APIs such as the X11 Window System [Scheifler and Gettys 1986].

Unfortunately, the weaknesses of standardisation are also well-known. While it may succeed in cutting out whimsical diversity, or ill-considered changes, many occurrences of diversity and change originate from highly justified desires to improve and innovate. For this reason, it is commonplace for a domain to offer many competing "standards", as well as many revisions or extensions of a given standard. Most software interfaces are not standardised at all, as this enables the greatest freedom to innovate and improve. Rather, they are peculiar to a single implementation and offer no guarantees of stability.

1.2.6 Information hiding and modular programming

We have seen how dealing with interface diversity and change can make development tasks expensive. For these reasons, both folklore and research in software engineering have often advocated keeping interfaces small. This is the basis of *information hiding* [Parnas 1972]: developers consciously limit what interface details should be visible from other components, in order to restrict the "surface area" of interdependency between components to a small and less change-prone set of details.

Most programming languages espouse some form of "modular programming". Invariably this means that input programs can be split into multiple constructs or files, where language-defined rules enforce restrictions on the permitted inter-reference between components. These rules partially derive from implementation concerns, but also encompass information hiding features. For example, Java and C++ have private and protected modifiers for class members, which exist only to hide change-prone members that should not be included in the interface. CLU [Liskov et al. 1977], with its notion of abstract data types—data structures whose concrete representation is hidden from their clients—was the first language to explicitly emphasise this principle.

Unfortunately, even small interfaces can suffer mismatch. Moreover, interfaces cannot be smaller than the domain that they model. Somehow, the programmer must choose a concrete expression of their domain. Any such concretion is an opportunity for introducing mismatch with a concretion chosen by another programmer. Therefore, information hiding is useful but not sufficient to address the problem of interface mismatch.

Another sense of "modularity" is the different but related issue of *modular reasoning*—the ability to scalably reason about large systems. Type systems are by far the most popular example: they abstract programs by considering the sets of values which each program fragment might yield [Pierce 2002], and use these abstractions to reason about compositions of program fragments. Just as modular programming concerns limiting the changes which can invalidate compositions, so modular reasoning is concerned (in part) with limiting the changes which can invalidate prior reasoning about a composition. These issues are separate from our work, except for an indirect practical relevance: the machine-readable *interface specifications* available in real code reflect certain biases in conventional programming languages. Specifically, most languages provide type annotations, but little about control properties of interfaces (such as their protocol specifications [Yellin and Strom 1997]).

1.2.7 Porting

Developers sometimes *port* code written for one environment (say, targetting a particular library, as in §1.2.1) to run in a different environment. This means editing the code to change the details that differ between the two interfaces. The result is either a *patch*, detailing the changes to the original code, or perhaps a *fork*, meaning a separately-maintained copy of an entire modified codebase.

Patches record source-level differences, usually in the form of line-by-line differencing (as provided by the Unix diff command [Hunt and McIlroy 1976]). Since patches identify locations in source code using fragile contextual details (usually a combination of line numbers and unmodified surrounding lines), they must be meticulously maintained as the original codebase evolves through independent modifications. Otherwise, the patch will no longer apply. In the case of a fork, the porting programmer has decided to avoid this maintenance effort, at the expense of foregoing the benefits of any independent modifications. (He may choose to *backport* those changes himself, effectively patching in the opposite direction.) Among open-source code, “rotted” patches are common: these are patches that have not been maintained, so no longer apply to current versions of their target codebase.

Since porting involves modifying existing code, it presents a high risk of introducing unrelated bugs in adjoining nearby code, causing regressions in potentially unrelated features. These problems arise because patchsets and source-level edits are *overly invasive*—they modify and duplicate code which could be left alone—and *overly syntactic*—meaning that they can be broken by irrelevant or superficial changes which have no semantic effect on the program (such as renaming an identifier, or simply changing code layout).

Porting-based migration of code from one infrastructure to another is occasionally undertaken, but at huge cost. For example, during 2006–08 the KDE desktop suite migrated from using the DCOP inter-process messaging infrastructure to the broadly similar D-BUS, entailing modifications to dozens of applications, in most case impacting hundreds or even thousands of lines of code.[4] It is notable also that the motivation for this move was to improve integration between KDE and GNOME desktop suites, which

[4]See revision 546826 of the KDE Subversion repository, available at http://websvn.kde.org/?revision=546826&view=version as of 22 September 2010, or in the e-mail with message ID 1149060043.312648.25096.nullmailer@svn.kde.org.

prior to this porting effort had been classic instances of the silo phenomenon described in §1.2.1.

1.2.8 Automated source code transformation

Many development environments, such as Eclipse [Holzner 2004] or Microsoft's Visual Studio [Johnson et al. 2002], offer automated "refactorings" on source code. Refactorings, as originated by Opdyke [1992] and popularised by Fowler [1999] are systematic source-level transformations, designed to automate non-localised changes which are laborious and error-prone to perform by hand. In this sense, they address a problem which generalises from some of the difficulties of porting: the laborious, non-localised nature of source-level changes. Refactorings may be used to make the editing process somewhat quicker and less error-prone than hand porting. However, they do not address the problem maintaining a forked or patched version of the original code. If a codebase has been refactored to target some different interface, or provide some different interface, then unless support for the *old* interfaces is no longer required, the same maintenance issues emerge. Current refactoring systems also require careful testing or manual checking of generated code, since they lack precise specifications and are frequently implemented incorrectly [Schaefer and de Moor 2010].

1.2.9 Glue coding

An alternative to porting is provided by the adapter pattern [Gamma et al. 1995] and other "glue coding" approaches. Here, the aim is to avoid modifying existing code, but instead to implement the desired interface by a thin layer of code which consumes an alternative interface provided by some available component. This is done by coding "wrapper" functions which effectively map from one interface to another.

This approach has several desirable properties. Unlike porting, it does not involve dangerous modifications to existing code, since it involves only adding new code. Unlike patching, it does not introduce fragile dependencies on the surface form of existing code, since it composes using the language-level composition mechanisms rather than textual or syntactic substitution. Unlike porting and patching, it is compositional in the sense that many different adapters can be linked against the same underlying interface without copying or modifying code. This means that adapters could be used to link a given client against multiple different libraries *within the same program.* (By contrast, applying multiple patchsets to the same underlying code is in general infeasible, except perhaps by replicating the target code multiple times in the codebase.)

The latter class of system, in which the same client links against many libraries within the same program, resembles very much a plug-in system, and indeed, abstraction layers (§1.2.2) can be seen as a kind of adapter—albeit one that is usually simplified by the fact that the adaptation was foreseen. This foresight allows the adapted-from interface to be carefully designed in a highly abstract way, in order to accommodate multiple conforming implementations with a minimum of subsequent effort. By contrast, unforeseen adaptation tasks are more likely to suffer complexity associated with "undoing" concretions introduced by the adapted-from interface.

Glue coding shares the main benefit of information hiding that dependency is limited to interface details, so glue code is fragile only to the extent that the glued interfaces are change-prone. Nevertheless, since one of our motivations is that interfaces can and do change over time, this still represents a nontrivial cost.

Other than this unavoidable maintenance cost, there are two key limitations of the glue coding approach. The first is simply that it is not always possible to avoid modifications to target code, because the necessary points of interposition may not be available. For example, if the adapted-to interface specifies that the client can supply a notification callback to be called when some internal event occurs, and the adapted-from interface does not generate any such call, it is not possible for an adapter to generate the call because it simply will not receive control when the relevant event occurs. To satisfy these compositions, some sort of invasive modifications are clearly required. Simple patching to insert the relevant notification (often called "hooks" or "instrumentation") can suffice; once this is done, the adaptation can be resumed in the usual glue coding fashion.

The second limitation is that glue coding is notoriously tedious and error-prone. Reifer et al. [2003] estimate that a line of glue code costs three times as much to maintain as a line of code in a version of the same system coded from scratch. Although this says nothing about the relative quantities of code required for glue-based implementation versus a custom reimplementation—and we would hope the former would require far less—it still suggests that glue code presents some specific difficulties.

Despite these limitations, the modular and compositional nature of glue coding make it an appealing approach for tackling interface mismatch tasks. The rest of this dissertation considers how to address some of its drawbacks, by making it a more efficient option for the programmer.

1.3 Examining tool support for glue coding

We now make some simple observations, familiar to most developers, which motivate the subsequent direction of the dissertation.

1.3.1 Aptitudes of conventional languages

Many tool developers have questioned whether general purpose programming languages are a good tool for creating various kinds of glue code. This is evidenced by the many domain-specific tools that have emerged for assisting with special cases of glue coding—particularly for generating marshalling code between network- and program-oriented encodings of the same data [O'Malley et al. 1994] and for foreign function interfaces in language implementations [Beazley 1996]. However, none is so general as to cater to a wide variety of interface mismatch scenarios.

It is generally understood that conventional languages are designed to abstract algorithmic, data-transforming or domain-modelling code. By contrast, we can observe that most glue code is algorithmically simple, defining few or no new data types, few new functions except for wrappers and occasional utility code, and making relatively little use of looping or recursion. Rather, it is concerned with *recognising* and *relating* the interactions defined by two *existing* mismatched interfaces. (These observations are illustrated by a great many examples throughout the rest of this dissertation, beginning shortly with Fig. 1.4.)

Glue code is often fragile in that its correctness depends on consistency with the external interfaces which it is adapting between. It is doubly fragile in that the same or similar patterns of glue are often repeated across many wrappers, frequently making it easy for the programmer to introduce errors, either by under-application of these patterns (applying logic at only a subset of the appropriate locations) or over-application (applying logic at inappropriate locations).

As illustration, consider the two example wrapper functions in Fig. 1.4. These form part of an adapter (discussed fully in Chapter 5) between a pair of filesystem interfaces, puffs (here the interface provided) and rump (the interface consumed). The seek call repositions an open file cursor, while remove deletes a directory entry; we have added comments, but the details of the code's function are not important. Instead, notice four problems with this style of code.

Firstly, there is a high volume of similar code: these functions show only two out of 28 wrapper functions in the complete adapter. Most of these 28 wrappers are of similar form, but different in detail, to the two shown. Overall, a large volume of similar code is required for what is a conceptually simple task.

Secondly, however, notice that the code is not *trivially* repetitive. Each wrapper applies a different subset of (implied) rules, e.g. for treatment of arguments. Consider the opc parameters in the figure: one case requires a bumped reference count and has different unlocking semantics from the other. The programmer must juggle these rules correctly amid this large quantity of similar code. The fact that these rules are not stated once, but rather are expanded into each wrapper function, represents a failure of modularity.

Thirdly, readability is poor: while the exported ("provided") interface is clearly marked out by the wrappers' signatures, the imported ("required") interface is buried inside the wrapper function bodies. This obscures what are in fact quite simple abstract correspondences between the two interfaces.

Fourthly, wrappers are inconvenient for developers: among other headaches, to compile this code the programmer must construct a hybrid build environment supporting compilation against *both* interfaces' data-types and helper functions. In the C language, for example, this is often nontrivial because of limited support for identifier scoping (i.e. lack of user-defined namespaces). In general, we claim that component implementation languages may not be accommodating of the peculiarities of adaptation tasks—unsurprisingly, because support for such tasks is not a design priority for conventional programming languages.

This example shows a relatively simple case of adaptation. In more complex cases, these difficulties escalate further. For example, in this particular case, functions across the two interfaces correspond one-to-one. In others, where functions have some more complex correspondence, the programmer might be forced to create a state machine in order to catch a particular temporal pattern of calls requiring special treatment. (We will see examples of this in Chapter 2.)

```
int p2k_node_seek(struct puffs_usermount *pu,
     puffs_cookie_t opc, off_t oldoff, off_t newoff,
     const struct puffs_cred *pcr)
{
  kauth_cred_t cred;
  int rv;

  cred = cred_create(pcr); // convert auth token
  VLE(opc);               // lock vnode ptr
  rv = RUMP_VOP_SEEK(opc, oldoff, newoff, cred); // call
  VUL(opc);               // unlock vnode ptr
  cred_destroy(cred);     // destroy temp auth token

  return rv;
}
int p2k_node_remove(struct puffs_usermount *pu,
  puffs_cookie_t opc, puffs_cookie_t targ,
  const struct puffs_cn *pcn)
{
  struct componentname *cn;
  int rv;

  cn = makecn(pcn);       // issue temp name
  VLE(opc);               // lock vnode ptr
  rump_vp_incref(opc);    // bump refcount
  VLE(targ);              // lock target vnode
  rump_vp_incref(targ);   // bump that refcount
  rv = RUMP_VOP_REMOVE(opc, targ, cn); // call rump
  AUL(opc);               // this time, vnodes were unlocked
  AUL(targ);              //... by rump, so just assert this
  freecn(cn, 0);          // free temp name

  return rv;
}
```

Figure 1.4: Example filesystem wrapper code

1.3.2 Plug-compatibility assumptions

All conventional programming languages and linkers use a plug-compatible model of composition: they compose partial programs whose interfaces are assumed to match exactly. This design follows naturally from the popular conception of programming as a precise mathematical activity, advanced by influential texts such as Dijkstra's *Discipline of Programming* [Dijkstra 1976]. Parnas observed that Dijkstra's idealised approach contrasted with the reality that programs are in fact not written to solve a definitive single problem but rather families of problems [Parnas 1978]. Analogously, our observations so far have shown that software components are not written for a single unchanging composition context but must integrate with diverse and changing interface requirements.

There is a gap in the support offered by tools assuming plug-compatibility: they provide no support for describing *how* to make a compatible composition out of a plugincompatible one. For example, a typical compiler will include a type-checker to tell the programmer explicitly when a composition might be type-incorrect, but will not do any work on the programmer's behalf to make a correct composition out of an incorrect one, nor accept any special instruction from the programmer about how to do this. Rather, traditional thinking holds that doing so is "just another programming exercise", most likely to be done by modifying the input components. By contrast, since we know that such incompatibilities arise from *diversity* and *change* in interfaces, we can observe two things. Firstly, this kind of programming task is qualitatively different from others, and so might benefit from different tools and languages. Secondly, such programming should be modularised *separately* from the input components, since it relates not to either component individually, but rather to their combination.

The closest to effective tool support for these scenarios lies in refactoring support in integrated development environments (IDEs), as outlined in §1.2.8. However, it works only when the relevant changes are made using the automated refactoring support of the tool, whereas many edits will lie outside the tool's repertoire. Moreover, it assumes that changes affect only code that is local and can therefore be automatically updated by the tool. By contrast, precisely the kinds of changes most in need of tool support are those whose effects cross project boundaries, since the developers who must deal with the changes are not the same developers who made them. These are not addressed by refactoring. (That is not to deny that other IDE features, such as incremental error-checking and autocompletion, do make the manual editing process somewhat more efficient.)

1.3.3 Homogeneity assumptions

Continuing development of new programming languages, libraries and other software infrastructure entails that components may also be mismatched in their expectations of *styles* or *packagings* of the components with which they are to be linked [Shaw 1995]. Even if two independent programmers devise what are effectively identical interfaces to some functionality, their code might nevertheless not be composable using conventional tools (such as a linker), owing to the concrete differences emerging from their choice of infrastructure (including programming language, libraries, operating system and so on). Effectively, each piece of infrastructure creates its own silo (§1.2.1). Since many abstractly similar infrastructures exist (e.g. similar programming languages, similar development frameworks, etc.), we would like the ability for tools to compose *heterogeneous* software. We currently lack this ability. Gaining it could hugely increase the degree of available compositionality by effectively breaking apart the silos defined by each separate piece of infrastructure.

1.3.4 The separation of functionality from integration

We can paraphrase the foregoing critical appraisal of tool design by saying that conventional tool designs do not separate *functionality* from *integration*. Composition-specific details are an integration concern, but owing to the plug-compatibility assumption, conventional tools force them to be addressed within components themselves—the same medium used for functionality. Similarly, by composing only homogeneous components, tools induce dependencies between raw functionality and the packaging- or infrastructure-specific details which ought only to be an integration concern. The desire to separate functionality from integration has been motivated by prior work [Dellarocas 1997; DeLine 2001]. However, much work remains on realising this separation in a practical tool.

1.4 Research approaches

Here we briefly characterise the classes of approach found in existing research literature, with approximate comparisons. A complete discussion of related work is left until Chapter 7.

Roughly, we may divide research approaches to this problem along two dimensions. The first is applicability: *clean-slate* solutions allow a broad re-imagining of the software development process, but require that any participant components are rewritten from scratch according to new practices, whereas solutions designed for *incremental adoptability* offer an approach which can be applied to existing code, invariably making some compromises to do so. The second dimension concerns the abstractions in whose terms compositions are described, which we divide into *white-box* and *black-box* categories.

1.4.1 Clean-slate versus adoptable solutions

Flexible Packaging [DeLine 2001] is an example of a clean-slate approach. It proposes developing software components in two parts: a *ware* describing the functionality, and a *packager* describing its interaction with external components. Different packagers can adapt the same ware to different sets of concrete operations, and can support a wide variety of interface mechanisms and conventions. This approach is extremely appealing, since it successfully achieves a separation between functionality (wares) and integration details (packagers). However, the work leaves open the question of how to apply the underlying principles to existing software in an *adoptable* way, without redeveloping all software in the form of wares and packagers. Answering this question is a primary goal of this dissertation.

1.4.2 White-box versus black-box approaches

We can generalise from our discussions of the relative merits of patching, porting and glue coding (§1.2) to classify research approaches.

“White-box” refers to any tool design which abstracts software components as structured artifacts whose entire internals are exposed to the tool. Such tools might perform conventional modular programming, or be specialised for expressing extensions, compositions or specialisations of software. Porting is a white-box approach, often supported using the Unix `patch` tool, which is applied to software by interpreting arbitrary pieces of software source code as lines of text, any of which may be replaced or deleted. Aspect-oriented programming [Kiczales et al. 1997], refactoring tools [Opdyke 1992], “semantic patching" tools [Fiuczynski et al. 2005; Padioleau et al. 2008], instrumentation tools such as Pin [Luk et al. 2005] and similar fall into this category. White-box tools often adopt source-level abstractions of components—such as lines of code, program statements, nodes in an abstract syntax tree, data structure representations, and so on. However, they do not always do so; Pin is one notable exception. Instead of a source-level view, its interface is based on control-flow information reconstructed from the binary image of a running component. Despite targetting binaries, it remains white-box.

By contrast, the black-box approach provides tools which only inspect and modify components up to some limited extent, circumscribed by the component's logical *interface.* An interface is an abstraction of a component intended to be both less change-prone than the component's internal implementation[5] yet nevertheless compositionally useful as a "contract" or point of agreement with other components. In other words, the essence of black-box models is precisely that of Parnas's information hiding [Parnas 1972], as embodied in most conventional programming languages' notions of interface or abstract data type. The adapter pattern [Gamma et al. 1995] represents a black-box approach to adaptation in conventional practice. Signature- and behaviour-oriented adaptation techniques, such as Nimble [Purtilo and Atlee 1991] or the work of Bracciali et al. [2005], constitute black-box approaches. Configuration and coordination languages like Knit [Reid et al. 2000] or Reo [Arbab and Mavaddat 2002] also present a black-box component abstraction to the programmer.

Both kinds of technique are capable of supporting a separation between functionality and integration. For example, an aspect-oriented pointcut language can identify the points in a component's execution where it should interact with some external component, while aspect advice can supply relevant integration code. Alternatively, a black-box approach could use a coordination language like Reo to describe logic or data-flows to be inserted between existing call sites and entry points, interpreted as message ports. The key difference between black- and white-box techniques may be summarised as follows. Black-box notations restrict what aspects of components are visible and limit what can be expressed, thereby constraining the programmer in order to protect modularity. Meanwhile, white-box techniques provide unconstrained, powerful primitives enabling a wide range of adaptations to be expressed, but in so doing, place a burden on programmer discipline to maintain a well-modularised system.

Fig. 1.5 highlights pictorially the contrast between white- and black-box descriptions of adaptation techniques.

Since black-box techniques more actively enable a modular separation between functionality and integration, and allow for more language-independent (and hence more widely adoptable) tool support, we pursue this approach within the scope of this dissertation. However, doing so makes the initial assumption that the necessary points of interposition—call-sites, join points, entry points and so forth—are exposed on the target components. If not, then white-box techniques must be employed to expose them. We will call this property of a compositional task "well-abstractedness".

[5]. . . or "diversity-prone", in the case of reimplementations, e.g. of a standard interface.

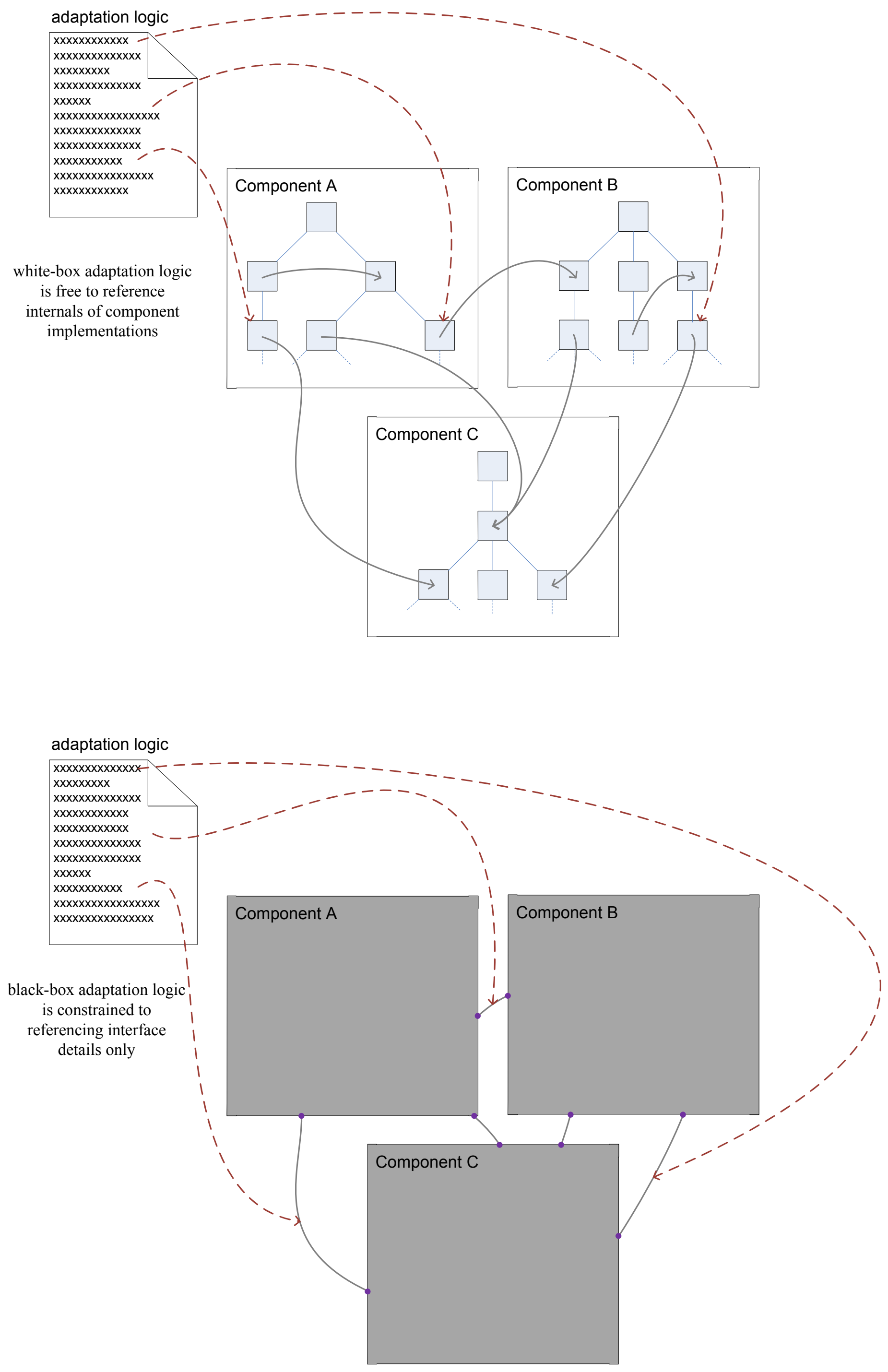

adaptation logic
Component A
Component B
Component C
white-box adaptation logic is free to reference internals of component implementations
adaptation logic
Component A
Component B
Component C
black-box adaptation logic is constrained to referencing interface details only

Figure 1.5: Contrasting black- versus white-box approaches

1.5 Goals

Having considered contemporary practices and prior research approaches, we now set out a list of goals which the work presented in this dissertation was conceived to satisfy. These are motivated by the identified weaknesses in prior work; no prior work simultaneously satisfies them.

1.5.1 List of goals

1. **Black-box abstraction of adaptation tasks** We require a solution which does not involve modifying the source code of input components, and preserves the benefits of information hiding between component implementations and their exported interfaces. This is a goal which can be satisfied by construction, but will constrain the design of our system in significant ways.

2. **Practicality** The contribution of this work is inherently practical; its success requires the outcome to be a useful development tool. We make this objective more precise by defining three sub-goals: implementability, adoptability and convenience. Our solution must permit a working implementation. Furthermore, in the interest in maximising the gains of compositionality, we require a solution which is *adoptable* in that it applies to a large volume of existing components (cf. clean-slate approaches, §1.4.1). Finally, we wish to ensure that our system constitutes a *convenient* tool, by minimising the extent of time-consuming overhead imposed on the developer. The particular inconveniences which our design avoids derive from its composition of *binary* components, as detailed in Chapter 2. All these goals are satisfied by the construction of a system according to our design.

3. **Sufficient expressiveness** Since we intend to make a contribution to *practical* software engineering problems, it does not suffice to demonstrate isolated individual ideas at a conceptual or theoretical level. Instead, we wish to design and implement a practical tool which, by a combination of features, has sufficient expressiveness to realise a reasonably *broad selection* of real or realistic programming tasks.

4. **Support heterogeneous components** Software is developed in a multitude of languages and coded in a multitude of styles. To maximise compositionality, we must make few assumptions about the nature or origins of components, with regard to the languages, libraries or coding styles with which they were developed. Moreover, we must provide means of describing or capturing these stylistic differences on a per-component basis, such that these recurring per-component characteristics may be abstracted *separately* from concerns arising from combinations of components.

1.5.2 Non-goals

As a matter of practicality, we must focus attention on some goals at the expense of others. It is important to state these explicitly. We will not pursue any specific targets in respect of the following criteria.

Safety Statically-assured freedom from various classes of run-time error is a common goal of much programming research. Since we target programmer productivity, rather than reliability per se, we do not set any goals in respect of reliability. (Of course, more productive programmers can spend longer on finding and fixing bugs in their code.) We add that while at first glance it may appear risky to perform programming tasks at the binary level, we firmly believe that it need not be less safe than any existing source-level approach. For reasons of simplicity, this dissertation does not address provision of *guaranteed* safety. However, binaries admit exactly the same sorts of type-checking and other compositional reasoning as source-level representations do (given appropriate metadata, in certain cases), so this problem may be tackled separately. This is discussed further in §1.7.

Performance We will not attempt to evaluate our solution based on the performance of the code it generates. Rather, it will suffice to identify the added costs in execution time and memory, discuss to what extent they are avoidable, and present approaches for reducing the overhead relative to that of an initial implementation.

Automation We are not seeking a tool for producing compositions automatically. Our requirement of working with existing codebases, which may have a highly limited extent of machine-readable specification, means that achieving semantically correct compositions requires human guidance. Therefore, we focus on techniques for reducing the incidental complexity of providing this guidance, relative to current language and tool designs. Future work could certainly build on this towards a more automated approach, by combining techniques for inferring likely composition logic with appropriate automatic testing or verification infrastructure.

1.6 Thesis statement

Having stated our goals, we can now state the thesis which the remainder of this dissertation will substantiate. The thesis of this dissertation is as follows.

> Using a special-purpose language, based on *relations*, to compose heterogeneous mismatched software components, is significantly more effective in practice than conventional programming languages.

We introduce the abstraction of relations in §1.7, and develop it throughout Chapter 2. Section 1.7 also more precisely defines the evaluation criteria, denoted above by “effectiveness in practice”; these are fully detailed in Chapter 5.

The substantiation of this thesis represents an advance in the state of the art. To explain this advance, we now summarise prior research work sharing some or all of this thesis’s goals.

system	ref.	b/w	existing	het'ous	expressive
Nimble	[Purtilo and Atlee 1991]	black	yes	some	low
subject composition	[Ossher et al. 1995]	black	some	some	low
Yellin & Strom	[Yellin and Strom 1997]	black	some	some	mid
BCA	[Keller and Holzle 1998]	black	yes	some	low
COMPOST	[Assmann et al. 2000]	white	yes	some	mid
Flexible Packaging	[DeLine 2001]	black	no	yes	high
Bracciali	[Bracciali et al. 2005]	black	some	some	mid
Object expanders	[Warth et al. 2006]	black	yes	some	low
Concept Maps	[Järvi et al. 2007]	black	some	no	high
Twinning	[Nita and Notkin 2010]	white	yes	no	mid
Cake	this dissertation	black	yes	yes	high

Table 1.1: Brief comparison of prior research approaches

1.6.1 State of the art

We briefly summarise how these goals compare with the achievements of related research work. For brevity, here we consider only the most directly comparable work, meaning that which presents new linguistic abstractions useful for adaptation tasks (hence the absence of the adapter pattern and other conventional techniques).

The substantiation of these comparisons can be found in Chapter 7, which also features a wider range of prior work. In summary, however, we can highlight the following shortcomings of prior work.

Limited expressiveness Many systems provide only features capable of expressing a few simple classes of adaptations. A common weakness is the lack of support for context-sensitive or many-to-many adaptation requirements [Purtilo and Atlee 1991; Keller and Holzle 1998; Warth et al. 2006; Nita and Notkin 2010]. Conversely, systems focussing on protocol adaptation [Yellin and Strom 1997; Bracciali et al. 2005] focus on such requirements at the exclusion of others—notably, little consideration of data structures and other decompositions of data—which are necessary to build a practical tool.

Lack of practical demonstration Some systems which aspire to relatively high expressiveness have not been applied to substantial case studies. Sometimes this is because no implementation is available, perhaps because the work's contribution is primarily a model or a formalism [Ossher et al. 1995; Bracciali et al. 2005].

Limited applicability to existing code Clean-slate solutions [DeLine 2001] are conceptually useful but of limited practical use unless a significant volume of code is written according to their approach. Other approaches which target specific programming languages [Keller and Holzle 1998] or subsets of programming languages [Haack et al. 2002] also bring analogous restrictions, although less severe.

Low tolerance of heterogeneity Many solutions target very specific notions of component, such as code written in a particular language [Keller and Holzle 1998; Järvi et al. 2007; Nita and Notkin 2010]. Since this reduces the selection of candidate components available for composition, it is less desirable than solutions which accept a wide range of input components.

White-box techniques White-box or "grey-box" techniques [Assmann et al. 2000; Nita and Notkin 2010] are easier to design with a high degree of expressiveness (since the ability to transform program internals naturally permits powerful adaptations), and often more straightforward to implement (since they can be compiled by direct syntactic manipulation). However, we have surveyed well-established arguments for the benefits of black-box solutions where they can be applied (§1.4.2).

For further discussion, the reader is referred to Chapter 7.

1.7 Approach

The following paragraphs summarise the key aspects of the approach adopted by our work.

Rule-based design We devise a rule-based language, called Cake, for describing adaptation logic. Rather than forcing the programmer to write wrappers, rules form a higher level and more declarative abstraction. Each rule localises (in the ideal case) a single piece of application-domain knowledge about the adaptation task. The burden of composing rules into wrappers falls on the compiler for our language, not on the programmer.

Computational simplicity Our solution should not incorporate expressiveness beyond that required for realisation of real use-cases. This complements the declarative rule-based approach: the expressiveness of our rules should not need to be extended to the point where it gains the feature-set of a general-purpose programming language. The motivation for this is that a more constrained domain is both simpler for the human programmer to use, and simpler for automated tools to reason about. Although neither automatic reasoning nor automatic generation of adapters are goals of this work, we aim for a solution amenable to future work towards these goals.

Emphasise open-source components This dissertation chooses (somewhat arbitrarily) to focus on components produced in the open-source community. These are often written in C, C++ and other unsafe languages with explicit storage management. This presents additional complexities in both tool design (e.g. the requirement to support mismatches arising from memory deallocation obligations, which would not be found in a fully garbage-collected environment) and implementation (e.g. to work around the lack of precise object metadata at run time). However, the core problem being addressed, namely interface mismatch among components, is specific neither to binaries nor to unsafe languages.

Target binaries Development tools should aspire to convenience of use. One source of inconvenience often encountered in composition-based development tasks is that the programmer must construct build environments capable of compiling and linking each target codebase. Moreover, an environment that can compile and link *all* target codebases *together* is required. This is nontrivial for various reasons. For example, some codebases may demand particular versions of compilers or header files, potentially in a conflicting fashion. Source-level approaches to composition tasks often involve slow edit–run–debug cycles because they entail frequent rebuilding of large portions of the codebase (for example when editing header files). Our approach is designed to avoid these complexities by working directly with compiled code. This also limits recompilation delays to that for the tool-generated code.

Evaluate against prior coding One way to ensure that a tool is practically useful is to use it to repeat tasks already tackled by conventional means. This also provides a useful like-for-like point of comparison. To evaluate our approach we apply it to tasks having available conventionally-coded solutions drawn from the open-source development community. By comparing both aggregate properties (measurements) and detailed properties (features of the resulting code) across the two solutions, we can gain confident assurances of the relative merits of our tool.

1.8 Contributions

In summary, this dissertation presents the following contributions.

Requirements By example, we present a set of expressiveness requirements for a black-box adaptation tool, as a step towards Goals 1 (black-box composition) and 3 (expressiveness).

Relation-based language design We then presents the design of a language, Cake, which abstracts adaptation logic as a set of rules describing *interface relations*, applying in a black-box fashion to existing binary code. This is a further step towards Goal 1 (black-box composition) and a step towards Goal 2 (practicality).

Code generation We describe how our implementation of Cake compiles abstract rules into wrapper code, which composes with existing binaries in a black-box fashion. This completes the satisfaction of Goal 1 (black-box composition), and is a step towards Goal 2 (practicality).

Run-time support We detail the run-time support required to execute Cake-generated compositions. This support consists primarily of the combination of several known dynamic analysis techniques. This completes the satisfaction of Goal 2 (practicality).

Application to found use-cases We describe experiences applying Cake to three real-world adaptation tasks drawn from well-known open-source codebases. We demonstrate material simplifications conferred by Cake, relieving the programmer of various coding obligations. This is a step towards satisfying Goal 3 (expressiveness).

Measurement In summary of our experiences, we present aggregate measurements of the existing implementations versus the equivalent Cake code, making certain corrections for syntactic disparities between the languages used. This completes the satisfaction of Goal 3 (expressiveness).

Extension to support styles We outline an extension to the basic Cake language which accommodates a class of *stylistic variations* among component interfaces. This class is surveyed in detail, and shown to encompass a wide variety of concerns arising from choices of programming language, implementations thereof, and coding style. This completes the satisfaction of Goal 4 (heterogeneity).

Related work We survey related work, explaining how existing research tools are not adequate for realising the tasks to which Cake has been successfully applied.

Future directions We describe several plausible extensions to Cake which could yield additional benefits.

By these contributions, we have shown that black-box adaptation using Cake is abstracted in a language which is sufficiently expressive to capture many real use-cases, is implementable, caters to heterogeneous input components, and results in code which is shorter overall and simpler in numerous details. The thesis is therefore substantiated. The remainder of this dissertation describes each of the listed contributions in detail.

1.9 Technical background

Out of necessity, the work presented in this dissertation illustrates general ideas in a specific technical context. In particular, it builds heavily on two pieces of technical background: firstly, common programming practices within open-source codebases, particularly their use of the C and C++ languages; secondly, on the composition specifically of compiled native binary code. We briefly discuss the peculiarities of these here.

1.9.1 Challenges of C and C++ programs

In the open source world, much software continues to be written in C or C++. This contrasts with much contemporary software engineering research into composition and adaptation tools, which, to a first approximation, mostly targets source- or bytecode-level representations of Java-like languages. Open-source software is also, by its nature, among the most likely to display the kind of decentralised, evolving or unanticipated composition practices for which the new tool support we have motivated is intended. There is therefore currently a significant gap in the existing research work towards supporting these activities on open-source and similar code.

C and C++, with their explicit resource management and direct-to-machine compilation, introduce two major difficulties to implementation: firstly in the direct access to untagged memory, and secondly in the variety of mechanisms determining object lifetime (deallocation). Untagged memory implies that it is nontrivial to discover a valid high-level interpretation for a given piece of memory. This is the same complication underlying conservative garbage collection [Boehm and Weiser 1988], which must conservatively interpret stored words as addresses even if user code happens only to interpret them as (say) integers. This also complicates several other dynamic analyses on C and C++ programs. We discuss our approach to this problem in Chapter 4. (Note that our approach is *not* afflicted with the various classic difficulties of *statically* analysing C and C++ code, such as unconstrained aliasing or implementation-defined ordering of side effects.)

C and C++ also share the property that their standard libraries are relatively small in size, compared with more recently defined languages such as Java and Python whose standard libraries are explicitly intended to be extensive. In turn, this means that many third-party libraries for recurring utility functionality have gained popularity among different groups of C and C++ programmers. Consequently, there is particularly great diversity visible in C and C++ code: even relatively simple programmatic abstractions, such as strings and lists, have multiple concrete realisations in different libraries, and this can be a source of interface mismatch.

1.9.2 Tools manipulating binaries

This dissertation proposes a tool which performs composition of *binaries*, in the form of relocatable object code compiled for some concrete machine-level architecture. Binaries offer the advantage of unifying many source languages. They also offer convenience benefits, originating in the fact that binaries are the form in which software is deployed and executed. We briefly survey the existing tools and practices for working with binaries.

Debuggers Symbolic debuggers are a well-established programming tool. They perform an impressive feat: recovering a *source-level* view of a running *binary* program. There are many benefits to this. Programs can be run "for real" in their intended environment while still providing the developer with source-level backtraces, interactive state inspection, step-through, breakpoints and so on. On discovering a bug, the problem can be tackled *in situ*. Debuggers are built on considerable infrastructure, stretching from hardware (which must support single-step, breakpoints, etc.) through operating systems to compilers and runtimes; our implementation approach actively exploits this established infrastructure.

Binary composition Tools for producing compositions of binaries offer convenience benefits analogous to those of symbolic debugging.[6] Existing deployed code can be incorporated into novel compositions "as is", without recompiling entire codebases. The developer's edit–run–debug work cycle is shorter as a result, and composition tools can be placed at the hands of the end user, maximising the potential for user-led innovation. The philosophy is longstanding: Unix [Ritchie and Thompson 1974] espouses much the same principle in the design of its stream-processing tools, where the user is invited to compose novel pipelines specialised to their particular tasks. More recently, interest in

[6]They may also borrow much of the same infrastructure, in the form of compiler-generated debugging information, in their implementation.

"mash-ups" of web-based application has provoked research into supporting infrastructure [Wang et al. 2007].

Established tools The evidence for the tractability of binaries does not end with symbolic debuggers. In recent years there has been considerable documented success of other binary tools including link-time optimisers [De Sutter et al. 2005], binary instrumentation tools [Luk et al. 2005], binary-rewriting virtual machine monitors [Devine et al. 2002] and reverse-engineering tools [Balakrishnan et al. 2005]. The convenience of binary techniques, in avoiding recompilation when producing a new composition, may also be seen analogously to the convenience of dynamic software update [Neamtiu et al. 2006] in avoiding restarts when deploying a new software revision, and offers similar benefits to both users and developers.

Reliability concerns A potential disadvantage of working with binaries concerns reliability. Since binaries are manipulated *after* compilers have performed semantic analysis on source code, various safety or correctness properties previously enforced by the compiler could be violated by errors introduced at the binary level. A typical example is type-unsafe linking: since conventional linkers do no type checking, they will happily link code with mismatched binary interfaces. Many programmers are familiar with bugs that can be introduced in this way, for example when linking version-skewed object code. This can easily cause crashes and unpredictable run-time behaviour.

Reliability solutions Fortunately, reliability fears can easily be mitigated. Previous work has addressed the specific problem of link-time type-checking [Stroustrup 1988; Banavar et al. 1994]. Meanwhile, well-engineered binary manipulation tools have been demonstrated to show high degrees of reliability. The popular VMWare virtual machine monitor [Devine et al. 2002] rewrites binaries dynamically at instruction-level granularity, yet is routinely used in mission-critical enterprise deployments. Other tools, such as Valgrind [Nethercote and Seward 2007] and Pin [Luk et al. 2005] perform similar feats with high standards of robustness. In a black-box approach such as the one adopted in this dissertation, binaries need not be modified internally, but instead may be simply linked with new intermediate code. The risks are therefore correspondingly fewer. With appropriate metadata—which may include debugging information—programs targetting binary abstractions of software are amenable to exactly the same classes of automatic checks as source-level code.

Relationship to black-box approaches Binary and black-box approaches are ideal complements. Since internals need not be modified or inspected, absence of source-level descriptions is not a problem. In fact, such approaches can often be safer than source-level coding, in that they avoid the need for the programmer to disturb the source-level internals of well-tested code. Instead, by coding against the code's interface, the programmer only risks incorrect behaviour in his own code.

1.10 Outline of subsequent chapters

Chapter 2 answers the key questions about the design of the Cake language. What are the requirements for such a language? What form does such a language take? This includes syntax and informal semantics.

Chapter 3, as an interlude, discusses open limitations and possible extensions to the Cake language.

Chapter 4 addresses how to implement such a language for native binaries, both in compilation and at run time.

Chapter 5 demonstrates the application of Cake to real tasks, and the extent to which the Cake language results in simpler, better-modularised code.

Chapter 6 describes extensions Cake to support a heterogeneous variety of input components, as motivated by the design discussion in Chapter 2 and experiences in Chapter 5.

Chapter 7 presents a comparison between Cake and related work.

Chapter 8 summarises the earlier chapters, draws conclusions in substantiation of the thesis, and outlines directions for future work.

Chapter 2

The Cake language

We have established the problem of interface mismatch, and motivated improved tool support for the integration tasks which emerge from it. This chapter introduces a special-purpose programming language designed to fill the gap in tool support. This language is called Cake.

Cake is a language designed to make black-box adaptation a more effective option for the programmer than manual glue coding (in the style of the adapter pattern [Gamma et al. 1995]). We begin by motivating the language design by example, then survey its features.

2.1 Motivation

Currently, non-invasive adaptation is often eschewed by developers faced with integration tasks, in favour of invasive editing or from-scratch redevelopment. To understand why, recall that adapters currently consist of *wrapper* functions like the two in Fig. 1.4. For convenience we repeat the code here (Fig. 2.1). As before, the details are not important. Instead, observe several difficulties with this style of coding, summarised as follows.

Repetition A large volume of similar code is required for a conceptually simple task.

Poor modularity The code is not *trivially* repetitive. Each wrapper embodies overlapping sets of underlying logical rules for how certain kinds of value must be treated in certain contexts, e.g. how particular arguments must be treated. Consider `opc` in the figure: one case requires a bumped reference count and has different unlocking semantics from the other. The programmer must elaborate these rules correctly into what becomes a large volume of similar code, rather than expressing them directly and individually.

Inconvenience Among other headaches, to compile this code the programmer must construct a hybrid build environment supporting compilation against both interfaces.

Complexity This shows a very simple case, where functions correspond one-to-one. In others, complexity of the task easily escalates further.

```
int p2k_node_seek(struct puffs_usermount *pu,
      puffs_cookie_t opc, off_t oldoff, off_t newoff,
      const struct puffs_cred *pcr)
{
  kauth_cred_t cred;
  int rv;

  cred = cred_create(pcr); // convert auth token
  VLE(opc);              // lock vnode ptr
  rv = RUMP_VOP_SEEK(opc, oldoff, newoff, cred); // call rump
  VUL(opc);              // unlock vnode ptr
  cred_destroy(cred);    // destroy temp auth token

  return rv;
}
int p2k_node_remove(struct puffs_usermount *pu,
  puffs_cookie_t opc, puffs_cookie_t targ,
  const struct puffs_cn *pcn)
{
  struct componentname *cn;
  int rv;

  cn = makecn(pcn);      // issue temp name
  VLE(opc);              // lock vnode ptr
  rump_vp_incref(opc);   // bump refcount
  VLE(targ);             // lock target vnode
  rump_vp_incref(targ); // bump that refcount
  rv = RUMP_VOP_REMOVE(opc, targ, cn); // call rump
  AUL(opc);              // this time, vnodes were unlocked
  AUL(targ);             //... by rump, so just assert this
  freecn(cn, 0);         // free temp name

  return rv;
}
```

Figure 2.1: Example filesystem wrapper code (repeated)

In short, wrappers are an unnecessarily complex approach to adaptation. Cake is a language designed to fix this problem. Figure 2.2 shows some Cake rules sufficient to generate the wrappers in Figure 1.4. Again the details are not important, but notice several advantages.

Separation of concerns The Cake programmer writes rules which we call *correspondences*. Each rule localises a particular piece of domain-specific knowledge about the adaptation task. The compiler is responsible for composing rules into wrappers. In particular, notice here that rules concerning functions are kept separate from rules concerning values. Such rules form the basic Cake language (to be described in Section 2.2).

```
// rules concerning functions
p2k_node_seek(_, vn, oldoff, newoff, cred) ⟶ RUMP_VOP_SEEK(vn, oldoff, newoff, cred);
p2k_node_remove(_, vn as vnode_bump, tgtvn as vnode_bump, cn) ⟶
RUMP_VOP_REMOVE(vn, tgtvn, cn);

// rules concerning values
values puffs_cookie_t ⟶ ({VLE(that); that}) vnode;
values puffs_cookie_t ⟵ ({VUL(that); that}) vnode;
values vnode_bump ⟶({VLE(that); rump_vp_incref(that); that}) vnode; // also bump refcount
values vnode_bump ⟵vnode; // unlock not required
values puffs_cred (cred_create( this )) ⟶ kauth_cred;
values puffs_cred ⟵ (cred_destroy( this )) kauth_cred;
values puffs_cn (makecn(this)) ⟶ component_name;
values puffs_cn ⟵ ( freecn ( this , 0)) component_name;
```

Figure 2.2: Cake rules generating equivalent wrappers

Expressiveness Cake rules advance on prior work by supporting *context-sensitive* and *many-to-many* relations between interface elements. For example, a single function may map to different calls on the opposing interface (depending on what calls have come before), or to a sequence of calls (perhaps with later calls' arguments depending on earlier results). Similarly, sets of values or objects occurring together may be treated as a group, and corresponded by a single rule. These and other advanced features greatly extend the power of the Cake language (and will be described in Section 2.3).

Object structures The example in Fig. 2.2 passes only isolated objects across the interface. However, Cake can handle the exchange of arbitrary object graphs across mismatched interfaces. This can eliminate considerable code. For example, consider writing adaptation logic to walk a linked list, converting each element in turn. The Cake programmer need only specify how separate classes of objects relate; the Cake runtime automatically explores the graph, applying rules to the objects it finds (as described in Chapter 4).

Simpler, shorter code The rules in Fig. 2.2 may appear to be only a little shorter than the wrapper code. However, the entire p2k adapter contains not two but 28 wrapper functions. Each rule above contributes to *many* of these wrappers, and often many wrappers can be generated from a single rule. The result is shorter and simpler code, as we show in three case studies (as described in Chapter 5).

2.2 The design of Cake

The bulk of this section concerns the features found in the Cake language. However, before describing *what* the design of Cake looks like, we add a brief note about *how* the design has come about.

2.2.1 Design approach

Cake is a relation-based declarative language that is deliberately *special-purpose*: it does not attempt to replicate the feature-set of conventional languages. Given this vague description, however, it is not clear what the limits of the tool's application might be. For example, for what kind of adaptation task would the programmer be advised to abandon Cake and resort to a more conventional language?

Answering this question appears difficult because there are no pre-existing formalisms that capture a well-defined space of interface mismatch. Without defining one—which we deliberately do not attempt—we are not able to conclusively answer the question. However, our motivation for *not* defining one is precisely that even if we did have one, we would not be able to answer the question! A theoretically "complete" language can easily be unusable for some class of task. For example, many special-purpose languages such as Sed [Dougherty and Robbins 1997] or XSLT [W3C, 1999] are Turing-powerful [Kepser 2004], but in practice wholly unusable for tasks outside their narrow remit. With respect to our goals (§1.5), a theoretical completeness result would not answer any useful question.

Therefore, the approach in designing Cake has taken an open-ended approach. We do not seek to define a "complete", finished language. Rather, we allow examples to guide us towards a design that is sufficient for a large number of use-cases. Much later we will reflect (in Chapter 8) on future avenues towards retrospectively formalising these features; this ordering, of placing use-case discovery firmly *before* definition of any formalism, is consistent with our practical approach.

2.2.2 High-level view

Fig. 2.3 gives a high-level view of an application of Cake. The original components are bridged by some adaptation logic generated from Cake code. Note that while our examples show only two components, Cake applies equally well to compositions of arbitrarily many components, by decomposing the problem into the relevant set of *pairwise* interactions.

The Cake language consists of the following parts:

- **top-level constructs** for introducing modules which exist, and identifying modules which should be derived (§2.2.2);
- an **interface description and annotation language** for supplementing the descriptive information contained within modules (§2.3.3);
- a very small **algebra of composition operators** for describing increments and compositions of components (§2.2.2);
- within the algebra's `link` operator, a **rule-based syntax** for drawing correspondences between semantically similar or equivalent elements in mismatched interfaces, from which the bulk of adaptation logic is derived (§2.2.9 onwards);

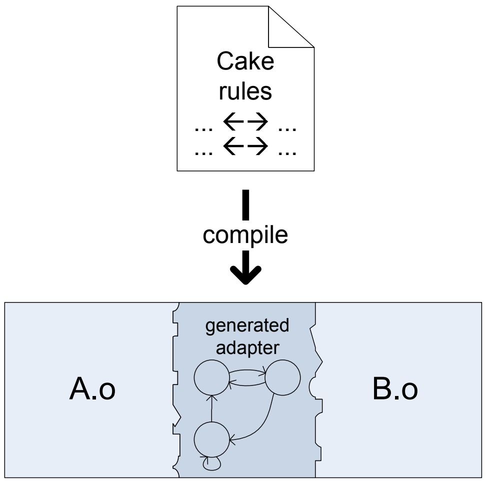


Figure 2.3: High-level view of an application of Cake

- an **imperative "stub" language** with generally familiar C-like syntax, late-bound semantics and deliberately constrained expressiveness (§2.2.9 onwards), used to describe sequences of calls (possibly with data dependencies) and "helper" computations;
- a **reactive execution model** in which Cake-generated code runs only in response to certain events (§2.2.10 onwards).

These are described in incremental fashion by means of a running example, which we now begin. A discussion of more subtle semantic issues is left until §2.4.

Our discussion of Cake develops a fairly large terminology; Appendix A provides a glossary which the reader may find useful. We will also see a variety of language features, the most important of which are summarised in Appendix B for subsequent ease of reference. A full grammar is in Appendix C.

2.2.3 Introducing the running example

We use a relatively ambitious running example to illustrate the design of Cake as a tool and a language. Can we take a client and library implementing hitherto unrelated interfaces and, by writing a succinct description of their corresponding features, glue the unmodified binaries together?

Consider a simple program which uses a library to decode some video. There are many possible choices of library; we consider a client written against the libmpeg2 library[1]. Suppose we wish to link this instead against the ffmpeg family of libraries[2]. There are many plausible motivations for this: perhaps to reduce the dependency footprint of a larger system, perhaps to exploit the larger feature-set of ffmpeg (which can decode video in

[1] http://libmpeg2.sourceforge.net/
[2] http://ffmpeg.org/

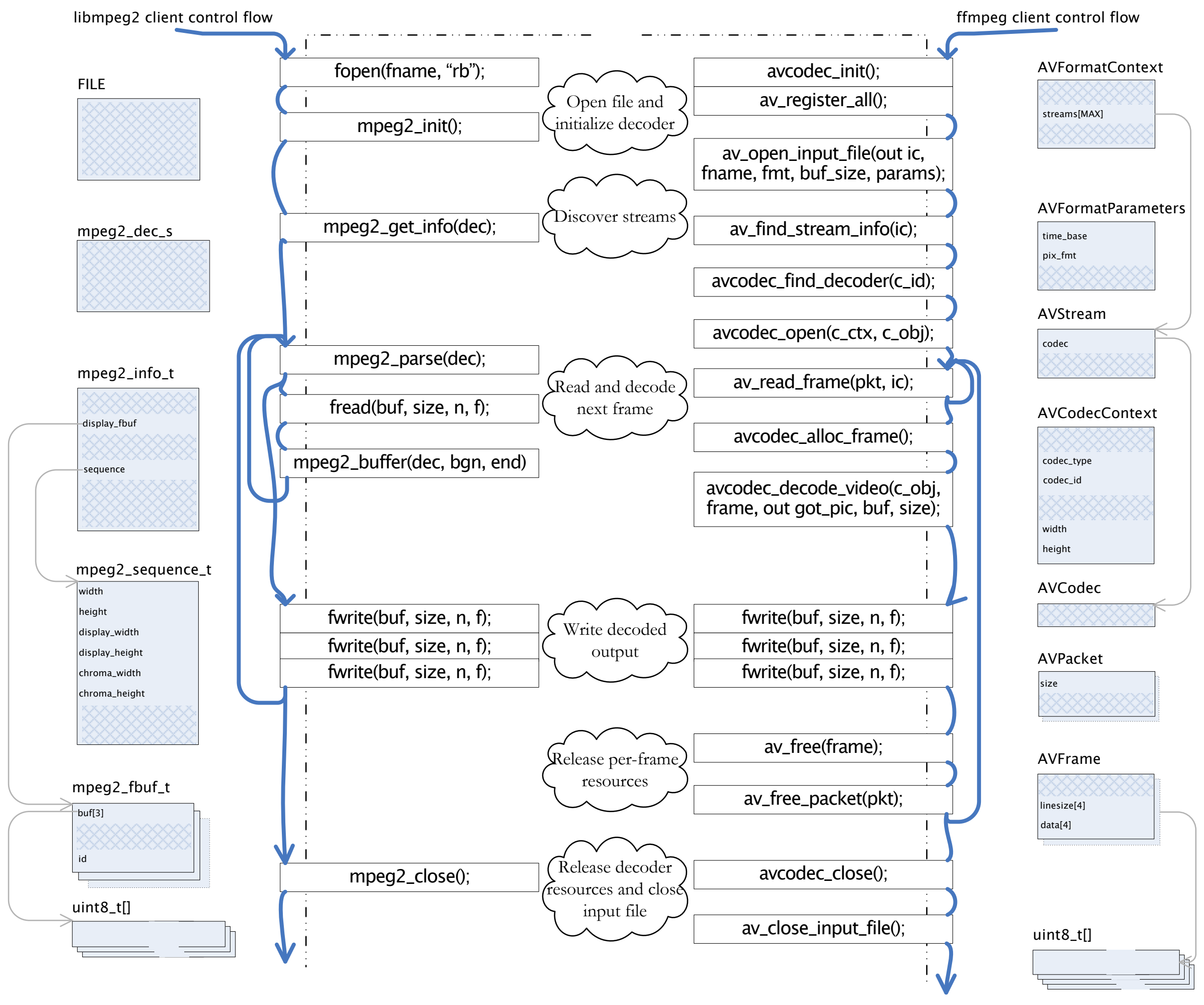


Figure 2.4: Example comparable usage patterns for libraries libmpeg2 and ffmpeg

other encodings than MPEG), or perhaps for differences in reliability or performance. Fig. 2.4 shows equivalent usage patterns of the two interfaces. Note that the correspondence between the two is nontrivial: in most cases there is no one-to-one correspondence between either the objects or the function calls used by the two interfaces.

2.2.4 Insights

Cake takes a black-box approach to adaptation. This is reflected in how Fig. 2.4 describes the two components only in terms of the function calls and data structures that they exchange.

A convenient formalisation of this notion of interface is the *trace* of a component's interactions, of the sort displayed by tools such as ltrace[3]. Fig. 2.5 shows the ltrace output

[3]http://ltrace.alioth.debian.org/

```
mpeg2_init()       = 0x9cd6180
mpeg2_info(0x9cd6180)        = 0x9cda380
mpeg2_parse(0x9cd6180)       = 0
mpeg2_buffer(0x9cd6180, 0xbfd17d88, 0xbfd18d88) = 0x9cd6180
mpeg2_parse(0x9cd6180) = 1
# --- snipped ---
mpeg2_parse(0x81bf180) = 0
mpeg2_buffer(0x81bf180, 0xbf9e2a58, 0xbf9e2a58) = 0x81bf180
mpeg2_close(0x81bf180) = 1
```

Figure 2.5: Example trace of a libmpeg2 client (libmpeg2 calls only)

for our client's interaction with the libmpeg2 library. Abstractly, a trace is simply a sequence of named calls or *events*, with each event communicating zero or more values from the traced component to its environment, while in the reverse direction each event receives an unnamed response optionally communicating a single value.[4] Cake code consists largely of rules which, abstractly, describe a *transducer* over this trace—that is, an automaton which both recognises and generates. At run time, conceptually, Cake-generated code feeds each component a trace generated from those output by the other components. Note that our discussion of traces is purely conceptual; a Cake programmer never needs to generate or manipulate traces in any way.

We note also from Fig. 2.5 that two orthogonal dimensions of *structure* exist in traces. Firstly there is "spatial" structure within each call, appearing horizontally—this structure includes the argument tuples, together with any substructure within those objects (and, implicitly, any heap structure reachable from them). It is familiar as function signatures and (concrete) data types in conventional languages. Secondly, there is a latent *temporal* structure appearing vertically, manifested in the ordering and co-occurrence relationships between calls over time. This is less commonly abstracted in programming languages, but is usually called a *protocol* or *channel contract* [Hunt and Larus 2007] when it is.[5] We will exploit both kinds of structure when writing Cake rules, but make unusually strong use of temporal structure. When referring to the *context* of an event in the trace, we will mean its *temporal* context unless explicitly stating otherwise.

2.2.5 Requirements

What kinds of rules are required for realistic adaptation tasks like our video decoding example? From Fig. 2.4 it is clear that simple mappings of function signatures and object fields are not sufficient, for several reasons.

- Correspondences between events are not one-to-one. In ffmpeg there is usually more than one call for each libmpeg2 call, so we require a way of mapping one call to many. Sometimes this relationship is reversed, so we need to recognise a sequence of many calls and map it back to a single call.

[4]This asymmetry in arity is of course the usual convention for procedural interactions, and indeed for functions in mathematics.

[5]Temporal structure is often captured in state machines, in which case state-oriented descriptions such as "typestate" [Strom and Yemini 1986] are used.

- Arguments to one call may not be sufficient to perform the corresponding call. For example, av_find_stream_info() corresponds fairly closely to mpeg2_info(), but the former needs a reference to the input file rather than the decoder object. Since only the decoder is passed to mpeg2_info(), and not anything representing the input file, we must somehow recover a reference to the file from the *context* of the mpeg2_info() call rather than its content.

- Components differ in the shapes of their data structures. Single fields or single objects may correspond to many fields or many objects. Moreover, objects may be passed indirectly, perhaps over many levels of indirection from the immediate arguments.

These imply that our transducer needs to be *stateful*, that it must support *context-sensitive* adaptations (where "context" must include at least temporal sequences of calls) and that it must be able to navigate *object structures* by following pointers. These requirements are manifested in two key features of the Cake language: *context predication* (to be described in §2.3.1) and its *object graph semantics* (to be described in §2.4.1).

2.2.6 Characterising Cake-generated code as a transducer

At run time, Cake-generated code maintains two kinds of state. Firstly, *blackboard state* enables matching of calls in a context-sensitive fashion. Secondly, *association state* tracks sets of semantically related objects collaborating across sequences of calls. These two kinds of state are analogous respectively to state machines and to aggregate or associative look-up structures that are often hand-coded in glue logic. In Cake, the programmer does not manipulate this state directly, but embodies the necessary state transformations in abstract declarative rules.

2.2.7 Toolchain context

Fig. 2.6 illustrates Cake's place in the toolchain. The Cake compiler inputs a collection of components (in the form of binaries) and some Cake code, and outputs Cake-generated source code, build rules for assembling the output binary (out of this code and the original binaries), and possibly some extended and relinked versions of the original binaries.

Fig. 2.7 shows the outline of a Cake source file. There are two main top-level constructs: exists and derive. The first of these identifies an existing component within the host file system—typically a relocatable object file—and optionally adds descriptive information to supplement the debugging information already present. Cake's interface model is based on DWARF 3 [Free Standards Group, 2005] and, in particular, its notions of "types" and "subprograms". The availability of debugging information is a huge convenience which we will assume when evaluating Cake as a practical tool. However, Cake does not demand debugging information—all such information can be supplied within the exists block. Certain annotations, beyond the expressiveness of DWARF, may also be added within the Cake source file (as described in §2.3.3).

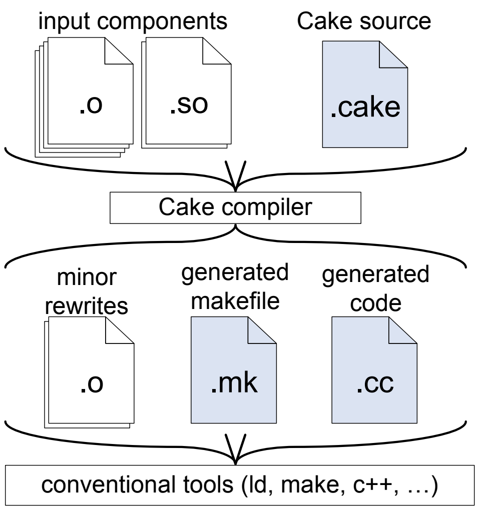


Figure 2.6: Cake's tool flow

name	function	arguments
link	link mismatched components	list of component identifiers; block of Cake correspondence rules
make_exec	generate executable from object file	object identifier
instantiate	link static data into object file	object identifier; structure to instantiate; name for instantiated object; symbol prefix for object members' linkage names

Table 2.1: Algebra of component derivation operators

Cake's other essential top-level construct is derive. This describes a new component to be created by assembly and adaptation of existing ones. Derived components are expressed in a simple algebra of built-in functions and operators. Table 2.1 lists the operators supported. The most important of these is link, which applies to a list of component names. Correspondences appear inside a block opened by a link keyword, and these account for the vast bulk of any typical Cake source file. Since correspondence rules always relate a *pair* of interfaces, rules appear in pairwise blocks, of which there may be many (in cases where link is applied to more than two components).

```
exists elf_reloc("foo.o") foo { /* optional info ... */ };
exists elf_reloc("bar.o") bar { /* optional info ... */ };
derive elf_reloc("foobar.o") foobar = link[foo, bar] {
  foo ⟷ bar
  {
    // your correspondences here...
  }
};
```

Figure 2.7: Skeleton of a simple Cake composition

```
1 fopen (fname, "rb" )[0]  ⟶  av_open_input_file(
2                                 out _, fname);
3
4 values FILE ⟷ AVFormatContext {};
5
6         mpeg2_init() ⟶ { avcodec_init();
7                             av_register_all (); }
8                      ⟵
9         (new mpeg2_dec_s);
```

Figure 2.8: Some simple Cake correspondence rules

2.2.8 Syntactic conventions

Arrows in Cake signify correspondence rules, and point in the direction of data flow. In Cake source code, correspondence arrows are rendered using angle brackets and double-hyphens. For example, the bidirectional arrow is `<-->`. In this dissertation they are typeset directly as long arrows ($\longrightarrow$, $\longleftarrow$, $\longleftrightarrow$). Aside from this, Cake's syntax is familiar from other languages, and is mostly C-like. For ease of recognition, we typeset all arrow operators specially: `->` denotes indirect member selection as in C, and is typeset $\hookrightarrow$; meanwhile `=>` denotes functional abstraction (as in ML [Milner et al. 1990]), typeset $\Rightarrow$; its converse `<=` (typeset $\Leftarrow$) binds names to function return values (described in §2.3.1).

2.2.9 Simple correspondences

Corresponding events Lines 1–2 in Fig. 2.8 define an event correspondence, stating that a call to fopen() with second argument "rb" should be translated to a call to av_open_input_file().[6] This is a pattern-matching construct; pattern-matching is central to Cake's design. The out keyword signifies that the first argument is an "output parameter" into which the logical "return value" of the call will be written; Cake automatically maps this to the return value expected by fopen() (as explained more fully in §2.3.7). Finally, the [0] qualifier matches only the first fopen() call in the client's execution—since it may want to open other files, for purposes other than video decoding. (A more flexible means for this kind of selective matching can be found in the notion of slices, discussed in §3.6.)

Corresponding values Line 4 says that a AVFormatContext object (on the ffmpeg side) can be created from a FILE object (on the libmpeg2 side) and vice-versa. In this instance, no further rules are specified and no fields are propagated between the two. This is sufficient since the FILE object is completely opaque to the client. If the objects were not opaque, we could add rules inside the braces to describe how their fields relate. If those fields were to include pointers to other objects, the Cake runtime would explore these and apply correspondences to construct and maintain the corresponding object graph as

[6]Readers familiar with the ffmpeg API may notice that we have simplified the arguments to the second call, for clarity of exposition.

pointers are passed around. In combination with the previous rule, Cake can now generate a wrapper for fopen() which calls av_open_input_file() with appropriate parameters and substitutes the AVFormatContext object with a FILE object on return.[7]

Compound statements and return Lines 6–9 describe initialization of the library state. The pair of ffmpeg initialization calls is given as a compound statement in Cake's "stub language", a simple loop- and recursion-free imperative language. Although syntactically C-like, this language is completely independent of the components' source languages (recalling that Cake deals only with binaries). A special *postfix arrow* syntax is provided to describe handling of a return event, on lines 8–9, saying that a new object of class mpeg2_dec_s should be allocated on return to mpeg2_init. Again, this object is treated opaquely, so we do not need to describe treatment of its fields.

2.2.10 Remarks on simple Cake usage

We may remark on the usage seen so far.

Dual scoping As befits a language describing relations, Cake has *dual scoping*: different sides of an arrow denote different components, in whose respective scopes names are resolved. The left-hand side of our rules always represent the libmpeg2 client, and the right-hand side represent always the ffmpeg libraries. This means that arrows may point left-to-right or right-to-left, according to which data flow the rule describes. Event correspondences are described using pattern-matching: the arrow-tail side (the "source") represents an *event pattern* that the event matches, perhaps supplying names; these are then bound on the other side (the "sink" side) to the elements they matched in the call. We can bind events to stubs in cases, as in lines 6–7 above, where the function correspondence is not one-to-one.

Rule selection Implicitly, for each event correspondence, the Cake compiler will select appropriate value correspondences for the values being communicated by the event. In this sense, value correspondences implicitly *quantify* over all event correspondences between the target components (in the sense of Filman and Friedman [2005]). In this example the Cake compiler can automatically deduce what value correspondences need to be applied; occasionally it is necessary to manually instantiate a value correspondence (see §2.4.4). Note also that these value correspondences only apply to interactions between our specific pair of components; they say nothing about e.g. how to treat FILE objects passed across other interfaces.

[7]Although these rules are called "value correspondences", a better name would be "structural correspondences", since two other Cake constructs, formally speaking, define correspondences between values: tables (§2.3.6), which handle enumerated data types and similar cases, and event correspondences, which can implicitly define relationships between address-taken functions (§2.3.9).

Execution model Cake's execution is purely reactive: Cake-generated code runs only when some event is triggered by one of the composed components. (We will discuss in due course certain possible extensions which would relax this—§3.5.8, §4.4.6.)

Programmer knowledge Like any programming tool, Cake depends on the programmer to understand the semantics of the domain. In writing the above rule, the programmer exploits two facts about the client's usage of the libmpeg2 interface: that it is accompanied by C library calls such as fopen() to do the file I/O, and that the *first* fopen() call opens the video file—signified by the [0] suffix to the pattern.[8] Similarly, the programmer is responsible for writing rules which, in combination, access the ffmpeg interface correctly, e.g. by inserting the av_register_all() call. A trade-off ubiquitous in all kinds of programming task arises here: the programmer must choose how generic or how task-specific to make his code. More general code is more likely to be reusable in future composition tasks, but requires a greater initial investment of effort.

Bilateral design Our language is designed to specify adaptations bilaterally—that is, with reference to a pair of interfaces at once and symmetrically. By contrast, many adaptation techniques have a *unilateral* nature: they describe adaptations performed on a single interface or piece of code, in the implicit hope that this yields a correct composition with whatever *other* interfaces are implicitly being composed against. For example, writing a wrapper function is a unilateral approach, since the *wrapped* interface is distinct from the *wrapping* interface syntactically and semantically. Moreover, it is left implicit that the wrapping interface is one actually consumed by some external code. By contrast, a bilateral approach references both interfaces simultaneously and explicitly. Among the motivations for bilaterality are readability, tool-checkability (since both interfaces are available for the tool to check compatibility against) and safety (since the adaptation is explicitly associated with all the interfaces for which it was intended). Greater expressiveness is also possible, in that a bilateral approach can express *bidirectional* correspondences with a single statement. (However, in the current Cake language, only simple correspondences may be bidirectional.) On the other hand, sometimes bilaterality is an unwanted constraint. For example, many adaptations are not specific to a single composition context but which rather present the underlying interface differently in some *generic* way—perhaps turning a synchronous interface into an asynchronous one, or altering the conventions for error reporting. Chapter 6 will describe extensions which allow Cake to capture these cases.

Higher-order perspectives Cake makes a clear separation between code (handled by event correspondences) and data (handled by value correspondences). Readers familiar with higher-order programming languages might wonder why this distinction is necessary. Surely functions can be treated as just another kind of object or value? Certainly they can, but functions are nevertheless a distinct kind of object, at least in that they support

[8]We briefly describe a cleaner approach to this class of rule, based on a more dynamic notion of components defined by *slices* of traces, as future work (§3.6).

an "application" operation, and in other ways too.[9] Cake's provision of two different kinds of correspondence is effectively a case analysis which exists to provide a set of primitives tailored for the qualitatively differing requirements presented by different kinds of object. It does not imply that functions may not be unified with other objects where this treatment is the most expressive. Cake's handling of function pointers (to be discussed in §2.3.9 and §4.4.3) exploits exactly that view of functions.

2.2.11 Correspondences for free: name matching

Cake automatically draws *implicit* correspondences between compatible like-named elements in linked interfaces. For example, if one module requires a function `foo()` and another provides such a function, an event correspondence is automatically drawn between them. This reproduces the behaviour of a conventional linker. (Since our current video decoding example is an instance of *unanticipated composition*, few names match, so the gains from name-matching are modest at best. However, name-matching is very helpful when applying Cake to *interface evolution*, where many interface elements can be matched without programmer intervention.)

Cake extends name-matching to structured data types. If two interfaces both define a class `bar`, then these will be corresponded; if one `bar` contains fields `amplitude`, `breadth` and `curvature`, and the other `breadth`, `curvature` and `density`, Cake will correspond `breadth` and `curvature`, and leave the others uncorresponded. This means that minor mismatches in size or layout of structures are automatically adapted around. For example, the implementation of the C library call `fstat()` [Kernighan and Ritchie 1988] often needs to adapt between kernel- and user-format `stat` structures, owing only to layout differences and extra fields. Cake could perform this adaptation automatically, assuming the corresponding fields have matching names. In the rare case where a given name-matching is not wanted, it can be overridden by mapping the name to an alternative element (if one exists) or `void`.

Differing identifiers often mask similar underlying structures. A complementary name-matching feature is the `pattern` construct, where a single rule can express multiple logical rules that are identical up to identifier substitutions. Identifier patterns (not to be confused with *event patterns* already described) and their rewritings are specified in a manner reminiscent of the Unix `sed` tool's `s`-command [Dougherty and Robbins 1997]. The fictitious example in Fig. 2.9 expresses three similar event correspondences in one rule. Value correspondence patterns are also allowed, to capture corresponding families of data structures in a single rule.

Conversely, meaningful identifiers are not always explicitly connected to their contexts of use. For example, integer fields may implicitly model enumerations or (as bitvectors) sets, whose elements are encoded as symbolic constants defined separately from field itself. This is especially common in C, because it provides no set abstraction, and in its earliest versions did not provide enumerated data types.) Function arguments may also be best

[9]Even in higher-order languages, functions are not usually modified or internally inspected at run time. Functions are arguably not instantiated at run time either, under lambda-lifting [Johnsson 1985] or similar approaches.

```
pattern /edit_(cut|copy|paste)/ (w, sel , ctxt)
  ⟶ clipboard_op_\1 (w, sel, ctxt);
```

Figure 2.9: Matching families of related function calls in a single rule

understood by their name rather than positionally, even though different declarations of the function (e.g. in C) may supply different names or no names at all. Cake supports a names annotation for applying a vocabulary of names to function arguments (to enable name-matching between provided and required functions) or integer fields (to enable name-matching between symbolic integer values). The syntax is similar to that used for selecting value correspondences: names may be supplied either in annotations within an exists block, or immediately when describing correspondences in a link derivation. Fig. 2.10 shows a realistic example using the former option.

2.3 More powerful features of Cake

We saw in §2.2.5 that simple correspondences are insufficient for real tasks like our video decoding example. This section discusses the features of Cake which make it sufficiently powerful to tackle these real-world use-cases.

2.3.1 Corresponding sequences of events: event context

The problem Often when performing an adaptation, considering each call independently is not enough: the correct action depends on what calls have come before. To this end, Cake event patterns may be prefixed by a *context predicate*: the rule only applies where certain preceding calls have occurred. Automatic management of the state necessary to match such patterns is another way in which Cake saves programmer effort. In our example, we use this facility when the client retrieves an object storing metadata about the video file: Fig. 2.11 shows one side of a rule telling Cake that a call to mpeg2_info() follows earlier calls to fopen() and mpeg2_init(), whose arguments and return values are significant.

Name binding Since it may be necessary to refer to values passed or returned during the preceding calls, context predicates can bind names to such values, just as event patterns bind names to function arguments. The Cake programmer uses the let keyword to bind names to return values and useful auxiliary values in this way. This does *not* denote assignment, in that the same name may not be re-bound within a rule. In patterns like Fig. 2.11 which bind names to return values of contextual calls, we can use the shorter syntax varname $\Leftarrow$ syntax instead of let. This appears in several subsequent examples.

Resolving ambiguity There is a potential ambiguity in context matching: *which* preceding call is relevant? When a call to mpeg2_info() occurs in our libmpeg2 client, the

```
// ... from headers used by client
/* Window types */
typedef enum
{
  GTK_WINDOW_TOPLEVEL, // == 0
  GTK_WINDOW_POPUP    // == 1
} GtkWindowType;
// ...
struct GtkWindow
{
        // ... IMPLICITLY, this uses values from the enum
        guint type: 4; /* GtkWindowType */
};

// ... hypothetical evolved headers used by library
/* Window types */
typedef enum
{
  GTK_WINDOW_UNDEFINED,// == 0 --added a NEW element which CHANGES numbering
  GTK_WINDOW_TOPLEVEL, // == 1
  GTK_WINDOW_POPUP    // == 2
} GtkWindowType;
// ...
struct GtkWindow
{
        // ... IMPLICITLY using values from the enum
        guint type: 4; /* GtkWindowType */
};
```

```
// Cake code linking the mismatched client and library
exists /* ... */ client
{
        declare { GtkWindow { type: guint { names GtkWindowType }; } }
};
exists /* ... */ client
{
        declare { GtkWindow { type: guint { names GtkWindowType }; } }
};
derive /* */ program = link [ client , library ]
{
        /* ... */ // annotations above ensure that name-matching is done on GtkWindow.type
};
```

Figure 2.10: Implicit use of enumerations in C code, and Cake names annotation

```
// here "..." matches any intervening call sequence
let f = fopen(fname, "rb"), ...,
let dec = mpeg2_init(), ...,
mpeg2_get_info(dec) ⟶// to be continued...
```

Figure 2.11: Matching calls in the context of preceding calls

```
/* ... continued */ ⟶ {
    av_find_stream_info(f) // in-place update to f
 ;& let dec...vid_idx = find( // Cake algorithm
      f↪streams, // among the file 's streams...
      fn x ⇒ // lambda! find the video stream
       x↪codec↪codec_type == CODEC_TYPE_VIDEO)
 ;& let codec_ctxt = f↪streams[dec...vid_idx]
 ;& let codec = avcodec_find_decoder(
                              codec_ctxt↪codec_id)
 ;& avcodec_open(codec_ctxt, codec)
 ;& codec_ctxt }
```

Figure 2.12: Describing data-dependent call sequences in Cake's stub language

client has opened its input stream and created a decoder object, but not yet associated the two with each other.[10] However, in ffmpeg the corresponding av_find_stream_info() call requires an input stream as an argument, despite mpeg2_info() passing only a decoder object. Somehow, we must match the incoming call with the relevant preceding fopen() call which opened the input stream. What if there have been *many* preceding fopen() calls? Cake assumes that related calls occur close together: it matches the *nearest* preceding fopen() (having appropriate arguments). This is expressed using the ellipsis (. . .) to extend our event pattern over unspecified intervening calls. The ellipsis acts much like "`.*`" within a regular expression [Dougherty and Robbins 1997], matching any intervening sequence, but ellipsis matches the *shortest* such sequence, rather than the longest as commonly matched by regular expressions. If we had left out the ellipsis, this would match only if the two calls occurred in direct succession (among all calls across this particular interface).

2.3.2 Generating data-dependent call sequences: stubs

Cake's stub language offers some special features for handling complex data-dependent sequences of calls. These are illustrated in Fig. 2.12, which provides the right-hand side of the the mpeg2_info() rule begun in Fig. 2.11. This demonstrates several features of Cake's stub language. Stubs, being short snippets of sequential code, are ubiquitous in Cake, but rarely contain more than a handful of statements. (Simple sink expressions, such as line 1 in Fig. 2.8, are in fact *singleton* stubs consisting of a single call.)

Error discovery Manually determining the success or failure from every function call can get very tedious. Every expression in the Cake stub language has a "success" or "failure" outcome, logically *separate* from any result value it may yield. Cake determines the success of a function call in a *style-dependent* way (as explained more fully in §2.3.3). The default style assumes that functions returning signed integers are successful iff they return zero, and that pointer-returning functions are successful iff they return non-null.

[10]By implication, the decoder object's fields are not initialized by the library until some time later, when the file header has been read.

This style is typical of a majority of C APIs, and many APIs in other languages. Calls that return neither a signed integer nor a pointer are considered always to succeed. Alternative error-reporting conventions are generally best captured in alternative styles, as outlined in Chapter 6.

Error handling Stubs are not expected to contain logic complex enough to warrant try–catch exception handling. Instead, expressions can be joined with short-circuit boolean connectives ;& and ;—, in an idiom similar to that found in Unix shell programming. Unlike the shell, success exists independently of the result value, so the connectives are distinct from the boolean operators && and ——. In the few cases where the style does not detect error status correctly, the programmer can explicitly describe success conditions using the success pseudo-variable and constants void (which yields no value but always succeeds) and fail (which always fails).

Binding Just as let binds names to values in context-predicated event patterns, it can bind names to values in stubs. These enable data dependencies between calls in a stub. The out keyword also binds a name, and is used when calling functions have output parameters (§2.3.7).

Associations Often, a stub must navigate a data structure to find relevant arguments to a call. The dot (.) and short arrow (->, typeset $\hookrightarrow$) have approximately C-like "access member" semantics in Cake. Analogously, the ... syntax is overloaded to denote "access associated": it enables formation and dereferencing of *associations* between objects or values. Associations are the mechanism for many-to-many value correspondences in Cake, and are discussed in §2.3.4.

Algorithms Traversal of data structures algorithmically is not something Cake is designed to express. However, simple algorithms are often indispensable when performing adaptation. Cake provides an effectively built-in selection of algorithms in the stub language. Here we see find denoting linear search. Algorithms are defined outside of Cake in an implementation-specific way. (Currently, we exploit the fact that Cake's back-end generates C++ code, and borrow most of the C++ standard library's algorithms directly. A *style-dependent* notion of lists and arrays, described further in §2.3.8, allows generation of appropriate C++ iterators.)

Lambdas Since algorithms sometimes take functions or predicates as arguments, simple functions may be defined as lambdas in the stub language. The expressiveness of this is deliberately constrained: lambdas may not contain other lambdas, and cannot refer to themselves, so cannot introduce recursion in the stub language.

2.3.3 Practicalities

We have now seen the basics of the Cake language. In this interlude we discuss several practical issues arising in the use of Cake.

Target representation Our chosen binary representation is *relocatable object code.* This means compiled native code, before linking, in a modern container format such as ELF [System V, 1997]. Most of our work has applied Cake only to static linking, but its approach applies equally to dynamic linking, in the sense that the necessary interposition techniques are supported by dynamic linkers (and indeed, explicitly provided for in the ELF standard).

Source languages Cake can compose components deriving from several source languages, limited primarily by the availability of implementations generating Dwarf information. Until Chapter 6 we will target only components written in C; this is partly because our implementation currently lacks understanding of some incidental features found in binaries originating in other languages (such as name-mangling, and various lesser-used Dwarf constructs), and partly to save consideration of alternative component styles (as just introduced in in §2.3.2) until later. Adding support for additional procedural languages is in most cases straightforward, although at run time, some cooperation with memory allocators is required (as explained in Chapter 4).

Obtaining debugging information Compilers usually require a command-line flag to enable generation of debugging information (typically `-g` for C compilers). Most software builds released to end users do not contain debugging information, but distributors often supply it as an optional extra.[11] There is considerable value in providing debugging information to users, for example in enabling generation of higher-quality bug reports. In the worst case, reverse engineering tools for recovering debugging information [Slowinska et al. 2010] may be useful, but commonly we envisage that compiler-generated information will be available.

Interface description As described earlier (§2.2.2), Cake allows programmers to supplement or replace available debugging information within `exists` blocks. For this, we devised a simple textual syntax for the relevant subset of Dwarf, of which Fig. 2.13 shows a small fragment. Our textual syntax does not include "source-side" features of Dwarf such as source code coordinates or line-number calculation tables.

Annotations The same syntax extends Dwarf by accepting certain annotations. For example, attributes `out` or `inout` can be made to function arguments, affecting how Cake applies value correspondences to values flowing into and out of a function call (to be explained in §2.3.7). (Some of these annotations could be useful to debuggers as well as to Cake, making them candidates for future versions of Dwarf.)

Checking interface prerequisites Interface description and annotations may be supplied under three levels of qualification: `check`, which simply checks that the underlying object file *already* satisfies the description; `declare`, the most common qualifier, which merges programmer-supplied description with any existing debugging information, but

[11]This is currently the case in Debian and certain other GNU/Linux distributions

```
// C declaration
// "foo is a function from int (call it 'a') to int"
int foo(int a);
// "count_t is a synonym for int"
typedef int count_t;
```

```
// Cake description:
// "foo is a function from int (call it 'a') to int"
foo: (a: int) => int;
// "int is 4 bytes of the 'signed' base type encoding"
int: class_of base signed <4>;
// "count_t is a synonym for int";
count_t: typedef int;
```

Figure 2.13: Interface description syntax

raises an error if contradictions are found; and override, which overrules any contradictory information in the object file.

Comprehension As with any programming tool, we assume that the programmer understands the interfaces he is coding against. In addition to debugging information, the programmer might use various means to gain this understanding: API documentation, source code, other code exercising the same interfaces, patterns mined from such code [Wasylkowski et al. 2007] or reverse-engineering tools [Balakrishnan et al. 2005]. The latter is especially relevant when Cake is used to compose binaries for which source code is not available. Although these means each have their shortcomings, we consider these as separable problems; in this work we assume that the combination of these techniques is sufficient to gain the necessary understanding.

Styles All components introduced by an exists block are interpreted according to a *style*. Styles are an abstraction mechanism designed to seamlessly support mixing and matching of object code adopting different sets of interface conventions, perhaps originating from multiple *packagings* (e.g. component systems, application plugins, etc.), language implementations or coding styles. Styles determine various higher-level interpretations which the Cake compiler applies to object code, including error-handling, treatment of lists, string handling and so on. The examples in this chapter use only one style, the "default style", which corresponds to the conventions typically found in components written in C. However, Cake is designed to accommodate multiple user-defined styles. These can greatly simplify the rules required when composing components with differing systematic ways of encoding common abstract meanings—such as list or set data-types, initialization functions, error reporting, and so on. Cake's support for styles is discussed in Chapter 6.

Instantiate Many clients dynamically load back-end components, such as plug-ins. To use Cake across these interfaces requires a small extra feature. Since the client does not call the back-end *directly*, but through an indirect dispatch table, we provide an instantiate primitive in the algebra of component derivations (Table 2.1), used alongside link in derive expressions. This constructs an instance of a given data structure—usually a dispatch table—and creates a new symbol for each element in the structure. This lifts table entries to first-class symbolic function names which can be used like any other in a link block.

Conveniences The inline construct is similar to exists, but allows a component to be supplied not by reference to an existing file, but by inclusion of a snippet of foreign source code embedded directly in a Cake source file. These snippets are lexed but not parsed by the Cake compiler, so any language with compatible lexical structure (up to balanced opening and closing braces) may be used; they are de-lexed and output as source files alongside Cake's other output, and compiled at the same time as Cake-generated code (as described in §4.2).

2.3.4 Many-to-many value correspondences

In our running example of libmpeg2 and ffmpeg, the structures maintained during decoding by the two libraries contain mostly the same information, but split differently among various objects. In general, while objects or values need not correspond one-to-one among different interfaces, we can often say that a *group* of objects corresponds to another group. Many-to-many value correspondences describe how to create and update values in one group from the various values in the other group. Fig. 2.14 illustrates this and some other advanced features of value correspondences.

Associations Each many-to-many value correspondence creates *associations* at run time. Each instantiated association is a tuple binding together several objects, optionally supplemented by some primitive values stored in the tuple (e.g. the video stream index vid_idx, in Fig. 2.14). Bindings are formed in stubs by applying the let keyword in combination with the "access associated" connective, written "...". These tuples constitute a dynamic relation maintained at run time, analogously with join tables in a relational database. A tuple persists as long as any bound object does. Association tuples may also contain auxiliary values—as opposed to pointers to the associated objects. Association tuples are the only state which may be explicitly mutated from Cake code, using the set keyword (syntactically similar to the let keyword).

Initialization versus update Value correspondences may distinguish *initialization* from *update*, as seen in the first rule in Fig. 2.14. When an object flows across an interface for the first time, Cake may need to instantiate one or more *corresponding objects* (co-objects). Initialization rules use an arrow suffixed with a question mark. When initializing the right-hand side in the figure, p will point to a new AVPacket object. Rules without the question mark are called *update rules.* In Fig. 2.14, no update rule is needed because the client never updates any state corresponding to AVPacket's fields. Alternatively, sometimes a co-object's fields have no analogous fields in the original object. Cake will initialize these fields (using the initialization rule), but will subsequently leave them alone (since there are no update rules), avoiding repeatedly re-initializing the fields at each traversal of the interface (which might clobber updates made earlier by code on the

```
values (dec: mpeg2_dec_s, info: mpeg2_info_s,
        sequence: mpeg2_sequence_s, fbuf: mpeg2_fbuf_s)
            ⟷ (ctxt: AVCodecContext, vid_idx: int, frame: AVFrame,
                  p: AVPacket, s: AVStream, codec: AVCodec)
{
  // ensure an AVPacket exists on any flow L-to-R
  void ⟶?(new AVPacket tie ctxt) p;
  // picture dimensions are in sequence and ctxt
  sequence ⟷ ctxt {
    // LHS "width" and "height" done by name-matching
    display_width ⟵ width;
    display_height ⟵ height; // here we assume a
    chroma_width ⟵width / 2; // 4:2:2 pixel format,
    chroma_height ⟵ height / 2; };

  // info.sequence always points to sequence object
  info.sequence (&sequence)⟵? void;

  // special conversion required for buffers
  fbuf ⟷ frame { // scanline treatment requires " artificial data types"
    buf[0] as packed_luma_line[height] ptr
      ⟷ data[0] as padded_line[ctxt.height ] ptr;
    buf[1] as packed_chroma_line[chroma_height] ptr
      ⟷ data[1] as padded_line[ctxt.height / 2] ptr;
    buf[2] as packed_chroma_line[chroma_height] ptr
      ⟷ data[2] as padded_line[ctxt.height / 2] ptr;
  } };

  // how to convert YUV "scanlines" between framebuffer formats
  // -- special correspondences for the artificial data types"
  values packed_luma_line ⟵ padded_line {
    void (memcpy(this, that, display_width)) ⟵ void; };
  values packed_chroma_line ⟵padded_line {
    void (memcpy(this, that, chroma_width))⟵ void; };
```

Figure 2.14: A sophisticated value correspondence

co-object side). The separation is asymmetric: if there is no separate initialization rule, an update rule will be used, whereas the reverse is not true.[12]

Primitive values Cake inherits from DWARF an understanding of all the common encodings of primitive values such as integers, booleans, characters or floating-point data. It can therefore deduce meaning-preserving conversions when passing primitive values between components.

[12]Initialization rules effectively define an object initialization or "construction" mechanism. Currently, component-provided constructor functions must be manually invoked from initialization rules where appropriate. Programmers could be relieved of this burden by pushing knowledge of constructors into styles, much like treatment of other language-specific concerns, as described in Chapter 6.

Tying The tie keyword can be used when allocating objects in Cake, to specify that the allocated object should be deallocated at the same time as the tied-to object. This is a common requirement in Cake, since the lifetime of state created by adaptation logic is often tightly dependent on some component-managed object. Tying therefore greatly reduces the need for explicit object freeing in Cake. Tying may be thought of as a generalisation of stack-allocated objects, contained subobjects, or the application of "resource acquisition is initialization" to heap memory allocation in C++ [Stroustrup 1997]. In all these cases, one object's lifetime is tied to that of some other allocation. Implementation of tying relies on the Cake runtime's ability to interpose on object deallocation, which is also used heavily by the Cake runtime internally (as described in §4.4.2).

Internal reference The unusual-looking rule describing info.sequence is used to describe the pointer structures *within* a group of objects. When creating an mpeg2_info_s structure, Cake needs to know that its sequence field should point to the mpeg2_sequence_s structure that is also participating in the association. Since this does not depend on any value from the right-hand side, void appears (to denote "no value"), but the syntax is otherwise identical to any other correspondence. In this way associations allow, but do not require, that each group of objects be related by internal pointer structures. Put differently, associations can be used to abstract away the distinction between explicit relationships between objects, using stored pointer structures, and implicit relationships captured by external associative data structures.

Applying functions A bracketed stub-language expression on one side of an arrow may be used to apply a function to the outcome of the source side, before it is propagated to the sink side. This is provided for cases where some computation is required in order to acquire the correct sink-side representation. To see why this is useful, consider a sink-side field storing an index into a table. Suppose the source provides only a pointer to the corresponding element; in this case, a search through the table is required to discover the index that needs to be stored. We can perform the search in a bracketed call to a helper function or Cake algorithm, effectively performing a conversion between pointer and table-index representations of the value. Such functions may be applied either *before* or *after* the conversions implied by other rules, according to which side of the arrow the expression appears on. Similarly, position relative to the arrow also determines which component's scope is used to resolve the names appearing in the bracketed expressions.

This and that The this and that keywords denote pointers to the local and remote-side representations of the value being described by a correspondence. These pointers can be useful when applying functions as values traverse the interface. In the example of Fig. 2.14, both interfaces describe some buffers containing the decoded data, but with a subtlety: the layout of the buffers is not quite identical. In ffmpeg each line is padded, whereas in libmpeg2 there is no padding. We supply special value correspondences for these, overriding the default handling of uint8_t arrays, using memcpy() in a padding-sensitive fashion on this and that to move the data.

2.3.5 Quantification and guards

Fig. 2.14 has one more distinctive feature: it introduces *artificial data types* in its padding-sensitive treatment of buffer copying. These describe special treatment for values occurring in particular contexts (either spatial or temporal); in the example, the artificial data types are packed_luma_line, packed_chroma_line and padded_line. These are introduced using the as keyword, and by default are assumed to be synonyms for the data types they implicitly replace (char[] in the example). However, the purpose of defining these synonyms is that can be given special value correspondences to change the handling of particular data items. In effect, the use of artificial data types is a form of annotation, used to select among alternative value correspondences (discussed further in §2.4.4). They are unified with C typedefs and other type synonymy features encoded in DWARF, meaning that existing type synonyms may be used as artificial data types.

Value correspondences are implicitly *universally quantified*; for a correspondence concerning data types A (locally) and B (in the remote component), the correspondence quantifies over all contexts where an A is passed into the remote component and a B is required there. This “where” clause is said to be a *guard condition* restricting an otherwise universally quantified statement. Similarly, we can say that event patterns quantify over all matching call events, but are *guarded* by the argument pattern that must be satisfied, and furthermore, may be *temporally* guarded by a context predicate. Artificial data types are a technique for guarding quantification of value correspondences, by identifying special contexts with the as operator. This restriction can limit the selection of both temporal contexts, by using the as annotation in a context-predicated event correspondence, and also *spatial* (or “structural”) contexts, by using the as annotation in value correspondences.

These guard mechanisms are sufficient for many applications of Cake. In Chapter 6, where we write rules that apply more generally than to a specific pair of interfaces, we will require a more general way to guard rules using explicit *guard predicates* and *metavariables* (variables ranging over matched values, calls or data types). Although these features logically belong in the core Cake language, we save discussing them until they are actually needed in Chapter 6.

2.3.6 Relating individual values

The value correspondences we have seen so far have described *structural* and, in some cases, *computational* relationships between whole *classes* of value at a time. In most cases, we have not had to express mappings down at the level of individual values. Sometimes, however, we must write correspondences at this case-by-case level. Consider explicitly mapping the individual elements of two enumeration types, where these do not have matching name vocabularies. Cake provides a table construct for this purpose. Syntactically this is much like a value correspondence, but containing mappings between constant values (identified as literals or symbolically) rather than fields.

```
strncpy:
 (dest: out char[len], buf: char ptr, len: size_t)
  ⇒ void /* was: char[len] ptr -- see note in text */;


strndup:
 (buf: char[] ptr, max_len: size_t)
  ⇒ char[] caller_free (free) ptr;
```

Figure 2.15: Enabling allocation adaptations on strncpy() and strndup()

2.3.7 Input and output parameters

Pointers are used to perform certain forms of parameter-passing. Cake's default style assumes that singly-indirected arguments denote "in-place update"—a value passes *out* of the call as well as *in*. Appropriate value correspondences will therefore run on both entry to and exit from the called function. These semantics ensure that balanced operations can be expressed (e.g. to insert locking and unlocking as an object traverses an interface).

A singly-indirected argument might also denote an *output parameter* (typically passing an uninitialised location in the caller's stack frame). Since the in direction may perform a (meaningless) conversion on contents of the uninitialized stack location, the argument can be annotated as out. Similarly, for objects no longer valid *after* a call (e.g. if deallocated during the call), we can annotate the pointer as an in ptr argument, preventing any value correspondence from running in the output direction.

Commonly, output values flow into caller-allocated memory. However, some interfaces return callee-allocated results instead. Given simple annotations, Cake can automatically adapt between simple mismatches in these caller-versus-callee allocation semantics. Consider the C library functions strncpy() and strndup() (shown in Fig. 2.15). In the first, a caller supplies its own buffer for output data to be placed in. The second instead returns a pointer to a new buffer, which the caller must free when finished. Thanks to the caller_free annotation, Cake will adapt a call to the first function so that it instead calls the second, by post-copying the callee-provided buffer into the caller-provided buffer and then freeing the former. Cake can also adapt a call to the second function so that it calls the first, by pre-allocating a buffer and returning it. Note that the real strncpy() returns a pointer to the output buffer dest passed as one of the arguments; we explicitly annotate the return value as void here because otherwise the Cake compiler would consider the function to have two output arguments—the return value and the output buffer parameter—and would flag an ambiguity, because it would not be clear which of these two outputs should be matched against the caller_free output of strndup().

The out keyword is used to bind names to output parameters of functions invoked by stubs, as seen in §2.2.9. It is also used similarly to let, to capture the output value of some expression and bind a name to it. Unlike let, out is used when the identifier is not fresh, but has not yet been bound to a value. Specifically, out some_ident = 42 implies that some_ident is an output parameter defined in the signature of the *source* function of the enclosing event correspondence.

When programmers design an API, the division between use of return values and output parameters is a somewhat arbitrary choice. It is therefore a common source of interface mismatch. Similarly, the ordering of function arguments is often arbitrary. In simple cases where there is a single output value, but there is a disagreement in whether it is passed as a return value or a pointer-based output parameter, the Cake compiler will automatically correspond these appropriately. (This happens in our libmpeg2 example in Fig. 2.8.) To allow stub code to straightforwardly overcome more complex cases of these mismatches using name-matching, in stub code Cake defines the pseudo-variables, in_args and out_args. These group the sets of input and output arguments (strictly, the set of variable bindings extant at the start of the stub, and the set of output bindings expected by the stub's context) as if they were supplied as a single structure each. (Note that these sets are overlapping in the case of inout parameters.)

These structures enable name-matching between arguments and structured values, by effectively granting named collections of arguments the same status as structured values (which are simply named collections of fields). Consider the fragment of Cake code in Fig. 2.16.The exists blocks set up a simple mismatch between a client expecting to receive a mean–variance pair in the form of a return value and an output parameter (respectively) and a library providing a call that returns the two values in a structured return value. Our first attempt at the simplest plausible Cake rule is incorrect, because Cake will not break apart the structured return value without instruction to do so. The second rule provides this instruction: the macro-like ellipsis after out_args implies that out_args should be treated as a set of bindings. This is much like the multiple simultaneous assignments supported by Python and other dynamic languages, of the form (var1, var2, ¡...¿) = expr which implicitly destruct a right-hand side tuple.

The final two rules in the figure illustrate other uses of the pseudo-variables. In the first, we see a pattern rule where in_args is used in a macro-like fashion to pass through unnamed parameters. Implicitly, structure elements are assigned to parameters by name rather than positionally—Cake never performs implicit positional matching, since this is inherently fragile. In the final rule, we see in_args used simply as a structure: some_other_function takes a single structure argument, and Cake will behave as if a value correspondence were defined between this argument and in_args.

2.3.8 Arrays and lists

Pointers may point to single objects or to arrays. As seen in Fig. 2.15 with char[], array syntax can be used to name a local field or argument which holds the length of the array. This effectively adds a data constraint annotation to the underlying DWARF information. Cake usually detects the size of arrays at run time (as described in Chapter 4) and applies value correspondences to each discovered element; in rare cases, length annotations are necessary to facilitate this discovery.

Cake also has a style-dependent notion of *iterables*. In the default style this includes arrays (either statically-sized, length-affixed, or explicitly terminated as with null-terminated strings) and singly-linked lists (recognised as any single-recursive data structure; assumed to be linear, not cyclic). These allow algorithms (find seen in §2.3.2) to be applied uniformly to any iterable data structure recognisable from the style's definition.

```
/* an "exists" block -- here simply for exposition, to define the signatures */
exists client // ...
{   declare
    {
      get_average : (arg: double, variance: out double) ⇒ average: double;
    }                                              // ^^^ named return value
};
exists library // ...
{   declare
    {
        dist_parms: class_of structure { average: double; variance: double; };
        get_parms : (t: double) ⇒ dist_parms;
    }
};

/* in a "link" block */
client ⟷ library
{
    get_average(arg, _) ⟶ get_parms(arg);
    /* INVALID: this would require breaking apart the
     * returned object into its constituent fields , but
     * the programmer has tried to use the same syntax
     * as if the return value were passed as a single
     * struct. */

    get_average(arg, _) ⟶ { out out_args... = get_parms(arg) };
    /* OKAY: the use of "..." signifies expansion of out_args
     * to a set of name bindings. */

    // arguments in event patterns -- name-match collections of arguments
    pattern /server_(.*)_request/ (...) ⟶ dispatch_\\1(in_args... );

    // use as a struct -- name match between collections of fields
    some_function(arg1, arg2) ⟶ { some_other_function(in_args) };
    /* some_other_function's argument is a structured object,
     * which Cake will put in correspondence with in_args.
     * This is valid Cake: we are just using in_args as if it were
     * an arbitrary structure type. */
}
```

Figure 2.16: Uses of in_args and out_args

2.3.9 Function pointers

Functions are just another kind of object. Although their internal structure is opaque to Cake, we already have a mechanism for describing correspondences between functions—namely, event correspondences. Passing a function pointer is equivalent to giving the recipient a capability to raise events across the interface between a pair of components. Therefore, Cake will handle the flow of function pointers appropriately as long as event correspondences are provided which specify a single unambiguous treatment of the pointed-to function for all calls to it. This means there must be some event correspondence defined

```
1  client ⟷ library
2  { register_callback (f, arg) ⟶ add_handler(f, arg);
3    notify_user_cb(message, aux) ⟵ _(message, aux); }
```

Figure 2.17: Adapting callbacks

across that interface whose sink expression is a simple invocation of the passed function (rather than, say, a multi-expression stub).

Fig. 2.17 shows how the developer describes how a function pointer may be passed from client to library by register_callback(). Line 2 adapts a minor mismatch in the callback registration interface: the client requires a call named register_callback whereas the library provides only a similar function called add_handler.

The event correspondence in line 3 is unusual because it does not specify a name for the called function, but simply uses the "_" syntax, meaning "some call" . This is because the call-site in the library is an indirect call, so does not statically name the function it is calling. Without line 3, or some other rule calling notify_user_cb from library, the Cake compiler would not generate code to interpose on the callback. This kind of code is "wrapper code", and is explained more fully in §4.2.6. The presence of this rule enables function pointers (such as notify_user_cb) to be correctly adapted as they are passed to the library, by substituting a pointer to Cake-generated wrapper code. (The implementation of this substitution is described in §4.4.3.)

For simplicity, our example here assumes that the callback interface itself, i.e. the signature of functions passed as the f parameter, is well-matched between the two interfaces. However, if additional adaptations are required on the function, they can be added as a lambda expression, e.g. wrapped around f in add_handler. Note that the range of adaptations permitted in this way is strictly smaller than those permitted by the whole of the Cake language—in particular, context-sensitive adaptations may not be expressed here.

2.3.10 Completing the example

One hurdle remains in our running example, which is to match up the decoder loops of the two interfaces. This requires no new Cake language features, so we present it as a commented fragment of Cake code, in Fig. 2.18. This is an exercise in matching of control structures. The key requirement addressed by these additional rules is the matching of the calls in the decode loops embodied in the respective interfaces. Since the loop in the called code proceeds strictly frame-by-frame, whereas our caller has a more general zero-or-more-frames design, the rules ensure that each iteration yields exactly one frame—a case supported by both client and library.

The question arises: what if we had been linking an ffmpeg client with the libmpeg2 library? This is the converse case of our example. Since the mpeg2_parse() function is free to return STATE_BUFFER an unbounded number of times, it would appear that the only way to capture this would be to introduce some kind of looping into Cake, so that the generated code could repeatedly buffer and parse new data until a frame was available. Section 3.5.6 discusses some possible solutions to this problem. In general, the Cake language is richer in its support for adapting between different decompositions of data, than in that for adapting between different control structures.

2.4 Semantic questions

Our examples so far have said little about Cake's precise semantics. We have no formal semantics for Cake, but here answer informally some questions raised by previous examples.

2.4.1 Component-managed state

When control passes from one component to another, we define Cake's semantics by saying that the program should behave *as if* the component's entire accessible object graph is carried over and transformed into a graph suitable for the component receiving control, according to the set of value correspondences defined. This is called the *object graph semantics*. The graph remains in that form until control flows back out of the component. This is an unsurprising behaviour for single-threaded components. Note that since components may save pointers between invocations, this transformation must apply to *any* object reachable from the component receiving control, and not only to objects reachable from any pointers passed as arguments to the call.

This informal description makes sense for single-threaded programs. To address the case of multithreaded programs, where control may be active in multiple components at any one time, we add that concurrent access from multiple components *is* permitted, and may occur at any time, just as if Cake were not being used. However, we give very weak assurance about *visibility of updates* between components, except to say that when some thread of control passes from component A to component B, all updates made by A become visible to B. Our ideal semantics would make updates immediately visible even without a control-flow crossing, but implementing this efficiently is not realistic on conventional hardware, since interposition on memory access is only supported through slow and coarse-grained memory protection techniques. We discuss these issues in the context of our current implementation in §4.4.2.

Another question mark surrounding the semantics of multi-threaded Cake compositions concerns the potential for *conflicting updates*. This follows from the practical concern that the *actual* behaviour of our implementation does not perform complete transformation at each control-path crossing, but rather is optimised by performing *replication* of objects which may be shared (logically speaking) across a mismatched interface. We discuss these subtleties, again in the context of our current implementation, in §4.4.2.

2.4.2 Run-time errors

Cake has no type system, nor does the language define any other rules for statically detecting the risk of undefined or abstraction-violating computation. Run-time failures

```
/∗ The loop in ffmpeg proceeds frame-by-frame, whereas in libmpeg2 each iteration
 ∗ might yield zero frames (in the STATE_BUFFER case) ∗or∗ one or more frames
 ∗ (in the STATE_SLICE case). Solve this by ensuring that each iteration yields
 ∗ exactly one frame---a case supported by both library and client. ∗/
mpeg2_parse(dec)[0]   ⟶ { void }
                      ⟵
            STATE_BUFFER;
/∗ Notice use of [0]: "the first call to mpeg2_parse() returns STATE_BUFFER" ∗/

/∗ Reading from the input file handle must also be mapped to an ffmpeg library
 ∗ call. Since success of fread() entails a return value of nmemb, we must return
 ∗ this, irrespective of the size of the frame actually read. ∗/
let f = fopen (fname, "rb" )[0], ...,
let dec = mpeg2_init(), ...,
fread(_, _, nmemb, f) ⟶ { { av_read_frame(dec...packet, f) ;& nmemb } ;| 0; };

/∗ Since ffmpeg handles input for us, no action is required on mpeg2_buffer(). ∗/
mpeg2_buffer(_, /∗buf∗/_, /∗buf + siz∗/_) ⟶{ void };

/∗ The client calls mpeg2_parse() to request decoded frames. This translates to a
 ∗ call to avcodec_decode_video(). Since our earlier call to av_read_frame() may
 ∗ have returned a frame from a different stream (e.g. an audio stream in the
 ∗ same file ), we have two cases to consider. These map directly to the libmpeg2
 ∗ constants STATE_BUFFER ("must read more data") and STATE_SLICE
 ∗ ("one or more decoded frames available"), distinguished by an if--then-- else. ∗/
f <- fopen (fname, "rb" )[0], ...,
dec <- mpeg2_init(), ...,
size <- fread(_, _, nmemb, f),
mpeg2_parse(dec) ⟶{ let frame_avail = (
  if dec...packet.stream_index == dec...vid_idx
  then { av_free( dec...frame ); // this is null-safe
         set dec...frame = avcodec_alloc_frame();
         avcodec_decode_video2(dec...codec_ctxt, frame, out got_picture, dec...packet );
         true } else false )
  }--
  ⟵ // return event
--{ if frame_avail then STATE_SLICE else STATE_BUFFER };
/∗ Notice the special reverse-arrow syntax for returning. Moreover, the special
 ∗ "--{" ("continuing") syntax retains all name bindings from the preceding stub. ∗/

/∗ Finally, we relate the state tear-down calls of the two interfaces. ∗/
mpeg2_close(dec) ⟶{ av_free( dec...picture ); avcodec_close( dec...codec );
                    av_close_input_file( dec...ic ) }
                 ⟵
 { delete dec };
```

Figure 2.18: Remaining rules in the libmpeg2–ffmpeg example

```
widgets_A ⟷widgets_B
{ // an event correspondence
  find_widget(descr)  ⟶  get_matching_widget(descr);
  values Widget ⟷ Widget
  { /* ... */ };
  values Window ⟷Window
  { /* ... */ };
}
```

Figure 2.19: Rules sensitive to static versus dynamic binding

in Cake-generated code, such as failures of event pattern matching or of field access (in stubs) can occur as a result of programmer error. Future work could improve on this using conventional type-checking techniques.

Field access may raise run-time errors in Cake rather differently from in, say, C code. This is because member access in Cake is semantically different from that in C: it is "dynamic". In other words, the set of identifiers that may be used after a member selection operator ("↪" or dot ".") depends not on static type information but on the dynamic class of the object being accessed. The next section discusses dynamic binding in Cake more fully.

2.4.3 Dynamic matching and binding

Suppose we write some Cake rules to adapt between two different implementations of a similar windowing toolkit, as shown in Fig. 2.19. Consider what happens when a pointer to a Widget flows across the interface, say as a parameter to some function. The pointer's target might be a Widget, or might be a Window, since we assume that Window is an object that either *is a* Widget (as defined by language-level subtype polymorphism) or *contains a* Widget at offset zero (implying that a pointer to a Widget might be usable as a pointer to a Window, without any adjustment).

We might attempt to fix this by declaring that Cake has "static rule binding" semantics, meaning only the rules of Widget would apply, because only Widget is used to describe the pointer's referent statically. However, this is unacceptable. Firstly, it assumes some sort of static type system providing a bound on what the pointer points to (i.e. that it is not simply a `void` pointer, in C terminology), whereas we wish to allow the possibility of Cake implementations supporting many diverse languages. Secondly, if the object really *is a* Window, the receiving component is entitled to reinterpret the pointed-to memory as a Window, and will most likely see garbage in the extra fields defined by Window (since the rules for handling those fields were not executed). (This betrays the fact that Cake's implementation necessarily involves deep copying.) Although we could attempt to shift this burden to the programmer by obliging him to insert downcasts within Cake rules, this would quickly become unworkable for the usual reasons: for many interfaces, the set of subclasses is *open*, meaning that the Cake programmer cannot conveniently enumerate them, e.g. in a switch-style construct, as would be required.

Debugging information describes the layout of memory locally to any allocated structure in a running program, including not only statically allocated data structures but also the local layouts of stack frames and structured values. However, the *global* layout of memory (including both heap and stack structure) evolves dynamically, so cannot be precisely described by metadata generated at compile time. This is the root of imprecision.

(We prefer not to call these DWARF-provided descriptions *type information* because they do not imply a static type system. However, DWARF terminology uses the term "type" pervasively to refer to data types, which in DWARF effectively serve as memory layout descriptions. Consequently we will occasionally use the word "type" in this DWARF sense.)

Cake has dynamic matching semantics. If the pointed-to object "is a" Window, then Window's rules must apply. For implementability purposes, we overcome this problem of static imprecision by imposing *well-behavedness* criteria on the input components. Specifically, we do so using the following steps.

- We define an "is a" relationship between DWARF types, informed by common programming idiom (such as object containment) as well as any explicit subtyping relationships recorded in DWARF information.
- We assume that the available DWARF information may be imprecise, but not incorrect. Most significantly, the *allowable* imprecisions are limited to the "is a" relationship defined earlier.
- We assume the availability of a dynamic points-to analysis. This is discussed further in Chapter 4.

We describe our algorithm in Chapter 4. However, we define our "well behavedness" criteria more precisely here. The following must hold.

- We have debugging information available at run time, sufficient to describe any object that our Cake-generated code or runtime might encounter (either as a function argument or return value, or by pointer traversal from these). This need not cover the whole program: code which shares no objects with Cake-composed components need not be described. Note that this information, or parts of it, may be supplied by the Cake programmer, as described in §2.2.7.
- Given DWARF information describing an allocated region of storage, any information about the encoding of data contained *within* that region must be sufficiently accurate to select correctly among the defined correspondences which might apply to the data stored in that region (but *not* necessarily to select correspondences for *pointed-to* data). Clearly, this assumption would be invalidated by practices such as packing pointers into integer fields. However, it does not require that pointers are precisely typed (since pointers are handled by our next assumption). Undiscriminated C `union`s are a problem, since debugging information is rarely sufficient to select what correspondences to apply to the union's contents. Additional annotation can address this in some cases (as described in Appendix E).

- Any DWARF information describing memory *outwith* the information's local region—i.e. information describing pointed-to objects—may be imprecise (and not incorrect) only up to the *is-a* relation, which we define fully in Chapter 4.

- Components which arithmetically adjust pointers to navigate within arrays must only send and receive pointers to the array that are precisely typed (with respect to the element type) or untyped (i.e. `void`), according to the debugging information as augmented by any annotations added by the Cake programmer. This is not a significant restriction, because the ability to perform correct pointer arithmetic rests on knowing the precise array element size. We explain this restriction more fully in Chapter 4.

These assumptions effectively bound what questions a Cake implementation must be able to decide about the intended interpretation of data being passed around at run time. In particular, it must be able to decide what correspondences should be applied to the contents of some block of memory. In order to do so, it may assume that debugging information is available, not incorrect, and not unreasonably imprecise. We describe how our runtime implementation answers these questions in the next chapter.

The dynamic "is a" test is reified as an `is` operator, available for use in the Cake stub language. For example, we can write the expression `w is Window`, yielding a boolean. Since the Cake runtime takes care of common cases of exploring and copying object graphs, this is used only occasionally, but we will see an example in Chapter 6.

2.4.4 Rule precedence

The final semantic issue we consider is of rule precedence. Clearly, rules may overlap in the values or function calls which they match.

Event correspondence precedence

Currently, Cake takes the following approach to precedence in event correspondences.

- At most one event correspondence rule will be fired by any event occurring at run time. (However, an event may advance the call sequence recognition state with respect to more than one rule. These rules are the context-predicated event correspondences described in §2.3.1.)

- The rule matched is the "most specific match", as this is likely to be understood by the programmer. For example, a rule that matches any value, in some position (say a particular argument position) is less specific than an otherwise identical rule which matches a single value in that position. Similarly, a rule matching in a specific call-sequence context is more specific than a rule specifying no such context predicate. It is an error if there is no unique such most-specific rule.

- The Cake compiler will raise a compile-time error if ambiguous matching of event patterns is specified by the programmer. Since event patterns only ever match arguments in "match all" or "match specific value" forms, ambiguity is straightforward for the compiler to detect. Therefore, no additional run-time errors are introduced from pattern ambiguity.
- Incompleteness of rules is not addressed until run time. This means that unlike the case of multiple ambiguous rules, which is detected by the compiler, *omission* of a rule *can* introduce run-time errors. These errors most commonly occur on function calls for which there is no correspondence defined.

Value correspondence precedence

Value correspondences need not be so strongly disambiguated at the point where they are defined. This is because additional disambiguating context is usually available at the point where they are applied. Specifically, all value correspondences are implicitly invoked within an event correspondence or another value correspondence. In these contexts, both a "from" and a "to" data type are usually available. Ambiguity in either one of these may therefore be tolerated, if their pairing is sufficient to select a unique correspondence. A second reason for tolerating ambiguous declarations is that it is usually convenient to supply disambiguating annotations inline in the relevant Cake code, using the `as` keyword (§2.3.5).

Initialization is a special case. When a value correspondences application requires initializing a new object, no "to" object is available, so there is greater potential for ambiguity. We rely on either unambiguous correspondences or explicit annotations in these cases.

2.5 Summary

We have seen a tour of the Cake language, motivated by practical examples, and discussed its informal semantics. The next chapter is an interlude discussing the language's limitations and how it might usefully be extended to overcome them. Less curious readers may skip directly to Chapter 4, for a discussion of implementation, or Chapter 5 for a practical evaluation of the language.

Chapter 3

Limitations of the Cake language

As we would expect from a design originating in examples, currently Cake is a relatively simple language which handles many common cases well, but has some apparent limitations. In this interlude chapter, we briefly consider these. This is intended both to satisfy the reader's curiosity, and to help develop a more thorough understanding of the Cake language. It will also serve as a source of ideas for plausible future extensions to Cake.

3.1 Event context predicate connectives

Event context predicates (§2.3.1) allow matching of function calls occurring in the context of a particular temporal sequence of calls. The language describing such sequences is particularly simple: apart from atomic event patterns, it provides only the comma connective (concatenation) and the ellipsis ("any sequence", a restricted form of iteration). Alternation and negation would be useful additions to this language. To see why, consider the rules in Fig. 3.1, adapting from an escape-coded character I/O interface (on the left) to a non-escape-coded version (on the right).

Fig. 3.1 illustrates a recurring idiom in Cake, where sequences of events are matched both immediately, individually (as in line 2) and also subsequently as context (lines 3–4). This repetition is somewhat undesirable, but necessary: owing to the "most specific match" rule, without the more specific rule on line 2, passing a backslash would fire line 1 and cause the character to be immediately emitted.

Alternation and negation in event patterns would be useful to eliminate subtleties and ambiguities between rules. Consider a sequence of backslashes being emitted by the left interface, as in the trace shown in Fig. 3.2. What determines whether the middle

```
1  putchar(a)      ⟶ putchar(a);  // common case
2  putchar('\\')   ⟶ {};  // do nothing  right  now...
3  putchar('\\'),   ...,     // ...  but  react  in  subsequent  calls
4  putchar(a)      ⟶ escapechar(a);
```

Figure 3.1: Non-obvious context matching precedence

```
putchar('\\');   # matches line 2, adding blackboard state
putchar('\\');   # matches line 4, clearing blackboard state
putchar('\\');   # matches line 2, adding blackboard state
```

Figure 3.2: Trace showing subtle rule matching behaviour

of the three calls matches line 4 (a contextful event pattern with universally quantified argument) or line 2 (a context-free pattern with specific argument)? Currently this is resolved by the "more specific" relation, which prefers to match contextful any-value rules over context-free specific-value rules. Intuitively, this is justified by a desire to consume pattern-recognition state as soon as possible, much like the shortest-sequence-match rule (§2.3.1), rather than letting it accumulate. However, in general it would be preferable not to leave the rules' semantics down to this sort of tie-breaking. Additing negation and alternation could allow the programmer to clearly resolve the ambiguity, much like in a regular expression, by predicating the rule on line 2 on a context where the previous call was *not* an unescaped backslash.

3.2 Cake and classes of formal language

The problem Cake's event patterns solve is more complex than simply recognising a sequence of calls. This is because in a procedural setting, execution cannot proceed without somehow deriving a return value for *each* call in turn. For this reason, Cake matches *single calls*, but optionally distinguished by their *temporal context*.

The contextual part of an event patterns—i.e. the qualifier, not the matched call—can be seen as a restricted form of non-greedy regular expressions, extended with data-dependent matching. Concatenation is the main connective; alternation and negation could be added, and a limited form of iteration (as in regular expressions' "*" metacharacter) is available in the ellipsis (...). However, note that the primary motivation for capturing contexts in this way is to form bindings referencing specific values passed in contextual calls (§2.2.5). In a hypothetical iteration-based rule A(), (B(x))*, C(), the variable x lacks an unambiguous binding, because it is within an iterated subpattern matched zero or more times. It is often acceptable to approximate this rule as A(), ..., C(), which is supported by the current Cake language. Cases where only *some* patterns of intervening calls should enable a match are sufficiently rare that we have omitted such support from the current language, but it could be added straightforwardly. (Note that the nested variable x effectively describes a list comprehension; specifying the binding of x as the output of a reduction operation on this list could make such variables useful.)

It is interesting that our event pattern language fails to express the full set of regular languages; yet, its data-dependent matching features capture some languages that are not context-free (e.g. in Fig. 2.11). This apparent incongruity need not indicate a flaw. Instead, it is perhaps a reminder that, as recently expounded by Jim et al. [2010], these classifications of formal languages are frequently a poor fit for real-world problem spaces outside the narrow domain of programming languages. (Recall that the "language" we are contrasting here is the language of call traces, not the Cake language itself, which is context-free.)

As another example, consider the example by [DeLine 1999, p. 100] of procedurally transmitting a set of data items split in two different ways: one-per-call versus many-in-one-call. Handling mismatches between these in Cake requires an explosion of rules, since there is no way of matching a context of n specific calls where these may have occurred *in any order* without enumerating all possible orders. This reaffirms our observation that traditional abstractions of formal languages are not necessarily the best guide as to what connectives Cake should have. The join calculus [Fournet and Gonthier 1996] provides just such an order-independent conjunction and might be a useful basis for extensions to Cake.

3.3 Cross-cutting logic

Cake's grammar restricts code to a relatively rigid structure: adaptation logic consists of a set of rules within a `link` block. The only supported reference between these rules is reference to value correspondences, either from event correspondences or from other value correspondences.

Therefore, Cake will fail to capture commonality between rules if that commonality cannot be captured by a value correspondence. This is particularly significant in stubs. If a recurring pattern of sequential logic is required in multiple stubs, that logic may need to be repeated in-line in each stub. This may be avoided, by factoring the logic into a value correspondence, in cases where the logic is associated with the flow of a particular class of value across the interface. Typically, such rules would use bracketed stub expressions attached to the correspondence arrow (§2.3.4).

3.4 Calling conventions

"Calling conventions" refers to machine-level protocols for making procedure calls using a combination of stack, register and static storage. Cake is capable of adapting across mismatches in many *standard* calling conventions. Here "standard" means calling conventions defined by languages. This limitation is practically motivated: information about the language of a given function is available in DWARF, and most host systems will feature a C++ compiler that supports many such conventions. (Recall from §2.2.7 that Cake depends on a conventional toolchain on the host system; our implementation emits C++ code, as described in Chapter 4.) For example, the Cake compiler could automatically detect and adapt around the mismatch between a Pascal-style caller and a C-style callee, using the host C++ compiler's `extern "C"` and `extern "Pascal"` declarations. By contrast, neither Cake nor DWARF has any means of explicitly describing alternative calling conventions, so cannot be used to describe adaptations between arbitrary mismatched calling conventions.

3.5 Control structures

Cake's features mainly address data flow. Event correspondences describe the flow of data through calls, and value correspondences describe structural transformations on that data. Cake has considerably less expressiveness in resolving mismatched control structures. We discuss these limitations now.

3.5.1 Procedural bias

Although the abstract design of Cake is phrased in terms of *events* rather than procedure call and return, concretely the current language is biased towards procedural interaction. Specifically, there is distinct syntax for procedure return, in the form of the "return syntax" (§2.2.9). Moreover, the current implementation (as detailed in Chapter 4) interprets object code in a particular way, such that only outgoing calls are matched by event patterns, and return syntax is the *only* way to describe treatment of return events. In C-like languages, this is not a practical limitation of expressiveness. However, languages with more general control structures, such as coroutines or generators, will require an extended implementation.

3.5.2 Threaded control models

To support a wide range of mismatches arising from multithreaded control models, threading support could be added to Cake's stub language—for example to allow spawning of threads to satisfy asynchronous interface requirements given a synchronous implementation. Note that at present, these use-cases can be tackled simply by calling the thread management API like any other function from a Cake stub.

3.5.3 Blocking versus nonblocking calls

A likely mismatch between control structures is between nonblocking and blocking variants of the same interface. To relieve the programmer of the burden of performing these adaptations, in a manner similar to the handling of allocation expectations (§2.3.7), annotation-enabled detection of mismatches and generation of appropriate glue logic would be possible in at least some cases. For example, adapting from a nonblocking implementation to a blocking one could be achieved by inserting a polling loop, and Cake could insert such logic automatically given appropriate interface annotations. In the reverse direction, we could adapt a *cancellable* blocking implementation to a nonblocking one by spawning a thread and aborting it if no input was returned within some short time interval. If only a non-cancellable blocking interface were available, more invasive adaptation would be necessary—for example, instrumenting the blocking call's implementation to exit early rather than waiting.

3.5.4 Conversational adaptations

The combination of Cake's reactive execution model and the syntactic form of event correspondences constrains the control flow that can occur when responding to an event. When a call event occurs, some caller-side logic may run (in a bracketed stub expression preceding the arrow), followed by some callee-side logic (usually in a sink-side stub expression) and optionally some further logic on the return path (using the return syntax from §2.2.9). In this way, at most two transitions between caller and callee context are permitted. Further alternation of control between caller and callee components cannot be expressed in a single rule. We call these unsupported cases "conversational adaptations" after their back-and-forth nature. This lack of support is rarely a limitation in practice, since any need to invoke alternative caller-side functions can usually be factored into the value correspondences pertaining to the call.

3.5.5 Call sequencing constraints

Many interfaces contain call sequencing constraints. Cake supports call context matching (§2.3.1) which provides the theoretical expressiveness necessary to tackle finite-state cases of call sequencing mismatch, much like various prior work [Yellin and Strom 1997; Passerone et al. 2002; Bracciali et al. 2005]. However, Cake is somewhat less usable, because a greater amount of work is pushed to the programmer. Specifically, since Cake does not assume any description of the behavioural constraints to be available in the debugging information, the programmer must provide a full elaboration of the sequencing rules in the correspondences he writes. By contrast, the same prior work assumes that input components are annotated with sequencing constraints; from these they algorithmically synthesise a full adapter from a specification consisting, in effect, of only a subset of the rules that a Cake programmer would need to write. The motivation for this omission is purely practical: DWARF does not include any behavioural descriptions.

3.5.6 Generalised sequence mappings

We saw in §2.3.10 that adding some sort of looping construct within Cake appears necessary to handle some use-cases, e.g. where a single call in the caller's interface requires an unbounded number of repetitions in the callee, as in the converse case of our video decoding example.

Rather than adding explicit looping or recursion constructs to the stub language, a better approach would be to generalise event correspondences so that both sides of a rule can describe sequences of calls. This implies that Cake-generated code could generate sequences of calls as well as consume them. For example, we might write something resembling the rule shown in Fig. 3.3.

The right-hand side of this rule introduces a sequence-matching operator * for matching a specific sequence of calls, and use of boolean operators (here &&) to allow matching to be predicated on return values. Sub-patterns are grouped using braces; note that both sides of the rule use event-pattern syntax, so despite the braces, there is no stub. Thanks

```
f ⇐ av_frame_new(), ...,
av_read_frame(dec...packet, f ),
( dec...packet.stream_index == dec...vid_idx) &&
  avcodec_decode_video2(dec...ctxt, _, _, _)
                                              ⟷ { nmemb ⇐fread(buf, size , nmemb, _),
                                                   mpeg2_buffer(dec, buf, buf + nmemb ∗ size),
                                                   state ⇐ mpeg2_parse(dec)
                                                    && state == STATE_BUFFER }∗,
                                                 state ⇐ mpeg2_parse(dec)
                                                  && state == STATE_SLICE;
```

Figure 3.3: Sketched extension to sequence matching

to this, the rule is bidirectional—it could be used in compositions where either component is the caller. In the left-to-right direction, a simple pattern of calls from some ffmpeg client triggers the Cake-generated code; the compiler-generated code is then obliged to generate a sequence satisfying the pattern on the right.

Since the right-hand pattern is guarded on return values—only particular values of state will do—the code must exploratorily generate calls to mpeg2_parse() in the hope that these will eventually satisfy the right-hand pattern. In this way, elaboration of looping is managed by the Cake compiler and runtime, and described by the programmer only declaratively (using the * operator).

The main problem here is that this exploration process could fail—for example, if a call to mpeg2_parse() returned neither STATE_BUFFER nor STATE_SLICE. It is not clear what should happen in this case. An extreme solution would perform the exploration in a transactional memory, and roll back if a conflicting pattern emerged. At this point, other rules for handling avcodec_decode_video2() could be attempted, or else no action taken. However, the performance cost of such an approach could be great.

There is another problem with implementing this rule: the rule leaves treatment of return values and error-handling underspecified. For example, how should the call to av_frame_new() be handled within libmpeg2? Similarly, in the right-to-left direction, our pattern is predicated on the return values of the mpeg2_parse() calls, but we have no rules to specify how these calls should be handled. It would be unwieldy to incorporate all these details into a single rule. There is also insufficient description of error handling—what if the fread() call failed? These problems of underspecification suggest that the kind of rule shown could only work when treated as *partial* specifications, effectively acting as constraints on a synthesis process within the Cake compiler. Other rules could be provided alongside, entailing additional constraints. This implies an extra compilation step in which partial rules are combined into complete descriptions of the handling of each particular event; this process could fail if the collection of rules proved under- or over-constrained. By contrast, in the current design of Cake, these issues are avoided by requiring that a given event pattern, once triggered, is associated with a *complete* description (usually a stub) of the behaviour it generates in the facing component. We consider other facets of this limitation next.

```
client ⟷ lib // simple case
{
  a() ⟶ x(); // no initialization rule
  b() ⟶ y(); // ... makes for simple code
  c() ⟶ z();
};
```

```
client ⟷ lib // initialization causes blow-up:
{
  a() ⟶ { init (); x() }; // match uninitialized
  _ (...),
  a() ⟶ x();          // match initialized
                      // "_(...)" means "any call"
  b() ⟶ { init (); y() };
  _ (...),
  b() ⟶ y(); // repeat for each call
             // that might come first!
  c() ⟶ { init (); z() };
  _ (...),
  c() ⟶ z();
};
```

Figure 3.4: Cross-cutting sequencing constraints causing rule blow-up

3.5.7 Partial event handling

A semantic limitation of Cake is that each rule is “fired” by exactly one event at run time, and moreover, each event fires at most one rule. Put differently, the handling of one run-time event—excluding updates to sequence-recognition state—must be described *totally*, by a single event correspondence. This is sometimes a limitation, as we just saw in Fig. 3.3.

A simpler yet ubiquitous example is the initialization call. Suppose that a particular library requires an initialization call, but a mismatched client does not make any such call. In semi-automatic systems, a synthesis algorithm would infer the need to insert initialization calls before any library call, simply from the presence of a “happens before” constraint edge from the initialization call to other library calls. By contrast, in Cake this is a highly awkward case, because of the nonlocal (cross-cutting) nature of the initialization constraint. Since every call in the interface is affected, every Cake event correspondence consequently needs to incorporate knowledge of the constraint. Fig. 3.4 shows this difficulty for a simple pair of three-call interfaces of which one features an additional initialization call. An improved Cake language could factor the initialization logic out cleanly by allowing each event correspondence to provide a *partial* description of the event’s handling, using call sequencing constraints within an interface to schedule the partial handling logic.

Note that by contrast, capturing *non-crosscutting* sequencing constraints in Cake does not present this problem; this is simply the normal case of sequence-based event context predicates, of which our video decoding example from Chapter 2 contains several occurrences.

3.5.8 Caller–callee mismatch

One kind of mismatch in control structures arises from asymmetries in the procedural execution model. Data can pass between components both during procedure call and

producer polarity / out-port mechanism	data flow	consumer polarity / in-port mechanism	adaptation
caller / call-site	$\longrightarrow$	caller / resume-site	mailbox
caller / call-site	$\longrightarrow$	callee / entry point	none (passes by call)
callee / exit surface	$\longrightarrow$	caller / resume-site	none (passes by return)
callee / exit surface	$\longrightarrow$	callee / entry point	pump

Table 3.1: Cases of procedural control-flow mismatch for a simple data flow

procedure return; components may be mismatched in the ways in which they render a particular data flow procedurally.

To map the space of these mismatches, consider a *calling* component—we say it contains a *call site* (the instruction making the call) and a *resume site* (the place where control resumes after the call, usually the following instruction). The call site is a *out-port* for data—the arguments are "sent" by the component, whereas the resume site is an *in-port* in that it is where the component "receives" the return value. The respective converses of these for a *called* component are *entry points* (which "receive" a function's arguments, hence being an in-port) and *exit points* (which "send" a function's return value, so are out-ports).

Now imagine a simple producer–consumer relationship between two components A and B. For simplicity we consider only a single direction of data flow. Table 3.1 summarises the ways in which this can be programmed procedurally, including mismatched cases, and the adaptation requirements arising. Note that the source and sink mechanisms are determined entirely by the pairing of caller-versus-callee distinction (i.e. the style in which the components' control flow is coded) and data flow direction (which is always from a source to a sink, and from A to B). The "mailbox" and "pump" adaptation requirements were previously identified by Black [1983].

The first problem with expressing the "mailbox" and "pump" cases in Cake is that there is no syntax for "cross-wiring" call events with return events—respectively, wiring a call-output to a return-inport (mailbox) or return-outport to call-inport (pump). As a syntactic limitation, it is easily addressed.

Both mailbox and pump have more significant problems than providing appropriate syntax within Cake. In the case of mailboxes, the complication amounts to stack management complexity at run time. In the case of a pump, it amounts to looping. Both of these are omitted from Cake currently; we discuss these briefly now.

3.5.9 Realising mailboxes

Considering data-flow only, a mailbox can be expressed in Cake, albeit somewhat clumsily, using an association. Fig. 3.5 shows an example. The clumsiness arises because each side needs an object to "hang" its association state off: recall that association tuples are the only kind of mutable state that may be directly manipulated in Cake (§2.3.4). Since association tuples can only be created from component-managed objects, we suppose that object-returning functions someLeftFunction() and someRightFunction() are available.

```
values (o: someLeftObject, buf: char) ⟷ (o: someRightObject);

// LHS puts items in the mailbox
obj ⇐ someLeftFunction(), ...,
putchar(a) ⟶ { set obj...buf = a; };

                // RHS retrieves items from the mailbox
{ obj...buf ; } ⟵ obj ⇐ someRightFunction(), ...,
                                        getchar(a);
```

Figure 3.5: Describing a mailbox in Cake

```
/* Decompression code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (c == 0xFF) {
        len = getchar();
        c = getchar();
        while (len--)
            emit(c);
    } else
        emit(c);
}
emit(EOF);
```

```
/* Parser code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (isalpha(c)) {
        do {
            add_to_token(c);
            c = getchar();
        } while (isalpha(c));
        got_token(WORD);
    }
    add_to_token(c);
    got_token(PUNCT);
}
```

Figure 3.6: Tatham's coroutines example

Such a mailbox is not sufficient, however, because it does not ensure a correct interleaving of control flow between producer and consumer. For example, there is nothing to stop the consumer reading the same character repeatedly, or the producer from overwriting a character before it has been read.

One solution, currently supported by Cake, is to realise the mailbox as a synchronised single-cell buffer, with the help of a threading library. The producer and consumer functions can simply be bound to the buffer's accessors.

However, this introduces additional run-time complexity, in the form of a threading library, and additional programmatic complexity, since access to state which the two components may incidentally share (that is, aside from the buffer) will require synchronisation. It should be possible for Cake-generated code to ensure a well-matched interaction without resorting to this kind of opaque solution. The rest of our discussion describes a possible approach to this.

This kind of mailbox scenario is often used to motivate coroutines, as a minimally invasive form of cooperative multi-threading. To illustrate, we use an example due to Tatham [2000] and reproduced (with permission) in Fig. 3.6.

Considering these two snippets each as a separate component, we see that they are mismatched in their control structures: both expect to perform both input and output by calling functions, then to regain control. Input calls—`getchar()`—pass no value, and receive the input character by return. Output calls—`emit()`—pass a character value and return no value. Suppose we want to link the parser component to the decompressor, so that it inputs characters from the decompressor's `emit()` output stream. We clearly cannot link these two snippets directly; instead, we need to "cross-wire" them with respect to calls and returns. A call to `emit()` must generate a *return* event resuming execution of a corresponding `getchar()`, and conversely, the call to `getchar()` must *activate* the decompressor at a position immediately after where it last emitted a character. This precisely models the "mailbox" scenario in Table 3.1. (In this case, the "mailbox" is small, containing only a single character, so can be implemented by the stack- and register-variable locations used to pass data to and from coroutines.)

A deeper problem with mailbox adaptation is that of on-stack state. Each component is a caller, so expects its local variables to persist across communication with its opposing component. (This is one reason why Tatham's mismatch example is more common than the converse "pump" case—it arises precisely because the caller style of programming is more convenient, in that it allows maintaining local state.) Traditional coroutine implementations do this by a special control-transfer primitive which retains the yielding coroutine's stack frame for later resumption.

Can we compose the two components unchanged, without rewriting either into a callee style? Tatham presents a portable source-level approach to this in C, by providing macros allowing a called function to recover its state across successive calls using a combination of macro-generated jump labels, goto statements and static local variables. (He also outlines an extension to support reentrant and thread-safe state management.) Effectively the only source-level changes required (other than "begin" and "end" macros and making local variables `static`) are to identify which call-sites should resume a previously activated coroutine, versus which should instantiate a new one; and conversely, which return-sites should be resumable later, versus which should terminate the coroutine.

Cake would require support for an analogous adaptation at the binary level. There are two particular challenges to performing this. Firstly, we require a way of supplying analogous programmer guidance about which call- and return-sites should activate coroutines—where this must now be specified *separately* from the original source code, rather than by modifying it. Identifying call- and return-sites by source code coordinates is one plausible approach, although fragile with respect to source edits. Secondly, we require run-time support for saving the stack frame and program counter of a returning function, and restoring it on resumption. This is straightforward to implement for simple cases where stack frames do not contain pointers that need relocation on restoration. (By contrast, relocation might be necessary if frames contain internal pointers or pointers into other coroutines' stack frames, since this state may have moved elsewhere by the time a call is resumed. This may be implemented by dynamic analysis, assuming sufficient debugging information is available at run time to identify which fields in the saved stack frame are pointers.)

3.5.10 Realising a pump

Having considered mailboxes, there is the converse case to consider: instead of being the caller, what if both components expected to be the callee? In these cases, a "pump" rather than a buffer is required, in the form of a loop which repeatedly calls first the producer and then the consumer. Since Cake's stub language features no looping or recursion, this cannot be expressed directly in Cake.

As with the mailbox scenario, a somewhat unsatisfactory solution is available already in Cake, by use of a threading library. Specifically, the Cake programmer may code the pump conventionally in the user's language of choice, include it in an `inline` block in the Cake source, and link it as a separate component. The pump could then be invoked from event-correspondence stubs which coincide with availability of new data from the consumer, or could be started in a separate thread from some stub invoked at initialization time.

Alternatively, as an improvement on the use of `inline`, a generic pump could be coded in C++ and added to the repository of Cake algorithms (§2.3.2). This remains unsatisfactory because doing so requires implementation-specific knowledge of the Cake compiler.

As with mailboxes, the most satisfactory solution involves adding cross-wiring syntax would allow the user to wire a callee out-port—i.e. an exit-site—with a callee in-port—i.e. a call-site. This is sufficient to allow the Cake compiler to infer when a pump is required. Unlike the mailbox case, there is no stack management concern—since both components are callees, they must already explicitly save local state across invocations.

3.6 Component structure

Cake currently identifies "components" by interfaces visible statically in object code, delineated by file boundaries. Often, however, the same static set of functions and data-types can realise logically quite different components at run time. For example, two `FILE` objects (and the traces of C library calls manipulating them) might each constitute a distinct logical component to which different adaptation rules should be applied. A refined notion of component interfaces as "slices" of a trace, where slices are identified by event sequence patterns much like those already used to match events context-sensitively (§2.3.1), could support such use-cases. Unlike existing sequence patterns, slice patterns must accommodate potentially infinite sequences; as such, an iteration (star) operator as sketched in §3.2 would be essential, in order to capture the entire slice (of unbounded length).

3.7 Bidirectionality

Currently in Cake, only the simplest correspondences may easily be made bidirectional. In future work we hope to unify stubs and patterns somewhat, so that more rules can be naturally bidirectional. For instance, a stub which does `a(); b()` can be treated as a pattern which matches the sequence `a()`, `b()` in the reverse direction. We saw a sketched example of this sort of rule in Fig. 3.3. Stubs which restrict themselves to reversible programming constructs could be interchangeably rendered as patterns in this way. This could yield an analogue of the bidirectional *lenses* [Foster et al. 2005] supported for tree-structured data, but effectively extending these to programmatic interfaces.

3.8 Recursive and side-by-side application of Cake

The output of Cake is simply another component, albeit with added dependencies on Cake run-time services. Therefore, the Cake language may be applied multiple times in the construction of any program, and no special support is required for this. However, doing so presents several *implementation* complexities. We discuss these briefly in Chapter 4.

Motivating the design of Knit [Reid et al. 2000] was an observation that the basic linking model in most toolchains is an unsophisticated "grab bag" which does not scale well to large compositions or certain composition patterns. Instead, it demonstrates the adoption of a pre-existing hierarchical linking model, Units [Flatt and Felleisen 1998]. The same arguments apply to Cake; Knit has shown that hierarchical linkage can be implemented in a very similar practical context to Cake's.

3.9 Summary

This chapter has facilitated a better understanding of the Cake language, primarily by negative examples. In so doing, it has identified many areas for future work. The following chapter returns to more immediate issues, by discussing the implementation of Cake.

Chapter 4

Cake implementation

In Chapter 2, we described the Cake language. To satisfy our goals, it must be possible to implement a compiler and runtime for this Cake language. The compiler must operate on binaries (satisfying our convenience goal) and makes modest assumptions about its input code (to enable a wide range of input code). The runtime must support the dynamic semantics described in §2.4.3. Preferably, such an implementation should also integrate well with existing toolchains, to further ensure the convenience, maintainability and debuggability both of the resulting tool and of the code it generates.

In this chapter we present the technical details of such an implementation. We first outline our implementation strategy, then discuss the compiler implementation, and finally the Cake runtime.

4.1 Strategic decisions

Our language design brings several implementation constraints. In particular, we are committed to a *black-box* abstraction, to the use of *binaries*, and integrating as conveniently as possible with existing toolchains. The use of binaries is the biggest constraint: these already embody highly specific object layouts and encodings, and are described only by the available DWARF information, perhaps augmented by programmer annotation.

Interposition mechanisms Since Cake's execution model is reactive, somehow we must interpose on (or "hook") events of interest during our components' execution. Various mechanisms are available for this: interposition support of the linker; generation and handling of traps supported by the operating system; perhaps direct modification of code in memory. Operating system traps are highly dynamic, and more flexible than linker-based approaches, since availability of memory protection traps could allow us to interpose on access to state as well as data. However, they are also generally considered a "slow path" by implementors, and are also esoteric and system-specific. Direct modification approaches have similar disadvantages. We will therefore prefer to use linker-based mechanisms for interposition.

Replication versus canonicalisation Our input components understand logically similar (i.e. corresponding) data structures by many different in-memory representations. We could opt to store a unique "master" copy of this state, and re-encode it on demand from mismatched components, perhaps using memory protection techniques. On a trap, we could emulate the trapped instruction (although this would be slow). Alternatively, we could give it a demand-generated re-encoded copy of the data; this would need to live in memory long enough for a candidate component to read from it, but not so long that the master copy is left unduly out-of-date, or that the temporary copy constitutes a resource leak. It is unclear how to manage these copies. Alternatively, we can use replication: replicate state in each of its alternative representations. This increases memory use, but provides faster access (since no traps are handled). It also brings the usual questions surrounding replication: how to handle conflicting updates, timing of update propagation, and consistency. We choose the replication-based approach—it is straightforward in many simple cases, which will be sufficient for proof-of-concept. For more complex cases, the two approaches are likely to be complementary, as we discuss in §4.4.6.

Rewriting Even though our language presents a black-box abstraction, there is no reason why its *implementation* should not use invasive techniques which actively consume and rewrite the internal instruction sequences of our input components. Indeed, such a technique could be used to improve the performance of a system considerably. For example, we could avoid the need for replication by rewriting memory-access and address-calculation instructions to reflect a changed object layout. Rewriting approaches offer mainly performance benefits over simple interposition approaches. Since high performance is not an immediate goal, we leave these for future work.

4.2 The compiler

Cake models a program as a set of communicating object files—or more properly, groups of object files, which we call components. Communication occurs along the control path of the program; an "event" between two components' interfaces occurs when control flows out of one component and into another. Cake is implemented by interposing on these events: Cake-generated code runs when events occur. The defining characteristic of a component is that it is internally well-matched—no interposition is necessary on events *within* the component.

4.2.1 Assumptions

The current implementation of Cake assumes that inter-component data flow occurs only through function calls, *or* through shared objects whose sharing was established at run time through function calls (i.e. by passing a pointer in an earlier function call). This "functions only" assumption allows us to implement Cake almost entirely by interposing on function calls; we can interpose on object sharing by intervening at the point of the function call during which sharing is established. We discuss the specific treatment of objects further in §4.4.

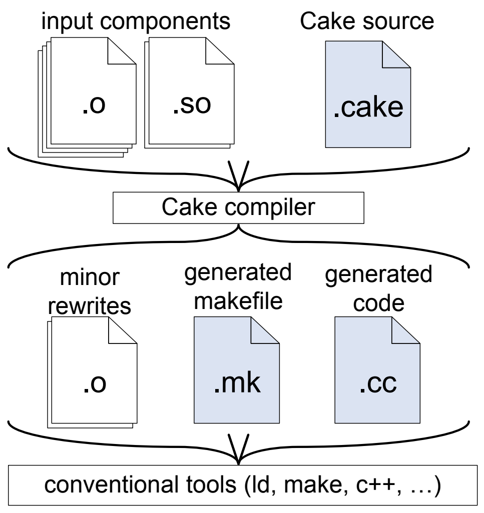


Figure 4.1: Cake's tool flow (repeated)

The "functions only" assumption might be violated by statically allocated shared variables, where sharing is established at link time. In practice, globals shared among components are rare, and mismatch in the usage of such globals is even rarer. (Where shared globals do exist, their interface is usually that of a standard library, for example the C library's errno, which is highly unlikely to suffer mismatch.)

4.2.2 Outline implementation

Fig. 4.1 repeats the diagrammatic view of Cake's tool flow shown in Chapter 2. The Cake compiler accepts a Cake source file as input, which in turn references various pre-existing object files and describes others which should be constructed according to correspondence rules. The compiler's execution consists of deriving a set of wrapper functions which interpose on communication between these object files, and a set of helper functions which implement the various value correspondences (defined in the Cake rules, jor implied by name-matching).

Our compiler generates C++ code. Interaction with the existing object files is achieved using a specially-created dwarfhpp tool. This tool generates specially-crafted C++ headers which reproduce the ABIs described in DWARF information. Fig. 4.2 provides a more detailed view of the common-case toolflow (excluding use of shared objects and rewrites to input objects).

4.2.3 Interacting with object code

Our tool dwarfhpp generates C++ headers which reproduce the binary interface exported by an object file, such that C++ code can be generated and compiled against that binary interface. The decision to generate C++ is helpful in supporting components written in *many* source languages, since C++ is a particularly featureful language. In other words, its feature set is sufficiently close to the union of most other procedural and object-oriented

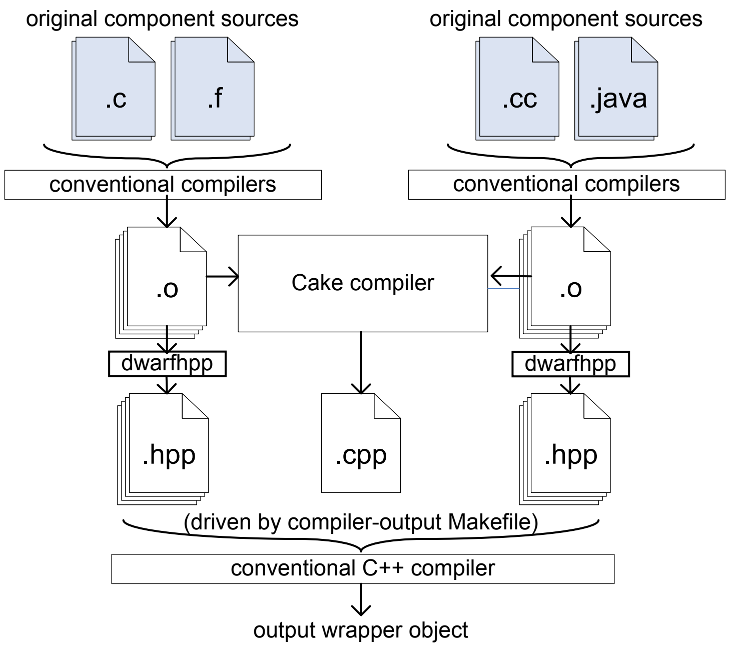


Figure 4.2: Detailed common-case toolflow

languages' feature sets that generated C++ headers can provide a fairly direct rendering of most DWARF-described features.

Note that the dwarfhpp tool is not restricted to binaries generated by the host C++ compiler. Indeed, the headers it generates need not particularly resemble the source that generated its input binaries. For example, the Cake compiler always generates structured data types as C++ structs rather than classes. Another difference is that the generated headers name as many definitions as possible: definitions left anonymous at source level, such as anonymous structure types, will have autogenerated names in the output headers, so that Cake-generated code can refer to them even if the component source happened not to.

The C++ language also proves useful in supporting features in DWARF that are not native to C++. For example, primitive values of non-native encodings, sizes or endiannesses can be reconstructed by adding an appropriate C++ data type definition to the compiler's internal library. Thanks to C++'s comprehensive data abstraction support, our Cake compiler can consume these using exactly the same code as it generates for compiler-native objects; it suffices to provide a class definition describing the non-native encoding up to all the operations which the Cake compiler might invoke on a variable encoded in this way (i.e. all the operators in the Cake stub language).

In a few cases, standard C++ does not support quite the level of control necessary to capture a DWARF-described binary interface. In these cases, dwarfhpp uses compiler-specific attributes, where available. In particular, attributes are required to ensure that structured data types' field padding and alignment match the DWARF descriptions. Consequently, currently our implementation supports the attributes of the GNU C++ compiler only, but many other compilers provide similar extensions; support for these could be added straightforwardly.

There is no guarantee that a given DWARF-described interface can be captured in C++ headers. For example, functions whose calling conventions are not supported by the host compiler's `extern "`*`language`*`"` options would be problematic. Similarly, nested subroutines would at least have to be rendered in some substantially different way (e.g. non-nested subroutines passing a static link explicitly). However, we are yet to encounter these limitations in practice. It is worth reiterating that our approach supports a vastly wider range of compiled code than simply output of the host C++ compiler, and in particular, it applies to code written in languages other than C or C++ (even if this support is limited in our proof-of-concept implementation).

4.2.4 Compiler overview

Fig. 4.3 shows a simplified view of the internal operation of the compiler, including its internal control structure, data structures and internal data flows. This is mostly straightforward; the remainder of this section will focus on the output files and the details of their generation.

4.2.5 Output of a Cake invocation

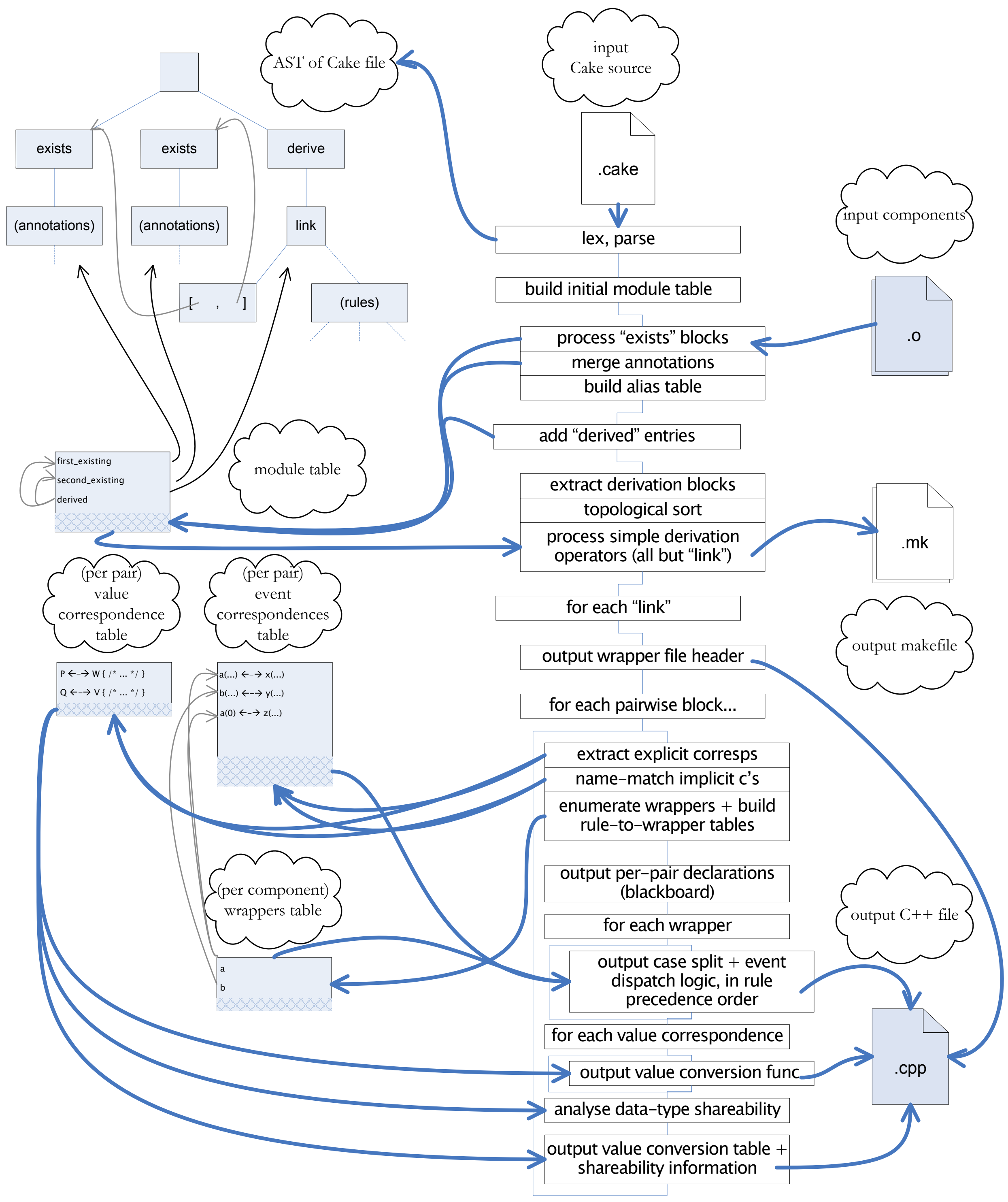
The Cake compiler outputs a set of wrapper functions, in the form of C++ code consuming dwarfhpp-generated headers, and a POSIX makefile containing linker invocations to interpose these wrappers. Wrappers are interposed statically using the GNU linker's `--wrap` option [Free Software Foundation, 2009]. This redirects references to a named symbol, so that instead they are relocated to a symbol `__wrap_`*`<symbol>`*, and renames the definition of the named symbol, if any, to `__real_`*`<symbol>`*.

4.2.6 Anatomy of a wrapper

Wrapper functions are the primary mechanism for Cake's implementation. Each wrapper function structured in a particular way, as shown in Fig. 4.4.

Limitation to similar signatures Since the linker performs wrapping on a per-symbol basis (§4.2.5), a single wrapper function must provide logic for handling *all* calls made to a particular symbol name. To fit generated code within C++'s statically-typed function call model, we also require that all such calls have compatible signatures, up to the data type of all immediately-passed values; indirected values are reified as pointers to untyped (`void`) memory.

Limitation to non-variadic functions Interposing on variadic functions is not currently supported. This is because discovering the interpretation of variadic function parameters at run time is not possible without some application-specific knowledge: even



AST of Cake file
input Cake source
.cake
lex, parse
build initial module table
input components
.o
process “exists” blocks
merge annotations
build alias table
add “derived” entries
extract derivation blocks
topological sort
process simple derivation operators (all but “link”)
.mk
for each “link”
output makefile
output wrapper file header
for each pairwise block...
extract explicit corresps
name–match implicit c’s
enumerate wrappers + build rule–to–wrapper tables
output per–pair declarations (blackboard)
for each wrapper
output C++ file
output case split + event dispatch logic, in rule precedence order
for each value correspondence
output value conversion func
.cpp
analyse data–type shareability
output value conversion table + shareability information
exists
exists
derive
(annotations)
(annotations)
link
[,]
(rules)
first_existing
second_existing
derived
module table
(per pair) value correspondence table
(per pair) event correspondences table
P ←–→ W { /* ... */ }
Q ←–→ V { /* ... */ }
a(...) ←–→ x(...)
b(...) ←–→ y(...)
a(0) ←–→ z(...)
(per component) wrappers table
a
b

Figure 4.3: Simplified view of Cake compiler internals

```
int __wrap_some_function(t1 arg1, t2 arg2 /* ... */)
{
    update counters

    append call to blackboard

    if (blackboard matches a context- or counter-predicated rule)
    {
        dispatch to that rule
        save return value on blackboard
    }

    (similar tests for other possible matches of this kind)

    else if (call matches a rule requiring specific arguments)
    {
        dispatch to that rule
        save return value on blackboard
    }

    (similar tests for other possible matches of this kind)

    else if (call matches a fully general rule)
    {
        dispatch to that rule
        save return value on blackboard
    }

    else // probably an error
    {
        dispatch to __real_some_call(arg1, arg2 ...);
    }

    update blackboard with call return value
    purge obsolete blackboard state

    synthesise return value from value saved by dispatch logic
}
```

Figure 4.4: Anatomy of a Cake-generated wrapper

distinguishing an integer from a pointer is not possible in general. In contrast, the interpretation of conventionally-passed arguments may be ambiguous across pointers, or in the use of unions, but is otherwise precisely described by debugging information. Annotations capturing common variadic argument encodings could be used to supplement debugging information and allow the removal of this limitation. (For example, a DWARF routine to decode C-style format strings could be used to identify how many parameters had been passed to a printf() call.)

Each wrapper function's structure is explained by the following sequence of steps.

1. **Update counter** To support the count-based predicates (of the kind seen in §2.2.9, involving fopen()), each invocation of a particular call must increment a counter. These counters are simply declared as `static` C++ local variables within the wrapper functions, and incremented on each call. Counters are used rarely; primarily, they exist as a more compact alternative to the blackboard, useful when only the occurrence of a call, not its arguments or relative ordering, is significant.

2. **Match and update call sequence (blackboard) state** If the call matches a pattern in some event correspondence containing a context predicate, then we must update the call sequence state accordingly. Call sequence recognition state is kept on a blackboard (one per component pair)—effectively a list of the previous calls that are being remembered, including return values for those that have been completed. We discuss this further in §4.2.7.

3. **Dispatch to firing rule** If the pattern state now matches some complete event pattern with context, that rule becomes the fired rule. Name bindings are formed to the relevant values stored in the pattern (by defining C++ references pointing into the blackboard state). If the pattern state does not match a complete pattern, we look for a context-insensitive matching rule, from most-specific to least-specific pattern. The code dispatched to is always a Cake stub—simple event correspondences described without using stub syntax are converted to single-expression stubs in Cake's abstract syntax. Stubs are emitted as C++ code generated straightforwardly from their abstract syntax. Each subexpression is lifted to a statement, and a name bound to its result. This means there is a strict and predictable sequential execution order within stubs. A success value is extracted from the expression's result; unsuccessful subexpressions abort their containing expression.

4. **Generate return value** Evaluation of the stub will yield some output value (perhaps void) and some success status. If no return rule was specified in the correspondence, style-specific conversions are applied to synthesise an appropriate return value from the yielded output and success status. Otherwise, another stub is generated according to the return event rule, and the return value and success status synthesised from this stub's output.

5. **Update and clear blackboard entries** The return value of the call, as seen from the caller's perspective (i.e. after synthesis) can now be written to the blackboard, to allow the complete call to be matched by context patterns. However, we only need to retain sufficient blackboard state to match the context predicates that were

actually used by the Cake programmer. These are finite by their nature. Therefore, we here clear whatever sequence recognition state is no longer required. We discuss this further in §4.2.7.

6. **Return** We return control, passing the synthesised return value (if any).

4.2.7 Notes on the blackboard

The blackboard can be thought of as a sliding window of call history. The window is only as big as required to match the context predicates specified by the Cake programmer, so it need not grow arbitrarily big as the call history grows.

Currently there is one blackboard per ordered pair of components. In a multithreaded setting, this would be a thread-local structure.

The current implementation is based on a simple run-time predicate interpreter. The blackboard is a queue of records. Each call pushes a record onto a queue, and each return from a call updates the call record with the return value (if any). Wrapper functions perform tests on the blackboard (see Fig. 4.4) by invoking the interpreter, passing as the interpreter's argument a predicate structure specific to the context predicates relevant to the wrapper. This predicate structure is a direct encoding of the context predicate expressed in the Cake code, and can be generated by the Cake compiler. The interpreter then looks for a match by walking the queue. This approach is not optimised for execution speed; applying efficient algorithms from the domains of language recognition or string matching could be a worthwhile contribution towards a high-performance implementation of Cake.

It is worth considering the growth of blackboard state. In the absence of rules which match data values across calls within a context-predicated rule, the blackboard state is finite: only finitely many context predicates are present in the Cake source file, and no more than one instance of matching may be ongoing for a particular context predicate at a time. In a multithreaded scenario, this becomes one instance per thread. The use of ellipsis ("...") in predicate rules does not worsen this, since any sequence of undistinguished calls may be recorded in constant space using a special "marker" blackboard record.

However, the encoding of data dependencies in pattern rules *does* introduce the possibility of (effectively) unbounded growth of the blackboard, although such cases always indicate buggy Cake code. Consider the rule we saw in §2.3.1, repeated in slightly simplified form in Fig. 4.5. A client which repeatedly called `mpeg2_init()`, yet never called `mpeg2_get_info()`, would cause unbounded growth, because the wrapper is trying to match calls to `mpeg2_get_info(dec)` with some particular argument value, each of which must be remembered separately. Without this data dependency, each `mpeg2_init()` call would replace the preceding one on the blackboard, so the growth would not be unbounded.

This theoretical blow-up is effectively attesting to the fact that Cake can adapt interfaces using dynamic allocation of objects: using these data-dependent predicates, each object can define a call history which is matched independently of similar call sequences happening on other objects in an interleaved fashion. This generalises to thread-level

```
// recurrence of "dec" introduces data dependency
let dec = mpeg2_init(), ...,
mpeg2_get_info(dec) ⟶/* ... some sink expression */
```

Figure 4.5: Data dependency risking unbounded blackboard growth

interleaving with the simple extension to a thread-local blackboard. However, note that each interleaved "slice" of calls is subject to the same Cake rules. The notion of *slices* identified as future work in §3.6 would address this restriction, by supporting the adaptation of interleavings where different rules should apply in each case.

A related issue is the detection of obsolete records. The obsolescence criterion for a blackboard entry is that for all context-predicated rules, for all sequence elements in that rule which might match the call denoted by the entry, an *obsoleting entry* exists subsequently in the blackboard. A later entry obsoletes an earlier entry with respect to a rule if it breaks the rule's pattern when anchored at the earlier record. For example, in a pattern A(), B(), supposing there is already an A() entry on the blackboard, any successive entry that is not B() will obsolete A() because the pattern from A() is now broken (noting that the ellipsis was not used). Later calls also obsolete earlier calls in cases where any call which might extend a pattern, as anchored at the earlier blackboard record, would also extend a *later* instance the same pattern, i.e. using an existing blackboard record appearing more recently on the blackboard than the earlier call. In this way, if a sequence is begun but not completed, and then begun over again, only the later instance can trigger a match. This restricts the interleavings of patterns that may be matched, but since patterns do not span multiple threads of execution, this is not a significant restriction.

As an example of the obsolescence rules, suppose rules exist for matching sequences:

1. A(),..., C(),
2. B(), ..., A() and
3. A(), ..., C(), ..., D()

... and the blackboard contains:

A(); B(); C(); A(); C(); D();

—this implies that rule (1) was fired after the third call and rule (2) after the fourth. However, the calls matched in rule (1) were not yet obsoleted, because a call to D() might fire rule (3) and hence match them again. By the second call to C(), however, these calls *have* been obsoleted, because any subsequent D() would match the more recent calls to A() and C() in preference to the earlier. Therefore, after the final D() call is made, the earlier A() and B() calls may be removed from the blackboard. (In fact, so may the later calls in this case, because no rules are defined that match a supersequence of the one just matched.)

4.2.8 Generated value conversions

Besides wrapper functions, the other main ingredient of the Cake compiler's output is value conversion functions. We discuss various aspects of these functions in this subsection.

Generation

Value conversions as template specializations The Cake runtime's header files describe a value conversion as a C++ template class overloading the function application operator `operator()`, defining a class of "function object" in C++ terminology.[1] All value conversions specialize this template. These templates are parameterised on the C++ argument types being converted, and on an additional integer rule identifier allowing the same C++ types to be related by multiple different conversion functions in different contexts (e.g. to support artificial data types, §2.3.5). These parameters are specified whenever Cake generates code to invoke a value conversion (perhaps in a wrapper function, or perhaps in another value conversion).

Prelude for default conversions A static "prelude" set of definitions provides various default behaviours. This includes default value conversions between differently-sized primitives values, and treatment of pointers (which invokes run-time logic to discover the precise class of pointed-to objects, enabling Cake's dynamic semantics described in §2.4.3). The prelude also includes C++ template descriptions of Cake's "default style", in the form of operations e.g. for extracting the success flag of a function call from its return value, for encoding booleans, sets and lists, and other matters of style. These definitions can be supplemented support alternative styles; Chapter 6 describes a way of doing this within an extended version of the Cake language.

Dynamic dispatch

One complication of using C++ templates to describe value conversions is that our dynamic matching semantics requires dispatching to a particular conversion at run time, rather than at compile time when templates are elaborated. It follows that we must ensure two things. Firstly, any value conversion template which might be required at run time must be instantiated at compile time when the generated code is compiled. Secondly, we must have a mechanism for dispatching to a particular template instance at run time. To satisfy both of these requirements, we generate a table mapping from pairs of data types ("from" and "to" conversion argument types) to the template function instances which perform the value conversion. By including in the table all conversions defined between all data types related in the Cake file, we ensure that all template instantiations that might be required at run time are performed. Run-time dispatch to particular value correspondences is then performed dynamically by table look-ups.

[1]See Stroustrup [1997], §18.4.

Identifying data types at run time

In generated code, we identify data types by the C++ type names generated by dwarfhpp. These directly reflect the component names given in the Cake source file, which then prefixes the components' internal namespacing structure. For example, if some component named renderer in the Cake source code contained a C++ data type widgets :: surface, the dwarfhpp-generated C++ name might be renderer ::widgets :: surface. However, the runtime discovers a Dwarf typename relative to the *run-time* component structure of the program. So if the renderer is linked statically into an executable `browser`, and its Dwarf information mentions that it was defined in a file `pane.o`, the runtime might identify it as `browser -> pane.o -> widgets -> surface`. To allow the runtime to look up conversion routines from these run-time identities, our table of conversion routines must be keyed using a naming scheme which unifies these two views. Erasing the unshared prefix (i.e. preceding widgets) would be incorrect, because, perhaps when applying Cake to interface evolution, we might be linking some client and library that use like-named data types which are incompatibly defined.

To solve this problem, we choose a subset of identifying information that is invariant under linking and deployment, and use this to prefix data type identities in both compiler and runtime. Specifically, keys in the compiler-generated table of value conversions are data type names prefixed with the full compile-time path of the *source* file containing the data type definition (usually a header file, for C source code) *and* the identification string of the compiler which generated the corresponding object code. Both of these strings are invariant under linking and deployment, and available to both the Cake compiler and the runtime (in Dwarf information). This scheme is not guaranteed to be free of collisions (e.g. if Cake is composing code compiled against mismatched but like-named headers), nor of false duplications (e.g. if multiple compilers were used to generate a single component). However, problematic collisions are unlikely. Compositions which share header files (up to their pathname) but which are nevertheless mismatched are likely to derive from significantly different compilation environments, meaning that the compiler itself is likely to be different (at least in a minor version number). Inclusion of the compiler identity string therefore avoids a collision. Since keying a table on two long strings is needlessly inefficient, our implementation actually uses a 64-bit hash of these strings. Again, collisions of these 64-bit values are a theoretical possibility but highly unlikely.

A final complication with handling data types in the runtime is that a single data type definition (say struct A, defined in some C header) will be repeated in many locations in Dwarf information (perhaps once for each source file that included the defining header). The runtime therefore maintains a name-canonicalisation table for data types, where these are grouped under an equivalence relation defined by prefix-stripped name equality *and* representation-compatibility (defined in §4.4.4). Canonicalisation is applied before doing conversion table look-ups.

4.2.9 Cake and dynamic linking

We have so far seen use of Cake as a static linking tool only. Since libraries are commonly deployed on most modern operating systems as shared objects, linked dynamically, we would like to be able to apply Cake to these. Interposing on dynamically linked symbol references requires a slightly different mechanism, but the generated code is otherwise identical. Wrapper functions are compiled into a shared object and loaded with the dynamic linker's `LD_PRELOAD`. Original versions of any wrapped functions, analogous to the `__real_`-prefixed symbols in static linking, are sourced using the dynamic loader's `dlsym()` API call.

4.2.10 Interaction with compiler-optimised code

Object code is a partially-specialised medium: the compiler has had opportunity to pre-compose certain fragments of source code. The two main examples of this are cross-module inlining and macro substitution. In both cases, code is embedded in locations remote from its definition, possibly in many such locations.

We distinguish *cross-module* embedding from intra-module embedding because the latter is both more common and unproblematic. For example, if a component uses an inline function internally, this inlining does not affect Cake usage. By contrast, if inlining or macro substitution occur across logical component boundaries—for example, a library defining inline functions in its header files—this can affect Cake usage. In the example, client binaries will embed inlined library code directly. This is a problem for a Cake programmer wanting to interpose on the inlined call, since as far as Cake is concerned, no such call occurs.

In general, we can say that inlining and macro expansion appear as binary interface alterations—that is, changes relative to the interface that would have emerged if the calls were left in functionally abstracted form in the object code. We consider two strategies for handling these alterations: working with them, or undoing them.

Working with alterations Since only small functions are typically provided as macros or inlines, only small alterations to the interface typically occur. For example, inlined getter and setter functions mean that the resulting interface consists of field accesses rather than function calls. Since Cake can adapt this class of mismatch, this is not strictly speaking a problem—particularly when inlining can be considered a predictable behaviour of the compiler (such as in C++, where member functions defined inside a class definition are reliably inlined in all implementations). Other cases of inlining may be less predictable, potentially causing a maintenance problem for the Cake programmer unless inlining is turned off.

Undoing alterations Undoing of optimisations is a necessary task for debuggers, when debugging optimised code. Accordingly, debugging information may be used to effect an undoing of optimisations, not only by debuggers but also by Cake. Modern debugging information, such as DWARF [Free Standards Group, 2005] records places where functions

```
widgets_A ⟷widgets_B
{ // an event correspondence
  find_widget(descr)  ⟶  get_matching_widget(descr);
  values Widget ⟷ Widget
  { /* ... */ };
  values Window ⟷Window
  { /* ... */ };
}
```

Figure 4.6: Rules sensitive to static versus dynamic binding (repeated)

have been inlined, similar to "scope descriptors" in the Self compiler described by Hölzle et al. [1992], and including descriptions of the parameters to the function from the local state within the enclosing function's code. Therefore, a tool such as Cake could rewrite these instruction sequences to instead perform an out-of-line call, thereby recovering the black-box view required by Cake. We have not implemented this rewriting, but this technique would be necessary in a production implementation of Cake.

Of course, it is preferable to write Cake rules against a stable interface, meaning one not dependent on some opaquely-chosen set of compiler optimisations. It is therefore best to disable inlining optimisations until link time, after Cake has done its work. Link-time optimisation is supported by many toolchains, including recent versions of the GNU toolchain. Since link-time optimisation allows whole-program analyses based on a complete call-graph, it yields better results than inlining during separate compilation. Specially-crafted link-time optimisations for Cake-generated code would make for interesting future work.

4.3 Implementing dynamic binding

Objects in native code are generally not self-describing at run time. Moreover, as discussed in §2.4.3, the debugging information available to describe them, much like static type information, is inherently imprecise. As a summary of the problem, we repeat the example from the previous chapter as Fig. 4.6.

Recall from §2.4.3 that we define *well-behavedness* criteria for input components. These exist to limit the number of difficult cases that a Cake runtime implementation has to address. Specifically, they limit the responsibilities of the runtime in three ways. Firstly, they bound how much memory the runtime needs to explore as pointers are passed across an interface—noting that a pointer might point to a single object or to an array thereof. Secondly, they provide constraints on what the possible *interpretations* of that memory will be, i.e. what data types it actually encodes (given that static type information is imprecise). Thirdly, they allow the runtime to decide which objects may be directly *shared* between mismatched components (according to an analysis we describe in §4.4.4).

The imprecision of static metadata on pointers is compounded by arrays. Debugging information does not reliably record whether a pointer points to an array or to a single value only (owing largely to the C idiom of handling arrays implicitly using pointers). Moreover, we cannot statically discover the size of many arrays. Cake's semantics (§2.4.1) nevertheless require that a receiving component may access the entire array. In summary, the Cake runtime must be able to decide two questions.

- Given a pointer to an object, what *byte-scale adjustments* might a component reasonably make, to reveal a pointer to a containing or inheriting object? (An adjustment of zero is valid, in the case where the inner object is at offset zero within the outer.)
- Given a pointer to an object, what *block-scale adjustments* might a component make, to navigate among objects in the same array? We call these block-scale adjustments because they are at least the size of a single object, whereas byte-scale adjustments never take the pointer outside the original containing object.

The two are not independent: to apply pointer arithmetic, a component must know the element size, so we assume that a component may *not* do *both* byte- and block-scale adjustments (unless the Cake programmer provides a precise type by annotation). To do both would be to navigate an array through a pointer whose static type did not reflect the true element size. In practice this occurs only when a function receives an array through a generic (untyped, `void`) pointer, and uses some prior knowledge to strengthen that pointer before using it. That prior knowledge is typically fixed statically for the receiving function, so can be described in a Cake annotation.

(Consider that these uses of `void` in C code occur because a caller often passes a pointer *through* some more generic code which treats that pointer opaquely, such as a callback registration function. The `void` typing therefore appears in header files, which must be generic with respect to all callers of the API. This is sometimes called an "existential use of `void`" [Neamtiu et al. 2006]. Since Cake allows caller and callee sides of an interface to be annotated independently, it does not suffer this problem: the eventual recipient will usually know a specific type for the pointer, even if intervening code is generic.)

4.3.1 Admissible reinterpretations

We call byte-scale adjustments *reinterpretations*. Cake's well-behavedness rules define criteria for what we call *admissible reinterpretations*. These criteria are designed to separate out common-case pointer adjustments, which can be treated automatically, from uncommon cases requiring annotation by the Cake programmer.

For a pointer whose target is statically typed with type τ, admissible reinterpretations are as follows.

- If τ is a DWARF base type, no reinterpretations are admissible.
- If τ is structured, reinterpretations to any *zero-offset containing type* are admissible. A zero-offset containing type is one which contains a subobject of type τ at offset

```
struct Abstract
{
  int val;
  struct Contained
  {
    float x;
    float y;
  } contained;
} abs;

struct LessAbstract /* logically a subtype of Abstract */
{
  Abstract base;
  // ... more fields
} lessAbstract;

void some_code(void)
{
  Abstract *abs_alias1 = (Abstract*) &lessAbstract; /* admissible! by zero-offset containment */
  Abstract *abs_alias2 = (Abstract*) &abs.val; /* not admissible! can't byte-scale adjust ptr to int */
  Abstract *abs_alias3 = (Abstract*) ( // arithmetic to recover a pointer to the enclosing object ...
                          (char*)&abs.contained // ... from &abs.contained
                          - offsetof(Abstract, contained)
                        ); /* also not admissible! but see text */
}
```

Figure 4.7: Abstraction-violating pointer adjustments

zero. We allow this to support the idiom often found in C-language object systems[2] which simulate inheritance by zero-offset containment.

- If τ is structured, reinterpretations to any DWARF-recorded inheriting type are admissible. This allows for downcasts using DWARF's special inheritance tag (which supports single or multiple inheritance).

Fig. 4.7 shows two examples of *uncommon* cases in C code, not admissible by these rules. The first is not admissible because pointers to DWARF base types, such as `int`, may not be reinterpreted. The second is not admissible because only pointers to contained subobjects at offset zero may be reinterpreted to the containing object type.

The first adjustment is fragile, because it assumes `val` is the first field in `abs`. However, we allow it because this kind of quasi-subclassing layout convention is commonplace (and because it happens not to render our problem intractable). The second adjustment is less useful because subtyping relationships are seldom defined on base types (and there is no other benefit to having a pointer to an `Abstract` be statically typed as an `int*`), so we disallow it.

The third adjustment is an attempt at more robustly coding the first. It would present a problem for Cake if the inner expression (of type `char*`) were passed through a Cake-

[2]A popular example is GObject [Krause 2007].

interposed function, or stored in a Cake-explored object, before being interpreted as an `Abstract*`, in that Cake would see a `char*` which was actually pointing at an `Abstract`. Since there is no relationship between the char and Abstract data types, this case is not admissible.

These uncommon cases have been described as "abstraction violating" in prior work [Neamtiu et al. 2006], which also shows them to be rare in real code. Ruling them out bounds how much memory the Cake runtime must dynamically explore, and also turns out to make the problem of recovering precise object descriptions more tractable. The Cake programmer can almost always use annotations to turn abstraction-violating cases into admissible ones. For example, the second adjustment above could be rendered admissible by redefining `Abstract`, within an exists block in a Cake source file, as inheriting the `contained` structure type rather than containing it. We benefit from annotating in a DWARF-based language, because this supports the concept of inheritance (thanks to the DWARF feature DW_TAG_inheritance [Free Standards Group, 2005]) even where C does not.

4.3.2 Discovering precise object descriptions

As described in §2.4.3, Cake's semantics rely on discovering precise descriptions (or "type information") about objects at run time. The well-behavedness criteria have provided useful constraints that will help us to do this. Here we briefly describes our approach.

Exploiting address-space layout To discover the *most precise* DWARF type for a given pointer, we use knowledge of address space layout to deduce whether the object is in heap, stack or static storage. For the latter two cases, a precise type is found in debugging information for the allocating stack frame or static variable definition. The same information also reveals whether the object is part of an array. Static- and stack-allocated storage is therefore easy to recover precise descriptions for; the heap-allocated case is harder. (We assume no use of the alloca() function provided in some C libraries, since this can introduce undescribed objects into the stack. However, such calls can be replaced by a combination of malloc() and *tying* as described in §4.4.2. We also exploit the assumption from §4.2.6 that we have no variadic functions to deal with.)

Debugging information and annotation We require that debugging information, or equivalent annotation, be available not only during Cake compilation, but also *at run time*. In practice this means that any annotations added to DWARF information by the Cake programmer have been stored on disk and can be found by the runtime. We could do so by generating updated DWARF information in output binaries. At present we take a simpler approach of storing the annotations found in the Cake source code textually as static string data in generated code, with a special name (prefixed `__cake_`). This data can then be found by the runtime, which parses it and applies the relevant updates to its in-memory DWARF database.

Heap difficulties Heap implementations differ in the metadata and bookkeeping information they maintain. However, most heap implementations are built on the host C library's implementation of malloc() and free(). In any case, to support components written in C, we must support discovering precise descriptions of object on this heap. Doing so is tricky because implementations do not store (nor indeed receive) any metadata about allocated objects.

Instrumenting malloc() Almost all implementations of the C library allocator support run-time instrumentation of the allocator, sometimes called "malloc() hooks". Any implementation of Cake must exploit these hooks in order to collect the necessary metadata. Any allocators layered over the C library, such as arena allocators [Hanson 1990], must also be instrumented to collect the same. The collected metadata must include the original *call site* which performed the allocation.

Allocation site assumption As with conventional (static) points-to analyses, we assume that the source code location where an allocation is "requested" by user code, in some sense, is sufficient to determine its class. Therefore, the collected metadata must include the *call site*, or more generally *call chain* of the malloc() invocation. These are respectively the program counter value at the point of the call and the sequence of such program counter values saved earlier in the call stack. The call chain is necessary in cases where a chain of malloc()-like functions ("malloc() wrappers") are defined, since these obscure the "requesting" call-site identifying the specific class of object being instantiated.

Storing heap metadata This collected metadata is stored in an appropriate associative data structure—a wide B-tree, space-optimised hash table or other bucket-indexed list structures provide sensible trade-offs. Storage and consumption of heap metadata represents one of the significant performance overheads of Cake. Closer collaboration with the host allocator could likely reduce this cost significantly—for example, storing the allocation site in compact form within the existing allocation bookkeeping information, and providing an efficiently indexed means of traversing *allocated* chunks of memory, analogously with how the allocator necessarily indexes *free* chunks of memory.

Mapping to Dwarf Our assumptions entail that the allocation call-chains gathered by our instrumentation can be mapped *onto* DWARF types. However, discovering this mapping is not straightforward. Most C code uses malloc(sizeof (T)) to allocate an object of class T. However, sizeof (T) is reduced to an integer by the compiler. To recover the allocated object's class requires either explicit annotation (a user-provided mapping from malloc()-sites to DWARF types), or analysis. Source-level analysis is straightforward: we use the debugging information to map the call site back to a source location, and extract T. However, since Cake is designed to avoid assuming availability of source, binary analysis is preferable. This can again exploit debugging information. We may follow data flow from the return value of malloc() into a location for which a DWARF type is available (typically a local pointer on the stack, but perhaps also a statically-allocated or indirectly-reached location).

Limitations This binary analysis is precise only when the static type of the written-to location is itself precise. Being close to the malloc() call, this is reasonably likely. Deviations can be flagged up in the vast majority of cases by a simple comparison between the size of the written-to value's type, as calculated from DWARF information, and the expected size, i.e. the malloc() argument or any factor thereof. However, this remains somewhat unsatisfactory—it only reaffirms the inescapable fact that we are unable to recover precise descriptions of heap objects without relying on either availability of source code, explicit annotations, or tolerance of imprecision. This inability is not unique to Cake: unsurprisingly, debuggers share precisely this limitation. The ability to automatically downcast a pointer to its "most precise type" would be a valuable feature within debuggers, so we regard this omission as a limitation of compiler and debugging infrastructure. It could easily be fixed in the compiler: perhaps by outputting a table mapping allocation sites to DWARF types, and emitting warnings where this could not be inferred from the source.

Implementations We have avoided implementing the binary analysis approach so far, in favour of a simpler option described in Appendix E. This avoids both analysis and annotation, at the expense of allowing imprecision. Prior work [Mock et al. 2002] has shown that a dynamic points-to analysis can be implemented efficiently and precisely using source-level instrumentation, although clearly, this approach assumes availability of source.

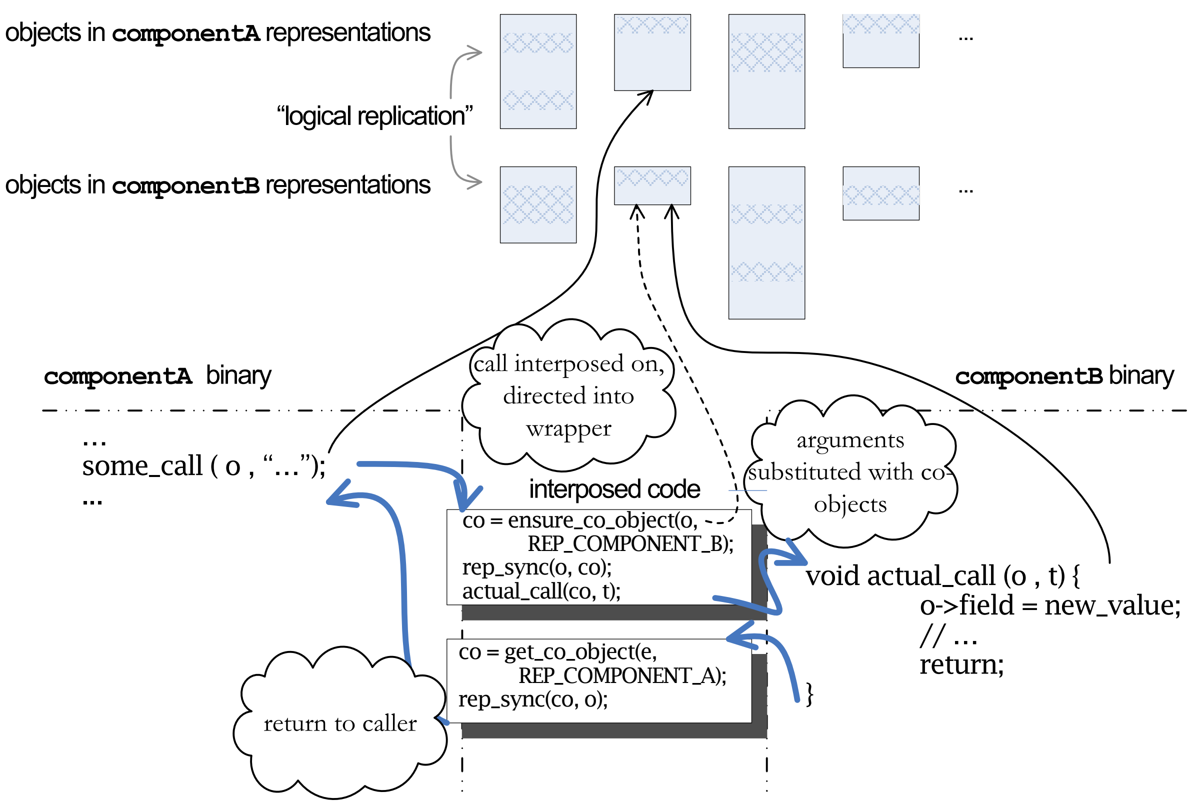
4.4 Adapting objects

We consider objects to be structured values with *two* key additional properties: identity and lifetime.

4.4.1 Object identity

Cake understands objects' addresses in memory as their identities. At run time, it maintains a table called the *co-object relation* which maps object identities to the identities of corresponding objects in use by other components—we call these *co-objects*. As pointers pass across an interface, Cake substitutes pointers to appropriate co-objects. Cake must also allocate co-objects if they do not exist—usually, for a given tuple in the co-object relation, exactly *one* object was allocated by user code; the others were allocated by Cake when a pointer to the first object, or some subsequent co-object, was passed. Fig. 4.8 illustrates this object exchange process.

Associations (§2.3.4) are implemented by mapping each object to a Cake-generated *umbrella object* which contains pointers to other objects in the association. The umbrella object is the co-object of *all* participating objects other components. The co-object relation is therefore asymmetric in these cases, as Fig. 4.9 illustrates. The "access associated" operator, "...", is implemented as field access on these umbrella objects.


objects in componentA representations
"logical replication"
objects in componentB representations
...
componentA binary
componentB binary
call interposed on, directed into wrapper
arguments substituted with co-objects
return to caller
interposed code

```
...
some_call ( o , "...");
...
```

```
co = ensure_co_object(o,
        REP_COMPONENT_B);
rep_sync(o, co);
actual_call(co, t);
```

```
co = get_co_object(e,
        REP_COMPONENT_A);
rep_sync(co, o);
```

```
void actual_call (o , t) {
        o->field = new_value;
        // ...
        return;
}
```

Figure 4.8: Object exchange in a wrapper, using the co-object relation

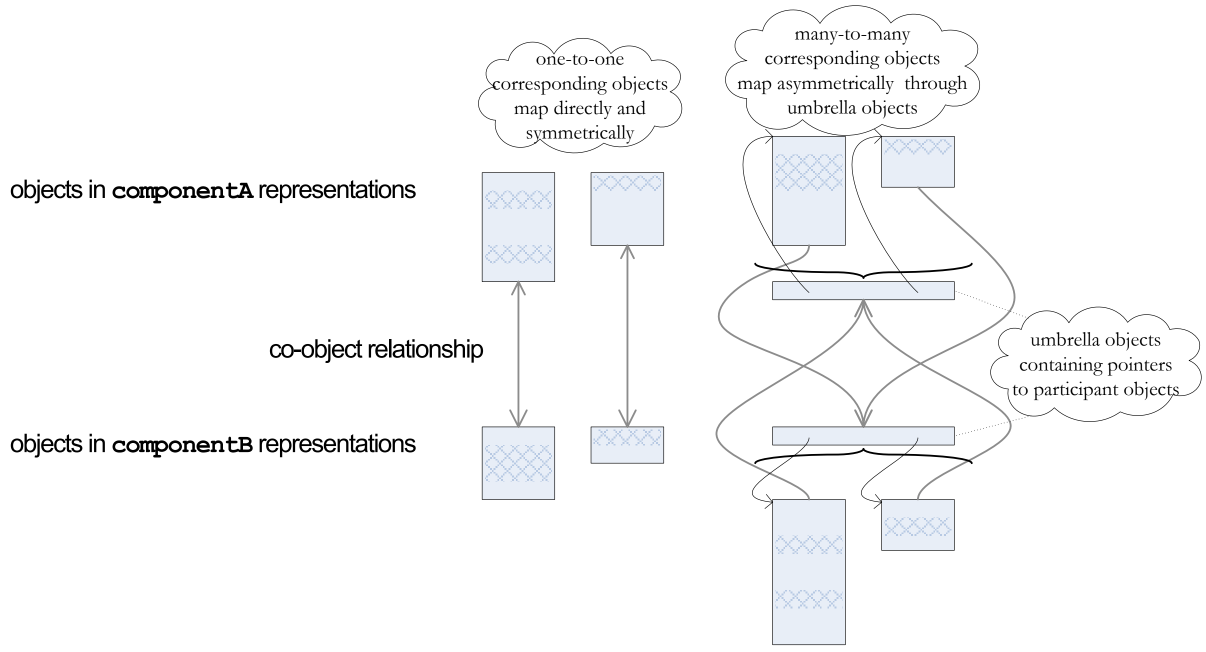

one-to-one corresponding objects map directly and symmetrically
many-to-many corresponding objects map asymmetrically through umbrella objects
objects in componentA representations
co-object relationship
objects in componentB representations
umbrella objects containing pointers to participant objects

Figure 4.9: Umbrella objects in the co-object relation

At present this unparameterised co-object relationship constrains an object to be participating in at most one association at a time. While this has been sufficient for us so far, it seems inevitable that this will prove restrictive in some cases. We could allow each object participate in one association (dynamically) per many-to-many correspondence rule (statically in Cake source code), by giving each rule a run-time tag and a compile-time identifier. The co-object relationship would then be parameterised on these tags. Since some instances of the access-associated operator will be ambiguous, if multiple rules are defined, a compile-time identifier can be used to select the rule, given a special syntax object..*ruleIdent*..associatedObjectName.

4.4.2 Object lifetime

Object identity and lifetime are interdependent, in that lifetime governs the re-use of identities. Therefore, any analysis making use of object identity must be sensitive to object lifetime. In practice this means that when our runtime tracks objects by storing pointers to them, we must also trap object deallocations, so that we can invalidate any data associated with the deallocated object.

When applying value correspondences to produce transformed versions of objects, Cake must allocate memory. We *tie* the lifetime of these allocations to the user-managed objects that caused them. The Cake stub language also supports explicit tying, as a convenient storage management mechanism—see §2.3.4.

Our interposition on deallocation must support not only heap-based deallocation (handled by the instrumentation of free() and similar functions, as previously described in §4.3.2) but also stack-based deallocation. To interpose on deallocation of stack-allocated objects, Cake must interpose on cleanup of the allocating stack frame. This is implemented by replacing the on-stack return address for the allocating frame with the address of a handler. This handler uses the stack pointer to identify which frame is returning, deallocates any tied objects, and jumps back to the intended return address. Our application of this technique is due to Amitabha Roy, who also provided an implementation.

4.4.3 Dealing with function pointers

In most respects, function pointers are not treated specially by Cake—they are simply pointers to objects, which are allowed to flow between components provided that suitable event correspondences are defined (§2.3.9). This means that functions are objects that may have co-objects. However, functions are allocated statically, unlike other object identities of concern to Cake, which we assumed could be discovered dynamically through function interposition (§4.2.1). When passing a pointer to a function across a mismatched interface for the first time, the Cake runtime cannot react by *instantiating* some new co-object. Instead, it must simply *substitute* a pointer to the relevant corresponding function.

As a consequence, the Cake compiler must *initialize* the co-object relationship with mappings between functions and their co-objects. In most cases, a function's co-object is a wrapper function. Specifically, for a function f in a component A, its co-object in component B is the unique wrapper function, if it exists, which is linked with call-sites in component B and dispatches to function f (and no other function). Naturally, there will only be any such function if event correspondences in component B dispatch to component A. If there is no such unique function, f has no co-object in B and it is an error to pass a pointer to f from A into B.

(Recall that this is only relevant for functions whose addresses are passed to other components. Non-address-taken functions, or functions for which no event correspondences are defined, will trivially not appear in the co-object relation.)

Since in general, the wrapper function might have a differing signature from f, and indeed, changed semantics, it remains the programmer's responsibility not to create compositions which pass the function into contexts where the wrapper will be invoked incorrectly. This amounts to understanding the semantics required by the *indirect call sites* in the receiving component. The constraint described in §2.3.9, that an unnamed event correspondence must be established from the receiving component back to the address-taken function in the originating component, exists partly to require the programmer to acknowledge this understanding.

The co-object relationships between functions and their wrappers is emitted as a static data structure in the C++ code generated by the compiler.

4.4.4 Sharing objects

As described so far, each component appears to have its own heap, completely separate from other components'. In fact, Cake allows sharing of objects between components, subject to the invariant that each component can only reach objects whose representation it understands. (We define this more precisely below.) The effect is a *partially split heap*—some objects are shared, and others are replicated (perhaps in alternative representations).

Enforcing this invariant is nontrivial because of the transitivity of reachability. In other words, given an object, it is not easy to tell whether it is safely shareable between two components. Although it may be laid out in a manner understood by both components, perhaps the same is not true of some objects reachable from it. Without analysing the components to discover their memory access patterns, we must assume it may perform any "well-behaved" access, which might reach an incompatibly laid-out object. We therefore must prevent these objects from being shared. We do so using a conservative approximation of this reachability relation. This is a static whole-program (or rather "whole composition") analysis, currently performed in the Cake compiler before generating code.

We start by partitioning the (infinite) set of run-time objects into equivalence classes based on their "most precise DWARF type" (§2.4.3). Our question then becomes whether Cake can allow two components to share an object of a given DWARF type. Firstly, consider the DWARF types of all objects which are *related* between each pair of interfaces. We call this the *master type relation* for that pair, and it is enumerated by the set of value correspondences established between the two components (including those made by name-matching). Next, we define a binary relation *representation compatibility* on DWARF types, recursively as follows.

- For a structured type: if the two structures define identical sets of field names at identical offsets, and for each like-named field the field's type is representation-compatible, then the structures are representation-compatible.
- For a pointer type: all pointers are representation-compatible. We account for reachability in a separate step (below).
- For a primitive type, the types are representation-compatible if and only if size and encoding match exactly.[3]

To incorporate sharing into Cake-generated code, we require a conservative approximation of safe sharing that can be computed statically. We begin with defining the "possibly shareable" set. This is those pairs in the master type relation that are representation-compatible (possibly after field renamings, originating from Cake rules, have been applied). Clearly, representation-incompatible objects should not be shared. However, not all representation-compatible objects are shareable, because they might contain pointers to objects which are not shareable (and we cannot be sure, without analysis, that the sharing component will not follow these pointers).

We generate the "definitely shareable" from the "possibly shareable" set by removing, until a fixed point, pairs where, given a pointer to some shared object, both components could reach some piece of memory about which their expectations are not representation-compatible.

For this, we require a conservative static approximation of reachability. For this, we re-use our "admissible reinterpretations" definition from §2.4.3, again capturing the notion of a "well-behaved" component. Define the *type reachability graph* as the connected digraph (V, E) where E includes (v_1, v_2) iff a pointer to some object of data-type v_1 can yield a pointer to data-type v_2 by *either* member selection *or* an admissible reinterpretation. We label each edge to identify which member was selected or what interpretation was applied.

To calculate the definitely-shareable set, we proceed iteratively on a working set initialised equal to the possibly-shareable set. We then remove any (α, β) if there exist some *non-shareable* α' and β'—i.e. a pair (α', β') *not* in the working set—reachable respectively from α and β by analogous paths in each component's type reachability graph. Here "analogous" means selecting the same member or performing the same reinterpretation, *again* allowing for field renamings which might have been used to recover representation compatibility (recalling that α and β are, by definition, representation-compatible in the context of a set of such renamings). The idea here is that we are exploring the same object from the point-of-view of both components, and stop once we discover a non-shareable pair of objects: from our invariant, we should not be able reach this object, so it was wrong to count the initial object as shareable. When selecting paths in the reachability graph, we can limit ourselves to the finite number of acyclic paths, since a data type whose shareability depends on only its own shareability (or more generally, on a cycle of mutual dependencies of this form) is trivially shareable.

[3] A subtlety here is enumerations, bitfields and other encodings layered onto primitive types. We rely on programmer annotation to interpret these, for example using the `names` construct (§2.2.11). If non-identical interpretations are applied in this way, rep-compatibility is lost.

Considering each component's reachability graph separately is sufficient under the assumption that for any memory access performed by a component, debugging information records a DWARF type describing that memory's layout as understood by the component. This is a reasonable assumption since it is also a necessary condition for a debugger to be able to read that memory.

Shareability of pointer, array and function data types is defined in terms of the data types from which they are constructed. An array type is shareable with its correspondent if their ultimate element types (i.e. the element type after collapsing arrays of arrays) are shareable. A pointer type is shareable with its correspondent if their target types are shareable. A function type is shareable if every data type in its signature is shareable. (This criterion can also be applied to omit wrapper generation, if all functions that would otherwise need to be wrapped have shareable signatures. We discuss this special case in §5.2.)

In practice, the effect of this shareability analysis is that utility data structures passed between components can be shared most of the time, because they are likely both to be representation-compatible locally and to reach few other objects. Meanwhile, completely unanticipated compositions of client and library rarely recover any sharing, because they rarely include representation-compatible data structures to begin with. Edge cases emerge when, say, linking a client and library whose interfaces are similar but, perhaps from interface evolution or compiler differences, are slightly incompatible at the binary level. In these cases, reachability has a big influence. If mismatched data types cannot be reached from most objects, then most objects may be shared and only a few incur the overheads of copying. The situation is reversed if representation-incompatibility occurs in pervasively-reachable data structures: few objects are shareable, because most of them reach incompatible objects.

4.4.5 Limitations of object graph exploration

When selecting correspondence for objects found during object graph exploration, the Cake runtime has relatively little contextual information available about those objects. This contrasts with objects passed directly by function call—i.e. up to one level of indirection from the actual arguments passed on stack—where treatment of objects may be specified in detail through several different means. For example, the programmer might specify using the `as` keyword (§2.3.4), or might either set up or traverse an association using the ellipsis operator, or might otherwise treat the object specially using arbitrary stub code. None of these are available during deeper object graph explorations—the runtime must rely entirely on the set of defined value correspondences. We say it is "context-impoverished".

In turn, the correspondences expressible in Cake with respect to objects discovered in this way are strictly fewer than for objects passed as parameters or return values. In particular, since creation of associations is always done explicitly in stub code, many-to-many correspondences may not currently be formed between objects reached only by object graph traversal. A more advanced runtime could potentially automatically partition the explored object graph and select appropriate many-to-many correspondences based on what classes of object were found in each component's view of the heap. Algorithms for doing this predictably, unambiguously and safely are a topic of future work. While Cake must handle graph-structured data, for tree-structured data this problem is relatively well understood from tree transformation languages, familiar both in the mainstream (e.g. through XML transformation languages XQuery [Boag et al. 2002] and XSLT [Tidwell 2008]) and in recent research literature [Foster et al. 2005]. Potentially, these languages could be generalised to support the graphs traversed by Cake, or perhaps annotations could enable graphs to be treated as trees in most cases.

4.4.6 Optimisations to object sharing

One of the key run-time costs of Cake is maintaining multiple copies of data, to satisfy the different object layouts in use by different components. This copying is no different from the marshalling or replication which is often done in hand-written glue code. In our case, copying is performed automatically by the Cake runtime. However, in the interests of performance we would like to eliminate avoidable copying by sharing objects in memory.

By their nature, these optimisations are more likely to be useful in applications of Cake to interface evolution, where many data structure definitions are similar or identical and only a few differ significantly, than in unanticipated composition tasks where all data structures are likely to be substantially different between the two interfaces.

Opaque and ignored pointers

One approach to minimising copying is to obtain more precise information about the interpretations *each component* makes of its objects. This might reveal cases where although the object layouts are concretely different, they are compatible up to the interpretations that a given component makes of an object. For example, if a component always ignores some field in an object, or treats a pointer opaquely, then it no longer matters whether the relevant field contents or pointed-to object are representation-compatible in other components. Our current Cake grammar allows the user to supply `opaque` and `ignored` annotations, but we leave implementation and evaluation of this approach to future work.

Reducing the volume of updates propagated

The "partially split heap" is effectively a replication-based approach, in which sharing is an optimisation. This means that our runtime must manage the propagation of updates between replicas, in the case where objects are not shared. Specifically, we currently use a policy of propagating updates between *all* replicas whenever control passes between components; this is correct in the single-threaded case, although slow (because of potentially high update volumes at each interface crossing). To reduce the update volume, points-to analysis could produce a tighter bound on which objects' updates may be needed during a given call.

(The update propagation step occurs in wrappers during the "dispatch to firing rule" step, as shown in Fig. 4.4.)

4.4.7 Object sharing between multiple threads

The "partially split heap" approach presents some familiar difficulties in the combination of replication with concurrent access to objects.

Our current update propagation policy is not compatible with all multi-threaded programs. Specifically, programs that may share "logical objects" (i.e. objects related across a mismatched pair of interfaces composed by Cake), and whose liveness relies on updates made in one component becoming visible in another component *before* control flow occurs between these components, will deadlock. This follows from our propagation policy: since updates are only propagated when control passes across the interface, updates in this case will never be propagated.

A background thread could ensure liveness, by propagating updates periodically. However, since this might gather an inconsistent snapshot of partially-updated objects, ensuring safety is a problem. This problem is analogous to that found in dynamic software update, when waiting for a safe instant to apply a patch which updates data structures to new definitions. The existing annotation-based approaches of "quiescent update points" [Neamtiu et al. 2006] and programmer-annotated "propagation points" [Neamtiu and Hicks 2009] are likely to yield a solution. However, it is not clear whether the latencies inherent in these approaches would be acceptable for applications of Cake, which generally have stronger timeliness requirements than dynamic software update.

A final problem in the multithreaded case is conflicting updates to separate replicas of logically shared state. In the absence of an abort-and-rollback operation, solving this will likely require that shared-writable objects are managed using an alternative replication-free approach. Specifically, we envisage using memory protection techniques to trap updates to these objects. Shared-writable objects' memory would be allocated from a special range of the virtual address space which has no physical backing, but generates traps. Then we handle traps by reading and writing a unique Cake-managed representation of the shared state. Higher read performance could be gained by maintaining read-only replicas in each component's native representation, while continuing to trap writes.

4.5 Status of the implementation

The compiler is a C++ program of (currently) around 20,000 lines, not counting the grammar definition or generated parser code. The compiler is the least fully implemented part of Cake; the basic structure as shown in Fig. 4.3, is implemented, and code for many kinds of correspondences is supported. Code generation for a few advanced Cake language features continues to be a work in progress. However, every major technique discussed in this dissertation has at least been implemented to proof-of-concept level in hand-coded mock-up form, where not already generated by the compiler.

However, it does mean that when we evaluate Cake on real code, the Cake code we write is not always compilable and executable, but is partly a by-hand translation. In particular, of the three Cake-language case studies presented in Chapter 5, only the first is currently fully supported by the compiler.

In turn, the compiler is built on a DWARF abstraction layer (written primarily for the construction of the Cake compiler, but separately re-usable) of around 13,000 lines of C++, not counting the third-party libdwarf[4]) on which it builds. The Cake runtime is around 2,600 lines of C and C++, and a few of assembly language.

4.6 Implementing recursive application of Cake

We described previously (§3.8) how the Cake language design accommodates the possibility of multiple applications of Cake within the construction of a single ultimate output object or executable. We briefly discuss a possible implementation of this.

Non-interfering cases First, we must distinguish *interfering* from *non-interfering* pairs of Cake applications. Two applications of Cake are interfering if the sets of symbol names they interpose on have a nonempty intersection. Non-interfering compositions are straightforward to support in our current implementation, as they affect unrelated interfaces. At present, the only barrier to supporting this in our implementation is that our encoding of the *initial co-object relation* (§4.4.3) is assumed to be unique by the Cake runtime, so at most one application of Cake can define such a relation within any one executable.

Procedural fan-in is non-interfering It is worth noting that wrappers attach to call sites, not their callees. Therefore, any code that is linked in to the program using Cake still retains its original set of procedural entry points, which are available for subsequent linking with or without Cake. This allows, say, two different client codebases to be linked against the same library in different ways within a single application (e.g. one mismatched case linked by Cake, another well-matched case linked directly).

Wrapper-chaining for interfering cases Interference occurs when different Cake applications are defining conflicting ways of handling calls to the same interface. Fortunately, interfering cases can nevertheless be resolved successfully, using *wrapper chaining*. If a wrapper receives calls which do not match any of the event correspondences defined in the originating Cake code, it tests for the definition of a "weak" (i.e. optionally defined) symbol prefixed by `__real_` and if so, passes the call to it (as shown earlier, towards the bottom of Fig. 4.4). This pass-through is useful when the earlier-linked Cake rules match only a subset of the possible argument values (by pattern-matching) or in a subset of contexts (as matched by context predicates). One example is `fopen()` in Fig. 2.8: other unrelated `fopen()` calls will be passed through. To implement wrapper chaining, the Cake compiler needs to detect cases where a `__real_`-prefixed symbol is found in the place of the expected name; it can then wrap this prefixed symbol name, forming a chain. Although we have not yet implemented this compiler extension, doing so would be straightforward. Multiple chainings can be supported by suitable symbol renamings. The effect is of defining a priority ordering of Cake applications, where earlier-linked applications of Cake get the first opportunity to handle a call.

[4]David Anderson's libdwarf is available at http://reality.sgiweb.org/davea/dwarf.html as of 2010/12/7.

Dynamic wrapper chaining Indirect call sites' targets are governed by dynamic data flow, as described in §4.4.3. Since dynamic flow of function pointers is governed by the co-object relation, which defines different relations for each *pair* of interfaces, a context-aware choice of wrapper is made at run time, and no problems analogous to wrapper chaining arise. This relies on the fact that later applications of Cake define new component identities as understood by the runtime—determined by their name within Cake source. Some care must therefore be taken to ensure these names are unique across the multiple Cake applications being integrated.

Limitations of wrapper chaining Wrapper chaining is no use in cases where a later Cake application is supposed to have priority over an earlier one, or where the event patterns in the higher-priority composition do not properly isolate the calls for local handling from those that should be passed through. This might owe to the limited expressiveness of the Cake event pattern language.

Chance interference Interference may occur between separate compositions of unrelated interfaces, simply by the chance event that overlapping sets of symbol names are used by unrelated code. The latter is reasonably plausible in languages such as C which lack explicit namespaces. There is no special provision for these cases; it may happen that existing event patterns happen to separate out the calls so that wrapper chaining will suffice, but there is no guarantee; in some cases, the unrelated earlier wrapper will render some calls non-interposable. This could be fixed by introducing a demultiplexing stage in each wrapper, using the on-stack return address to check that the expected component is calling, and to call down the wrapper chain if not.

Emitting Dwarf Another potential problem with multiple applications of Cake is that the Cake compiler does not emit DWARF information. Therefore, compiler-generated code may not itself be the subject of interposition (although this seems an unlikely scenario). Furthermore, any annotations added by the Cake programmer in one application of Cake will not be visible to subsequent applications, since we chose an ad-hoc implementation for propagating these in §4.3.2.

4.7 Summary

This chapter has discussed the implementability of Cake, and described a proof-of-concept implementation in reasonable detail. A high-performance, multithreading-capable implementation of Cake is still a target of future work, but we have presented some promising ideas in that direction.

Demonstrating implementability of Cake was a central requirement of its goal of practicality, which we have now fully demonstrated. The next chapter will evaluate the language's *expressiveness* by considering its usefulness in real use-cases.

Chapter 5

Evaluating the Cake language

Previous chapters have shown the design and implementation of the Cake language, and claimed that it is a convenient and expressive adaptation tool which can be applied to real-world tasks. Indeed, the thesis of this dissertation is that this tool is "significantly more effective in practice than conventional programming languages". This chapter provides experimental evidence which substantiates that claim.

5.1 Approach

Our evaluation comes in three parts.

In the first, we present some "null cases" of the Cake language, to illustrate how Cake replicates and extends conventional toolchains and their behaviour, and to show the impact of introducing Cake into the development process.

In the second, we present our experiences from a formative case study conducted while the Cake language was being designed and core elements of its implementation were being built. This is primarily a source of evidence that the implementation techniques discussed in Chapter 4 are both implementable and practically applicable. Some performance measurements are included and discussed.

In the third and most significant part of the evaluation, we comparatively evaluate the Cake language by identifying a series of three example tasks which have *already* been performed using conventional approaches, prior to the work in this dissertation, and have publicly available implementations. We then present Cake-based implementations reproducing as closely as possible the behaviour of the pre-existing conventional solutions. We compare this code to the equivalent Cake code. This comparison includes detailed description of the various facets of the Cake language design as they related to each task, and aggregate side-by-side measurements of code size as it differs between the Cake and original versions.

```
exists elf_reloc ("component1.o") c1;
exists elf_reloc ("component2.o") c2;
derive elf_reloc ("output.o") output = link[c1, c2];
```

Figure 5.1: Linker-like usage of Cake

5.2 Null cases

We begin by considering some hypothetical simple usages of Cake. As described in §2.2.11, the default behaviour of Cake resembles that of a conventional linker. Therefore, we begin by considering what happens when to using Cake to compose binaries whose interfaces are well-matched. We then proceed to cases where certain fairly shallow mismatches are added.

5.2.1 No mismatch

If a pair of components is not mismatched, then it follows that

- every function required by one component from the other is provided by that other component;
- the signatures of such functions match;
- every data type used by the two components to communicate exists in both modules;
- for each of these data types, its definition is the same, and therefore rep-compatible (§4.4.4), in both modules.

Such cases require no correspondences from the Cake programmer. The complete Cake code for all such use-cases looks invariably similar to that shown in Fig. 5.1.

Cake's default name-matching behaviour can draw implicit correspondences between all the relevant function calls and data type definitions. Furthermore, the shareability analysis (§4.4.4) finds every object to be shareable (since no path in one component's type reachability graph would not also be present in the other's). Therefore, at run time no co-objects need to be allocated.

By default, Cake's wrapper generation algorithm (§4.2.6) would generate trivial no-op wrapper functions for each relevant function call. However, a special case in the Cake compiler allows it to omit these wrapper functions in the cases where no interposition is necessary on a call or its parameters. This omission is safe only if all objects possibly reachable from the call's parameters are shareable. If this property is true of all function calls made between the two components, then the Cake compiler will not generate any wrapper code at all, and the output Makefile will contain a simple linker command exactly like the one used by a developer linking the components directly without the use of Cake. Therefore, in the well-matched case, Cake degenerates into a linker. Unsurprisingly, the output program need not be linked with the Cake runtime, and there is no run-time overhead.

```
exists elf_reloc ("component1.o") c1;
exists elf_reloc ("component2.o") c2;
derive elf_reloc ("output.o") output = link[c1, c2]
{
        c1 ⟷ c2
        {
                /* old name in caller */               /* new name in callee */
                do_something(...)                 ⟶ do_poke_sth(...);
                pattern /(read|write)_sth /(...)  ⟶ do_\\1_sth(...);
        }
};
```

Figure 5.2: Function renaming

5.2.2 Renaming functions

In the case where interfaces differ only in the naming of functions, the relevant Cake correspondences would include only event correspondences (§2.2.9), perhaps also using identifier patterns (§2.2.11) to cover many functions in a single rule. Fig. 5.2 shows a simple example.

This case is much like the previous one. The only difference is that rather than linking directly, function symbols may have to be renamed in the object code. The existing Cake compiler contains an implementation of this, using GNU objcopy[1] to rename the appropriate symbol in an object file requiring a function which is provided under a different name. Since this renaming occurs during the build process, there is, as before, no need to link with the Cake runtime. There is also no run-time overhead.

5.2.3 Renaming data types

We now consider the case where data types, or fields within structured data types, have been renamed. Aside from this, there are no changes in the layout of any data type.

Despite the renaming, all corresponding data types remain rep-compatible and shareable. We recall from §4.4.4 that the algorithm takes as input a list of corresponding data types, which need not be like-named, and accounts for renamings of fields. Therefore, the same compile-time checks as before can infer that no wrappers are necessary in this case.

5.2.4 More complex cases

Now we consider what happens when the definition of data structures within one or both components is changed, so that like-named data types are no longer rep-compatible.

Rep-incompatibility between data types used internally by one or both components need not result in any change from the previous cases, since they are not necessarily used in any interactions across the two components' shared interface. Specifically, if mismatched

[1]http://ww.gnu.org/software/binutils/

are not reachable from any parameter passed between the interfaces, according to the usual rules in shareability analysis (§4.4.4), then wrappers are not necessary. Note that the shareability analysis is conservative in determining reachability.

The point of departure from these degenerate cases occurs when a correspondence between rep-incompatible data types is introduced. Since some co-objects may now be allocated, the compiler is obliged to generate wrappers—not only for all functions from whose arguments an instance of these rep-incompatible data types could be reached, but in fact for all function calls occurring between the two components. This is because co-objects must be synchronised at every crossing of control from one component to another, to satisfy the "as if" semantics described in §2.4.1. In particular, since component code is entitled to save pointers to objects passed earlier, if we were to synchronise only at crossings which might be passing pointers to co-objects, updates to co-objects reached from these saved pointers would not be made visible in a timely fashion. So, in this case and in all more complex cases, the generated code follows the pattern described in Chapter 4.

5.3 Gtk+ case study

Our first case study is included for the sake of recounting experience gathered during the design and implementation of Cake, rather than for comparative measurement.

5.3.1 Outline

The study centres on a small utility called gtk-theme-switch[2]. The utility allows a user of applications written using the Gtk+ windowing toolkit [Krause 2007] to customise their appearance by selecting a "theme" among the selection of themes installed on the user's system. The utility comes in two versions—one for the version 1.2 series releases of the Gtk+ library, and one for the version 2.0 series. The API exposed by the library changed significantly between these two versions [GNOME Developers 2002], with consequent (even larger) changes to the underlying binary interface.

The utility itself is a small C program—943 and 993 raw lines respectively for the 1.2 and 2.0 versions. The differences in the two library interfaces were clearly significant enough for the developer to choose to maintain two separate versions, rather than abstract out a common core. Running the Unix diff utility on the two versions' sources reveals 210 line deletions and insertions are necessary to produce the 2.0 version from the 1.2 version. This is a nontrivial difference—enough to make it seem likely that the author of such a utility would much preferred to have only one version of the source to maintain, if adequate tool support for doing so were available.

Our goal in the study was therefore to generate code which could allow the binary of the old (version 1.2) utility to link with the new (version 2.0) library. When this study was conducted, the Cake language had not yet been conceived. This study was a formative

[2]The utility is available from http://freecode.com/projects/gtkthemeswitch, as of 2012/5/1.

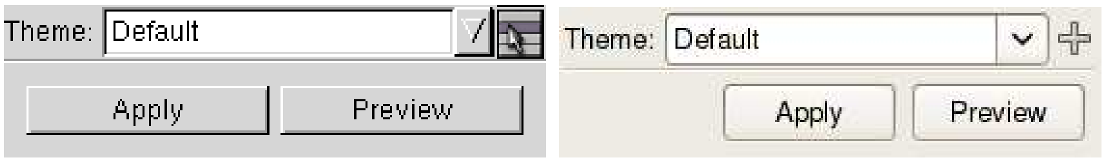


Figure 5.3: The Gtk+ client linked against the 1.2 (left) and 2.0 (right) library

influence on both, and represents the first implementability results for the key techniques in Chapter 4, including wrapper-based interposition (§4.2.6), object graph exploration and the replication-based approach to managing objects for each component (§4.4.4).

Fig. 5.3 illustrates the practical outcome of the study with screenshots of the same client linked against 1.2 (direct) and 2.0 (with adapter) versions of the library.

5.3.2 Relationship to the main Cake implementation

The Gtk+ study was implemented in an abstractly very similar way to other applications of Cake. Generated code, in the form of wrapper functions and value conversion functions (§4.2.5), executes on top of a runtime library responsible for constructing and synchronising replica object graphs (§4.4.4).

The role of the Cake compiler was filled by a set of ad-hoc scripts consuming the libraries' header files, and somewhat specialised to the naming conventions and layout found within them. A small amount of code was hand-written, to deal with interface features too complex for the scripts to handle. These included functions taking arrays as arguments, one-to-many function mappings, and functions returning objects by value. All data types corresponded one-to-one between the two interfaces, although nearly always with differing layouts, and often with different names; a unifying name, known as a "form", was used to refer to each corresponding pair of data types in the code.

The Cake runtime was developed during this study, as a generic support library for maintaining replicated object graphs, and retains essentially the same implementation in its current form. The "initial co-object relation" technique (§4.4.3) is used extensively to manage the flow of callback function pointers from client to library. The code is unoptimised and the data structures used were deliberately naive, to ease debugging.

This case study made no use of DWARF information. Instead, the generated code directly consumes the libraries' header files, using C++ namespaces to separate like-named definitions from the two sets of headers. Ad-hoc text formats were used to describe the signatures of called functions and layouts of replicated objects. The object layout tables were translated by another script into C99 source files, linked into the program, and used by the runtime navigate the in-memory structures of the object graphs it explores. These ad-hoc tables were constructed with knowledge of how each object's fields were used by the client, so have the effect of applying `opaque` and `ignored` annotations included in the Cake language (§4.4.6) to limit the extent of unnecessary exploration and replication. The study also does not use dynamic object discovery when allocating co-object structures.

```
rightclick:1{FORM_GTK_WIDGET}2{FORM_GDK_EVENT_BUTTON}
apply_clicked:1{FORM_GTK_WIDGET}
font_browse_clicked:1{FORM_GTK_WIDGET}
install_clicked:1{FORM_GTK_WIDGET}
install_ok_clicked:1{FORM_GTK_WIDGET}
preview_clicked:1{FORM_GTK_WIDGET}
set_font:1{FORM_GTK_WIDGET}2{FORM_GTK_WIDGET}
```

Figure 5.4: A description of the Gtk+ callback function signatures

Rather, it assumes that static type information in the header files is precise enough for the purposes of the adapter.

Fig. 5.4 shows the ad-hoc text file used to describe the function signatures of the client's callback functions. (A similar file, but much larger, was defined for the library.) Fig. 5.5 shows a wrapper function generated from this description (and other ad-hoc inputs describing the data types). Fig. 5.6 shows parts of the generated tables describing the layouts of objects.

(We will refer to the script-generated code as "Cake-generated code", even though strictly speaking it is not derived from Cake source code. It is also not entirely generated, in that it includes a small quantity of manually written code.)

5.3.3 Performance

For completeness, we consider the performance of the Gtk+ study here. This study was formative, and provided many of the early indications that the incorporated techniques could be made to work. However, we note that performance data derived from this system cannot be considered strongly indicative of anything in particular—certainly not "the performance of Cake", which, as we will remark at the end of this chapter (§5.9) is very highly dependent on the particular usage scenario, and yet to be explored within an optimised implementation. The version of the Cake runtime found in this system is both unoptimised and naive. In particular, the co-object relation (§4.4.1) is represented by a simple linked list, making lookups unnecessarily slow. However, because it is based on hand-crafted metadata tables rather than DWARF information, it is also free from certain unnecessary overheads (resulting from the use of an unoptimised DWARF library) that are present in the current head implementation of the runtime.

To measure its performance, we ran the system under the Callgrind tool from the Valgrind suite [Nethercote and Seward 2007], which extracts instruction-level profile data. In our case we use it to count the number of instructions executed in wrapper and non-wrapper code, as a proxy for execution time. We compiled a version of the Cake-generated code with all debugging printouts and assertions removed. We then performed a scripted list of interactions with the Gtk+ program's interface, finishing by exiting the program. From the Callgrind data, we then extracted the instruction count attributed to Cake-generated code versus that attributed to the wrapped library calls and dynamic linker overhead. We can assume that without Cake, the library would be exercised by a perfectly matched client making the necessary calls directly, so would incur the same cost in

```
#include <stdio.h>
#include "rep_man-shared.h"
#include "rep_man_tables.h"

extern const char *object_forms[]; /* table of strings */


extern void __real_apply_clicked(void * arg1,void* arg2);
void __wrap_apply_clicked(void * arg1,void* arg2)
{
        fprintf ( stderr , "%s:␣called␣%s\n", __FILE__, __func__);
        sync_all_co_objects(REP_GTK_20, REP_GTK_12);
        /* process arg1 */
        fprintf ( stderr , "%s:␣arg1␣(%p)␣is␣non-opaque␣pointer␣to␣rep-mismatched␣object␣(form␣%s,␣rep␣
            %d)\n", __FILE__, arg1, object_forms[FORM_GTK_WIDGET], REP_GTK_20);
        walk_bfs (REP_GTK_20, arg1, FORM_GTK_WIDGET, REP_GTK_12,
            allocate_co_object_idem, REP_GTK_20, REP_GTK_12);
        walk_bfs (REP_GTK_20, arg1, FORM_GTK_WIDGET, REP_GTK_12, init_co_object,
            REP_GTK_20, REP_GTK_12);
        fprintf ( stderr , "%s:␣calling␣__real_apply_clicked\n", __FILE__);
        /* call the 2.x function */
        __real_apply_clicked(find_co_object(arg1, REP_GTK_20, REP_GTK_12, NULL,
            FORM_STORED_SIZE(REP_GTK_12, FORM_GTK_WIDGET)),arg2);
        fprintf ( stderr , "%s:␣__real_apply_clicked␣returned␣into␣wrapper\n", __FILE__);
        fprintf ( stderr , "%s:␣syncing␣all␣co-objects␣from␣rep␣REP_GTK_12␣to␣rep␣REP_GTK_20\n"
            , __FILE__);
        sync_all_co_objects(REP_GTK_12, REP_GTK_20);
        /* handle return value */
        fprintf ( stderr , "%s:␣returning␣from␣%s\n", __FILE__, __func__);
        return;
}
```

Figure 5.5: A wrapper function generated by scripts in the Gtk+ study

executing the underlying library call (and dynamic linker activity), but without the wrapper overhead. Therefore, we can measure the overhead added by Cake as the difference between the wrapper cost and the underlying function cost.

(We take this approach, in preference to comparing "real" on-CPU execution time with that of the original Gtk+ 2.0 version of the program, owing to the difficulty of precisely repeating the same execution path in multiple runs of an interactive application. Even when following the same script of interactions, some run-to-run variability is observed in in the number of times various functions are called. Therefore, we prefer to analyse data gathered in a single run of the program. However, at the end of this section we do present some aggregate figures comparing separate runs of the two different versions.)

Primarily, the added costs in the Cake-generated version are those of object graph exploration and copying of data (including synchronisation of the logical replicas). Table 5.1 shows the results broken down by wrapper. Nine wrapper functions did not yield meaningful breakdowns from Callgrind, since they are part of call-graph cycles arising from the use of callbacks in the code. These are omitted from the table. From the others,

```
/* GtkFontSelectionDialog, 1.2 rep, has subobjects at these offsets . */
const size_t rep_gtk_12_gtk_font_selection_dialog_subobject_offsets[] = {
        offsetof (GtkFontSelectionDialog, window),
        (size_t) -1 /* terminator */
};

/* GtkFontSelectionDialog, 1.2 rep, has subobjects of these forms. */
const int rep_gtk_12_gtk_font_selection_dialog_subobject_forms[] = {
        FORM_GTK_WINDOW,
        INT_MAX /* terminator */
};

/* GtkFontSelectionDialog, 1.2 rep, has pointers at these offsets . */
const size_t rep_gtk_12_gtk_font_selection_dialog_derefed_offsets[] = {
        offsetof (GtkFontSelectionDialog, ok_button),
        offsetof (GtkFontSelectionDialog, cancel_button),
        (size_t) -1 /* terminator */
};

/* GtkFontSelectionDialog, 1.2 rep, has pointers to these forms. */
const int rep_gtk_12_gtk_font_selection_dialog_derefed_forms[] = {
        FORM_GTK_WIDGET,
        FORM_GTK_WIDGET,
        INT_MAX /* terminator */
};
```

Figure 5.6: Fragments of the table describing object layouts

we can see that there is a very large variance in overhead from wrapper to wrapper, largely explained by the wrapped library call's function. The most significant relative overhead is in g_type_check_instance_cast, whose core implementation involves, in the common case, only a one-word comparison on a tag field. When wrapped, it incurs the cost of a full synchronisation cycle, taking several hundred times as long to execute. The highest absolute cost is g_list_insert_sorted, probably reflecting the fact that from this function, a potentially large space of objects is exposed to object graph exploration algorithm—including the whole linked list and whatever can be reached from it. Conversely, in other cases, the costs of synchronisation and exploration are dominated by the library call's underlying function, such as in the gtk_label_new, gtk_combo_new and other functions constructing new widgets. Co-object churn is also a factor; wrappers called at times when few co-objects exist, such as during program start-up, will incur less synchronisation cost than those called at other times.

Finally, we repeated our scripted list of interactions on both the Cake-composed and original Gtk+ 2.0 clients, and counted the total instruction fetch count in each case. Over five runs of each, the median of the Cake-composed version was 2.93×10^8 instructions, against 2.24×10^8 for the plain Gtk+ 2.0 client, and all results within 2% of the relevant median. In other words, this application of Cake has caused an approximate 30% slow-down in overall application performance. This is entirely reasonable for many classes of application, confirming our claim that Cake's approach is a practically applicable one.

wrapper name	# calls	insn fetches	underlying	overhead	factor
__wrap_g_list_insert_sorted	44	13081442	34503	13046939	380.
__wrap_gtk_dialog_new_resizable	1	10274006	10199719	74287	0.0073
__wrap_gtk_combo_set_popdown_strings	1	5232109	4885513	346596	0.071
__wrap_gtk_combo_new	1	3462937	3414342	48595	0.014
__wrap_gtk_font_selection_dialog_get_font_name	1	3280671	3194282	86389	0.027
__wrap_g_type_check_instance_cast	36	2485524	3942	2481582	630.
__wrap_gdk_pixmap_colormap_create_from_xpm_d	1	1908432	1858428	50004	0.027
__wrap_gtk_box_pack_start	11	982313	314663	667650	2.1
__wrap_gtk_label_new	1	822123	802955	19168	0.024
__wrap_gtk_widget_realize	1	704275	687591	16684	0.024
__wrap_g_signal_connect	7	506962	21236	485726	22.0
__wrap_gtk_button_new_with_label	4	444998	192578	252420	1.3
__wrap_gtk_check_button_new	1	210082	182309	27773	0.15
__wrap_gtk_rc_get_style	1	171160	77396	93764	1.2
__wrap_gtk_entry_get_text	2	130174	1486	128688	87.
__wrap_gtk_widget_show_all	1	81539	36068	45471	1.3
__wrap_gtk_toggle_button_set_active	1	80249	20019	60230	3.0
__wrap_gtk_entry_new	1	79171	31368	47803	1.5
__wrap_gtk_hbox_new	2	77560	12640	64920	5.1
__wrap_gtk_tooltips_set_tip	1	76075	17772	58303	3.3
__wrap_gtk_pixmap_new	1	75191	14286	60905	4.3
__wrap_gtk_container_add	1	66869	9246	57623	6.2
__wrap_g_slist_length	1	61996	1771	60225	34.
__wrap_gtk_event_box_new	1	51395	5223	46172	8.8
__wrap_gtk_widget_set_events	1	48589	4800	43789	9.1
__wrap_gtk_window_set_title	1	45418	28504	16914	0.59
__wrap_gdk_pixmap_unref	1	38181	1773	36408	21.
__wrap_gdk_bitmap_unref	1	36429	232	36197	160.
__wrap_gtk_window_set_policy	1	33097	16427	16670	1.0
__wrap_gtk_widget_get_colormap	1	28501	2102	26399	13.
__wrap_gtk_tooltips_new	1	27467	8758	18709	2.1
__wrap_gtk_tooltips_set_delay	1	17054	1506	15548	10.
__wrap_gtk_font_selection_dialog_get_type	11	3483	2104	1379	0.66
__wrap_gtk_box_get_type	11	3355	1490	1865	1.3
__wrap_gtk_container_get_type	1	2484	1633	851	0.52
__wrap_gtk_toggle_button_get_type	1	2467	1520	947	0.62
__wrap_gtk_window_get_type	2	2291	1307	984	0.75
__wrap_gtk_dialog_get_type	5	2030	1541	489	0.32
__wrap_gtk_entry_get_type	4	1870	1478	392	0.27
__wrap_gtk_combo_get_type	1	1388	1286	102	0.079

Table 5.1: Wrapper overheads in the Gtk+ study. "Instruction fetch" counts instructions fetched (by Callgrind's simulated CPU) during the execution of each wrapper, including all called functions. "Underlying" counts the subset of these which are due to the underlying library and not to the Cake runtime. "Overhead" is the difference, and "factor" is the same expressed as an approximate multiplier of the underlying execution time.

5.3.4 Shareability

We also use the Gtk+ study as an opportunity to evaluate the usefulness of the shareability analysis described in §4.4.4. Recall that shareability is only especially useful for tasks arising from interface evolution. Since shareable objects must be not only representation-compatible, but not reachable from any representation-incompatible objects, they are unlikely to be found in completely unanticipated compositions.

We took the Gtk+ 1.2 library, starting with release 1.2.0, and considered several successive versions, up to the end of the 1.2 release series and also including the first release of the 2.0 series. We analysed the shareability of like-named data types across these library versions, to see how it declined as the interface evolved away from its starting point. Since the analysis is currently somewhat expensive to run, we restricted ourselves to the 118 named

version	like-named	and rep-compatible	and shareable	% shareable
1.2.0	118	118	118	100
1.2.2	117	116	93	79
1.2.4	117	116	93	79
1.2.6	117	115	69	58
1.2.8	117	115	69	58
1.2.10	117	115	69	58
2.0.0	95	47	42	36

Table 5.2: Shareability results for successive versions of Gdk. Percentages are relative to the total of 118 Gdk data types in the 1.2.0 version.

data types defined in the Gdk portion of the library (which abstracts the underlying drawing primitives provided by the windowing system), all of whose names begin with "Gdk".

As a sanity check, we first compared it with itself, finding as expected that every data type is shareable. (This is a property required to support the linker-like degenerate cases mentioned in §5.2.) We then proceeded to compute the shareability with each subsequent version two minor revisions apart, up to release 1.2.10. Table 5.2 shows the results. The final and penultimate columns show the number of shareable data types declining as the version gap increases. The second and third columns count pairs of data types satisfying intermediate conditions that are necessary (but not sufficient) for shareability. Specifically, we might find in a subsequent version that a given data type is not present, at least not with the same name (therefore not counted in any column); present but not rep-compatible (counted in the second column only); or present and rep-compatible, but not shareable owing to the reachability of some incompatible pair of data types (counted in the second and third columns).

5.4 Measurement methodology

Our remaining case studies consider tasks which have *already* been performed using conventional approaches. We discuss the details of task in depth, and summarise the outcome of each task by aggregate side-by-side measurements of code size as it differs between the Cake and original versions. Only language-independent software measurements are appropriate, since we are comparing Cake and non-Cake implementations. Therefore, we use the following three simple counts over source code syntax:

LoC (nb nc) a count of lines of code that are non-blank and non-comment;

Tokens a count of tokens in the code, as output by a lexer for the appropriate language;

Semicolons a count of semicolon tokens, proxying the count of "statements" or "declarations" in the code. This is meaningful since we only compare Cake code against C code, and these two languages make very similar syntactic use of the semicolon.

We state several observations about our methodology and its practicalities.

Measurements Although we use count-based measurements, we appreciate their shortcomings. Cake's lower counts certainly originate partly in improved abstraction, but perhaps also to incidental factors such as a reduction in boilerplate code. To combat this effect, we report an additional "adjusted" measurement for C code. This measures the code after manually erasing common C boilerplate—specifically, variable declarations and function prototypes. This is an ad-hoc adjustment, so still does not account for certain other areas where Cake's syntax may be more concise (for example, error-path control flow).

Code left as is Cake is a restricted language; it has a purely reactive execution model, and cannot directly express recursion or iteration. Some tasks therefore contain small pieces of code that cannot be implemented in Cake and must be written in another language. The "remaining" column in our tables refers to C code that could not be reimplemented in Cake and was left as is.

Percentages For ease of reference, we also calculate a percentage showing the relative size of the Cake implementation, by subtracting the C "remaining" counts from the C "adjusted" counts and then dividing the Cake count by this total. (In fact, this calculation is slightly unfair to Cake, in that the "remaining" counts were not adjusted for boilerplate, so subtract slightly too much from the total C code. The difference is not significant.)

Unevaluated benefits Even if Cake did little to simplify code, there are inherent benefits in Cake's black-box, binary approach which are not substantially evaluated here. Our goals with Cake are not simply to provide a marginally better way of coding adapters. Rather, we wish to enable a shift in development practices towards integration-based approaches and away from reimplementation and invasive editing. This desire directly motivates the adoptable, black-box, binary design of Cake. However, we clearly cannot evaluate these design aspects in small-scale studies.

Sources of error As remarked in §4.5, our current implementation is not capable of generating code for the whole Cake language. The Cake code written in the experiments in this chapter has been carefully written with direct and detailed reference to the original C code, and we present several detailed side-by-side comparisons. However, without the ability to compile and test the Cake versions, there will no doubt be bugs, and hence some degree of error in the measurements taken. As we will see, the results are such that any error introduced in this way cannot feasibly endanger our conclusions.

5.5 Bridging related components: libp2k

Our first task considers a pair of filesystem interfaces. Filesystems are a ubiquitous abstraction: they appear in kernel-level operating system code, but also in graphical desktop

```
// hunk 1: basic event correspondence patterns
pattern puffs_fs_(.*) { names (mount: _) }
      ⟷ rump_vfs_\1 { names (mount: _) };
pattern puffs_node_(.*) { names (mount: _, cookie: _ as vnode_unlocked ptr) }
      ⟷ RUMP_VOP_\U\1\E { names (cookie: _) };
```

Figure 5.7: Basic event correspondences for p2k

```
// hunk 2: basic value correspondences
values puffs_usermount ( puffs_getspecific ( this )) ⟶ mount;
values puffs_cred ({
  puffs_cred_getuid( this , out uid) ;| let uid = 0;
  puffs_cred_getgid( this , out gid) ;| let gid = 0;
  puffs_cred_getgroups(pcr, out groups[NGROUPS], out ngroups)})
  ⟶ (rump_cred_create(uid, gid, ngroups, groups)) kauth_cred;
values puffs_cred ⟵ (rump_cred_destroy(this)) kauth_cred;
values puffs_cn ⟶ (rump_makecn(that↪pcn_nameiop, that↪pcn_flags,
  that↪pcn_name, that↪pcn_namelen, that↪pcn_cred, curlwp)) component_name;
values puffs_cn ⟵ (rump_freecn(this, RUMPCN_FREECRED)) component_name;
```

Figure 5.8: Basic value correspondences for p2k

environments,[3] in web servers,[4] and elsewhere. The programming interfaces behind which filesystems are implemented are invariably abstractly similar, yet often concretely different, making them frequent candidates for black-box adaptation.

Our task in this section reimplements the libp2k adapter [Kantee 2009] from the NetBSD operating system. This adapts filesystems from an API embedded in a special environment for running unmodified kernel code, including filesystems, in user-space (rump) so that they may be used from NetBSD's native user-space filesystem implementation (puffs).

Figures 5.7–5.13 show a large portion of the Cake code for this task. We were fortunate to have a one-to-one correspondence between most calls in the two interfaces, with naming conventions which map straightforwardly; this is captured neatly in two pattern rules (Fig. 5.7).

There are some simple correspondences between objects in the two interfaces (Fig. 5.8). The puffs_cred rightward rule is rendered in the C implementation as a utility function cred_create() called from many different locations in the code. We can write this utility logic once as a value correspondence, and have it automatically applied wherever any Cake rule demands conversion from a puffs_cred to a kauth_cred_t.

Some rump library calls leave their vnode target unlocked on return, so we need not apply RUMP_VOP_UNLOCK() in those exceptional cases (Fig. 5.9). These calls are exactly those which may modify the filesystem's directory structure; such calls also require

[3]Two popular examples on Unix platforms are Gnome VFS, http://library.gnome.org/devel/gnome-vfs-2.0/unstable/ and KIO, http://api.kde.org/4.x-api/kdelibs-apidocs/kio/html/. These URLs are valid as of 2010/12/7.

[4]One notable example is Apache Commons VFS, at http://commons.apache.org/vfs/ as of 2010/12/7.

```
// hunk 3: more value corresps incl. special unlocked-return
values vnode_unlocked ⟶
 ({RUMP_VOP_LOCK(that, LK_EXCLUSIVE); that}) vnode;
// no need to update vnode value in the reverse direction
values vnode_unlocked ⟵(RUMP_VOP_UNLOCK(that, 0)) vnode;
values vnode_bump ⟶
 ({RUMP_VOP_LOCK(that, LK_EXCLUSIVE);
                  rump_vp_incref(that); that}) vnode;
values vnode_bump ⟵vnode; // unlock not required
puffs_node_create(mount, vn as vnode_bump, ni, cn, vap)
  ⟶ RUMP_VOP_CREATE(vn, ni, cn, vap);
puffs_node_mknod(mount, vn as vnode_bump, ni, cn, vap)
  ⟶ RUMP_VOP_MKNOD(vn, ni, cn, vap);
// ... similar for remove, link, rename, ...
```

Figure 5.9: Special value correspondences (1) for p2k

```
// hunk 4a: how to output parameters by "newinfo"
values puffs_newinfo ({puffs_newinfo_setcookie(this, that); this})
 ⟵(RUMP_VOP_UNLOCK(this, 0)) vnode;
// Some calls return a fuller set of newinfo
values puffs_full_newinfo ({puffs_newinfo_setcookie(this, that);
 puffs_newinfo_setvtype(this, vtype);
 puffs_newinfo_setsize(this, vsize);
 puffs_newinfo_setrdev(this, rdev); this}) ⟵
  ({let (vtype, vsize, rdev) = rump_getvninfo(this); this}) vnode
```

Figure 5.10: Special value correspondences (2) for p2k

the reference count of any modified vnode to be pre-incremented to avoid premature reclamation. This is captured by the vnode_bump rules.

This example illustrates a common idiom in Cake programming. Having introduced a general rule, here for vnode_unlocked, we introduce a new artificial named data type, here vnode_bump, with alternative value correspondences. Then the programmer supplies this name in the contexts where the special treatment is required—either as annotations in a declare block, or inlined in the relevant event correspondences using an as declaration. Here, we chose also to base the more common-case rule, vnode_unlocked, on an artificial data type, rather than using vnode directly. This forces the Cake programmer to be explicit about which rule is required in every context where puffs passes a vnode.

Some rump functions return output values through parameters. The puffs interface requires these to be passed through an opaque object, puffs_newinfo, populated using setter functions. We can express this by firstly describing which rump calls' arguments are outputs (Fig. 5.11), and secondly providing value correspondences between puffs_newinfo and the relevant rump structures (Fig. 5.10).

In the above rules, as an optimisation, two calls (read() and readdir()) use shared (multi-reader) vnode locking, rather than exclusive locking. This is done by two additional rules (Fig. 5.12), and replicates the behaviour of the C implementation.

```
// hunk 4b: tell Cake which calls need " full " newinfo...
exists elf_archive (" puffs.a " ) puffs ; // this hunk would appear at
derive elf_archive puffs_inst = // ... the top of the .cake file
  instantiate ( puffs , puffs_ops, pops, " puffs" );
 puffs_inst { declare {
    puffs_fs_fhtonode : (_, _, _, out puffs_newinfo as puffs_full_newinfo ) ⇒ _;
    puffs_node_lookup : (_, _, out puffs_newinfo as puffs_full_newinfo , _) ⇒ _;
} };
```

Figure 5.11: Annotations for enabling special value correspondences in p2k

```
// hunk 5: shared locking
values vnode_lkshared ⟶({RUMP_VOP_LOCK(that, LK_SHARED); that}) vnode;
values vnode_lkshared ⟵({RUMP_VOP_UNLOCK(that, 0); that}) vnode;
```

Figure 5.12: Special value correspondences (3) for p2k

Code in librump originated in the kernel, where client reading and writing of file data to or from user processes requires address-space traversal. The four relevant calls use a special interface called uio for passing this data. To us, this is just a new way of packaging parameters for input and/or output, and is handled by a few more correspondences (Fig. 5.13).

The rules shown generate complete implementations of all but six of the 28 p2k wrappers. The omissions are explained by special error-handling requirements, one-to-many function mappings, and function correspondences which do not follow the naming convention. These are easily handled by a few more event correspondences, shown for completeness in Fig. 5.14.

As usual with dynamically loaded components, puffs calls into p2k indirectly, through a table of function pointers passed during initialization. We instantiate this table using Cake's instantiate helper (§2.3.3). The only logic required which Cake could not adequately express was about 40 lines of C code in p2k.c which load the filesystem (the p2k_run_fs() function). This consists of a combination of calls to the core puffs and rump

```
// hunk 6: input/output by uio
values uio_outbuf (buf: uint8_t [] ptr, resid : size_t ptr,
  off : const off_t) ⟶ (rump_uio_setup(that↪buf,
 ∗that↪resid , that↪offset , RUMPUIO_READ)) uio;
values uio_outresult ⟵ (rump_uio_free(this)) uio ;
values uio_outres_len_off ⟵ ({rump_uio_getresid(that↪resid);
      rump_uio_getoff(&that↪readoff);
      rump_uio_free(this)}) uio ;
puffs_node_read(mount, vn as vnode_lkshared, uio as uio_outbuf(buf, resid , offset ),
      _, resid out_as uio_outresult, cr , ioflag ) ⟶ RUMP_VOP_READ(vn, uio, ioflag, cr);
// similar : readlink , readdir , "uio_inbuf" and write
```

Figure 5.13: Special value correspondences (4) for p2k

```
// "inactive" notification requires special action in rump
puffs_node_inactive(mount, vn as vn_no_lk) ⟶ {
 rump_vp_interlock(vn);
 RUMP_VOP_PUTPAGES(vn, 0, 0, PGO_ALLPAGES);
 RUMP_VOP_LOCK(vn, LK_EXCLUSIVE);
 RUMP_VOP_INACTIVE(vn, out recycle);
 }--
⟵
--{ if recycle then puffs_setback(
      puffs_cc_getcc(mount, PUFFS_SETBACK_NOREF_N1
   )) else void; };

// reclaim maps to call with non-analogous name
puffs_node_reclaim(mount, vn as vn_no_lk)
 ⟶ { rump_vp_recycle_nokidding(vn); void };

// unmount requires special action
puffs_fs_unmount(mount, flags)
 (let rvp in_as vn_no_lk = puffs_getroot(mount)↪pn_data)⟶ {
  rump_vp_recycle_nokidding(rvp);
  rump_vfs_unmount(mount, flags) ;|
  ( rump_vfs_root(mount, out rvp2, 0); assert(success && rvp == rvp2); ) };

// puffs sync needs two calls in rump
puffs_fs_sync(mount, waitfor, cr) ⟶
 { rump_vfs_sync(mount, waitfor, cr); rump_bioops_sync(); };

// fhtonode and nodetofh map to non-analogous names
puffs_fs_fhtonode(mount, fid, _, out ni as puffs_full_newinfo) ⟶
 rump_vfs_fhtovp(mount, fid, ni);

puffs_fs_nodetofh(mount, vn as vnode_nolk, fid, fidsize )
 ⟶ rump_vfs_vptofh(vn, fid, fidsize );
```

Figure 5.14: Final correspondences for p2k

interfaces to register the filesystem with puffs and allocate rump-side state, rather than logic for implementing a particular puffs operation using the rump interface. (This code is “conversational” in the sense of §3.5, which explains why it cannot be expressed in Cake.)

Table 5.3 shows the aggregate comparison of Cake's p2k with the original implementation. We can see that the Cake implementation is roughly 30%–45% the size of the C implementation, event after adjustments for relatively more verbose C syntax and the small amount of C code retained as is.

In summary, Cake can express the p2k component in a fraction of the code size. Evident from the code snippets, but not measured, is an additional benefit that the Cake implementation localises each concern of the two interfaces' syntactic and semantic differences more clearly than the existing C code. For example, treatment of unlocking and reference counting is handled by discrete and localised rules, rather than being scattered throughout the code.

	C	adjusted	Cake	remaining C	%age
LoC (nb nc)	605	523	129	54	27%
tokens	3469	3137	1285	347	46%
semicolons	358	277	74	33	30%

Table 5.3: Comparing p2k implementations in Cake and C.

5.5.1 Performance

As with the Gtk+ study, we briefly consider the performance of the p2k study for completeness. However, as before, we do not consider the figures strongly indicative of any useful property of the Cake language (as distinct from the current state of its implementation).

Unlike the Gtk+ study, the p2k study runs on top of the DWARF-based version of the Cake runtime, and has fully dynamic binding semantics. The current implementation of this is immature and completely unoptimised code, which, for ease of continuing development, remains deliberately naive in many places. For example, it uses linear search through certain key data structures (including the search for named DWARF elements), neglects to cache various reusable results (such as offsets of fields within objects, computed from DWARF information), and uses inefficient string-based representations for other important objects (notably data type identifiers). It also makes a large number of avoidable heap allocations when iterating through DWARF entries, of which frequently thousands are traversed during a single query.

As a simple worst-case benchmark, we measure the cost of running the puffs_fs_sync wrapper function dispatching to the tmpfs temporary filesystem implementation from rump. We do this measurement for our two implementations of this function: one implemented by the original libp2k code, and one in the Cake-generated version. This is a worst-case comparison because the fs_sync call in tmpfs is a no-op which simply returns zero (since the data in a temporary filesystem resides in memory only).

NetBSD 5.0 (the only operating system on which the case study executes) offers an API for measuring resource consumption, based on the BSD getrusage() system call. However, this is known to be unreliable.[5] Unfortunately, the Valgrind approach of the Gtk+ study is not available to us, since no port of Valgrind is available for NetBSD 5.0. Therefore, we proceed with the getrusage() method, but note its lack of accuracy.

The vfs_sync operation normally executes every 30 seconds, triggered by system-wide period synchronisation of filesystems. Our approach to measurement was simply to instrument the puffs dispatch loop with getrusage calls before and after calls into the wrapper, and compute the elapsed time as the difference. The same instrumentation suffices for both libp2k and Cake-based implementations. We let both systems run for 60 seconds before taking the first measurement, to allow any lazy initializations to take place. (In

[5]For example, it can often generate negative elapsed time measurements. This is discussed by Woods in a mail to the tech-kern@netbsd.org mailing list, available at http://mail-index.netbsd.org/tech-kern/2011/10/28/msg011778.html as retrieved on 2012/5/1.

	time/μs	
	libp2k	Cake
	69	123712
	73	126829
median	78	129572
	82	131756
	93	134241

Table 5.4: Execution time of the fs_sync call

particular, the Cake runtime initializes itself on the first call into it, rather than at program start-up, so the first fs_sync call takes markedly longer than subsequent ones.) We then recorded the next five measurements.

Table 5.4 shows the results. These measurements were taken in a VirtualBox 4.1.2 virtual machine running NetBSD 5.0.1, configured with 768MB memory (and otherwise unloaded), on an IBM Thinkpad T60 with a 32-bit Intel Core 2 Duo T2400 CPU and 2GB memory (similarly unloaded). We ignore the computed elapsed system time, since it was two orders of magnitude smaller than the user time, and negative.

Very roughly, they show that the Cake version takes around 1000–2000 times as long to execute as the libp2k version.

This might seem excessive, but considering that the underlying library call does nothing and that the adaptation logic itself does very little (primarily locking and looking up fields), it is understandable that the cost of dynamic binding is extremely significant in relative terms. This is especially true given the unoptimised state of the implementation. We note that the figures are roughly in line with the Gtk+ figures, where a wrapper for a trivial library call took several hundred times the execution time of the wrapper itself.

Some other details are worth comparing between the Gtk+ study and this one. Whereas in the Gtk+ study, the effect of opaque and ignored annotations (encoded into the ad-hoc input files to code generation) limited the extent of object graph exploration, in the p2k case we took an analogous measure by observing that no wrapper function requires deep object graph exploration, and limited the depth of exploration to 1 (i.e. one level of indirection from the arguments). However, a key difference between the two studies is that the Gtk+ study used compile-time information about the structure of objects passed by each function, so did not exhibit any overhead from dynamic binding. By contrast, the p2k study includes the additional overhead from dynamically discovering the objects being passed via each pointer argument, and other dynamic lookups to discover the relevant value conversion functions.

5.6 Migration between support libraries: ephy–webkit

Another area of continuing evolution in contemporary codebases is that of web browsers and associated software. The Epiphany web browser[6] migrated during 2007–08 from a single Mozilla-based HTML display widget to a more flexible codebase supporting both the Mozilla renderer and a Webkit-based one. We compare Epiphany's internal WebKitEmbed adaptation layer with a Cake implementation.

The developers of Epiphany chose to strip out the adaptation layer after Webkit migration was completed, around July 2008, and target Webkit APIs directly. We therefore used Subversion revision 8300 (28 June 2008) as the reference codebase for Cake reimplementation. Although there is no relevant discussion of the decision to remove the abstraction layer, either in the changelogs or mailing list archives, clearly the developers anticipated no future need to change the target API. Performance could be a plausible motivation, but the interface in question is not particularly performance-sensitive. More likely, the change was intended to simplify the codebase, in the assumption (arguably optimistic) that there would be no future need to support alternative renderers. (This highlights the anticipation difficulties surrounding abstraction layers, as we remarked in §1.2.2.)

The adaptation logic we reimplemented is effectively circumscribed by the definition of class WebKitEmbed. For simplicity, we left as is some additional adaptation code handling cookie management, password management and certain other functionality, since this contained only no-op implementations in our chosen revision. Similarly, we retained the utility classes WebKitEmbedPrefs and WebKitEmbedHistoryItem for use by our adaptation logic; these could be implemented in Cake, but owing to their small size, their C code is dominated by boilerplate, so would not give useful measurements. A particular complication with this boilerplate is that it is derived from the GObject library on which Epiphany is based, and is generated by heavy use of C preprocessing; we discuss this in the next subsection.

5.6.1 Objects, associations and their construction

Epiphany uses subclassing (using the GObject library [Krause 2007]) to connect an Embed object with a Webkit instance: the subclass's fields point to Webkit resources. In Cake we use an association: the Embed object is associated with the relevant Webkit objects. Recall that this is implemented using an aggregate *umbrella object*, as described in §2.3.4. In the Cake implementation of the adapter, the umbrella object's additional fields accommodate the state which was added by subclassing in the original implementation. Contrast the Cake fragment of Fig. 5.15 with the C fragment of Fig. 5.16.

In the C case, a WebKitEmbed object *is a* EphyBaseEmbed and *has a* WebKitWebView, an EphyHistory and some other state. In the Cake version we factor this slightly differently: a given EphyBaseEmbed and EphyHistory are *associated with* a WebKitWebView. "Is associated with" is effectively the same as a *has-a* relationship, as conventionally encoded by a stored pointer. However, in our Cake code this is encoded not by directly embedding

[6]http://www.gnome.org/projects/epiphany/

```
values (embed: EphyBaseEmbed,
        history : EphyHistory)  ⟷   (web_view: WebKitWebView,
                                      scrolled_window: GtkScrolledWindow,
                                      load_state: WebKitEmbedLoadState,
                                      loading_uri : char [])
{ /* ... */ }
```

Figure 5.15: Associating corresponding objects in Cake

```
struct WebKitEmbed {            // from webkit-embed.h line 56
  EphyBaseEmbed parent_instance;

  /*< private >*/
  WebKitEmbedPrivate *priv;
};

struct WebKitEmbedPrivate       // from webkit-embed.c line 65
{
  WebKitWebView *web_view;
  GtkScrolledWindow *scrolled_window;
  WebKitEmbedLoadState load_state;
  char *loading_uri ;
  EphyHistory * history ;
};
```

Figure 5.16: Subclassing-like containment and pointer-based association in C

a pointer to the subordinate object, but associatively, by maintaining references in the table implementing the co-object relation (§4.4.1).

Instantiation These rules present a complication: how do we ensure that the Epiphany client binary, which was compiled to instantiate and manipulate instances of some subclass, will instead instantiate and manipulate this association state using the logic in our Cake rules? This requires a combination of interposing on *instantiation* of state and interposing on *access* to that state. As often in Cake programming, more than one mix of these two tactics can lead to a viable solution. We can choose not to interpose directly, instead relying on the behaviour that an EphyBaseEmbed object will be a subobject instantiated within some enclosing object. The form of this enclosing object will be determined by whatever alternative HTML rendering library the client binary was compiled against (the Mozilla renderer in our case). Such an object is nevertheless usable—its EphyBaseEmbed subobject will be used as is, and its extra fields from the containing object will go unused.

5.6.2 Method dispatch

We must next make sure that calling code calls through our dispatch table (created in the Cake code with an instantiate helper invocation, not shown). In the GObject library, the dispatch table is located by a complex sequence of steps—by library calls which access

```
// instantiate the interface dispatch tables we implement
derive elf_archive ("ephy+.a") ephy = instantiate(
        instantiate ( elf_archive ("ephy.a"), EphyCommandManagerIface, man_impl, ""),
                EphyEmbedIface, embed_impl, "");
// ...
/* within ephy ⟷ webkit link sub-block ... */
// globally, use these tables for these interfaces
g_type_interface_peek(_, EPHY_TYPE_EMBED) (let impl = ephy_impl)⟶ { impl };
g_type_interface_peek(_, EPHY_TYPE_COMMAND_MANAGER) (let impl = man_impl)⟶ { impl };
```

Figure 5.17: Interposing new dispatch tables in Epiphany

a GClass meta-object, in turn containing pointers to dispatch tables of all implemented interfaces. Constructing an alternative such object within Cake would mean replicating knowledge of these GClass objects' semantic constraints, so that we could construct one respecting the invariants required by other GObject code. For example, each GClass has a type identifier number issued by the GObject library, according to certain rules which are encapsulated in C preprocessor macros.

So, although we would like to define a new GClass around our dispatch tables in Cake, this would mean replicating knowledge of these rules in the Cake file. This would be fragile, since these rules are regarded as implementation details of the GObject library. The usual way for a C programmer to define a new GClass is by C preprocessor macros which hide change-prone implementation details. Since these macros are not usable from Cake, we lack an equivalently robust solution. Chapter 6 identifies this as a *stylistic* issue which future versions of Cake could tackle in a library of GObject-specific rules. For now, we handle dispatch more straightforwardly by interposing on retrieval of the dispatch table, access to which happens conveniently to be functionally abstracted (Fig. 5.17). These rules are limited to cases such as ours where only one implementation of the named interfaces is used within the whole program.[7]

5.6.3 Minor benefits

Simple rules Many of the calls between the two interfaces map very directly, as shown in Fig. 5.18. Some are left unimplemented by Epiphany; these are mapped to empty stubs in Cake.

Pattern-matching succinctness Pattern-matching on event correspondences simplifies the load and manager_do_command functions.

Lack of boilerplate Another small benefit in the Cake implementation is a relative lack of boilerplate. Epiphany's use of the GObject library necessitates somewhat verbose C code to perform downcasts and populate a dispatch table. By contrast, Cake can

[7]On the other hand, this limitation could be removed by adding context predication to the g_type_interface_peek() rules, or by defining a GClass object specific to our association.

```
ephy_load(embed, url as raw_url, flags , preview_embed)
  ⟶ { set embed...loading_url = url;
        webkit_web_view_open(embed...web_view, url); };

ephy_stop_load(embed) ⟶
  webkit_web_view_stop_loading(embed...web_view);
ephy_can_go_back(embed) ⟶
  webkit_web_view_can_go_back(embed...web_view);
ephy_can_go_forward(embed) ⟶
  webkit_web_view_can_go_forward(embed...web_view);
ephy_can_go_up(embed) ⟶{ false };
```

Figure 5.18: Simple rules mapping from Epiphany to Webkit calls

succinctly instantiate the table using instantiate. Furthermore, since associations are formed dynamically and navigated using run-time metadata, downcasts are unnecessary. This reflects the usual trade-off between static-typed and dynamic languages, albeit in a fashion exaggerated somewhat by the verbosity and C-induced clumsiness of the GObject programming style.

5.6.4 Handling the history list

The two components exchange history item objects by copying a doubly-linked list from the HTML widget (which keeps a local history for the current navigation session) and the host browser (which copies entries into its persistent history key–value store). Mostly-automatic handling of this list-passing is a key benefit of the Cake implementation, drawing on Cake's object graph exploration. We now compare the C code (Fig. 5.19) and Cake rules (Fig. 5.20) for these two cases.

Utility code and boilerplate Note an unfortunate naming choice in the Epiphany codebase: WebKitHistoryItem is a data type defined by Epiphany code rather than by WebKit. WebKit's own history item data type is WebKitWebHistoryItem (note the extra "web" fragment). Epiphany also defines a common interface EphyHistoryItem, of which WebKitHistoryItem is an implementation. Strictly speaking, WebKitHistoryItem is part of the adaptation logic we are aiming to replace with Cake code, rather than being core Epiphany code. We treat it as preexisting utility code for the purposes of this study, for the same reasons that explained our handling of EphyBaseEmbed instantiation: Cake does not currently provide a good way of replicating GObject boilerplate. In this case, the utility code is almost entirely boilerplate—the implementation contains a pair of trivial getter functions only. Aside from the boilerplate, replicating this logic within Cake would clearly not be problematic, since it amounts to a pair of very simple function correspondences. It is therefore reasonable to proceed with this study by treating this code as shared utility code. Of course, we do not count this code when measuring the volume of C code replaced by our Cake rules.

```
/* enumeration for distinguishing a "direction" of history */
typedef enum
{
  WEBKIT_HISTORY_BACKWARD,
  WEBKIT_HISTORY_FORWARD
} WebKitHistoryType;

/* utility function for doing the list-copying */
static GList*
webkit_construct_history_list (WebKitEmbed *embed, WebKitHistoryType hist_type)
{
  WebKitWebBackForwardList *web_back_forward_list;
  GList *webkit_items, *iter, *ephy_items = NULL;

  g_return_val_if_fail (WEBKIT_IS_EMBED (embed), NULL);

  web_back_forward_list = webkit_web_view_get_back_forward_list (embed↪→priv↪→web_view);

  if (hist_type == WEBKIT_HISTORY_FORWARD)
    webkit_items = webkit_web_back_forward_list_get_forward_list_with_limit (web_back_forward_list,
                                                                            WEBKIT_BACK_FORWARD_LIMIT);
  else
    webkit_items = webkit_web_back_forward_list_get_back_list_with_limit (web_back_forward_list,
                                                                         WEBKIT_BACK_FORWARD_LIMIT);

  for (iter = webkit_items; iter != NULL; iter = iter↪→next) {
    EphyHistoryItem *item = webkit_history_item_new (WEBKIT_WEB_HISTORY_ITEM (iter↪→data));
    if (item)
      ephy_items = g_list_prepend (ephy_items, item);
  }

  g_list_free (webkit_items);

  return ephy_items;
}

/* get the "backward" piece of the history list */
static GList*
impl_get_backward_history (EphyEmbed *embed)
{
  return webkit_construct_history_list (WEBKIT_EMBED (embed),
                                        WEBKIT_HISTORY_BACKWARD);
}
/* get the "forward" piece of the history list */
static GList*
impl_get_forward_history (EphyEmbed *embed)
{
  return webkit_construct_history_list (WEBKIT_EMBED (embed),
                                        WEBKIT_HISTORY_FORWARD);

}
```

Figure 5.19: C code for transferring WebKit history items to Epiphany

```
values
{
        /* history item handling */
        GList_of_EphyHistoryItem ⟷GList_of_WebKitWebHistoryItem
           {
                data as WebKitHistoryItem ptr ⟷ data as WebKitWebHistoryItem ptr;
        };
        // update rule exploiting underlying shared representation
        WebKitHistoryItem (this↪data)⟶ WebKitWebHistoryItem;
        // init rule only: we never send updates to webkit
        WebKitHistoryItem (*(webkit_history_item_new(that) tie this))⟵? WebKitWebHistoryItem;
}
ephy_get_backward_history(embed) ⟶{
        let full_bf_list = webkit_web_view_get_back_forward_list(embed...web_view);
        webkit_web_back_forward_list_get_back_list_with_limits(
                full_bf_list ,
                WEBKIT_BACK_FORWARD_LIMIT) as GList_of_WebKitWebHistoryItem
};
ephy_get_forward_history(embed) ⟶{
        let full_bf_list = webkit_web_view_get_back_forward_list(embed...web_view);
        webkit_web_back_forward_list_get_forward_list_with_limits(
                full_bf_list ,
                WEBKIT_BACK_FORWARD_LIMIT) as GList_of_WebKitWebHistoryItem
};
```

Figure 5.20: Cake rules for transferring WebKit history items to Epiphany

Pointer annotation Note that in both cases the lists are passed as GList objects, but with different payload types. We rely on explicit specialisation of the void pointers in each GList node. This is done using the artificial data types GList_of_EphyHistoryItem and GList_of_WebKitWebHistoryItem. This explicit specialisation is necessary for two reasons: firstly to allow our runtime to follow pointers from list node to payload object (§4.3.2) and secondly to allow our object-sharing analysis (§4.4.4) to correctly infer that the list nodes are not shareable (because following analogous paths from the two components' nodes can reach incompatible data types). If the specialisation were omitted, the list nodes would be erroneously shared, the list recipient would perform an invalid downcast on the void pointer, and meaningless type-incorrect data would be read.

C implementation We see the webkit_construct_history_list helper calls two functions: webkit_web_view_get_back_forward_list to retrieve a complete history list in Webkit form, and then either webkit_web_back_forward_list_get_forward_list_with_limits or webkit_web_back_forward_list_get_backward_list_with_limits to produce a copy of a bounded subset of the list as a GList of WebKitWebHistoryItems. A loop copies the elements of second list into a new list of WebKitHistoryItems which is freed by the caller subsequently. The earlier GList of WebKitWebHistoryItems is freed in the wrapper.

Cake contrast In the Cake code there are three notable absences: no loop, no null test, and no list-freeing. The former is because the Cake runtime explores the list automatically.

	C	adjusted	Cake	remaining C	%age
LoC (nb nc)	525	513	172	0	33%
tokens	2525	2455	958	0	39%
semicolons	175	163	74	0	45%

Table 5.5: Comparison of ephy–webkit in Cake and C.

The latter is explained by the Cake runtime's treatment of co-objects: when allocating new object structures in applying a value correspondence, the newly created objects are recorded as co-objects of the originating objects. When *either* of these co-objects is freed, the Cake runtime frees the other. So it suffices for the client to free the objects it is passed—which it must, or else there is a memory leak in the client code—and the Cake runtime will ensure that the intermediate list is also freed.

Null test difficulty The lack of null test is a simplification enabled by the fact that webkit_history_item_new() returns null only for a null argument, which can occur only when the input GList node has a null payload pointer. This reveals a shortfall in expressiveness of Cake—we cannot express the fact that null-payload nodes should be omitted without breaking the list structure. We could attempt to correspond a skipped node with void ("no value"), but this breaks the list, instead of continuing it with "the next node with non-null payload"—which is the desired behaviour, but currently inexpressible. (This is an instance of the "context-impoverished" limitation identified in §4.4.5.) The importance of this in our particular study is moot, since in our experience, WebKit does not return lists containing null-payload nodes. We therefore work around the problem by assuming that null-payload nodes do not arise; for fairness, we comment out the null test in the C code so that it does not appear in the code measurements.

Object lifetime subtleties The "free one frees all" semantics of co-objects (§4.4.2) clearly presumes that there is no mismatch in the lifetimes of objects across the mismatched interface. This is sufficient in our case, but does entail one case where an object is freed less eagerly in our implementation than in the C version. Specifically, note the call to g_list_free() in Fig. 5.19, which frees the list returned by the Webkit call. In the Cake implementation, the freeing of this list is delayed until the *corresponding* list is freed (within the receiving Epiphany code).

This case study proves a fair demonstration of Cake. Table 5.5 shows the summary measurements, showing that the Cake implementation is again 30–45% the size of the C version after the usual adjustments.

5.6.5 API influence and anticipation

Arguably, this case study is not a pure case of unanticipated composition, in the sense that it is aided by some amount of both *anticipation* and *common ancestry* in the design of the interfaces involved.

Anticipation is evident in the fact that Epiphany contained the EphyEmbed abstraction layer. This abstraction layer predates the introduction of alternative rendering engines by some time. Indeed, it features in the earliest revision that remains publicly available at the time of writing, from December 2002—over four years before the Webkit renderer was first supported (in July 2007). No other renderers than the Mozilla one were supported during this interval. Clearly, the authors anticipated some future need to support alternative renderers.

This need not have made the Webkit composition task any easier, if the Webkit interface turned out to be wildly different from that chosen for the abstraction layer. However, this proved not to be the case. Although nontrivial quantities of code were required to implement the abstraction layer over Webkit, there were also some suspiciously convenient properties about the interface pairing, as revealed in Fig. 5.18. Why are the function names in the EphyEmbed and Webkit interfaces so often similar, when supposedly the former existed long before the latter? In particular, the suffixes _can_go_back and _can_go_forward are shared by calls in the two interfaces. This seems unlikely to be a coincidence.

The explanation is a different phenomenon concerning *common ancestry* or, more generally, *influence* in interface design. The first popular embeddable browser component for X11 applications was the GtkMozEmbed component from the Mozilla project. This component is the one originally targeted by Epiphany's abstraction layer. Later, when the Gtk+ APIs for Webkit were designed, their developers chose to make the interface of WebKitWebView resemble that of the established GtkMozEmbed in several ways. Of the eighteen documented non-boilerplate methods defined by a GtkMozEmbed widget,[8] six are present in essentially identical form in the WebKitWebView interface, and another three with slightly different names. Of these nine functions, all but one are used in the adapter logic (in both Cake and original versions), suggesting that the GtkMozEmbed influence has had some significant impact on simplifying the task.

The p2k example from the previous section also exhibits this influence phenomenon. The original Unix VFS interface [Kleiman 1986] has been hugely influential in both kernel-level filesystem interfaces (such as rump) and user-space interface (such as puffs, or Linux's fuse[9]). Even when there is no apparent ancestry shared by two interfaces, cultural factors can lead to similarities. As will motivate the extensions presented in Chapter 6, certain interface styles are favoured by certain communities of developers with a shared familiarity with particular tools, particular languages and particular well-known interfaces. Case studies encompassing deep mismatches of style would be less likely to benefit from this kind of influence. This reinforces our belief that styles merit considerable attention in future work.

[8]The relevant documentation was retrieved from http://www-archive.mozilla.org/unix/gtk-embedding.html on 2012/5/1.

[9]http://fuse.sourceforge.net/

5.7 Evolving interfaces in distributed systems: XCL

Codebases in long-lived distributed systems accumulate complexity over time. Occasionally developers choose to redesign the client interfaces to shed this complexity and better serve contemporary needs. Such an initiative began in the X Window System [Scheifler and Gettys 1986] around 2003, when a new client library XCB was proposed to replace the original Xlib. For clients of Xlib, an adaptation layer called XCL [Sharp and Massey 2002] was devised. We took a small but representative subset of the XCL source code (around 600 raw lines out of 6000) and reimplemented it using Cake.

Since XCB is designed to be more minimal than Xlib, there is a small abstraction gap between the two. As a result, some utility code from XCL whose purpose was to bridge that gap was retained unmodified for use with our Cake implementation. Meanwhile, many data structures are shared verbatim between Xlib and XCL, so there was only limited opportunity to exploit the expressiveness of value correspondences.

5.7.1 Main pattern rules

The main adaptation requirement between the two interfaces is summarised by a Cake pattern rule which matches a large number of Xlib calls (Fig. 5.21). Unlike Xlib, XCB provides an asynchronous interface, where a request call yields a handle which is fed to a reply call, returning a caller-freed reply object. The naming conventions in these reply objects match mostly, but not exactly, the output parameters in the Xlib calls. Two pattern rules capture these: an event pattern rule captures the family of calls, and a value pattern rules captures the family of data structures by providing the necessary name mappings.

Explicit freeing The stub pattern in Fig. 5.21 includes an explicit free() of the reply object. We might ask whether a Cake caller_free annotation could allow the compiler to infer this call. However, unlike in the example in §2.3.7, there is no single caller-provided storage area matched to the callee-output value in the event pattern. Rather, the returned object is explicitly received by the stub. Consequently, its address has a chance to escape to other code; since our compiler does not perform escape analysis, it cannot ensure that the reply object need not remain alive after the call, so cannot infer the free() operation.

Special cases Nearly all Xlib calls take a Display pointer as first argument. In the lower-level XCB API, the display abstraction is replaced by a XCBConnection object in most cases, but a utility call allows retrieving a connection from a display object. Fig. 5.22 shows these rules. We use the common Cake idiom of introducing an artificial data type to add special-case treatment for some calls—a few XCB calls require the display to be locked before proceeding, and unlocked on return.

Stronger and weaker data abstraction Xlib defines a series of simple value types as C typedefs of primitive data types. Meanwhile, in XCB these are more strongly abstracted

```
pattern /X(.*)/(dpy, ...)  ⟶
            { let request = XCB\\1(dpy, in_args ... ) ;&
              let reply = XCB\\1Reply(dpy, request) ;&
              { out out_args... = *reply;
                free(r); true }
              ;| false;
            };

values _ ⟵ pattern /XCB(.*)Rep/ // Cake compiler will define an Xlib-side struct...
  // ... with an arbitrary name
{
  // Xlib calls sometimes have a "root" Window output parameter...
  // typedef CARD32 XID; typedef XID Window;
  // ... while in XCB..Rep structs, it's a WINDOW
  // typedef struct WINDOW { CARD32 xid; } WINDOW;
  root ⟵ root.xid ;

  // similar
  child ⟵ child.xid ;

  // identifier styles differ
  borderWidth ⟵ border_width;
};
```

Figure 5.21: Basic pattern-based correspondences in XCL

into simple structures, forcing the C programmer to make explicit any non-opaque treatment of the encapsulated value. A series of trivial value correspondences relates these.

Annotations To use the special-case value correspondences for Display, we annotate the interfaces with the artificial data type names introduced by the correspondences, as shown in Fig. 5.23.

String handling Fig. 5.23 also shows a new data type, encap_string, being declared among the annotations. This is useful because while Xlib uses the native C treatment of strings, simply as pointers to null-terminated character arrays, some XCB calls use the closer-to-protocol form of length-prefixed arrays. The encap_string definition allows a simple value correspondence to convert between these, as shown in Fig. 5.24.

Error reporting A recurring feature of stubs in the Cake code for XCL is the affixing of a true expression at the end. Recall from our discussion of Cake's default style (§2.3.3) that integer-returning calls as treated as successful iff they return zero. In Xlib, many calls are integer-returning but encode success as TRUE (one) and failure as FALSE (zero). We therefore must explicitly give the stub's output value rather than synthesising it from the success of the previous call. This, together with the string handling and asynchronous dispatch styles seen previously in this case-study, suggest that a more appropriate way to abstract some of the XCL composition task would be to capture the styles of each

```
// common cases: Display to XCB connection, with and without locking
values Display           ⟶(*XCBConnectionOfDisplay(that)) XCBConnection;
values Display_unlocked ⟶({LockDisplay(that);
                            XCBConnectionOfDisplay(that)}) XCBConnection;
values Display_unlocked ⟵({UnlockDisplay(that); void}) XCBConnection;

// rarer cases
values Display_unlocked ⟶({LockDisplay(that); that}) Display_locked;
values Display_unlocked ⟵({UnlockDisplay(that); that}) Display_locked;

// typedefs become structs
values pattern /Window|Pixmap|Cursor|Font|GContext|Colormap|Atom/
        ⟷ \\U\\1\\E // GNU sed-style pattern rewrite...
        { *this ⟷ xid };
values pattern /VisualID|Keysym|Keycode/ ⟷\\U\\1\\E // .. same here
        { *this ⟷ id };
values Time ⟷ TIMESTAMP
        { *this ⟶ id };
values CARD8 ⟷BUTTON
        { *this ⟷ id };
values Drawable ⟷ DRAWABLE
        { *this ⟷ font.xid };
values Fontable ⟷ FONTABLE
        { *this ⟷ window.xid };
```

Figure 5.22: Value correspondences between Xlib and XCB

interface separately from the individual pairwise correspondences. We revisit this idea in Chapter 6.

Cross-rule commonality This study exposed another flaw with the current Cake language: it has no means to factor out cross-rule commonality which cannot be captured using value correspondences. In XCL there is some such commonality. For example, several Xlib calls for setting window properties map to the XCBChangeProperty call, which takes many arguments. In XCL, there is an XSetProperty function which abstracts away most of these arguments, and series of other Xlib calls are implemented using this function. In Cake we were forced to implement each as a verbose call to XCBChangeProperty instead, meaning the Cake version required slightly more code than the C version, as shown in Fig. 5.25.

Table 5.6 shows the aggregate comparison of the original implementation and Cake's. The savings on code size are somewhat less in this study than in the previous two, with overall code size around 70–80% of the original size. This is explained not by a single large failing, but by the sum of small factors identified so far: the abstraction gap, the asynchronous style of dispatch in the XCB interface, the fact that Xlib's return conventions do not match Cake's default style, and the failure to capture cross-rule commonality. There is also a fairly large number of value correspondences required up-front; had we had the resources to implement the whole of XCL in Cake, we might expect this effort to be amortised over a larger quantity of code.

```
exists elf_archive (" rxvt.a") client_of_xlib
{
    declare {
        XFillRectangle : (dpy: Display_unlocked ptr, ...) ⇒ _;
        XGetGeometry: (dpy: inout Display, d: Drawable, root: Window ptr,
            x: out int , y: out int , width: out unsigned, height: out unsigned,
            borderWidth: out unsigned, depth: out unsigned) ⇒ _;
        XTranslateCoordinates: (..., child : out Window) ⇒same_screen : _; // named return value!
        XQueryTree: (_, _, root: out Window, parent: out Window,
            children : out Window[nchildren], out nchildren ) ⇒ _;
    }
};

exists elf_archive (" libxcb.a ") xcb_library ;
exists elf_archive (" xcl_util.a ") xcl_util
{
    declare {
        _XFlushGCCache: (dpy: Display_locked ptr, ...) ⇒ _;
        XCBPolyFillRectangle: (dpy: Display_locked ptr, ...) ⇒ _;
        XCBPolyRectangle: (dpy: Display_locked ptr, ...) ⇒ _;
        encap_string: class_of struct {
            len : size_t ;
            bytes : char ptr;
        };
    }
};
alias any [xcb_library , xcl_util ] xcb;
```

Figure 5.23: Annotating interfaces to use artificial data types' value correspondences

```
// implicitly , the LHS encap_string is a char[] ptr
// ... and the RHS is a structure defined in the previous figure
encap_string ⟶ encap_string
{
    void ⟶ ( if ∗that then strlen (that) else 0) len ;
    void ⟶ (that) bytes ;
};

// specific event correspondences invoke the length-prefixed treatment
XLoadFont(dpy, name as encap_string) ⟶
                                        { let f = XCBFONTNew(dpy);
                                          XCBOpenFont(dpy, f, name.length, name.bytes);
                                          f.xid };
XStoreName(dpy, w, name as encap_string) ⟶
                                        { XCBChangeProperty(dpy, PropModeReplace,
                                              w, XA_WM_NAME, XA_STRING, 8,
                                              name.length, name.bytes);
                                          true };
```

Figure 5.24: String handling

```
// longhand in Cake, repeating the XCBChangeProperty call
XSetWMName(dpy, w, tp) —→XCBChangeProperty(
        dpy, PropModeReplace,
        w, XA_WM_NAME, tp↪→encoding,
        tp↪→format, tp↪→nitems, tp↪→value);


// shorthand in C, using XSetTextProperty convenience
void XSetWMName(Display *dpy, Window w, XTextProperty *tp)
{ XSetTextProperty(dpy, w, tp, XA_WM_NAME); }
```

Figure 5.25: Cross-rule commonality repeated in Cake yet captured in C

	C	adjusted	Cake	remaining C	%age
LoC (nb nc)	368	303	182	42	69%
tokens	2501	2264	1401	232	68%
semicolons	181	142	100	19	81%

Table 5.6: Comparison of an XCL subset in Cake and C.

Full versions of all the Cake code discussed in this Chapter may be found in Appendix D.

5.8 Relation to thesis statement

The thesis of this dissertation claims "using a special-purpose language, based on *relations*, to compose heterogeneous mismatched software components, is significantly more effective in practice than conventional programming languages". With the exception of the word "heterogeneous", this chapter (as a culmination of those preceding it) has substantiated this statement. The "significance" of our results rests on the fact that in the worst case, a 25% reduction of code size, and in most, 60% or more was achieved. This scale of reduction is clearly significant, in that it could not be achieved by minor syntactic variations on conventional languages. To further substantiate this, our detailed discussion of each task has related these code-size improvements specifically to the language's design elements as described in previous chapters. For the cases where the Cake language happened not to enable a desired benefit, we have highlighted the responsible root shortcomings of the current Cake , and in most cases described incremental improvements which would allow a language very similar to Cake to overcome these limitations.

The one aspect of the thesis statement which remains unsubstantiated is the claim that Cake can compose *heterogeneous* components. This is the subject of the next chapter.

5.9 Closing remarks

Any language design raises a host of questions about the various properties of the language and its programs. Aside from questions relating to the thesis statement, the experiences presented in this chapter have raised several other issues deserving consideration. This section briefly notes a selection of such questions and suggests how each of them might be resolved.

Performance Achievable performance using Cake depends greatly on the "cut" of the interfaces being composed. We have several reasons to believe that Cake's generated code can be acceptably efficient in many cases. It is often structurally similar to hand-written code (particularly the `p2k` study). Link-time optimisations can be applied after Cake has done its work. The relatively slow uptake of link-time optimisation suggests that cross-library calls are rarely performance-critical (cf. intra-library calls). Finally, as previous chapters have described, there is huge scope for adding further annotations and analysis to allow generation of faster code. Similarly, the previous chapter has described the potential for reducing the overhead imposed by Cake's runtime (such as that of heap instrumentation described in §4.3.2).

Applicability and scale Cake's range of applicability can only be discovered in longer-term studies. However, its underlying model is highly general and certainly not limited to the kind of single-process procedural interactions with which it has been demonstrated so far. For example, Cake's design might apply particularly well to distributed message-passing systems. (This is a particularly interesting possibility, since in systems where storage is naturally replicated more than it is shared, the performance penalties and object-sharing complications detailed in §4.4.4 do not arise.) Regarding the scale of target codebases, we note that our evaluation case studies are relatively small. However, one would expect interface size or "surface area" to grow sublinearly with both component size and program size ("volume"). Again, deeper and longer-term studies will be required to establish this.

Binaries and styles One usability issue not accounted for in our measurements is that the programmer must understand two versions of their interface: source-level and binary-level. With C code (the "default style" target) these two views are usually very similar, although they can be obscured by use of the preprocessor (e.g. to redirect function calls or modify their arguments). The problem becomes more significant when targetting components written in higher-level languages, as the gap between language level and object code level widens. The work presented in Chapter 6 will partially address this problem.

Chapter 6

Extending Cake with component styles

In previous chapters we have described the core Cake language and shown it to be effective with respect to a sample of real programming tasks. However, both the language design and the selection of tasks were somewhat narrow. Cake's "default style" (§2.3.3) limited many of the interfaces to those typical of C code. Meanwhile, this limitation was reflected in the selection of tasks which were addressed in Chapter 5.

In this Chapter, our main contribution is identifying a class of diversity in software component interfaces which we call *stylistic variation*. We explain how it encompasses the diversity observed between binary components that are *heterogeneous* with respect to the languages, tools and coding styles used to produce them, and how it is useful to capture this within a composition tool. We then consider the requirements for extending Cake so that it can abstract away this diversity, and sketch some modest and conceptually straightforward extensions which achieve this.

6.1 Introduction

As described so far, the Cake language describes correspondence relations between concrete, existing component interfaces. However, there are other useful approaches to abstracting composition tasks. Cross-cutting similarities occur across large numbers of component interfaces, even when these are dissimilar in their function. We will call this phenomenon *stylistic variation*, and explore examples shortly. Stylistic variation presents an opportunity to amortise certain kinds of Cake programming effort over large numbers of composition tasks, by capturing styles independently of composition context. Our goal is to allow stylistic diversity to be abstracted away, so that the programmer can focus on composition-specific issues.

Object code has the benefit of providing a unifying view of many target components. However, since this view is relatively low-level, different language implementations may choose many different ways of encoding the same higher-level meanings within object code. Therefore, styles must capture not only interface conventions chosen by the programmer, but also conventions chosen by tool implementors.

```
struct wc; // implemented in C

// struct is treated opaquely by client

struct wc *word_counter_new(const char *filename);
  // returns NULL and sets errno on error
  // in Cake, annotate caller_free (word_counter_free)

int word_counter_get_words(struct wc *obj);
int word_counter_get_characters(struct wc *obj);
int word_counter_get_lines(struct wc *obj);
int word_counter_get_all(struct wc *obj,
  int *words_out, int *characters_out, int *lines_out );

void word_counter_free(struct wc* obj);
```

```
class WordCounter // implemented in Java
{
  /* fields not shown... */

  public WordCounter(String filename)
    throws IOException { /* ... */ }


  public int getWords() { /* ... */ }
  public int getCharacters () { /* ... */ }
  public int getLines () { /* ... */ }
  public Tuple<int, int , int> getAll () { /* ... */ }

};
// implicitly , deallocation is done by unreferencing + GC
```

Figure 6.1: Two stylistic variants of the same component

6.2 Motivation and simple examples

We begin with a very simple artificial example, then progress to more realistic scenarios.

6.2.1 Simple example

Suppose two programmers independently develop a simple component for counting the lines, words and characters in a file; Fig. 6.1 shows what they might write.

It is quite clear that these components are functionally identical. However, they are somewhat different superficially. Output parameters have been encoded differently, as have character strings. One component provides an explicit resource management API, implicitly also handling initialization and finalization, whereas the other provides only an explicit initialization API. Naming conventions for multi-word identifiers also differ. Moreover, the components are written in different languages, so compilation will introduce further differences. Calls to the Java component will use virtual function dispatch and exception handling, while C code will not.

These conventions are not invented anew by each programmer. Rather, they are imported from a wider shared repertoire, perhaps defined by a language, a toolchain, or simply a coding style. If we can capture these conventions in a one-time effort for each style, hence abstracting away these recurring differences, we can avoid repeated effort by users of Cake (and similar tools) in carrying out successive programming tasks.

In more realistic examples, there will be not only stylistic differences, but also differences in how each programmer has modelled the domain. These are precisely what the usual kind of Cake rule—which we call "bilateral rules"—is suited for. Fig. 6.2 shows the relationship between styles and a more familiar application of Cake. Styles may be thought of as "views" or "lenses" which abstract interfaces "vertically", recovering a more abstract interface from a more concrete one. Bilateral Cake rules can then be written "horizontally" *at the more abstract level.* In rare cases such as Fig. 6.1, where there are *only* stylistic differences, then at the abstract level, the components will link simply by name-matching and no bilateral rules will be necessary.

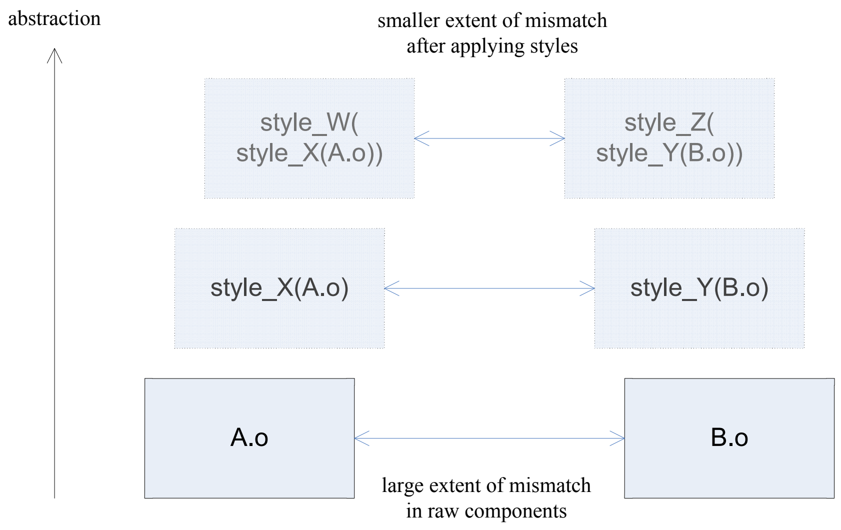

abstraction
smaller extent of mismatch after applying styles
style_W(style_X(A.o))
style_Z(style_Y(B.o))
style_X(A.o)
style_Y(B.o)
A.o
B.o
large extent of mismatch in raw components

Figure 6.2: Styles as interpretations underlying familiar Cake coding

6.2.2 Examples from earlier chapters

Earlier chapters in this dissertation have already provided real examples motivating awareness of styles within Cake and similar tools. In Chapter 2 we identified several classes of convention which vary from component to component, including error reporting conventions (§2.3.4), output parameters (§2.3.7), and representations of common abstractions such as lists, strings and sets (§2.3.8). In Chapter 5 we noted some areas where Cake's "default style"—which attempts to capture the common-case conventions found in many C APIs—did not optimally abstract the composition tasks presented by the interfaces concerned. Specifically, in the Epiphany study (§5.6), we imported some utility code for dealing with history lists, because re-creating the GObject boilerplate could not be done conveniently or robustly within Cake. Meanwhile, in the XCL study (§5.7) we found that error-reporting conventions did not match Cake's default style (in that they returned nonzero on success), while the two interfaces used two different but commonplace string representations (length-prefixed versus null-terminated).

6.3 Dimensions of stylistic variation

The usefulness of styles derives from their recurrence across a large population of components. What interface conventions recur in this way? By its nature, this question can be answered only empirically.

To this author's knowledge, there are no existing studies on stylistic variation as a whole, although partial surveys of variation within a particular concern have sometimes emerged from other work relating to that concern [Lamb 1990; Ellis et al. 2007] but motivated by other factors than composition.

In the absence of a formal study, we must rely on experience gathered in previous chapters and general programming experience to obtain an overview of this space of interface conventions. Table 6.1 presents a catalogue of stylistic concerns gathered in this way. Note that our catalogue need not be exhaustive; the Cake extensions we will develop are for *user-defined* styles, using the list as a guide, but not limited to styles chosen from this list specifically.

6.3.1 Properties of interest

Table 6.1 deserves a detailed explanation. We consider each of the table's columns in turn.

Abstract concern This column identifies a shared intention underlying a set of alternative interface conventions. These concerns are the root of our interest in styles: if there were only one way for a given abstract concern to be realised concretely, then there would be no need for composition tools like Cake to support multiple styles.

Sample concretion approaches These describe broad equivalence classes of approaches. Within each class, the differences between each approach are relatively superficial. Each group member might therefore be considered a particular parameterisation of one overarching logical style. Note also that these concretion approaches are not mutually exclusive; some styles will combine many of these at the same time. For example, many C library calls report errors by some combination of the return value, the `errno` global and an error discovery function like `ferror()`. We will consider both composition and parameterisation of styles in due course.

Concrete examples and reference This columns lists real APIs, well-known programming idioms or documented tool implementations instantiating a given concretion approach. Since we are implicitly interested in conventions appearing at the object code level, notice how these concretions descend right down to the binary level, and are a mixture of tool- and programmer-selected conventions. For most examples, a bibliographic or section reference to further details is provided.

Classification We label each stylistic concern by a subset of *mechanism*, DWARF, *Platonic*, *dataflow*, *structural* and *runtime*. These will be explained inline as we walk through the rows of the table, and are highlighted in bold type when first introduced.

6.3.2 Abstract concerns

Most rows of the table are self-explanatory, but we add the following notes on selected rows.

Procedure call Local procedure calls may be implemented in a variety of ways using a mixture of registers, the stack and static storage. The **mechanism** tag remarks that these lie below Cake's baseline abstraction level; the Cake language has no means to describe them (§3.4). However, a Cake implementation may support them using mechanism-specific knowledge. *Remote* procedure calls are usually rendered as interprocess communication system calls, with arguments and return values encoded in particular ways. Subsequent rows cover these argument encodings.

Error reporting Certain data values indicate errors; we may need to gather them from diverse sources (return values, error discovery functions, etc.), and decode them using various conventions. The tag **Platonic** notes that since error return is a concept built-in to the Cake stub language, Cake defines a standardised abstract interface which any style definition addressing this concern should satisfy. The "mechanism" tag also applies, specifically to exception handling, because this is generally implemented by intricate stack manipulations which, like calling conventions, lie below Cake's level of abstraction. The **data flow** tag denotes that error conventions are a concern which can be directly observed in the trace of data flowing across an interface; we explain the significance of this at the end of this section.

Name mangling and word-separating The Cake compiler performs name-matching. Different ways of encoding the same names therefore represent a stylistic concern which Cake should abstract. Often, demangled names are available in debugging information, so do not require separate user-provided description. However, ad-hoc naming and word-separating conventions could usefully be interpreted in the Cake compiler, much like pattern-based Cake rules (§2.2.11), allowing more correspondences to be formed automatically.

Unstructured and composite values Encodings of unstructured and structured data are described by DWARF, and Cake inherits an understanding of these. However, not all stylistic alternatives are supported: for example, string-based encodings of values would require extending DWARF with some sort of grammar-based description of variable-length encodings. Dimensions which would require DWARF extensions to be captured within Cake are tagged with **Dwarf** in the Classification column. We use "functionally abstracted" to denote interfaces, like getter and setter interfaces in this example, which hide access to state behind function calls; it need not indicate a *functional programming* style per se.

Call demultiplexing Polymorphic code [Strachey 1967] invariably embodies a concretion process in which an abstract operation is mapped to one of many concrete implementations. We call this process "demultiplexing". Examples include static overload resolution [Stroustrup 1997], virtual function calls (demuxed by an indirect table look-up), and other, more flexible dispatch processes [Lindholm and Yellin 1999; Alpern et al. 2001; DeMichiel and Gabriel 1987; Chambers 1992; Clifton et al. 2000]. User-level solutions also exist, such as a simple `switch` statement (e.g. on a "type" or similar field, as

Abstract concern	**Sample concretion approaches**	**Concrete examples**	**Reference**	**Classification**
procedure call	local; pass on stack and registers	C calling convention; Pascal; stdcall, fastcall,	Solomon 98	mechanism
	remote or out-of-process: pass through IPC	Sun RPC; other RPC protocol; Unix command by fork-exec	Birrell 84	data flow
error output	thrown exception	exceptions in C++, Java, ...	Stroustrup 97	dataflow, Platonic
	return value is error flag	return true for success, false for error	§5.4	
	use subspace of return value	return null pointer, negative integer, etc		
	error callback or jump	(various APIs); longjmp() to handler	Kernighan 88	+mechanism
	set global	C library errno	Kernighan 88	
	error discovery function	C library ferror()	Kernighan 88	
input parameters	immediate	parameters on stack (most language impl'ns)	SCO 97	dataflow, DWARF, Platonic
	contained	packed into tuple (ML idiom), array (Java varargs) or other structure	Milner 90	
	indirected	pass by reference (C++, ...)	Stroustrup 97	
	lazily by passed thunk	call by need (Haskell, ...)	Jones 03	
output parameters	single return value	direct register- or stack-based return (C, ...)	SCO 97	dataflow, DWARF
	return packed structure (tuple, array, etc.)	tuple return (e.g. ML), structured value return (C, ...), array return (Java, ...)		
	write to locations passed as input	most C APIs' output parameter e.g. libc's asprintf(), stat(),...; large struct return in some C ABIs e.g. gcc	Kernighan 88	+mechanism
	return into static storage	older C struct-return ABIs, e.g. pcc; hypothetical Fortran 77 ABIs		
	lazily by returned future (asynchronous call)	call by need (Haskell); asynchronous APIs e.g. XCL	§5.4	
name mangling	no mangling	plain C code		dataflow
	ad-hoc mangling	C namespacing prefixes, e.g. in GObject-based libraries; C overload suffixes e.g. libc's pow(), powf(), powl()	Krause 07	
	systematic mangling (namespacing, overloading etc.)	Hungarian notation; C++ Ann. Ref. Man mangling conventions; GNU Java conventions	Ellis 90	
name word-separating	no separation	most C library calls	Kernighan 88	structural
	punctuation-based	underscores (most C APIs); hyphens (many Lisp APIs)		
	casing-based	Pascal casing, camel casing		
unstructured value encoding	native machine encoding	most compilation-based language implementations	SCO 97	dataflow
	alternative binary encoding	network representation e.g. in BSD sockets API; “tagged pointer” encoding in e.g. OCaml or Lisp implementations	Leroy 97	
	injection into native encoding	boolean encodings in {0, 1}; enumerations in C++ etc.	Stroustrup 97	
	string representation	Unix shell; text-based data encodings e.g. XML; text-based network protocols e.g. HTTP	Ritchie 74	
composite value encoding	concatenation or alternation of primitives locally in memory	field layout in most compiled code		dataflow
	linked structures	aggregate objects e.g. in Java; C++ virtual inheritance;	Stroustrup 97	
	ad-hoc embedding within variable-size primitive	ad-hoc tuple or list encodings in shell programming, text processing etc.; length-bounded lists implemented as arrays e.g. in C		
	systematic embedding within variable-size primitive	objects as hash tables e.g. in Python	Van Rossum 03	
	functionally abstracted (get/set)	JavaBeans and many Java/C++ APIs		
call dispatch (demux)	direct call to single target (trivial dispatch)	(any non-virtual, non-overloaded, singleton functions)		dataflow
	unary look-up in object	object directly stores pointers to target functions (e.g. Berkeley DB API) object reaches pointers to target functions by single indirection (e.g. vtable) object reaches pointers to target functions by multiple indirection (e.g. GNU Java interface tables)	Stroustrup 97	
	functionally abstracted look-up	JNI method id retrieval (unary);	Liang 99	
	manual dispatch by caller	overload selection in C++ or Java	Stroustrup 97	
call multiplexing	explicit “sub-call” argument	Unix execve() family (executed command represents sub-call) Unix's syscall(), ioctl(), fcntl() etc.	IEEE 88	dataflow
	implicitly encoded sub-call	I/O or IPC primitive encoding higher-level protocol messages, e.g. send(...) encoding a Sun RPC message	Stevens 98	
data sequence iteration	pointer increment	C array iteration	Kernighan 88	dataflow
	pointer follow	linked list iteration		
	functionally abstracted	C++ iterators; Java iterators; (many ad-hoc examples)	Stroustrup 97	
data sequence termination	stored end location	C++-style iterator end() value	Stroustrup 97	dataflow
	stored sequence length	length field in Java arrays; C++ arrays impl'd with cookie	Arnold 05	
	implicit sequence length	arrays in C	Kernighan 88	
	special delimiter value	null-terminated character strings in C	Kernighan 88	
	functionally abstracted	Java-style iterators' hasNext() call	Arnold 05	

set data type	bit-array encoding membership (for sets with enumerated domain only)	C++ I/O flags (in std::ios); (many other C, C++ and Java APIs)	Stroustrup 97	dataflow
	(instance of sequence, with arbitrary order)			
	(instance of sequence, sorted)			
	functionally abstracted (set ADT)	Python set builtin	Van Rossum	
	(unit-valued specialization of mapping data type)	Java HashSet, TreeSet etc. classes; C++ set class	Stroustrup 97	
mapping data type	open-indexed table (compact mapping from integers)	(any array-based implementation of a mapping); argv argument to main() in C programs	Kernighan 88	dataflow
	open associative table (sorted or unsorted; open-coded lookup)	directory as output by POSIX readdir()	IEEE 97	
	functionally abstracted (e.g. dictionary / hash table)	C++ map; Java HashMap, TreeMap etc.; Python dictionary builtin	Stroustrup 97	
	function (read-only mapping)	strerror() in C library	Kernighan 88	
string data type	(character-valued specialization of sequence)	Haskell strings; C strings (delimited sequence of chars)	Kernighan 88	dataflow
	purpose-crafted abstract data type or builtin	strings native to Java, Pascal, many other languages	Arnold 05	
object initialization	user implicitly responsible	(many C APIs)		dataflow, runtime, structural (because of static initialization)
	initialize to all-zeroes representation			
	ad-hoc functional abstraction, separate from memory acquisition	(many C APIs)		
	ad-hoc functional abstraction, integrated with memory acquisition	GObject-style constructors (<classname>_new())	Krause 07	
	systematic functional abstraction (generally integrated with memory acquisition)	C++ constructors, Java constructors	Stroustrup 97	
object finalization	ad-hoc functional abstraction called by user	(e.g. close()-style function in many C APIs)		dataflow, runtime, structural
	systematic functional abstraction, called by user	IDisposable pattern in Microsoft .Net libraries; SWT dispose() methods	Gunnerson 05	
	systematic functional abstraction, called by resource management handler	Java finalizers, C++ destructors	Gosling 05	
contextual initialization & finalization	ad-hoc functional abstraction called by user	("registration"- and "deregistration"-style calls in many APIs)		dataflow, runtime, structural (static again)
memory acquisition	system functional abstraction or builtin	C malloc(), C++/Java new, ...; implementation-specific calls in most purely functional languages (Haskell, pure fragment of ML, ...)	Kernighan 88	dataflow, runtime
	ad-hoc functional abstraction per ADT	GObject-style constructors (<classname>_new()) (other in-ADT allocation functions)	Krause 07	
	ad-hoc functional abstraction broader than ADT	abstract factory	Gamma 95	
memory release	explicit free using system-provided functional abstraction or builtin	C free(), C++ delete, ...	Stroustrup 97	dataflow, runtime
	explicit free using per-ADT functional abstraction	(<classname>_delete() functions supplied by many APIs)		
	implicit free by unreference (+GC)	garbage collection in Java, Lisp, ML, ...	Gosling 05	
	implicit free by explicit or tool-generated reference count management	GObject refcount management API; boost shared_ptr	Krause 07	
abstract resource management	ad-hoc functional abstraction	various APIs featuring _new() and _delete() calls, or open() and close() calls, e.g. POSIX files, semaphores, shared mem, timers, ...	IEEE 97	dataflow, structural
	systematic functional abstraction	resource acquisiton is initialization (in C++ standard library containers andmany other C++ APIs)	Stroustrup 97	
synchronisation	mutex-, semaphore- or monitor-based interfaces (varying granularity)	(conventional Java or pthreads-based code)	IEEE 97	dataflow
	transaction primitives	(code written against TinySTM or other software transactional memory API)	Harris 03	
	fork/join (no shared state)	(code written against Cilk or other fork/join API)	Joerg 96	
formatted (human-readable) object representation	ad-hoc functional abstraction	(C API with _print() functions or similar)		runtime, structural
	systematic functional abstraction	Java toString() override; C++ operator<<(std::iostream&, ...) overload	Stroustrup 97	
run-time self-description	stored pointer to description object	GObject's GClass conventions	Krause 07	runtime, structural
	tagged pointer	OCaml tagged pointer	Leroy 92	
	functionally abstracted description interface or builtin	C++ typeid Python __attrs__ dictionary	Stroustrup 97	

Table 6.1: Dimensions of stylistic variation, with examples

seen in C code) or the complex Visitor pattern [Gamma et al. 1995]. Components may be mismatched in *what* demultiplexing is performed (e.g. on what arguments), *how* (i.e. by what mechanism), *how much* (i.e. how fine-grained) and *by whom* (caller, callee or an intermediary). Many implementations yield nontrivial relationships between families of calls (e.g. "all polymorphic calls in class *C*") and the data structures used to demultiplex those calls (e.g. "the vtable for class *C*"). This explains the **structural** tag: demultiplexing concretions often rely on *static structural constraints* which cannot be expressed in familiar Cake rules.

Call multiplexing Demultiplexing's converse is where a single physical call provides access to *multiple* logically distinct operations. This multiplexing is usually simple, in that the logical call is encoded fairly directly, for example as an argument, as with Unix's ioctl() [IEEE POSIX, 1988], or else Unix's execve() where the logical operation is encoded as a command and its arguments.

Object initialization Initialization calls show great stylistic variation. This may be integrated with memory allocation, as with the C++ new operator, or not, as with C APIs reliant on malloc(). This concern is labelled **runtime** because it has a special interaction with the Cake runtime: since Cake rules require the runtime to initialize and update logical replica objects (§4.4.4), knowledge of this style can ensure that correct initialization procedures are followed for runtime-created objects (e.g. invoking a relevant C++ constructor).

Contextual initialization Object initialization often brings obligations to initialize or somehow update state *in the object's environment.* For example, many APIs require that certain objects are "registered" by calls on other objects before they are usable, in a separate step from the object's own initialization. The p2k study in Chapter 5 discovered a fragment of C code which could not be reimplemented in Cake precisely because of registration calls. Similarly, dependency constraints on initialization are common, meaning some objects must be initialized *before* some other may be created or used.

Formatted object representation Many components can generate human-readable representations of objects. These may be captured by language-defined conventions such as Java's toString() method or C++'s overloadable ¡¡ operator. They may also be ad-hoc, e.g. a print_*¡data_type¿*() function in C code.

Synchronisation Abstract requirements such as "ensure this sequence of operations appears atomic" or "wait until condition X becomes true" are communicated across many concrete interfaces. Often these use fairly simple functional abstractions (e.g. Unix's pthread_mutex_lock() and family), but occur in very complex patterns of use.

Run-time self description Many languages allow introspectively discovering descriptions (or "type information") of objects. In object code these appear as accesses to compiler-inserted fields or calls to runtime functions. In other languages, some libraries layer descriptive information into data structures, by adopting particular layout conventions and defining related functions, as in the GObject library [Krause 2007]. These bring structural constraints, since they depend on a systematic static embedding of descriptive information. They are also useful to the Cake runtime, as they provide mechanisms for discovering *object descriptions* which might be more efficient or precise than Cake can discover on its own (§4.3.2).

Abstract resource management Many programs consume abstract resources, such as file handles, network connections, mutexes, slots in a fixed-size data structure, and so on. Management of these is distinct from memory management. For instance, a garbage collector will not release them automatically without special instruction. Interface conventions for management functions show considerable stylistic variation. The Microsoft CLR [Meijer and Gough 2001] defines a special IDispose interface for objects encapsulating abstract resources; calling this correctly is the programmer's responsibility. Meanwhile, the "resource acquisition is initialization" style in C++ works by ensuring that abstract resources' lifetimes are tied to an object's.

We have seen a large variety of stylistic concerns, some apparently quite different from others. Rather than attempting to address them all here, we narrow our focus on *data flow* concerns. This is the common case: the majority of table rows have this tag, and these are concerns that most reflect the usual operation of the Cake: affecting how data is interpreted and converted as it flows between mismatched components. Styles involving mechanism, runtime-specific concerns, Dwarf extensions and structural constraints are left for future work.

Subsequent discussion of an extension to Cake is split into four parts. Firstly, we establish an understanding of the requirements on an stylistic extension to Cake, by considering the abstract nature of styles and contrasting it with existing Cake features. Secondly, we introduce some concrete syntax and simple examples of styles, considering an application to a value correspondence-like style rule. Thirdly, we use this example to discuss a process by which styles rules can be composed into familiar bilateral Cake rules with fairly little programmer guidance. Fourthly, we discuss a technique which can make a certain class of Cake rules apply bidirectionally, and consider its application to functionally abstracted stylistic concerns in the form of event correspondences.

6.4 Understanding styles

This section begins our outline of a design for an extended Cake language which captures component styles. It seems sensible that this should appear as a new kind of toplevel statement, the style definition. Beyond this, we must ask: how do styles differ from the kind of Cake code we have been writing before? How are they similar? This section presents several answers to these questions.

6.4.1 Lack of context

Recall that all Cake rules that we have seen so far occur in a link block inside a derive statement (§2.2.7). In other words, they are specified relative to a concrete composition context. By contrast, a style definition must lie outside any such context. Styles may nevertheless require certain properties of the components they apply to—for example, to require existence of particular named functions or data types.

Styles are Cake's re-use mechanism. Unlike bilateral rules, styles are not specific to a particular task, and may be re-used. Indeed, they are not useful otherwise. Like any re-usable programmatic definition, lack of context means that a style definition cannot fix all its internal details. Rather, it must be parameterised. For example, many styles are found concretely as functionally abstracted interfaces. Rather than fix on the precise names of each function in that interface, we would prefer to parameterise styles on those names, so that the names can be supplied at style instantiation time, without preventing *other* details of the abstract interface to be specified earlier in the style.

Lack of context brings a final conceptual difficulty: interface mismatch "at the Cake level". Now that developers are building libraries of styles rules, for use later in unknown contexts, there is the potential for different developers to factor stylistic concerns differently, or choose different encodings of the same abstract concerns, and later find that their styles do not compose with those of other developers. Any tool designed to capture styles should attempt to mitigate this phenomenon. The notion of *Platonic forms* partially addresses this problem (as explained in §6.7.4).

6.4.2 Styles as relations

By the time Cake sees a component, at the object code level, each concern in Table 6.1 will appear in one of its rightmost (most concrete) forms in the table. In order to *abstract away* these concrete details from an interface, we wish to recover a *leftmore*, more abstract view.

One key insight for adding this ability to Cake is that we are able to use Cake's existing linguistic abstraction, namely *interface relations*, in the form of event correspondences and value correspondences, to do so. Rather than relating two concrete interfaces, as before, we relate an interface in a rightmost form (as seen in the table) with a more abstract, leftmore version *of itself.*

6.4.3 Composition of styles

No nontrivial component uses only a single style. Programmers select styles on a concern-by-concern basis, so many styles may be evidenced side-by-side within a single component ("independent composition"). Moreover, styles may build on other styles. For example, run-time self-description might be exposed as a polymorphic operation building on a particular demultiplexing implementation; compound values might be encoded into strings, for a particular string implementation; a functionally multiplexed entry-point might define a resource management interface; and so on.

Style definitions must support both kinds of composition. In practice this means that the Cake compiler is charged with *composing* style rules to form bilateral rules which incorporate stylistic advice. This *elaboration* process is described in §6.6.

6.4.4 Reversibility of styles

Like other Cake rules, there is potential for styles to be bidirectional. This means rules which describe not only how to *recognise* a particular abstract meaning encoded within a concrete component, but also how to *generate* an alternative concretion of that abstract meaning. Since style definitions are intended to be re-usable, there is an added incentive to make the extra effort necessary to support both directions, and to package these in the same definition.

6.5 Expressing and applying styles

Having established that our design must support compositional application of styles, we now consider how this should work concretely.

A simple example of styles concerns encoding of booleans. This features in Table 6.1 in the "unstructured value encoding" row, "enumerated injection" sub-row. Although DWARF, and hence Cake, has a built-in notion of booleans (the DW_ATE_boolean base encoding), not all components describe booleans in this pre-abstracted way. For example, C code (predating the C99 standard's introduction of a _Bool data type) encodes booleans as integers, with zero indicating false and nonzero indicating true. On the other hand, an opposite convention exists in Unix shell programming: zero indicates truth, and nonzero indicates falsehood. In Cake there are no default conversions between boolean primitives and integers, precisely because these conversions are subject to stylistic variation.[1] Fig. 6.3 shows two Cake style definitions capturing these two alternative conventions. As one would expect, the style uses Cake's table construct (§2.3.6) to relate enumerated sets of values. The styles are parameterised on an identifier integer_typename—partly because languages other than C name integers differently, and partly for other reasons explained shortly.

To allow both layered and independent composition of styles, we require the programmer to apply styles to an input component *in a particular order*. This puts the user in control of both layered composition (where ordering is used to select intended interactions) and independent composition (where reordering can avoid *unintended* interactions). This ordering is specified at the exists block which introduces each component, also shown in Fig. 6.3.

To summarise: at their simplest, styles are libraries of correspondences. Unlike bilateral rules, they are designed to be useful across many composition tasks. However, many styles are written much like correspondences seen in earlier chapters. Instead of relating two components, they relate two *views* of the same component: a more concrete view (always on the left) and a more abstract (on the right). Styles may be parameterised (in

[1]Here we are distinguishing the core of Cake from the default style, which *does* define such conversions precisely as in the c89_booleans style in Fig. 6.3.

```
style  c89_booleans(integer_typename)
{
        table integer_typename ⟷ boolean
        {
                0 ⟷  false ;
                _   ⟶  true;  /* note the order-dependent (impure) pattern matching */
                1 ⟵  true;
        };
};

style  shell_booleans(integer_typename)
{
        table integer_typename ⟷ boolean
        {
                0 ⟷ true;
                _   ⟶  false ;
                1 ⟵  false ;
        };
};

exists c89_booleans(BOOL)( // ⟵apply style,  instantiating  parameter to "BOOL"
            elf_reloc ("componentA.o")  //  ⟵   basic  component specifier
            )
            componentA; // ⟵   overall   identifier   for  component
```

Figure 6.3: Two styles and an applying exists block

a macro-like fashion) to widen their applicability, and these parameters are supplied at exists-time.

What is the effect of applying the style? The short answer is that when the component is used subsequently in a link block, the compiler may form correspondences that would not otherwise be formed—in this case, correspondences for handling integers encoding booleans. Beyond an ordered list of styles, the programmer should not need to give any further guidance to the compiler on which rules to apply. Rather, the compiler should automatically infer a sensible composition of rules. The key conceptual challenge of this is to support “multi-hop” Cake rules: values or function calls being corresponded over a *sequence* of rules, chosen automatically from style definitions. This is performed in an the *elaboration* process, as we describe next.

6.6 Elaboration of styles

Consider composing two C components, componentA and componentB, each representing booleans as integers. With no other styles defined, we have no information about which integers in the C components are actually encoding booleans. Therefore, the usual (implicit) int ⟷ int rule will apply when ints are passed between components. Now suppose componentA uses the C conventions, whereas componentB uses the shell conventions. This composition is mismatched, but since the int ⟷ int still applies, Cake will not address

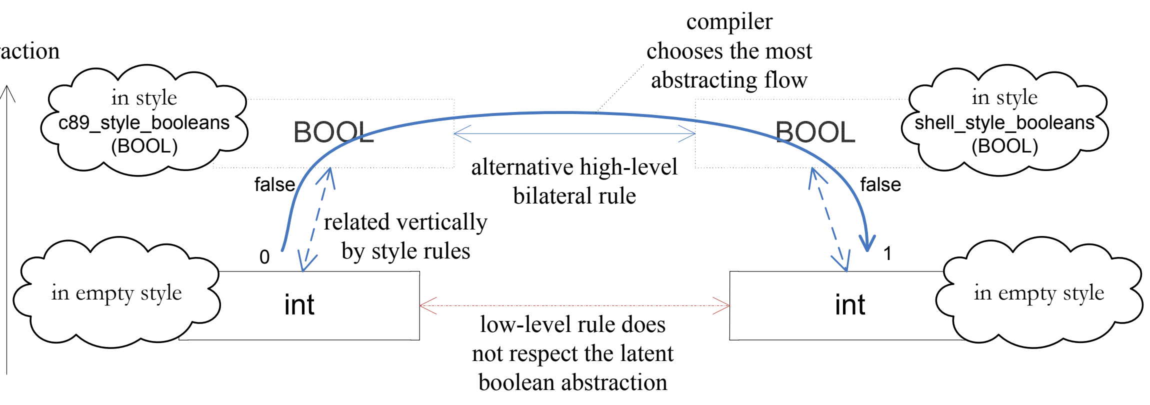

compiler chooses the most abstracting flow
abstraction
in style c89_style_booleans (BOOL)
BOOL
false
0
related vertically by style rules
alternative high-level bilateral rule
BOOL
in style shell_style_booleans (BOOL)
false
1
in empty style
int
low-level rule does not respect the latent boolean abstraction
int
in empty style

Figure 6.4: Alternative flows enabled by styles

this mismatch. To fix this, we need to somehow annotate the input components to identify which integers are really booleans. Then we expect the Cake compiler to perform some automatic analysis to work out what rules should apply to these integers. This analysis is the *elaboration* of styles.

6.6.1 Elaboration disambiguated using annotation

To begin with the simplest case, let us assume that this "annotation" has been done for us, in that the C programmer has used typedef to create a synonym for integers, used exactly when they represent booleans. (Recall from §2.3.5 that Cake unifies data-type synonyms with annotations used to select among value correspondences, and calls these *artificial data types*.)

After applying this style, we have two possible "flows" for BOOL instances: one treating the BOOL in the style-specified way, and one treating it as a plain integer. Fig. 6.4 illustrates. This ambiguity is a generalised version of the value correspondence ambiguity problem (§2.4.4). Instead of selecting a single value correspondence, we are selecting a *several* in a sequence of "hops". We call such a sequence a "flow". (In the earlier version of Cake, without style support, flows consisted of a single hop only, so were of no special interest.) A flow is structured as a sequence of "abstracting" style rule applications (i.e. rules read from left to right), then a bilateral rule (either explicit or formed by name-matching), and then a sequence of "concreting" style rule applications (i.e. rules read from right to left). As before, we gain flexibility by resolving the selection of rules only in a particular composition context. This allows styles to define what *may* happen, independently of context, and leave the tool to determine what *must* happen once a specific composition is being formed.

6.6.2 Semantics of elaboration

In this example, implicitly there is at least one bilateral rule passing a BOOL from one component to another; when compiling this, the compiler must select a particular *sequence* of value conversion rules, including potentially any number of style rules on each component's side. Loosely, the language's semantics are to always choose the "most abstract" option, meaning the "tallest stack" of styles (as in Fig. 6.4).

More precisely, a "flow" is defined a sequence of style-defined value correspondences either *upward* from the value-source component, or downward to a value-sink component. The relative ordering of styles in an "upward" direction is from innermore to outermore styles, as specified in the exists statement (see Fig. 6.3). From any single style at most one rule will be selected in any flow. Cake always prefers a *more abstract* flow, where a flow A is more abstract than a flow B if B is a strict prefix of A when written in upward order. This defines a partial order; it is an error if the compiler finds two possible flows (G, H) without *either* G more abstract than H *or* vice-versa. The presence of such ambiguous flows usually indicates that the programmer applied too many styles simultaneously to the input components.

This search for the "most abstracting" composition of styles is the core of the elaboration process. Note that the process is performed statically, within the compiler. Its output is conceptually similar to a set of bilateral correspondences such as a Cake programmer could have written by hand. However, these correspondences may internally by implemented by many stages of functional interposition, many levels of value conversions, and potentially many levels of logical replication (if objects are not shareable).

As before, the set of "source" and "sink" data types is bounded by the sets of data types defined in each of the pair of components, augmented by any artificial types introduced by annotations. Annotation fragments may introduce only statically named artificial data types (that is, the name following the as may not be computed). This, together with the fact that styles do not recurse (each style named in a given component's exists block is applied at most once during elaboration) ensure that elaboration is a necessarily finite process.

6.6.3 Adding annotation using quantification

If we didn't have a convenient typedef in our previous example, we would require another way to identify the contexts where an int models a boolean. For example, perhaps an API contains many calls with parameters named flag, which are declared as int but actually model booleans (under one of our conventions). Capturing these styles means generalising our notions of *quantification* and *context*, so that we can identify precisely those places in unannotated code where particular rules should apply. The rule in Fig. 6.5 is *guarded* by a square-bracketed *guard predicate*. This constrains the application of the rule to contexts where the guard pattern can be instantiated. Here, it requires that flag is the name of a parameter to some function; the same flag is then subject to a new rule, annotating it to be treated as the data-type syn_name.

Note that this kind of rule is not strictly a value correspondence: it is a "fragment", inserting an annotation which influences subsequent selection among other value correspondences. This is analogous with how explicit annotations are used in bilateral rules, again with the as construct. Unlike explicit annotations, fragments perform *quantified*

```
style boolean_as_integer_synonym(syn_name)
{ /* guard in square brackets */  /* ... followed by guarded rule */
    [_: _ (..., flag : int , ...)]  flag ⟶ flag as syn_name;
};
```

Figure 6.5: A style with guard predicate

```
style boolean_as_integer_synonym(syn_name)
{
        [@x in /.* flag. */, // "in" meta-predicate allows "x" to range over matched identifiers
         _: _ (..., @x: int , ...)]  @x ⟶ @x as syn_name;
};
```

Figure 6.6: A style using a metavariable-based guard predicate

annotation: the annotation may apply in any context matching the guard. When constructing possible flows, guard-satisfying fragments are considered alongside value correspondences as potential abstracting steps of the flow. Instantiation of the fragment within a flow leaves a "dangling" use of some artificial data type name (named by the parameter syn_name) which may or may not be resolved by outermore styles; if not, then no flows containing the fragment will be considered. This hints that elaboration of flows may be implemented using a backtracking search algorithm.

Note that use of the name flag, in addition to naming a quantifying variable, provides a name constraint: it matches parameters called *flag*, only. This is consistent with the Cake language as used throughout this dissertation: when identifiers are preceded by the colon, they are bound to a like-named element in the component's debugging metadata; in other lexical contexts, a name such as flag is universally quantified over any value appearing in the relevant program context. In contexts involving the colon, we can use an identifier having the @ prefix, called a *metavariable*, to provide the quantifying behaviour. Metavariables can be useful for describing more complex guard conditions that require multiple references to a recurring matched value, without any constraint that the same name be used to denote that value in each relevant entry in the underlying debugging information.

Contexts appearing in guard expressions, like normal Cake rules, may reference certain elements of static or dynamic context. For example, in Fig. 6.5 we reference flag's context within an argument tuple, which constitutes a piece of context defined relative to the input component's static form (i.e. the argument list of some function in an input component's debug information); it does not specify any dynamic context such as what calls precede the call being matched. In general, static context is available for any rule, but availability of dynamic contexts varies between rules. For example, only event correspondences may be guarded on dynamic call context, whereas value correspondences may not, because blackboards are only inserted at specific places within generated code.

```
// abstract a single ioctl
style ioctl_call (subcall_ident)
{
        ioctl (fd, subcall_ident, args... ) ⟶ device_op_ ## subcall_ident(fd, args);
};

// abstract all ioctls that we know about
style ioctl_all
{
        [@x: const int, #@x in pattern /.IO.. ∗|..( GET|SET){1,5}/]
        // here ˆ "const int" should really reference DW_MACINFO data (see text)
        ioctl (fd, @x, args... ) ⟶ device_op_ ## @x(fd, args);
};
```

Figure 6.7: A simple stylistic event correspondence

6.7 Reversibility

It is worth re-stating an underlying motivation for styles: that the bilateral Cake programmer should be oblivious as to whether the composed components use the same or differing styles. This means not only that Cake can provide an abstract view of components, but that that it can compose components that are *different* underneath that abstract view. Given this requirement to support heterogeneity, it is especially important that styles can be applied in both directions: both *recognising* (relating concrete with abstract) and *generating* (relating abstract with concrete). The recognition step "undoes" some concretion into an abstract form, and the generation step "replays" a (usually) different concretion.

Rules for relating values are generally written to be reversible as a matter of course (for example, in the c89_booleans style, where we provided rules covering both directions) because values often flow in both directions. Styles may also be used to abstract function calls. In this case, reversibility presents more problems. We now discuss these problems and outline a technique for handling various common cases.

6.7.1 Event style rules

At their simplest, event styles are simply ways of recognising concrete calls, or patterns of calls, and interpreting them as (usually) a simpler, more abstract call. As a trivial example, consider a caller invoking a control operation on a device. In Unix, this is done with the ioctl() call, which is parameterised by a "sub-call" argument identifying the operation. Fig. 6.7 shows style rules for abstracting occurrences of ioctl() into their own function call. The first style abstracts occurrences of a single sub-call; the second attempts to abstract *all* known sub-calls.

As an aside, we note that the extended quantification introduced by styles allows Cake to perform nontrivial metaprogramming. Both examples use the identifier pasting operator ##, taking after the C preprocessor (but necessarily implemented entirely within the compiler), to compute a composite identifier for the abstract call. The second style also uses # to reify an identifier as a string, in order to perform meta-level reasoning, here testing that the identifier matches a pattern. In the second style we are attempting to quantify over all symbolic constants denoting ioctl() subcalls. (Unfortunately, as written, the style will not quite work, because these symbolic constants are coded as C preprocessor macros rather than compiler-visible named constants. However, macro definitions can be expressed in DWARF information, albeit manifested somewhat differently to symbolic constants. Therefore, only a small extension to the Cake language is required for the example to work correctly.)

6.7.2 Abstraction as normalisation

We note that the underlying ioctl() is rendered as a multiplexed entry point so that the set of defined opcodes can be transparently widened later without resorting to adding a new top-level function call to the API. However, this has the side-effect of misusing the functional abstraction somewhat: abstractly separate functions are no longer represented as concretely separate functions. (Note that in no conceivable usage is the subcall argument a computed value.) Our intention in writing this style is to reduce the "mismatch gap" with *alternative* interfaces providing the same logical functionality. Normalising away ioctl()'s peculiarities is sensible because any alternative device control interface (which we might want to link with a client calling ioctl()) is unlikely to be multiplexed in the same way; the target "normal form" is one where each logical function appears as its own "physical" function.

Note that our approach to this normalisation risks introducing ambiguity: if we found two or more symbolic values for the same integer value (say FIONREAD == 0x541B and also TIOCINQ == 0x541B, to use a real example), the second event correspondence would become ambiguous, because there would be two different ways of abstracting a call matching ioctl(_, 0x541B, ...). This is exactly the same ambiguity as would arise if we had written, in the Cake of Chapter 2, two different event correspondences for such a pattern.

For resolving this ambiguity, we take the usual approach: delay decisions on which abstractions apply until composition time, and then search the space of style applications to find the "most abstract" resolution that satisfies the requirements of the linked interfaces. This might resolve the above ambiguity if, say, a later style rule further abstracted device_op_FIONREAD() and not device_op_TIOCINQ(). As before, cases where there is no such unambiguous resolution are an error which is pushed back to the user.

6.7.3 A multi-call example

As a more advanced example of styles interpreting function calls, consider a caller written in C but consuming a Java library, using the Java Native Interface [Liang 1999]. Fig. 6.8 shows the C code a JNI programmer might write to call a function, and Fig. 6.9 shows a Cake fragment that might abstract the resulting object code. It uses a familiar context-predicated Cake event pattern (§2.3.1), but where each element in the sequence has its own guard predicate. Aside from the guard predicates and metavariables, Fig. 6.9 looks much like a familiar event correspondence. (Clearly, in a realistic scenario we would write

```
JavaVM *jvm;
JNIEnv *env;
JavaVMInitArgs vm_args;
long status = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
if (status != JNI_ERR)
{
    jclass cls = (*env)↪FindClass(env, "java/lang/System");
    if (cls)
    {
        jmethodID mid = (*env)↪GetStaticMethodID(env, cls, "currentTimeMillis", "(J)V");
        if (mid)
        {
            jint result = (*env)↪CallStaticLongMethod(env, cls, mid, 5);
        } // else handle error
    } // else handle error
} // else handle error
```

Figure 6.8: JNI C code, such as could be abstracted as in Fig. 6.9

```
style jni_static_longcall (classname, funname, argsig)
{
  [status != JNI_ERR]
     (status, jvm, env) ⇐ JNI_CreateJavaVM(out _, out _, _), ...,
  [cls != 0, @FindClass == (*env)↪FindClass]
     cls ⇐ @FindClass(env, #classname), ...,
  [mid != 0, @GetStaticMethodID == (*env)↪GetStaticMethodID]
     mid ⇐ @GetStaticMethodID(env, #funname, #argsig), ...,
  [@CallStaticLongMethod == (*env)↪CallLongMethod]
       @CallStaticLongMethod(env, mid, args... ) ⟶ classname ## _ ## funname ## _ ## argsig(args...);
}
```

Figure 6.9: Recognising and abstracting a concrete sequence of calls

rules for capturing not only invocations dispatched by CallStaticLongMethod but also ones dispatched by other JNI calls.)

We would like Cake compositions to effectively perform an "undoing" one concretion—for example, "undoing" JNI concretion into an abstract form—and then generating an alternative concretion using other rules in the other direction. This enables a "mix and match" compositionality between heterogeneous components, provided only that we can abstract them to a common form that can be related with (hopefully trivial) bilateral Cake rules or simple name-matching. (This is similar to the "objectification" transformation of COMPOST [Assmann et al. 2000], but additionally retains Cake's usual benefits of a language-agnostic, black-box and binary approach.)

"Reversing" an event style rule implies that it should be bidirectional. We noted in §3.7 that fully general bidirectional correspondences are not yet expressible in Cake. The main challenge identified there was to unify event context patterns (§2.3.1) with stubs (§2.3.2). However, rules like the JNI rules in Fig. 6.9 constitute a simpler case, in that they have only a singleton call on the right-hand (abstract) side. Therefore, reversing

```
JNI_CreateJavaVM(out _, out _, my_vm_args) ⟶{};
```

Figure 6.10: An empty right-hand side rule for enabling reversibility

them requires only to *interpose on* this abstract call when it is recognised, by emitting it as a call that is then directed to a wrapper function, and then synthesise a special kind of stub which reproduces a context satisfying the predicate on the left.

To generate this special kind of stub, we must first *check* which of the context predicates already hold—this is done by keeping a blackboard, as previously (§4.2.7). Then, for any components of the predicate which *do not* hold, we may synthesise a call to make it so. The key precondition for this synthesis is that both the selection of this call and its arguments must be fully determined given only the abstract function name, its arguments, and the contents of the blackboard (that is, details of whatever calls have been made previously).

In our example, most of the JNI calls' arguments can be recovered—in our case, from the name of the abstract function, which encodes the arguments to FindClass and GetMethodID. However, we cannot do this for JNI_CreateJavaVM (since the abstract function provides no way to infer the vm_args argument), so we have to write a special rule to perform this call. This appears as a simple event correspondences with an empty right-hand side, and crucially, providing a specific argument value for vm_args. Fig. 6.10 shows such a rule, which implicitly depends on a context where the JNI-calling component defines a my_vm_args structure. (If no structure defining VM arguments exists, it may be added statically using instantiate, described in §2.3.3.)

Summary The result of this reversibility is that Cake can play the part of a "reversible stub compiler". This contrasts with the stub compilers in the literature [O'Malley et al. 1994; Eide et al. 1997] which perform only concretion, not abstraction. Of course, there are numerous limitations to the approach described here: it effectively only handles expansion of functional abstractions, and does not address the (often intricate) optimisations with respect to data representation and data copying that those compilers perform.

6.7.4 Other concerns

Our exploration of styles has left many questions unanswered. Here we briefly review some issues previously alluded to.

We mentioned a need for certain abstract concepts, namely those present in Cake's stub language, to be well-defined for *any* input component. For this purpose, we define 'Platonic forms" for these concepts.[2] These are are *standardised* functionally-abstracted interfaces to various common abstract concerns. Platonic forms are defined for those concepts appearing in the Cake stub language which are either not in DWARF, or commonly

[2]This is a reference to Plato's theory of forms, in which for every object or concept appearing in the concrete universe, an ideal abstraction of that concept exists. Note that unlike Plato, we only claim that there are ideal abstractions for certain particularly well-understood concepts.

```
// sequences 1: null-terminated arrays
[true ⟷ s is __cake_sequence]
[
  // function rules -- RHS are Platonic forms
  { s[n] } ⟵ get_nth(s, n);
  { &s[0] } ⟵ begin(s);
  { *p == '\0' } ⟵ is_at_end(s, p);
  { p+1 } ⟵ next(s, p);

  // values rule -- a value correspondence from abstract to concrete...
  values _[] ⟵ __cake_sequence
  {
    // initialization *and* update rule (mutating)
    void ({foreach( this , fn p ⇒ set *p = get(p - &this [0]));
            set this [len] = '\0'}) ⟵
                      ({let len = count(this); assert (len < array_length(that));
                        let get = fn n ⇒ get_nth(this, n)}) void;
  }
]
```

Figure 6.11: Extract from an experimental definition of the default style

not encoded by DWARF-emitting compilers. This includes the success status of a function call (§2.3.2), the ability to generate iterators from objects representing sequences (suitable for passing to Cake algorithms as shown in §2.3.2), output parameters (§2.3.7) and the string data type. Fig. 6.11 includes a fragment of a definition for Cake's default style (§2.3.3), where we can see a functionally-abstracted Platonic form for sequences (similar to many iterator APIs) in the first few rules.

Being an application of standardisation (§1.2.5), Platonic forms suffer the usual limitations. However, standardisation is arguably appropriate here because the styles in question are limited to the closed set of concepts embodied in Cake's stub language. These concepts are simple, recurring, and well-understood. We envisage it being feasible to define them carefully enough that minimal future revisions are required (although, naturally, such careful definitions are a work in progress).

It seems infeasible that we could define a standardised Platonic form for every stylistic concern. There is no urgent need to define standardised abstract forms for concepts not appearing in the stub language. However, a "standard library" including other abstract forms might be useful in eliminating the diversity generated by future independent Cake programmers. It is useful to observe that a repertoire of named component styles exists between developers. Programmers talk of "a Java component", "a GObject component", "a C library" and so on. These imply particular expectations about how those components' interfaces differ, including at the object code level. These "named styles" are not quite like the style definitions we have seen in Cake. Rather, they represent *predicates* over the component—"does it conform to a particular template?"—instead of *transformations* stating how various interface elements "may be viewed abstractly as. . . ".

In this sense, named styles remain opaque to Cake. However, it might emerge that Cake programmers prefer to write style rules whose purpose is to relate one named style to another. An extended form of Cake's interface description sublanguage (§2.3.3) could be used to guard styles (using the `check` qualifier of §2.3.3) on *structural* properties characteristic of that named style. By "reversing" the `check` process, effectively requesting that a given structure is *instantiated*, we could satisfy boilerplate requirements such as the GObject boilerplate problem of §5.6. This could also form a basis for treatment of call demultiplexing styles (as featured in Table 6.1).

As a final hint at the potential for named styles, we draw an analogy between Cake and the familiar tool Make [Feldman 1979]. Makefiles include rules for building one kind of file out of another, just as Cake source files include rules for transforming interfaces from one style to another. Complex makefiles are manageable to write partly because the Make process happens over multiple hops—rather than requiring n^2 rules for converting among n kinds of file, the problem becomes linear because a small number of popular intermediate representations emerge (e.g. PostScript when building documents, object code when building programs, etc.). Potentially the same approach could mitigate diversity in Cake, provided that certain styles emerged as popular intermediate styles. The ability to *infer* a sequence of style transformations that transform a component into a named style, by analogy with Make's ability to infer a sequence of rules that yield a named output file, was the envisaged feature that gave Cake its name. However, such a feature remains future work. In particular, determining the parameters at which each style transformation should be instantiated is a problem. This does not arise in Make, because Make rules are simpler and tend only to be parameterised by values that can be shared by all instances of the same rule (e.g. the typical `CFLAGS` variable).

6.8 Summary

This chapter has both introduced the concept of stylistic variation, and considered addressing it within an extended Cake language. We first provided a broad overview of the problem of stylistic variation, using many examples. We then considered its relationship with existing Cake features, and finally presented in detail two particular issues: how to perform the elaboration of rules using separately-defined styles, and how to reverse Cake rules from *recognising* concretions into *generating* them.

Stylistic variation is a large and multi-faceted phenomenon. The material presented in this chapter has been only a preliminary exercise in capturing and abstracting it. Nevertheless, by accounting for stylistic variation within the design of Cake, we have made real progress towards a fundamentally more powerful class of composition tool.

Chapter 7

Related work

Other work has pursued similar goals to Cake's. This chapter explains in detail how Cake either *differs from* or *advances on* prior and concurrent work.

Relevance criteria To limit our consideration to relevant work, we define two inclusion criteria. Firstly, the work must define a new linguistic abstraction useful for adaptation tasks—this might be a language, or language feature, or library. Secondly, the work must be useful in accommodating diversity and (optionally) change in interfaces. It is not sufficient to address *only* change—we do not consider work that addresses purely evolution of interfaces, since our primary focus has been on *unanticipated compositions* of software.

Distinct approaches Our discussion will highlight the various properties of systems and their designs, and contrast them with Cake. Several properties identified in the Introduction are of interest:

- black- versus white-box techniques (§1.4.2);
- binary versus source-level approaches (§1.9.2);
- clean-slate versus adoptable approaches (§1.4.1);
- support for heterogeneous components (§1.3.3).

Linguistic dimensions Furthermore, we summarise the distinguishing properties of the Cake language design with the following selection, which will be among our points of feature comparison with related work:

- unilateral versus bilateral abstractions (§2.2.10);
- support for context-sensitive cases of adaptation (§2.2.5, §2.3.1);
- support for many-to-many cases of adaptation (§2.2.5, §2.3.4);
- treatment of structured values (§2.2.5, §2.4.1).

Breakdown The related work is scattered across many diverse sub-areas of research. We visit work so as to roughly group it by these sub-areas. This is intended to assist with the orientation of readers already familiar with *some* of the related work. It also means that similar approaches *tend* to be covered together, although there are several exceptions.

7.1 Conventional programming practice

Adaptation is familiar to programmers in conventional object-oriented languages thanks to the adapter design pattern described by Gamma et al. [1995]. This itself was proposed as a more flexible alternative to implementation inheritance: while the latter effectively defines a difference or *delta* over the overridable methods of a statically selected superclass, the former provides a *wrapper* over a dynamically chosen target object. The adapter pattern is simply the documented practice where an object with one interface is made composable with clients of a second interface by interposition of an adapter object. It is, of course, a useful programming pattern which has been employed for decades. We are interested in techniques for specifying and performing adaptation *more effectively* than these conventional approaches.

Adapters also appear in the world of distributed middleware as a general-purpose point of interposition. CORBA, a distributed object middleware standard, features both a "basic object adapter" and later a "portable object adapter". These are server-side objects responsible for dispatching incoming requests to other server-side objects, which may or may not implement the interface being consumed by the client [Pilhofer 1999]. Again, this provides a highly general interposition mechanism but does not contribute any special techniques for describing adaptation.

7.2 Scripting languages

There is a common distinction in programming practice between "scripting" languages and other "systems" or "component" programming languages. Although expounded in comparatively recent literature, notably by Ousterhout [1998], the distinction goes back at least to the 1970s and the Unix shell [Ritchie and Thompson 1974]. Attributes usually associated with scripting languages include brevity, lack of static type-checking, expressiveness, and support for dynamic code evaluation. The key underlying characteristic is the trade-off of safety and performance for brevity and dynamism. The combination of dynamism and brevity makes it quicker to alter scripts—i.e. to perform *edit-based* source-level adaptation on script components—than to modify components written in conventional languages.

7.2.1 Piccola

The composition language Piccola [Achermann and Nierstrasz 2001] is founded on this latter observation: its motto "applications = components + scripts" intends that scripts should be used for parts which are less stable, since they are likely to be more convenient to modify invasively [Nierstrasz and Achermann 2000]. Piccola is a scripting language with formal semantics based on an extended pi-calculus. Its main contributions are two unifying models: of computation, as *agents*, and of communication, as behavioural exchanges of structured messages modelled as *forms*. These are shown to capture a variety of previously-existing styles of black-box composition, including GUI component composition, Unix-style pipelines and mixin layer composition. Moreover, Piccola captures these in the form of well-modularised composition *operators*.

Although Piccola's core design is imperative (syntactically resembling Python), its dynamic and higher-order nature encourages styles to be coded as "component algebras", making component wiring explicit and declarative, which further eases invasive editing. Piccola's design also explicitly provides for definition of customised compositional styles, in the form of new *operators* and *rules*. These might be useful not only to define new abstractions for well-matched component composition, but to adapt across mismatches in a variety of abstractions (bilateral or unilateral). However, devising such abstraction is left to the programmer; Piccola provides no special support for implementing such adapters themselves, and implementing composition operators falls back on fairly conventional imperative coding.

It is not clear whether operators of similar meaning to various advanced classes of Cake rule, supporting context-sensitive or many-to-many adaptation, could be coded in Piccola; in any case this would be complex enough to constitute a contribution in its own right. Moreover, although Piccola's design and underlying formalism is highly general, it cannot be considered to support a wide range of heterogeneous input components. Rather, this is the Piccola implementor's concern, and the presented implementation targets Java bytecode only. The Java interface includes ad-hoc bridging logic between the Piccola abstract VM and the Java VM, and it is unclear whether this support extends to arbitrary Java interfaces.

7.2.2 Other scripting approaches

Mainstream scripting languages As well as being more amenable to invasive edit-based adaptation, scripting languages often have special features for adapting the components they script. Mainstream scripting languages such as Perl [Wall and Loukides 2000] and Python [Van Rossum and Drake Jr. 2003] have extensive support for regular expression-based string matching and rewriting. This is useful for adapting data between different encoding conventions; it is also error-prone. These languages also support invocation of external code using a wide variety of communication mechanisms: calling other scripts, accessing the file system or network, invoking external programs, manipulating environment variables, and so on. This "Swiss army knife" philosophy makes them convenient for glue code which integrates components that are *heterogeneous* with respect to their chosen communication mechanisms.

Orchestration The word "orchestration" often refers to scripting of network-enabled services, most typically Web Services. Owing to the widely distributed nature of such services, orchestration languages such as BPEL [Peltz 2003] and Orc [Misra and Cook 2006] provide special features relating to concurrency, latency and failure. For example, Orc provides a parallel composition operator, support for pipeline-like streamed continuous communication, and both pruning (as a feature) and timeout (as an idiomatic derived form) for late-responding services. Orchestration may therefore be seen as scripting tailored to wide-area distributed execution. There are many similarities between orchestration and recognised scripting languages. For example, the Unix shell has special features for parallel and redundant execution in its `&` and `||` operators. As with scripting, orchestration's main contribution to adaptation is its provision of a domain, separate from component code, which localises integration details and is convenient for adaptation by source-level edits. Applying the Cake approach to mismatched components in widely distributed systems would likely entail adding support for Orc-like primitives for handling concurrency, latency and failure.

7.3 Adaptation described as such

Only a subset of work related to adaptation actually uses that word to describe itself. This section discusses that subset.

7.3.1 Early work

Nimble Black-box adaptation of procedural interfaces was first directly addressed by the Nimble system [Purtilo and Atlee 1991]. It provides a new pattern-based language for rewriting the arguments and return values of procedure call. It can reorder, duplicate or omit arguments, or supplement them based on default values, and can also call on external functions where necessary to transform values. However, its expressiveness is primitive relative to subsequent work (including Cake). For example, with respect to function calls, it lacks the statefulness necessary to express adaptations over *sequences* of invocations. Meanwhile, with respect to values, it has no automatic treatment (such as Cake's) of complex structures.

Protocol adaptation Yellin and Strom [1997] improve on this in their work on finite-state protocol adaptation. This work brings a semi-automatic approach in which each component's source-code is annotated with a finite-state protocol description, and a complete adapter is synthesised from a high-level partial specification. This specification takes the place of a subset of event correspondences which would be required in equivalent Cake code. In some cases, the protocol descriptions allow omission of rules or context that would need to be added explicitly in Cake (as discussed in §3.5). Since their contribution is a theoretical rather than a practical one, it is not demonstrated that the synthesis algorithm described is sufficient for a range of real-world use-cases (and indeed, our experience with Cake has shown that a wider range of features is useful, particularly in the treatment of structured values). Passerone et al. [2002] add a game-theoretic interpretation of the adapter synthesis algorithm of Yellin and Strom, showing that the process of synthesis is equivalent to that of testing for synthesisability.

Event-based adaptation The work of Rine et al. [1999] is similar to Nimble, but includes a slightly more heterogeneous notion of component. Adapters are the default, rather than a special case: each component is accessed only through an adapter. Adapter logic is specified in a configuration file which can specify signature-level remappings similar to those offered by Nimble. Unlike Nimble, procedural requests and responses are fully separated into effectively an asynchronous event-based model—the return path of a method call is modelled with a separate signature invoked by the callee, and may be separately adapted. Adapters also control the implementation of event-based communication: they are responsible for constructing the system-level communication paths between each other at initialisation time. Auxiliary information in the configuration file can select details of communication implementation, from among a fixed set: procedures, pipes or message queues.

LayOM LayOM [Bosch 1999] is an implementation of a "layered object model" for C++, based on a concept of adaptation called "superimposition". Adaptation primitives are captured as operators known as *superimposing entities*, each defining a "layer" or black-box delta over the underlying object, and include Nimble-like function interposition, but also member renaming, interface restriction, run-time configurable method dispatch, and interposition on field accesses. Additionally, LayOM supports a kind of compositionality within adaptation: existing operators can be specialised or combined by the user into new ones. This contrasts with the bilateral core of Cake, where a fixed number of relatively powerful "operators" (function and value correspondences) are provided, but these do not compose within a single application of Cake. Meanwhile, it resembles the style-based extension to Cake (Chapter 6), which supports unilateral adaptations which are expressly designed to compose (but LayOM applies this unilateral approach to all classes of adaptation task, not simply stylistic concerns). In LayOM new operators are introduced as a preprocessing stage for C++: the LayOM new operators are described using new lexing, parsing and code generation rules. Consequently the process of introducing new operators is fairly involved, only C++ components are supported, and only relatively syntactic adaptations are easily supported. For example, there are no many-to-many or context-sensitive operators of the kind supported by Cake.

BCA Binary component adaptation for Java (BCA) [Keller and Holzle 1998] implements a load-time adaptation for Java binaries. Adaptations to a class definition, such as member renamings and method additions, are specified in a "delta file" using a special language syntactically resembling Java. However, the range of adaptations available is narrow, being essentially limited to addition of new code, renaming of symbols, and changes to typing metadata (e.g. the effect of retrospectively adding an "implements" clause to a class definition). Later work under the heading of "program transformation" (§7.9) extends these abilities within similar tool designs. Cake shares its black-box, binary, adoptable approach with BCA, but describes bilateral correspondences rather than unilateral deltas. More significantly, Cake's context-sensitive and many-to-many correspondences, and automatic treatment of object structures, expand on BCA's expressiveness considerably.

7.3.2 Protocols with dynamic communication structure

Bracciali et al. [2005] takes a similar approach to Yellin & Strom, by tackling semi-automatic adaptation given protocol descriptions of the input components. Their system replaces Yellin & Strom's abstraction of finite-state automata with a replication-free pi calculus [Milner et al. 1992]. This remains finite-state, but by including the *channel* abstraction, can explicitly capture the dynamic evolution of communication structure.

This work offers comparable expressiveness to Cake in the case of event correspondences ("function mappings" in the paper's terminology). However, it is impoverished in its treatment of values, which are modelled simply as tuples of opaque names, offering no treatment of object structures and also no way of incorporating algorithms within mappings (cf. Cake's algorithms, §2.3.2). Even finite-state algorithms are not fully supported, because even though these could be encoded into the subset of pi-calculus being used, mappings (unlike behaviours) may take only a restricted number of forms (i.e. a strictly smaller subset), enumerated by example.

A final expressiveness limitation is a lack of support for data-dependent mappings. Consider a Cake stub containing an `if P then call_foo() else call_bar()` statement, where `P` is dependent on a function called earlier in the stub. None of the four classes of correspondence described in the paper (pages 47–49) can express this, because the mapping cannot vary its behaviour according to the response to the earlier outgoing message. Although the programmer could use non-determinism to encode both possible actions, the only way then to resolve this nondeterminism is to modify the annotated behaviour (i.e. protocol) of one or other component. This clearly breaks the mapping abstraction, by encoding a compositional concern within the description of a single component.

The main benefit of this work, which Cake does not provide, is its ability to synthesise adapters using behavioural constraints on participant components (much like the work by Yellin & Strom). Similarly, the contribution is primarily in formalism and algorithm, rather than providing evidence of its effectiveness as a practical tool.

7.3.3 Automatic, purely functional adaptation

AxML [Haack et al. 2002] supports adaptation over a purely functional subset of Standard ML. Programmers annotate code with specification axioms, after which it is available for consideration by the automatic adaptation system. These specifications are built from arbitrary predicates, whose meaning is opaque to the tool—rather, they are defined by the programmer as conventional ML code. Specification axioms are expressed in propositional logic, extended with universal quantification, using these predicates. Adaptation is triggered by writing the signatures of the required data structures or functors in the special form of a "synthesis request" which describes, using the same vocabulary of predicates, the signature type and specification of the required code. The adaptation is performed by source transformation, at compile time.

A key advantage of AxML is that it offers assurance that derived compositions are semantically correct—up to the expressive power of the specification axioms and the correctness of the programmer-supplied implementation of the specification predicates.

Synthesised code specialises and composes existing code using a deliberately constrained set of language features, including supplying additional function arguments, currying or uncurrying, functional abstraction and function application, record formation and record selection. These provide, in effect, a similar degree of expressiveness to the basic features of Cake correspondences and stub code, with the exception that Cake does not support adding extra layers of functional abstraction (i.e. increasing the degree of parameterisation of some code).

Like Cake, AxML explicitly precludes recursion on the grounds that this is usually not required within the tool's remit "to transform modules into similar modules". AxML can support one-to-many function compositions in the sense that the synthesis algorithm considers compositions of available functions when searching for a satisfying term. However, since synthesis requests are framed in terms of individual functions, this does not support many-to-many relations. Since it operates on purely functional code, there is no treatment of object graphs (as opposed to tree-structured record nesting), and unsurprisingly, no context-sensitive treatment of function calls for similar reasons.

A key philosophical difference with Cake is that AxML insists on automation, whereas Cake explicitly embraces programmer guidance. AxML therefore presents a very different set of trade-offs. AxML's search procedure is incomplete, terminated by time-bounding, and will fail in many cases which could be manually expressed in Cake (e.g. cases requiring Cake algorithms or complex data-dependent stubs).

A final consideration of AxML is that since it relies on a vocabulary of opaque predicates that is shared between adapted-from and adapted-to interfaces (as annotations and synthesis requests, respectively), there is an unaddressed potential for mismatch in that higher-level domain, for example if different parties define predicates that are differently named or differently parameterised predicates but nevertheless semantically equivalent. To avoid this, responsibility for annotating existing code could be pushed to the composition author—although this would mean repeated effort, this would be no worse than the equivalent burdens of repeating rules or annotations that can easily occur in Cake programming.

7.3.4 Other recent systems

FLAME [Eisenbach et al. 2007] is a tool for "flexible dynamic linking", targetting the Microsoft .NET Common Language Runtime. It performs a very restricted form of black-box adaptation, affecting only system linkage structure. The authors describe how bytecode for the CLR embeds not only the names of external classes, but also often embeds names of component implementations ("assemblies"). Mismatches of these names prevent linking. FLAME modifies the compiler to embed metadata into the generated code, specifying which class or assembly names are to be treated as substitutable metavariables. The resulting adaptation capabilities are a proper subset of those of BCA (§7.3.1). FLAME adaptations are unilateral and provide no special programmatic abstraction beyond simple name substitutions.

7.4 Adaptation, evolution and refactoring

We discussed the relationship between refactoring and adaptation in the Introduction (§1.2.8). In general, refactoring tools are not suitable for describing or constructing unanticipated compositions in a maintainable fashion. However, the logs kept by refactoring tools have useful application to evolution-oriented adaptation. ReBA [Dig et al. 2008] and ComeBack [Savga et al. 2008] are both systems for generating adapters to bridge mismatches originating in evolution, where that evolution was performed using a logging refactoring tool. These logs are used to generate adapters automatically. Since these tools are not useful for producing unanticipated compositions, we do not discuss them further.

7.4.1 Twinning

Twinning [Nita and Notkin 2010] is a system for describing bilateral relations between APIs in Java (and, potentially, similar languages). It shares much in common with Cake, and was developed concurrently. Relations are expressed as a list of *mappings*, roughly analogous to correspondences in Cake or mappings in the work of Bracciali et al. [2005].

Unlike Cake, Twinning requires a one-to-one correspondence between the data types defined by the two APIs, meaning it would not suffice for some of the Cake examples we have seen (at least the libmpeg2–ffmpeg example in Chapter 2 and the Epiphany–Webkit study in Chapter 5). A special-case behaviour relaxes this in the case of exception types, where one-to-one correspondences are especially rare.

Twinning is a source-level technique: mappings are expressed in terms of fragments of source code, paired with a list of formal parameters (in whose terms the fragment is expressed) and a type name (which must match against the type of the expression). Twinning is therefore sensitive to the syntax of the host language, and is most effective for languages providing a small number of syntactic forms—preferably only one—for each logical feature. This property is true of Java, but less so of more complex languages such as C++.

Twinning is formally a white-box technique since it describes matching and replacement over abstract syntax trees. However, for a simple host language such as Java, and restricted to only a simple set of matching forms—such as simple function invocations and return statements—it can be said to alter only the "edges" of source code, making it comparably robust to a black-box approach in simple cases. This does not hold for more complex mappings, such as mappings relating particular sequences of calls or particular patterns of data-dependency between calls. These are supported by Twinning, but only in a fragile fashion: the AST matching expression is vulnerable to breakage on insertion of unrelated calls within a sequence, or on hoisting a nested subexpression out to its own statement. It seems likely that these fragility problems limit the extent of matching which can be performed in practice.

By contrast, since Cake is based on a black-box approach, and matches dynamic events rather than static source code fragments, it can express complex adaptation rules that are robust to these kinds of internal change. However, Cake does so at a cost in performance:

whereas the dynamic analysis Cake uses to match patterns brings run-time performance cost, Twinning's source code substitutions are unlikely to affect performance.

Unlike Cake, Twinning also includes tool support for the idea of "deep adaptation". This amounts to "pushing" an interface mapping into the original codebase, as a form of refactoring. The same mapping language is used as in the shallow case. Effectively, the tool will refactor the original codebase to yield a design including an abstraction layer (§1.2.2), together with implementations of that abstraction targetting both original APIs. By providing an automated refactoring algorithm to insert this layer, "deep adaptation" overcomes abstraction layers' common drawback of requiring anticipation (at least in cases that can be captured by the mapping language). A similar tool could be developed for a pairing of Cake and some source language of interest. However, this would require more complex analysis than in the Twinning case, since Cake rules are described in dynamic terms, which are further from the source code than Twinning's AST-based rules.

7.4.2 Adaptation as horizontal extension

Concept maps, a feature experimentally added to the C++ language [Järvi et al. 2007], have been demonstrated as useful in practical adaptation tasks. Specifically, they resolve mismatch of data structures across the interfaces of C++ generic libraries (i.e. libraries for template metaprogramming). Concepts effectively define "interface-like" abstractions of structural polymorphism in C++. A concept map is a declaration that extends a particular class so that it satisfies a particular concept, supplying additional code as necessary. Since templates are elaborated statically, run-time performance is mostly identical to hand-crafted implementations. There is no special support for context-sensitive or many-to-many adaptations.

Concept maps belong to a wider class of features enabling what is sometimes called "horizontal extension": the modular addition of new elements to previous units of code, such as classes or other data-type definitions. The next section discusses these more generally.

7.5 Open classes and horizontal extension

C++ concept maps are an example of a *horizontal extension.* They are not the only example: type classes in Haskell [Wadler and Blott 1989], Classboxes [Bergel et al. 2005], object expanders [Warth et al. 2006] and *open classes* in Scala [Odersky et al. 2008] are all essentially similar features. These techniques differ in how and when the extensions are elaborated, how name conflicts and other scoping issues are handled, and in what static checking they permit—we do not detail these distinctions here.

These techniques are invariably unilateral, in that they describe extensions to single components, rather than how to compose a pair of interfaces. They are also black-box, in that they are expressed relative to an interface-like abstraction of the extended data type, according to the host language's information hiding support. In other words, not all details of the original data type will be accessible by the extension.

These extension-oriented language features are designed as small increments on top of existing abstractions such as classes, fields and methods; adaptation tasks are then performed using a combination of the new feature and existing conventional constructs in the host language. This contrasts with Cake's substantially different rule-based approach, which represents a significantly more specialised style of coding.

This incremental approach means that these features do not add support for context-sensitive descriptions of logic of the kind offered by Cake. Any complex relations between the adapted-to and adapted-from interfaces, such as many-to-many relationships between functions or object, must fall back to manual coding in the host language. However, the extension feature may help keep such logic modularly separated from the target code, so this is still an improvement over languages not offering such features.

One semantic distinction with other adaptation tools is that these features allow separate modularisation of the code necessary to support an *extended* interface, rather than an *alternative* one. Hiding or replacing the original interface is not supported. In turn, the interface complexity of the code overall will escalate as more horizontal extensions are added.

A related language feature is the *mixin* [Bracha and Cook 1990]. This complements open classes by defining a subclass-like extension which can attach to any potential superclasses fulfilling some contract demanded by the mixin. A larger-scale composition abstraction is provided by *mixin layers* [Smaragdakis and Batory 1998], which define many such mixins which may be composed (or not) as a unit, intended to model the long-standing practice of layered design. Like horizontal extension features, mixins support the extension of interfaces in a way that is more flexible than traditional subclassing-based primitives. (Whereas horizontal extension allows new interfaces to be added to a specific pre-existing class, without requiring the definition of a new class, mixins are composable units that are useful *only* for defining new classes, but do so without restricting their target to a specific pre-existing class.)

7.6 Linking and interconnection languages

Many researchers have proposed languages or tools which give explicit description of the relationships between modules in a large system. Such languages are usually called "module interconnection languages" or "linking languages". Often they are designed to be useful as high-level structural descriptions of a large system developed, frequently also embodying a novel development method or information-hiding technique [Parnas 1972]. The first example of such a language was probably MIL 75 [DeRemer and Kron 1975]. These descriptions are a convenient domain to adapt the *structure* (or architecture) of a system, as compared with manually altering the linkage relation within source code or binaries.

Knit The Knit linking language [Reid et al. 2000] confers explicit control over the linkage relation over a set of object files, by rewriting symbol names at link-time from a high-level description of object file instances and their intended linkage structure. It also advances on conventional linkage by adopting the Units model [Flatt and Felleisen 1998], which supports hierarchical information hiding and cyclic inter-module reference structures. In addition, Knit allows checking of global properties of a composition by requiring that it satisfies some formula over opaque predicates attached to each component. Knit also provides automatic scheduling of static initializers across components. Although operating logically at the object code level, it optionally supports transformation of the originating C source code for whole-program optimisation purposes.

Jiazzi Jiazzi [McDirmid et al. 2001] applies the same model and linking capabilities to the more complex case of Java linkage. By accommodating cyclical linkage relations, including the case where an exported class is a superclass of an imported class, it simulates open classes (§7.5).

Other tools As already described, renaming is also the technique used by Flexible Dynamic Linking (§7.3). DITools [Serra et al. 2000] provides a dynamic analogue to Knit, supporting load- and run-time rebinding of *dynamically-linked* symbol references, as described in a configuration file.

OMOS + Jigsaw None of the above systems provide any explicit programmatic abstractions of adaptation; rather, like FLAME (§7.3) they simply make interposition within existing code more convenient for the developer. The first linking tool with extensive adaptation support was the combination of the OMOS linker and the Jigsaw language [Bracha et al. 1993]. OMOS is a long-running linking service, while Jigsaw is a language of unilateral transformations over object files. Jigsaw is founded on the insight that module instances may be unified with objects [Seeley 1990]. Jigsaw extends OMOS by supporting linker invocation not only on concrete modules, but also on so-called "meta-objects" specified as transformations of existing concrete objects. Jigsaw specifies these algebraically in terms of an abstract data types for objects, including many operations used to transform and combine objects: symbol hiding, renaming, rebinding and copying, merging two modules, and others. These primitives are shown to support derived constructs such as functional interposition, although there is no support for modularising these derived constructs into more abstract adaptation functions (cf. LayOM, §7.3.1), nor for finer-grained transformations (e.g. at argument level, cf. Nimble and the like, §7.3.1).

7.7 Megaprogramming and mediators

The influence of MIL 75 and similar systems, combined with the explosive growth of complexity in software systems, led in the 1990s to renewed interest in tools suited to the challenges of building very large systems. The term "megaprogramming", coined by Boehm and Scherlis [1992] and elaborated subsequently by Wiederhold et al. [1992] identified the distinct nature of writing software at such large scales. Indeed, the statement of Wiederhold and Genesereth [1997] that "you cannot expect that large-scale information systems will have homogeneous ontologies" is echoed by our arguments in the Introduction to this dissertation, that diversity among interfaces is great (§1.1) and that traditional modular programming, based on information hiding, is therefore insufficient (§1.2.6). Like Cake's treatment of components, megamodules are assumed to be internally consistent, but are not expected to share their model of the domain with other megamodules.

Megaprogramming spawned the language CLAM [Sample et al. 1999] and its runtime infrastructure, CPAM. CLAM shares its key features with subsequent coordination-oriented scripting languages, including Orc, as discussed earlier (§7.2.2). Its emphasis is on writing applications which efficiently consume disparate and distributed data sources, each viewed architecturally as an encapsulated megamodule. Its most notable feature is the decomposition of procedure calls into asynchronous operations for sending and receiving data. By doing so, it enables more flexible data-flow and synchronisation behaviour, hence enabling greater efficiencies in a distributed setting—but exposing somewhat greater complexity to the programmer. By contrast, although Cake is also a language targeted at composition tasks, its goals are to simplify related programming tasks, rather than to enable greater efficiency.

CPAM [Melloul et al. 1999] is the coordination protocol underlying CLAM, designed to support a variety of pre-existing remote procedure call (RPC) protocols including CORBA, Java RMI, and others. (We will discuss RPC systems in §7.13.) As such, it shares Cake's desire to support heterogeneous existing code (the subject of Chapter 6). However, to support data interchange, it requires transmitted data to be encoded in a specific binary format. Indeed, the authors state that "a client doing composition only (and not computation in between) does not need to interpret the data it receives from a megamodule or sends to another megamodule." It follows that CPAM and CLAM cannot support transformations of data structures of the kind supported by Cake. Meanwhile, although CLAM can be implemented on top of existing infrastructures, such as CORBA and Java RMI, it does not follow that systems already built using these systems are directly usable by CLAM as megamodules. Rather, extra *wrapping* development effort is required. (In contrast, Cake's approach allows any DWARF-described object code to be used as-is within a Cake composition.) Although various incidental complexities are handled by automatic tools, analogous to stub and skeleton generation in the underlying RPC system, the core task of relating the interfaces being composed, described as "mapping of methods and parameters", is left to manual coding in the CPAM approach. By contrast, this is the very task addressed by Cake.

Related literature identifies *mediators* as an architectural feature of large-scale systems of the kind constructed by megaprogramming. Wiederhold [1995] describes mediation as "an extra software layer... inserted between the client and server [which] breaks the coupling". Again, much like Cake components may be seen as megamodules, Cake-generated adapters may be seen as mediators. The literature on mediators is primarily conceptual or architectural, and describes little specific tool support for programming mediators. An exception is FICAS [Liu et al. 2004], which considers mediators in web service compositions. (Web services are described in §7.13; we may see them simply as remote-procedure services for wide-area distributed use, hence fitting within the megamodule paradigm.) FICAS defines an infrastructure for service composition in which mediators—components containing adaptation logic—are implemented as mobile code. The choice of mobile code, implemented using the support for network class loading in Java, is to allow mediators performing data transformation and aggregation to be pushed close to data sources, for greater efficiency. Indeed, much like CLAM, this work's goal is to support widely distributed data-centric applications efficiently; there is no specific support for programming mediators. Rather, mediators must be programmed against a narrow interface of data manipulation primitives, based on a DataElement class providing primitives such as getBooleanValue, setValue and so on. This is certainly more constraining than programming adaptation logic directly in a conventional language, not to mention a specialised language like Cake. As with CPAM, services must also be wrapped to be usable within the FICAS system.

7.8 Decentralised modularisation

Several technologies provide alternative ways to modularise large codebases, and in so doing, provide a domain which expresses some kinds of adaptation.

7.8.1 Patch-based systems

Perhaps the most longstanding scheme used to modularise widely-scattered changes to codebases is the patch, as supported by Unix's patch command. Patches are purely textual; this makes them highly general, in that they apply to any kind of file, but also inherently fragile in that they rely on concrete file contents. Having no knowledge of higher-level abstractions, they also cannot abstract over multiple locations in source code.

Coccinelle [Padioleau et al. 2008] addresses these weaknesses to provide a "semantic patch" abstraction, with the aim of reliably and reusably capturing the semantic intent in patches as they are currently written. Its target domain is the patchsets arising from evolution in large codebases, such as the Linux kernel, where development is continuous and decentralised. The system's semantic patch language SmPL uses pattern-matching to avoid manual specification of each source code location requiring modifications. Since it targets only problems of evolution, we do not discuss it further.

7.8.2 Subject-oriented composition

The idea of "subject-oriented programming" introduced by Harrison and Ossher [1993] addresses diversity and composition among multiple large codebases. Its goal is to enable the black-box composition of new applications out of separate codebases, where this separation is motivated either by encapsulation concerns or by independent development. It therefore has considerable commonality with Cake.

Codebases, or "subjects", share a domain, but concern partially differing properties, operations and taxonomies (i.e. class hierarchy structures) of that domain's objects. Each subject may specify different instance variables and different method suites for a particular object, and arrange objects in a different class hierarchy. *Composition rules* are used to specify correspondences between the subjects' class hierarchies, instance variables, and dispatch strategies for method calls (since the latter may trigger execution of multiple subjects' code).

Ossher et al. [1995] presents a design of subjects composed of compiled code and a descriptive "label", the latter being effectively an interface description closely mirroring Cake's use of debugging information. It also proposes, as borrowed by Cake, the policy of performing name-matching by default.

Descriptions of subject compositions cannot include new code, except in the constrained language of composition rules; in particular, there is no feature comparable to Cake-style stubs. However, a more constrained form of one-to-many mappings of method calls to code execution is supported: a method invocation is in general mapped to *multiple* bodies of code, executed in some sequence (which may be arbitrary). A variety of treatments are provided for coalescing the return values of these code bodies for return to the caller (such as returning the last value in the list, asserting that all values are equal, or returning an array containing all return values). In contrast to Cake, no context-sensitive mappings are supported.

For describing value correspondences, the subject-oriented approach separates the *identity* of an object from the *multiple* collections of state and code which realise it concretely. It therefore allows many-to-many correspondences between structured values, subject to the constraint that composition must result in a single object identity for each run-time instance of a correspondence. This single identity is analogous to the unique logical identity of objects related under the *co-object* relation in Cake (§4.4.1).

Ossher et al. [1995] highlight a particular semantic requirement on code composed by subject-oriented composition: when manipulating multiple collaborating objects, the programmer must not rely on these being distinct (non-aliased) objects. This somewhat obscure detail highlights two differences in emphasis between subject-oriented composition and Cake. Firstly, the former is both more concerned with *compositing codebases* (to create a unified "super-codebase") rather than Cake's goal of reconciling interface differences. Secondly, the former is contributing primarily a model rather than an implemented system or tool. Indeed, it is difficult to see how binary subject composition could be implemented without an extensively modified toolchain. For example, the presence of many-to-many relations between various classes would entail significant changes to the rules used by the compiler for in-memory layout of objects, to prevent distinct fields being assigned to overlapping ranges of offsets within the composite object.

There is little emphasis placed on the definition of new composition rules themselves, except for a list of primitive rules consisting of essentially the same primitives as Jigsaw's [Bracha et al. 1993]. One new addition is the idea of allowing specification in the form of *general rules* (quantifying over all subjects, or classes, methods etc.) supplemented with *exceptions* for special cases. These ideas are borrowed directly by Cake, particularly in its idiomatic use of separate value correspondences for general and special cases (§5.5).

7.8.3 Aspect-oriented programming

Aspects in their most general form, as described by Filman and Friedman [2005], are modules expressing modifications or additions to existing code in a way which supports *quantification* and *obliviousness*. The former means that points of application are identified by some logical expression *quantifying* over the existing codebase; the latter means that the existing code may remain unaware of these changes, since they are modularised separately.

Typically aspects are advocated as a convenient way of modularising cross-cutting concerns within a single project [Kiczales et al. 1997]. However, the obliviousness property makes them useful for specifying adaptation. AspectJ [Kiczales et al. 2001] is the best-known implementation of aspects; it operates on source-level representations of Java programs. Points in a program's execution where control is transferred to aspect code are identified by quantifying expressions called *pointcuts*. This quantification is dynamic, in that it may be predicated on the program's execution context (e.g. on the class of the current method context's object). The code spliced in at join points is called "advice", and the splicing process is known as "weaving".

Pointcuts may be thought of as a white-box instrumentation language for the basic host language. Aspects are a fundamentally white-box technique, in that pointcuts range over arbitrary internals of a component; this is both powerful, yet potentially damaging to modularity [Steimann 2006]. Like many other techniques we have surveyed, particularly linking languages (§7.6) and horizontal extension mechanisms (§7.5), aspects are primarily an enabler of interposition and extension rather than specifically addressing composition or adaptation tasks. As with horizontal extension, the code advice consists of conventional code, in contrast to the rule-based abstraction of Cake. However, pointcuts themselves are clearly a rule-based abstraction. The context-sensitivity of Cake's event correspondences can be captured in very similar fashion by a suitable pointcut language. However, similar to Twinning (§7.4.1), pointcuts range over all aspects of component execution, rather than an interface abstraction (like *traces* in Cake), so may be less robust to internal code changes than Cake rules. There is no special treatment of data structures analogous to Cake's value correspondences.

7.8.4 Remodularization

Mezini and Ostermann [2002] describe a set of extensions to a Java-like language for integrating "generic functionality" with pre-existing codebases. "Generic" refers to functionality for which a well-known interface may be defined for such functionality, and various concrete realisations later mapped to that interface; this is similar to our "Platonic forms" (§6.7.4) but applied somewhat more widely. Although described as "remodularization", this is not in the sense of Tarr et al. [1999], but rather meaning simply adaptation: the goal is to "wrap abstractions from the world of the concrete usage scenario and map them to abstractions from the generic component world". The main contributions are improved type-checkability (using virtual types) and avoiding traditional problems of object identity sometimes encountered under the adapter pattern [Hölzle 1993]. This is done with a special state management device ("wrapper recycling") reminiscent of Cake's co-objects (§4.4.1). In this way it supports many-to-many relationships among objects; the overall effect is horizontal extension at component (cf. class) granularity, since it emphasises adapting *families* of related data types at once.

7.9 Program transformation

Several projects have developed systems for transformation of Java programs. These inhabit the space around both BCA (§7.3) and AspectJ (§7.8), but support more powerful white-box adapters. In all cases, these systems are potentially capable of addressing Cake-style adaptation tasks, albeit in a unilateral way, by means of transformers acting on each input component separately. However, they are not evaluated on such tasks, and each presents a significantly lower level of abstraction—of the sort upon which Cake-like tools could be built. We discuss them briefly in turn for completeness.

JOIE JOIE [Cohen et al. 1998] is a load-time transformation system. In it, "transformers" are Java-language components which inspect and modify existing class definitions using a reflection-like API. Transformers can be specified highly invasively, and are specified essentially as programs operating over bytecode (down to the level of instruction mnemonics). Unlike with AspectJ and other ahead-of-time techniques, there is no pattern language for identifying modification sites; regular Java-language iteration and if–else tests are required. There is a strong resemblance between JOIE transformers and COMPOST metaprograms (§7.10). However, JOIE's level of abstraction is lower, being at the binary bytecode level rather than the source level.

Javassist Javassist [Chiba 2000] similarly introduces an expanded reflection API to Java, designed as a general-purpose interface upon which to write tools such as BCA-like adaptation primitives, AspectJ-like pointcut or "hook"-based interposition rules, and load-time stub generation for remote method invocation systems. Unlike JOIE, its interface does not descend to the instruction level; rather, it provides only higher-level primitives, such as method wrapping and field access redirection.

JMangler JMangler [Kniesel et al. 2001] is yet another project with similar goals. Its expressiveness lies between those of JOIE and Javassist: unlike Javassist, it expresses all transformations which preserve binary compatibility of Java bytecode, whereas unlike JOIE, it rules out transformations which break compatibility. It also addresses the unanticipated combination of multiple independently developed transformers, where a change specified by one transformer may trigger changes in others. This is done by introducing a distinction between *interface* and *code* transformations: the former are shown to yield an order-independent fixed point when applied iteratively, which can be done mechanically, whereas the latter's fixed points are order-dependent and therefore must be combined under programmer guidance.

7.10 Software connectors

Architecture description languages (ADLs), much like linking languages, are designed to describe, explain and reason about large-scale structural properties of systems. They also promote re-use of high-level designs, and enable checking and traceability between implementation and design [Garlan and Shaw 1994]. As such, they not only offer the same structural view as that of linking languages, but have, in a few cases, introduced other features useful for adaptation—mostly concerning the notion of "connectors". Connectors may be thought of as re-usable abstractions of *how* two modules might be connected, both in terms of mechanism and of the patterns and conventions of their communication. Traditional programming languages do not cleanly capture such abstractions, and implementation of communication mechanisms is typically spread across program modules and in tool-generated code [Shaw 1994].

7.10.1 UniCon

UniCon [Shaw et al. 1995] resembles a module interconnection language, but introduces a distinction between components (which are C source files in the implementation presented) and connectors. Connectors are implementations of communication abstractions, such as local or remote procedure calls, pipes, real-time resource schedulers and shared variables. Components' binding points, or "players", are wired to the connectors' binding points, or "roles".

The UniCon compiler uses built-in knowledge of each connector type to build the complete system, first generating intermediate artifacts (such as RPC stubs or makefiles) and later invoking a conventional build system to construct the ensemble (a design borrowed by Cake). Since each connector is built-in, and none performs adaptation (i.e. each connector is somehow symmetric in the *requires* and *provides* relationships among its connected components), UniCon's only direct comparison to Cake is as a tool addressing a specific case of stylistic variation (i.e. that of these communication abstractions). However, UniCon is also notable as the foundation for Flexible Packaging, which we discuss separately (see §7.11).

7.10.2 COMPOST

COMPOST [Assmann et al. 2000] explicitly combines adaptation techniques with connectors. Connectors in this system are defined by metaprograms which not only generate adaptation logic in a black-box fashion, but can also more invasively refactor or "rebind" the source code of existing components, written in Java or C++.

COMPOST proceeds by source-level rewriting of input components, in two stages. Firstly, existing code using method calls is automatically rewritten into a generic "abstract" model of communication based on object exchange, effectively normalizing any method call into a predictable syntactic form. Secondly, this abstract code is rewritten to use a new concrete communication mechanism,. These rewriting procedures are expressed as metaprograms over Java or C++ abstract syntax, making them white-box according to our definitions (although the authors term them "grey-box"). Although writing an individual meta-program may be significant effort, the two-stage process allows libraries of abstracting and concreting transformers to be written and re-used independently. The meta-programs themselves are written conventionally, using previously published meta-object protocols for the target component languages.

Without a practical evaluation of COMPOST, it is difficult to gauge how these compare to Cake rules in complexity. However, since each metaprogram adapts a single component at a time (i.e. unilaterally), it works at the lower level of describing transformations rather than describing correspondences; since these transformations are based on abstract syntax, much like Twinning (§7.4.1), they are liable to be more fragile to changes in the component source. The ability to code many-to-many or context-sensitive correspondences as meta-programs is limited only by the host language and the design of its meta-object protocol; however, since this is unlikely to expose a history of call sequences, recognising sequences of calls would depend on inserting a manually coded recogniser machine implementation. (Note that the "context-sensitive" rules described in the paper are actually not sensitive to the dynamic sequential context of a call, but rather, are metaprogramming rules which apply at multiple locations in the target codebase, roughly analogous to Cake's pattern rules or the extended quantification introduced in Chapter 6.)

Similarly, the COMPOST programmer's ability to apply rules to specific classes of value in a program are dependent on the host language's ability to identify these. Since both C++ and Java are statically typed languages, precisely identifying objects at run time by their class would likely involve additional coding effort by the metaprogrammer.

7.11 Packaging

The term "packaging" was used by Callahan [1993] to refer to the "details of how software configurations are 'packaged' into executables". These details include both programming-level details—data encodings, control flow patterns and APIs required by communication mechanisms—and lower-level details (such as `make` rules) for building binaries compatible with loading and linking mechanisms. Packaging systems may support whole-system description, like ADLs, or else only single-component packaging, but in either case clearly provide adaptation techniques, in the sense of adapting between the different communication conventions and styles found among heterogeneous selections of components.

7.11.1 Polygen

The Polygen system [Callahan and Purtilo 1991] accepts declarative descriptions of modules in several procedural languages, together with a composite system description, and uses a "rule base" to generate a makefile which can construct a complete system. The rule base contains knowledge about different procedural packaging styles—such as rules for invoking RPC stub generators or language-specific wrapper generators. Like the later UniCon (§7.10), the system description can select among different implementations of various communication abstractions, and new implementations can be defined (using rules written in Prolog). It can also combine this with knowledge of what communication abstractions are implemented within a single process and which require separate processes, and solve these to produce an optimised composition, without embedding mechanism-specific knowledge into the components themselves. However generation of packaging-specific code is left to external tools, and not abstracted by Polygen itself. The implementation presented uses the Polylith software bus [Purtilo 1994] for these purposes, which includes special-cased stub generation support for a number of pairwise language- and protocol-level data conversions.

7.11.2 Flexible Packaging

UniCon's notion of connectors make it a suitable base for the orthogonalisation of packaging concerns from functionality [Shaw 1995], a goal shared with Polygen. DeLine's system Flexible Packaging [DeLine 2001] builds UniCon into a system for building unanticipated compositions of components, potentially involving heterogeneous communication styles.

Flexible Packaging is a clean-slate solution. It requires that each input component (or "ware") is programmed to a particular, highly generic channel abstraction. In making this requirement, it defers packaging commitments until integration time. However, coding to the channel-based style of wares represents a small commitment by itself, which cannot be deferred. By contrast, Cake embraces prior commitment: it assumes that its input components have been coded conventionally against some interface. Therefore, the Cake programmer must work harder to "undo" the premature commitment embodied in the component. The payoff is that Cake can be used with a wide variety of existing components.

Regarding expressiveness, there is no clear discussion of the extended UniCon's expressive capability. In fact, UniCon is treated as just one possible implementation choice for a packaging code generator, rather than being central to the approach. From the examples, it has clearly been extended with a variety new connectors, including spreadsheet accessors and byte-streams described by grammar combinators.

Flexible Packaging is designed to accommodate a wide variety of control structures within wares. This contrasts with Cake, which currently suffers certain limitations regarding control structures (§3.5). Flexible Packaging achieves this by explicitly basing its design on coroutines. Wares are written in Ciao, a version of C extended with channel-based communication primitives much like those in CSP [Hoare 1978]. Packaging generators, or *packagers*, are tools (such as the UniCon compiler) for generating *packaging* (such as the concrete output of the UniCon compiler) based on a *packaging description* (such as source code in the UniCon language). Ware and packaging each run as one or more coroutines in the assembled program.

Unlike Cake, Flexible Packaging provides features for checking that a given composition of ware and packager is compatible, including a CSP-based notation for channel behaviours. By contrast, Cake pushes this responsibility to the programmer. Note that there is no check that a particular ware actually provides the signature with which it is annotated.

The packager must ensure that generated packaging is well-matched with the ware. This includes channel names. These are derived from packaging descriptions; the integrator can often choose names which will generate packaging that is well-matched with respect to the ware's channel names. An explicit "channel map" may be used in other cases. For mismatches more complex than channel names—for example, mismatches in the grouping of channels or arguments on those channels, or mismatches in the encoding of data sent along channels—system integrators may write additional conventional code in the Ciao language to overcome these. There is no analogue of Cake's value correspondences, or any a high-level notation for describing conversions of data sent along channels. Also note that ware and packaging do not share state: they communicate only by channels.

The output of Flexible Packaging is a component, not a composition. In other words, packaging is a unilateral action, performed for each component separately. Moreover, the system does not explicitly ensure that a suite of generated components will be compatible, nor provide tools for adapting mismatches between them if they are not. If they are not, the developer's recourse is to a modified packaging description or, in more difficult cases, use of a wholly alternative packager (replacing UniCon compiler). Implicitly, it is hoped that the packager would accept a sufficiently expressive range of packaging descriptions that any context-specific details of integration—for example, the layout of particular data structure, or the message format to be used on a particular file descriptor—could be passed through in the description as parameters to the packaging generation process. However, this requires sufficient anticipation on the part of the packager author. In the worst case, a new or modified packager would be required to support some new integration context.

Put differently, the design of Flexible Packaging does not rigidly define a line between packaging style—the abstract or recurring properties of an particular packaging—and packaging detail which might be specific to one particular composition context. Rather, ensuring a sufficiently general packager is left to discipline and foresight on the part of the author. UniCon evidently suffices for a large number of examples presented in the paper, but there is no detailed discussion of its expressiveness.

7.12 Coordination

Coordination might be described as the composition of components under a special awareness of parallel execution, synchronisation and scheduling constraints. It has been described as "managing dependencies between activities" (cited by Papadopoulos and Arbab [1998]), by Gelernter and Carriero [1992] as "gluing together of active pieces", and by Wegner [1996] as "constrained interaction". The word "interaction" is often used to describe the domain of coordination; we can consider this word synonymous with "communication".

Exogenous coordination techniques [Arbab 1998] can constitute a useful medium for black-box adaptation. Specifically, they naturally permit interposition—a key function of the coordination domain—and may provide constructs for rewriting or reordering messages. "Exogenous" stands in contrast to so-called "endogenous" coordination languages such as Linda [Carriero and Gelernter 1989] and its many variants [Papadopoulos and Arbab 1998], which have no such special features. For our purposes, the contrast lies in the fact that the only way to adapt a Linda composition is to edit the composed components themselves.

Exogenously coordinated components never interact directly, but instead exchange opaque messages with a coordination engine, which is responsible for routing, synchronisation and scheduling concerns. In Reo [Arbab and Mavaddat 2002], the behaviour of this engine is specified as a network of channel primitives with specified synchronisation and data-flow behaviours. This network specifies not only the linkage relation between components, but much of its concurrent execution, synchronisation and scheduling behaviour of the system. Consequently, the network of connectors may be modified, independently of components, to prevent deadlock or improve parallelism. There is a clear parallel between the description of this network and a set of Cake rules.

Reo permits component aggregation, interposition and also protocol adaptations (in the sense of Yellin and Strom [1997]). However, the data sent along channels is not modelled; it is treated opaquely except for a filtering channel primitive. It follows that Reo's coordinators do not inspect or modify the messages themselves, so any adaptation of messages must be done in a manner opaque to Reo, by adding or changing components. Since this is a key function of Cake, pervading not only its value correspondences but also the pattern-matching behaviour of event correspondences (which rest on the ability to match particular calls and particular arguments), Reo is not a comparable tool for any realistic application of Cake. Rather, a Cake-like language could usefully describe adaptations at a layer above Reo within a distributed system. (Alternatively, combining Reo with aspect-oriented techniques has been proposed by Eterovic et al. [2004], using pointcuts to exogenously instrument components with channel endpoint logic; one could also add message adaptation logic in this way.)

7.13 Specialised code generators

Code generation tools have been applied in specialised domains to automate the generation of particular kinds of integration logic. By parameterising the generator on some description of the code to be generated, they can provide a medium for the programmer to express adaptations.

Language integration Wrapper generation for programming language integration is one example of specialised adaptation. Tools such as Swig [Beazley 1996] adapt a module written in one language such that it can be consumed by another. The process is parameterised by rules controlling how features of one language should be mapped to those of the other. This permits some flexibility in how each module concretely captures the interface of the other, and can therefore be used for adaptation. However, it it is unlikely to be sufficient to avoid further manual glue coding in cases where both *from* and *to* interfaces are defined *a priori*. Many languages' foreign function interfaces, such as Java's JNI [Liang 1999] and Haskell's FFI [Chakravarty et al.], provide similar wrapping capabilities, but offer still less flexibility.

Stub generation for remoting Stub generation in RPC systems [Birrell and Nelson 1984] or, more recently, in Web Services and Remote Method Invocation [Waldo and

Clemsford 1998], is another domain-specific form of adaptation. The automatically generated stubs (for the client-side invocation) and skeletons (for the server-side dispatch) are adapters from local procedural communication, passing messages on the stack, into a distributed version of the same, passing messages over some network socket. They are mostly black-box, although some implementations may force client code to add extra error handling, for errors associated with distributed execution. Subsequent work has greatly increased the degree of parameterisation and customisation available in the stub generation process, as well as increasing performance [O'Malley et al. 1994; Eide et al. 1997], but none approaches the expressiveness of Cake, and as usual, none provides support for context-sensitive or many-to-many adaptations. However, deep traversal of object graphs (§2.4.1) is a feature which the code generated by IDL compilers shares with Cake.

7.14 Data-only techniques

Database schema evolution and schema mismatch are well-known problems in their own right, and have been studied separately from issues of active communication and computation in mismatched interfaces. Although no such system is directly applicable to Cake-style problems on its own, we may compare these techniques' treatment of data with Cake's.

Web- and XML-oriented tools Much as scripting languages build in support for string rewriting, tools targetting tree-structured or relational data present techniques for addressing mismatch in the organisation of that data, analogously to Cake's value correspondences. Web-derived technologies such as XSLT [Tidwell 2008] or XQuery [Boag et al. 2002] have been created to operate on XML documents: the former as a customisable specification language for XML-to-HTML prettyprinters, and latter as a query language, where querying generalises into extracting and transforming subsets of the available data. Both have grown into Turing-complete languages [Kepser 2004] with useful adaptation facilities such as projection, renaming and (in XSLT's case) pattern-based rewriting of tree structures. As such they are more expressive than Cake's value correspondences. On the other hand, Cake's value correspondences apply, in general, over graphs rather than trees; extending the ideas in these languages to apply over graphs—perhaps as simply as treating certain back-edges opaquely—could be a useful extension to Cake, which is currently somewhat impoverished in its ability to select between value correspondences when traversing object graphs.

Bidirectional combinators Lenses [Foster et al. 2005] are a class of tree transformation combinators which ensure that transformations are *bidirectional*. This is done by constraining the language of combinators from which lenses are built. It is an open question whether this restricted set of transformations would suffice for the kinds of transformations required by Cake. However, at least for the subset that is supported, these combinator-based transformers have the benefit of being bidirectional by design, and providing behavioural guarantees about round-trip operations. By contrast, Cake rules are only bidirectional in simple cases, and ensuring their well-behavedness is a problem that is pushed to the programmer.

Schema mappings Much work in the database community concerns the description and discovery of mappings between different representations of semantically equivalent data. This work considers the same kinds of one-to-one and many-to-many mappings as do Cake's value correspondences. Miller et al. [2000] proposed query languages as suitable abstractions for expressing these mappings and also coined the term *value correspondence* with much the same meaning as it has in Cake (albeit unidirectional, and *guarded* much like the style-oriented rules in Chapter 6). Fletcher et al. [2006] build on this with a formal calculus of data mappings, although this does not represent a tool per se. Other work has progressed towards *automatically* discovering such mappings by a combination of techniques: ontology matching based on field names, learning-based generalisation of human-supplied mappings, and exploiting data integrity constraints [Doan et al. 2001; Nottelmann and Straccia 2005]. These techniques are a complement rather than a replacement for a Cake-like programming medium, as they generate approximate output that requires manual editing in a declarative notation.

7.15 Concluding remarks

This chapter has discussed a large quantity of prior work, overlapping with the goals and achievements of Cake in various ways. A recurring theme is the relative lack of support for many-to-many and context-sensitive relations between interfaces. Testifying to the novelty of Cake is the complete lack of prior work integrating these features in the context of a practical, adoptable tool supporting composition of heterogeneous components.

The work surveyed in this section has also suggested several areas for future work building on Cake. These are summarised, along with our conclusions, in the final chapter.

Chapter 8

Summary and Conclusions

This chapter briefly summarises the substantiation of the thesis, and directions for future work.

8.1 Summary of the thesis and its substantiation

The thesis of this dissertation has been that "using a special-purpose language, based on *relations*, to compose heterogeneous mismatched software components, is significantly more effective in practice than conventional programming languages".

The substantiation of the thesis was divided into a set of goals: that this special purpose language would (1) capture adaptation tasks using a black-box abstraction, with (2) demonstrable practicality of application, (3) an expressiveness that covers a broad range of tasks with *significantly* greater effectiveness than conventional languages, and (4) that the language would extend to support for *heterogeneous* components.

In Chapter 2 we described the Cake language design in detail, establishing its black-box design and the detailing the language features which enable the expressiveness and effectiveness on which future chapters rely.

In Chapter 3 we explored the Cake language further, noting areas where it could be straightforwardly extended to expand the range of use cases to which it could be applied.

In Chapter 4 we outlined an implementation of Cake. By demonstrating the implementability of Cake, and moreover its integration with existing toolchains and ability to use *existing* code, this chapter substantiates the goal of practicality.

In Chapter 5 we presented an evaluation of Cake on three case studies on real codebases, including comparison with pre-existing adapters coded conventionally for each study by those codebases' own developers. We showed both by a series of code examples how Cake alleviates many burdens that afflict the developers in conventional approaches, and by measurement, how the overall result is a substantially shorter body of code under the Cake approach. This, together with the previous chapters, demonstrates Cake's expressiveness and substantiates the larger part of the thesis.

In Chapter 6 we introduce the class of *stylistic* variations in object-code interfaces, and presented a simple extension to Cake which captures a large class of these heterogeneous interface conventions without altering the basic language design. This completes the substantiation of the thesis.

In Chapter 7 we established the novelty of the thesis's contribution by comparison with existing work.

8.2 Future work

Each chapter from Chapter 2 onwards has identified some future work. For brevity, we do not repeat these items here. Instead, we simply present a coarse-grained categorisation with the appropriate backreferences.

Abstracting alternative dimensions of mismatch Addressing the classes of control-based mismatch identified in Chapter 3 would make an interesting challenge. Refining our naive treatment of object lifetime (described in §4.4.2 and explored in §5.6.4) would allow handling of a further class of mismatch. A linguistic abstraction of low-level issues such as calling conventions (§3.4) or exception handling stack-walking protocols (§6.3.2) could prove useful in many cross-language scenarios.

Refinements to the language Bidirectional Cake rules, analogously with bidirectional tree transformations [Foster et al. 2005] could bring added elegance to the Cake language, by unifying stubs with event patterns. Also, a context model for objects reached during heap exploration, perhaps based on a formalisation of tree- or graph-structured data (motivated in §4.4.5) could yield a solution to the problem of nontrivial correspondences between complex object structures identified in §5.6.4. Meanwhile, decoupling concrete components from logical "slices" of traces would improve Cake's scalability to large programming tasks (§3.6).

Formalisations A formal model of interface mismatch would allow us to reason about the completeness of Cake as a language. Currently, no existing work establishes a formal distinction between interface mismatch (of the kind and scale addressed by Cake) and more general programming problems. We noted in §3.2 how models of formal languages appear not especially suited to capturing interface mismatches, but perhaps other domains offer more suitable foundations.

Checking compositions Cake is an ideal vehicle for checking properties of a composition. Given a more precise formal model of component behaviour, perhaps as strengthenings of DWARF-based interface descriptions, we could extend Cake to support checking interesting non-local properties of code. Meanwhile, applying more conventional type- and model-checking approaches to identifying errors in Cake code would improve its practicality as a tool.

White-box complement Cake only performs black-box adaptation, so assumes that input components expose the necessary interposition sites, i.e. that they are well-abstracted from a black-box composition perspective. Future work could investigate the complementary approach of using binary instrumentation systems, such as Pin [Luk et al. 2005] for turning ill- into well-abstracted tasks.

Relationship with other dynamic analyses Cake retains a lot of additional state to implement its specified run-time behaviour—including call history (§4.2.7) and logical replicas of objects (§4.4.4). It would be interesting to explore the overlap between this and heavyweight dynamic analyses done by value-shadowing tools such as Valgrind [Nethercote and Seward 2007]. For example, it might be useful to characterise the classes of analyses can be done with acceptable performance on conventional hardware.

Coverage of trickier use cases From a practical perspective, more sensitive treatment of unions (§2.4.3) by suitable programmer annotation, and interposition on variadic functions (§4.2.6) and inlined functions (§4.2.10) would be useful additions.

Performance optimisations Considering the time and space overheads of logical replication, identified techniques worth exploring include annotations for copying reduction (§4.4.6), automatic inference of these, and an efficient approach to the required heap instrumentation (§4.3.2). Efficient algorithms for implementing the blackboard remain an area for experimentation (§4.2.7). Combination of Cake with link-time optimisation (§4.2.10) and binary rewriting optimisations (§4.1) could also yield significant performance improvement. A performance study of practical applications of Cake would be valuable guidance for this work.

Concurrency An implementation of a thread-safe Cake runtime that can deal with shared-writable or change-prone objects (§4.4.7), safely propagate replica updates without control-flow crossings, and avoid conflicting updates is necessary to make Cake usable for multithreaded programs, and stands to improve performance at the same time.

Automation Cake's correspondences are effectively a somewhat strengthened *specification* such as might be fed to a converter synthesis algorithm [Passerone et al. 2002]. Still missing is a description of the *protocols* of the input components, so that the trickier aspects of control structure can be inferred automatically. Recent work on object usage pattern mining [Wasylkowski et al. 2007] extracts exactly this information; this could be a basis for greater automation of Cake coding. Ontology matching, learning-based schema matching and similar techniques could be applied additionally (§7.14).

Unaddressed stylistic issues Call demultiplexing and synchronisation were deliberately omitted from consideration in §6.3, but would be valuable to explore in future work. A formalisation of styles' preconditions and postconditions, analogous to ConceptC++'s treatment to the same issues for C++ templates, could make styles a more usable and scalable programming tool (§6.4.1). A higher-performance and more robust Cake run-time could be achieved through exploiting certain relevant styles, as identified in §6.3.2.

8.3 Summary

With this chapter, the dissertation concludes. Interface mismatch is a complex and highly recurrent problem. Like so many practical problems, it is unlikely ever to be truly "solved"—nontrivial mismatches will always occur. Nevertheless, this dissertation has contributed a practical step towards reducing its impact on the developer and on the cost of software, by showing that a great many mismatches can be addressed in a modular fashion by some simple abstractions captured in a simple, special-purpose language.

Appendix A

Glossary

This appendix briefly lists the named concepts and other terminology introduced in the dissertation, with references to their definition in the main text.

term	explanation	ref.
admissible reinterpretation	an adjustment to a pointer which doesn't violate the abstraction of the pointed-into structure	§4.7
algorithm (Cake)	an opaquely-defined algorithm over abstract sequences of data items, which can be invoked from a stub	§2.3.2
allocation annotation	a Cake extension to DWARF interface description, describing, for a function call which allocates an object, that the caller should free that object using a given function	§2.3.7
annotations	either a programmer-supplied fragment of DWARF interface description or the Cake extensions, or use of the `as` keyword to select the value correspondences applying to an alternative data type, usually an *artificial data type*	§2.3.3, §2.3.5
artificial data type	either a data-type synonym defined in an input component, or a data type not defined in an input component but introduced by the Cake `as` keyword	§2.3.5
association	a value correspondence involving multiple data-type instances in the same component, for at least one of the two components	§2.3.2
bidirectional rule	a Cake rule using the double-arrow, $\longleftrightarrow$, to describe handling of two directions of data flow at once	§3.7
bilateral rule	a Cake rule which relates two concrete interfaces, perhaps unidirectionally or bidirectionally	§2.2.10
black-box composition	any description of a composition made referencing only interface details, not internal implementation details of any component	§2.2
blackboard	the structure on which past calls across an interface are remembered at run time, for subsequent matching as event context	§4.2.7
bracketed stub expression	a piece of stub code, embedded in a value correspondence rule, defining a computation through which a source value is passed before reaching the sink	§2.3
co-object relation	the run-time record of which object identities are storing logically related state in separate (mismatched) components	§4.4.1

composition context	effectively, a Cake link block	§2.2.7
context	usually *event context*, but also composition context	§2.3.1
context predication	the practice of writing event correspondences containing event patterns on which the correspondences is predicated	§2.3.1
correspondence	either an event correspondence or a value correspondence	§2.2.9
data-dependent call sequence	a stub which uses let to bind a name to an intermediate value which is referenced later in the stub	§2.3.2
data-dependent context predicate	an event context predicate in which a bound name is referenced subsequently in the pattern, forming a requirement that a specific value seen earlier recurs later	§2.3.1
default style	the style definition which is used by default, and captures interface conventions of a reasonable proportion of components coded in C	§2.3.3
dynamic points-to analysis	the analysis which recovers, at run-time, the precise class of an object given a pointer to that object	§4.3.2
DWARF	the debugging information format on which Cake's model of component interfaces is based	§2.2.7
elaboration	the process of expanding styles into rules	§6
error discovery	the process of determining, after a procedure call returns, whether it "succeeded" or "failed" in abstract terms	§2.3.2
event	generally, an abstract instance of communication between two components, of a kind that can be interposed on by Cake; concretely, a procedure call or return from one	§2.2.4
event context	either structural context of an event, or its temporal context	§2.3.1
event correspondence	a Cake rule describing what should occur on a particular function call event	§2.2.9
event pattern	Cake syntax for describing a set of temporal event contexts	§2.2.9
event sequence	a sequence of procedure calls across same interface, and procedure returns in the opposite direction	§2.3.1
functions-only assumption	the assumption that components do not communicate through shared objects unless that sharing was established dynamically, hence allowing Cake to be implemented by functional interposition only	§4.2.1
functional interposition	the procedure of intercepting a function (procedure) call, in order to acquire control of its handling	§4.2.1
guard predicate	a piece of Cake syntax describing a property which must hold for a rule to apply in a particular context	§2.3.5
heterogeneity	the property of software components deriving from different languages, toolchains or coding styles, observable as superficial differences in their interfaces	§1.3.3
identifier pattern	a regular expression matching identifiers in Cake rules that use the pattern keyword	§2.2.11
initialization rule	a rule in a value correspondence that applies only when initializing an object	§2.3.4
input and output parameters	parameters or return values in a procedure call used to implement its logical inputs and outputs, respectively	§2.3.7
instantiate	one of the algebra of Cake composition operators, used to instantiate a named data structure	§2.3.3

interface description	the Cake textual syntax for DWARF information, augmented by some Cake-specific annotations describing interface features which DWARF does not	§2.3.3
logical replication	the technique of maintaining, at run time, multiple representations of the same logical state, implemented by the co-object relation	§4.4.4
metavariable	a variable appearing in a guard predicate which ranges over named variables or other named interface elements	§2.3.5
"most specific match" rule	the behaviour that events (calls) should fire the most specific Cake event pattern of all those defined which match the event	§2.4.4
name matching	the behaviour of the Cake compiler in implicitly drawing correspondences between like-named interface elements	§2.2.11
named style	a style defining a checkable structural condition on its input component	§6.7.4
object discovery	the process implemented by dynamic points-to analysis	§4.3.2
partially split heap	logical replication with the addition of sharing as an optimisation	§4.4.2
pattern	usually: event pattern; sometimes: identifier pattern	
Platonic form	an abstract specification of the interface which a set of style rules should generate	§6.7.4
quantification	the property that a Cake rule applies to any of an open set of dynamic events, so long as its guard predicate (if any) is satisfied	§2.3.5
representation compatibility	a binary compatibility relation on data structures	§4.4.4
rule	a correspondence	§2.2.9
sharing analysis	an analysis which decides whether two components may safely share a given object or instances of a given class, given that the objects are representation-compatible	§4.4.4
source and sink	the origin and, respectively, target of the data flow described by a correspondence rule	§2.2.10
structural context	a fragment of interface description describing the programmatic surroundings of a value's source-level definition	§2.3.5
structural guard	a guard predicate describing a particular structural context	§2.3.5
stub	an imperative fragment of Cake code	§2.3.2
styles	groups of correspondences written to apply to open-ended sets of components, cf. bilateral rules	§6
table rule	a special kind of value correspondences which relates elements of two data-types individually, rather than relating divisions of the data type structurally	§2.3.6
temporal context	the history of interactions between the relevant components when an event occurs between them	§2.3.1
trace	an execution history of interactions between two components, as a list of calls with their argument and return values	§2.2.4
tying	an instruction to the Cake runtime that the lifetime of an object should end when that of another object or allocation ends	§2.3.4

umbrella object	an object created by the Cake runtime to hold pointers to other objects related by an association	§2.3.2
update rule	a rule in a value correspondence that does not apply when initializing an object, unless no initialization rules are given	§2.3.4
value conversion function	a function generated by the Cake compiler to implement value correspondences	§4.2.8
value correspondence	a rule relating elements of one or more data types in one component with one or more in another component	§2.2.9
wrapper function	a function generated by the Cake compiler to implement event correspondences	§4.2.6

Appendix B

Cake recipes

This Appendix gives an overview of the main Cake language features, in the form of templated syntax.

B.1 Pairwise features

These features appear within `link` blocks (§2.2.7), and relate the interfaces of a pair of components.

Event correspondences Event correspondences (§2.2.9) are the relational analogue of function definitions in conventional languages. Rather than defining new callable functions, they relate a function call required by one component with a different call provided by the opposing component. Common argument bindings route arguments; optionally, `as` annotations control the selection of value correspondences applied to those arguments (§2.3.5).

```
f(a, b as t) ⟶ g(b, a as u);
```

Families of correspondences for events of similar names can be defined using an identifier pattern to capture multiple events and their correspondents. (§2.2.11).

```
pattern /do_(.*)/(x) ⟶ perform_\\1(x);
```

Event correspondences can be predicated on an *event context*, meaning the history of prior calls between the pair of interfaces in question.

```
 prior_call1 (arg1),  ...,
 prior_call2 (arg2),
h(c, d) ⟶ j(d, c);
```

Bracketed *stub expressions* are useful (for example) to add extra bindings, or perform any other auxiliary computation. They may appear on either side of the arrow...

```
p(x) (let y = get_y()) ⟶ q(x, y);
```

... but are more likely on the source side (here left), since the sink (here right-hand) expression may be an arbitrary expression in the stub language (§2.2.9).

```
r(z)  ⟶ { s(z); w(); }
```

Value correspondences Treatment of data structures is specified using value correspondences, which are prefixed by the keyword values or contained in a values block (§2.2.9). At their simplest, they relate two named data types in opposing components, meaning that when an instance of one type is provided (in a call, or within an object graph) by one component, an instance of the other should take its place in the opposing component.

```
values Abacus ⟷ BeanCounter;
```

Most often, the named data types are structured types, meaning they consist of fields (data members). Structured value correspondences relate fields within the types (§2.3.4). These are the relational analogue of structured data type definitions in conventional languages, such as C's struct.

```
values Abacus ⟷ BeanCounter
{
   rows ⟷ piles ;
};
```

As with event correspondences, bracketed stub expressions can be used to insert extra logic. The special variables this and that hold the local and remote representations of the source value ("local" meaning the same side as the stub; this may be used only on the source side, and that only on the sink side).

```
values Square ⟷ square_shape
{
        length (this ∗ 4) ⟶ perimeter;
        length (that / 4)⟵  perimeter;
};
```

Another kind of value correspondence is the table construct (§2.3.6), which is the relational analogue of an enumeration type in conventional languages. Rules inside a table relate individual named values.

```
table Colour ⟷ Pigment
{
   Red    ⟷ Vermillion ;
   Brown ⟷ BurntSienna;
};
```

Many-to-many correspondences between objects can be achieved using *associations* (§2.3.2). Syntactically, each component's view of the association is described by a tuple of named elements, each an instance of a named data type. Objects participating in the association are treated much like fields in a structured value correspondence.

```
values
(al: AdjacencyList, cm: ColourMap) ⟷
        (ns: ColouredNodeList, es: EdgeList)
{
        /* ... */
};
```

B.2 Annotations

Pre-existing components may have blocks of annotations attached to them at the point where they are introduced in Cake code, namely at their exists statement (§2.2.7). These annotatons can supplement the debugging information contained within the component, or to add Cake-specific annotations (§2.3.3).

```
exists elf_reloc ("componentA.o")
{
        declare {
                error_t: class_of enum {
                        NO_ERROR, NOT_FOUND, NOT_SUPPORTED, IO_ERROR
                }; // define new DWARF info
                write_output: (_, _) ⇒ error_t; // refine existing DWARF

                handle_t: class_of opaque void ptr; // Cake "opaque" annotation
        }
};
```

Appendix C

Grammar of the Cake language

The following grammar includes all the syntax used in the examples in Chapters 2–5.

C.1 Lexical structure

C.1.1 Skipped lexemes

⟨NEWLINE⟩ ::= '\r'? '\n'

⟨WS⟩ ::= (' ' | '\t')+

⟨LINECOMMENT⟩ ::= '/' '/'(~ '\n')*

⟨BLOCKCOMMENT⟩ ::= '/' '*' (~ '/' | (~ '*') '/')* '*' '/'

C.1.2 Other lexemes

⟨INT⟩ ::= ('1'..'9''0'..'9'*) | '0''x'('0'..'9' | 'a'..'f' | 'A'..'F')+ | '0'

⟨FLOAT⟩ ::= '0'..'9'+ '.''0'..'9'+

⟨STRING_LIT⟩ ::= '\"' (~ '\"' | '\\\"')* '\"'

⟨IDENT⟩ ::= ('\\'. | 'a'..'z' | 'A'..'Z' | '_'+'a'..'z' | '_'+'A'..'Z' | '_'+'0'..'9')
  ('a'..'z' | 'A'..'Z' | '0'..'9' | '_' | '\\'.)*

⟨PATTERN_IDENT⟩ ::= '/'('a'..'z' | 'A'..'Z' | '_''a'..'z' | '_''A'..'Z' | '_''0'..'9')
  ('a'..'z' | 'A'..'Z' | '0'..'9' | '_' | '|' | '*' | '(' | ')' | '.' | '?')*'/'

⟨METAVAR⟩ ::= '@'('a'..'z' | 'A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*

C.2 Interface description syntax

⟨membershipClaim⟩ ::= ⟨memberNameExpr⟩ ':' 'class_of' ⟨valueDescriptionExpr⟩ ';'
  | ⟨memberNameExpr⟩ ':' 'const' ⟨namedDwarfTypeDescription⟩ '=' ⟨constantValueDescription⟩ ';'
  | ⟨namedValueDescription⟩ ';'
  | '...' ':' ⟨valueDescriptionExpr⟩ ';'

⟨*memberNameExpr*⟩ ::= ⟨*definiteMemberName*⟩
| '_'

⟨*definiteMemberName*⟩ ::= ⟨*IDENT*⟩ (⟨*memberSuffix*⟩)*

⟨*memberSuffix*⟩ ::= '.' ⟨*IDENT*⟩
| '[' ⟨*constantIntegerArithmeticExpression*⟩ ']'

⟨*valueDescriptionExpr*⟩ ::= ⟨*primitiveOrFunctionValueDescription*⟩

⟨*primitiveOrFunctionValueDescription*⟩ ::= ⟨*argumentMultiValueDescription*⟩ '=>' ⟨*primitiveOrFunctionValueDescription*⟩
| ⟨*primitiveValueDescription*⟩

⟨*argumentMultiValueDescription*⟩ ::= ⟨*multiValueDescription*⟩

⟨*multiValueDescription*⟩ ::= '(' ⟨*optionallyNamedWithModeValueDescription*⟩ (',' ⟨*optionallyNamedWithModeValueDescription*⟩)* (',' '...')? ')'

⟨*optionallyNamedWithModeValueDescription*⟩ ::= valueModeAnnotation? ⟨*optionallyNamedValueDescription*⟩

⟨*optionallyNamedValueDescription*⟩ ::= (namedValueDescription)=>namedValueDescription
| ⟨*primitiveOrFunctionValueDescription*⟩

⟨*namedValueDescription*⟩ ::= ⟨*memberNameExpr*⟩ ':' ⟨*valueDescriptionExpr*⟩

⟨*namedMultiValueDescription*⟩ ::= '(' ⟨*namedValueDescription*⟩ (',' ⟨*namedValueDescription*⟩)* ')'

⟨*structuredValueDescription*⟩ ::= ('object' | 'struct') '{' membershipClaim* '}'

⟨*simpleOrObjectOrPointerValueDescription*⟩ ::= ⟨*structuredValueDescription*⟩ (⟨*valueDescriptionModifierSuffix*⟩)*
| ⟨*simpleValueDescription*⟩ (⟨*valueDescriptionModifierSuffix*⟩)*
| ⟨*enumValueDescription*⟩ (⟨*valueDescriptionModifierSuffix*⟩)*
| 'void' (⟨*valueDescriptionModifierSuffix*⟩)*
| '(' ⟨*primitiveValueDescription*⟩ ')' (⟨*valueDescriptionModifierSuffix*⟩)*

⟨*valueDescriptionModifierSuffix*⟩ ::= 'ptr'
| '[' arraySizeExpr? ']'

⟨*arraySizeExpr*⟩ ::= ⟨*constantIntegerArithmeticExpression*⟩

⟨*simpleValueDescription*⟩ ::= ⟨*namedDwarfTypeDescription*⟩
| '_'

⟨*byteSizeParameter*⟩ ::= '<' ⟨*INT*⟩ '>'

⟨*namedDwarfTypeDescription*⟩ ::= 'base' ⟨*dwarfBaseTypeDescription*⟩
| ⟨*IDENT*⟩

⟨*dwarfBaseTypeDescription*⟩ ::= ⟨*IDENT*⟩ byteSizeParameter ⟨*dwarfBaseTypeAttributeList*⟩

⟨*dwarfBaseTypeAttributeList*⟩ ::= ('{' (⟨*dwarfBaseTypeAttributeDefinition*⟩)* '}')?

⟨*dwarfBaseTypeAttributeDefinition*⟩ ::= ⟨*IDENT*⟩ '=' (⟨*IDENT*⟩ | ⟨*INT*⟩) ';'

⟨*enumValueDescription*⟩ ::= 'enum' ⟨*dwarfBaseTypeDescription*⟩ enumDefinition
| 'enum' ⟨*enumDefinition*⟩

⟨*enumDefinition*⟩ ::= '{' enumElement* '}'

⟨*enumElement*⟩ ::= 'enumerator' ⟨*IDENT*⟩ '==' ⟨*constantIntegerArithmeticExpression*⟩ ';'
| ⟨*IDENT*⟩ ';'

⟨*primitiveValueDescription*⟩ ::= 'const' ⟨*constantValueDescription*⟩
| valueIntrinsicAnnotation? ⟨*unannotatedValueDescription*⟩ valueInterpretation?

C.3 Cake annotations

⟨*valueInterpretation*⟩ ::= 'as' ⟨*unannotatedValueDescription*⟩ valueConstructionExpression
| 'out_as' ⟨*unannotatedValueDescription*⟩ valueConstructionExpression
| 'interpret_as' ⟨*unannotatedValueDescription*⟩ valueConstructionExpression
| 'in_as' ⟨*unannotatedValueDescription*⟩ valueConstructionExpression

⟨*valueConstructionExpression*⟩ ::= '(' ⟨*stubNonSequencingExpression*⟩ (',' ⟨*stubNonSequencingExpression*⟩)* ')'
| ⟨*empty*⟩

⟨*valueIntrinsicAnnotation*⟩ ::= 'opaque' valueIntrinsicAnnotation?
| 'ignored' valueIntrinsicAnnotation?
| 'invalid' valueIntrinsicAnnotation?
| 'caller_free' '(' ⟨*IDENT*⟩ ')' valueIntrinsicAnnotation?

⟨*valueModeAnnotation*⟩ ::= 'out'
| 'in'
| 'inout'

⟨*unannotatedValueDescription*⟩ ::= ⟨*simpleOrObjectOrPointerValueDescription*⟩

C.4 Literal values and compile-time constant expressions

⟨*patternConstantValueDescription*⟩ ::= ⟨*STRING_LIT*⟩
| ⟨*INT*⟩ '...' ⟨*INT*⟩

⟨*constantOrVoidValueDescription*⟩ ::= ⟨*constantValueDescription*⟩
| 'void'

⟨*constantValueDescription*⟩ ::= ⟨*STRING_LIT*⟩
| 'null'
| ⟨*constantSetExpression*⟩
| ⟨*constantIntegerArithmeticExpression*⟩

⟨*constantSetExpression*⟩ ::= 'set' '[' (⟨*constantValueDescription*⟩ (',' constantValueDescription*)*)? ']'

⟨*setExpression*⟩ ::= 'set' '[' (⟨*stubLangExpression*⟩ (',' stubLangExpression*)*)? ']'

⟨*constantIntegerArithmeticExpression*⟩ ::= ⟨*constantShiftingExpression*⟩

⟨*primitiveIntegerArithmeticExpression*⟩ ::= ⟨*INT*⟩
| ⟨*memberNameExpr*⟩
| '(' ⟨*constantIntegerArithmeticExpression*⟩ ')'

⟨*constantUnaryOperatorExpression*⟩ ::= ('-' | '+')* ⟨*primitiveIntegerArithmeticExpression*⟩

⟨*constantMultiplicativeOperatorExpression*⟩ ::= ⟨*constantUnaryOperatorExpression*⟩
(('*' | '/' | '%') ⟨*constantUnaryOperatorExpression*⟩)*

⟨*constantAdditiveOperatorExpression*⟩ ::= ⟨*constantMultiplicativeOperatorExpression*⟩
(('+' | '-') ⟨*constantMultiplicativeOperatorExpression*⟩)*

⟨*constantShiftingExpression*⟩ ::= ⟨*constantAdditiveOperatorExpression*⟩
(('>>' | '<<') ⟨*constantAdditiveOperatorExpression*⟩)*

C.5 Cake language syntax proper

```
⟨toplevel⟩ ::= declaration*

⟨declaration⟩ ::= ⟨existsDeclaration⟩ ’;’ ?
    | ⟨aliasDeclaration⟩ ’;’ ?
    | ⟨supplementaryDeclaration⟩ ’;’ ?
    | ⟨inlineDeclaration⟩ ’;’ ?
    | ⟨deriveDeclaration⟩ ’;’ ?

⟨aliasDeclaration⟩ ::= ’alias’ ⟨aliasDescription⟩ IDENT ’;’

⟨aliasDescription⟩ ::= ⟨IDENT⟩
    | ’any’ ⟨identList⟩

⟨identList⟩ ::= ’[’ ⟨IDENT⟩ ( ’,’ ⟨IDENT⟩ )* ’,’? ’]’

⟨supplementaryDeclaration⟩ ::= ⟨IDENT⟩ ’{’ claimGroup* ’}’

⟨inlineDeclaration⟩ ::= ’inline’ ⟨objectSpec⟩ ⟨wellNestedTokenBlock⟩

⟨wellNestedTokenBlock⟩ ::= big alternation of all distinct tokens

⟨objectConstructor⟩ ::= ⟨IDENT⟩ ( ’(’ ⟨STRING_LIT⟩ ’)’ )?

⟨objectSpec⟩ ::= ⟨objectConstructor⟩ IDENT
    | ⟨objectConstructor⟩ ’deriving’ ⟨objectConstructor⟩ IDENT

⟨existsDeclaration⟩ ::= ’exists’ ⟨objectSpec⟩ existsBody

⟨existsBody⟩ ::= ’{’ ( ⟨claimGroup⟩ | ⟨globalRewrite⟩ )* ’}’
    | ⟨empty⟩

⟨globalRewrite⟩ ::= ’static’? ⟨valueDescriptionExpr⟩ ’–>’ ⟨valueDescriptionExpr⟩ ’;’

⟨claimGroup⟩ ::= ’check’ ’{’ claim* ’}’
    | ’declare’ ’{’ claim* ’}’
    | ’override’ ’{’ claim* ’}’

⟨claim⟩ ::= ⟨membershipClaim⟩

⟨deriveDeclaration⟩ ::= ’derive’ ⟨objectConstructor⟩ IDENT ’=’ ⟨derivedObjectExpression⟩ ’;’

⟨derivedObjectExpression⟩ ::= ⟨IDENT⟩ ( ’(’ ⟨derivedObjectFunctionArgument⟩ (’,’ derivedObjectFunctionArgument)* ’)’
        )?
    | ’link’ ⟨identList⟩ linkRefinement

⟨derivedObjectFunctionArgument⟩ ::= ⟨derivedObjectExpression⟩
    | ⟨STRING_LIT⟩

⟨linkRefinement⟩ ::= ’{’ pairwiseCorrespondenceBlock* ’}’
    | ⟨empty⟩

⟨pairwiseCorrespondenceBlock⟩ ::= ⟨IDENT⟩ ’<–>’ ⟨IDENT⟩ pairwiseCorrespondenceBody

⟨pairwiseCorrespondenceBody⟩ ::= ’{’ pairwiseCorrespondenceElement* ’}’

⟨pairwiseCorrespondenceElement⟩ ::= ⟨IDENT⟩ ’:’ ⟨eventCorrespondence⟩
    | ⟨eventCorrespondence⟩
    | ⟨IDENT⟩ ’:’ ⟨valueCorrespondenceBlock⟩
    | ⟨valueCorrespondenceBlock⟩
    | ⟨singleValueCorrespondence⟩
    | ⟨IDENT⟩ ’:’ ⟨singleValueCorrespondence⟩
```

```
⟨eventCorrespondence⟩ ::= ⟨eventPattern⟩ ⟨infixStubExpression⟩ '−>'
        ⟨infixStubExpression⟩ ⟨eventPatternRewriteExpr⟩ ⟨leftRightEventCorrespondenceTerminator⟩
    | ⟨eventPatternRewriteExpr⟩ ⟨infixStubExpression⟩ '<−'
        ⟨infixStubExpression⟩ ⟨eventPattern⟩ ⟨rightLeftEventCorrespondenceTerminator⟩
    | ⟨atomicEventPattern⟩ ⟨infixStubExpression⟩ '<−>'
        ⟨infixStubExpression⟩ ⟨atomicEventPattern⟩ ⟨bidirectionalEventCorrespondenceTerminator⟩

⟨bidirectionalEventCorrespondenceTerminator⟩ ::= ';'

⟨leftRightEventCorrespondenceTerminator⟩ ::= ';'
    | '<−' ⟨stubNonSequencingExpression⟩ ';'
    | '<−' '−{' ⟨sequencingExpression⟩ ';'? '}' ';'

⟨rightLeftEventCorrespondenceTerminator⟩ ::= ';'
    | '−>' ⟨stubNonSequencingExpression⟩ ';'
    | '−>' '−{ ⟨sequencingExpression⟩ ';'? '}' ';'

⟨eventContext⟩ ::= ( '(' ( ⟨stackFramePattern⟩ '::' )+ ')' )?

⟨stackFramePattern⟩ ::= ⟨IDENT⟩

⟨eventPattern⟩ ::= ⟨atomicEventPattern⟩
    | ⟨contextBindingEventPattern⟩ ( ',' ⟨contextBindingEventPattern⟩ )* ',' ⟨atomicEventPattern⟩

⟨contextBindingEventPattern⟩ ::= ⟨bindingPrefix⟩ atomicEventPattern
    | '...'

⟨atomicEventPattern⟩ ::= ⟨eventContext⟩ ⟨memberNameExpr⟩ ⟨eventParameterNamesAnnotation⟩
        ( '(' ( ( ⟨annotatedValueBindingPattern⟩ ( ',' ⟨annotatedValueBindingPattern⟩ )*
        (',' '...' )? ) | '...' )? ')' eventCountPredicate? )?
    | ⟨eventContext⟩ ⟨identPattern⟩ ⟨eventParameterNamesAnnotation⟩
        ( '(' ( ( ⟨annotatedValueBindingPattern⟩ ( ',' ⟨annotatedValueBindingPattern⟩ )*
        (',' '...' )? ) | '...' )? ')' eventCountPredicate? )?

⟨eventCountPredicate⟩ ::= '[' ( ⟨INT⟩ | ⟨IDENT⟩ ) ']'

⟨eventPatternRewriteExpr⟩ ::= ⟨stubNonSequencingExpression⟩
    | '{' ⟨sequencingExpression⟩ ';'? '}−'

⟨identPattern⟩ ::= 'pattern' PATTERN_IDENT;

⟨annotatedValueBindingPattern⟩ ::= ⟨valueModeAnnotation⟩? ⟨valuePattern⟩ ⟨valueBindingPatternAnnotation⟩?

⟨valueBindingPatternAnnotation⟩ ::= ⟨valueInterpretation⟩
    | '{' 'names' ⟨memberNameExpr⟩ '}'

⟨eventParameterNamesAnnotation⟩ ::= '{' 'names' ⟨namedMultiValueDescription⟩ '}'
    | ⟨empty⟩

⟨valuePattern⟩ ::= ⟨memberNameExpr⟩ '[' ⟨constantIntegerArithmeticExpression⟩ ']' ⟨valueInterpretation⟩?
    | ⟨memberNameExpr⟩ ⟨valueInterpretation⟩?
    | ⟨constantValueDescription⟩
    | 'void'
    | 'this'
    | 'pattern' ⟨PATTERN_IDENT⟩
    | 'pattern' ⟨patternConstantValueDescription⟩

⟨binding⟩ ::= ⟨bindingPrefix⟩ ⟨stubLangExpression⟩

⟨bindingPrefix⟩ ::= ⟨bindingKeyword⟩ ⟨bindableIdentSet⟩ '='
    | ⟨bindableIdentSet⟩ '<='

⟨bindableIdentSet⟩ ::= '(' ⟨bindableIdentWithOptionalInterpretation⟩
        ( ',' ⟨bindableIdentWithOptionalInterpretation⟩ )+ ')'
    | ⟨postfixExpression⟩ ⟨valueInterpretation⟩?
```

⟨*bindableIdentWithOptionalInterpretation*⟩ ::= ⟨*IDENT*⟩ ⟨*valueInterpretation*⟩?

⟨*bindingKeyword*⟩ ::= 'let' | 'out' | 'set'

⟨*valueCorrespondenceBlock*⟩ ::= 'values' '{' valueCorrespondence* '}'

⟨*singleValueCorrespondence*⟩ ::= 'values' ⟨*valueCorrespondence*⟩

⟨*valueCorrespondence*⟩ ::= ⟨*valueCorrespondenceBase*⟩ valueCorrespondenceTerminator;

⟨*valueCorrespondenceTerminator*⟩ ::= ';'
  | ⟨*valueCorrespondenceRefinement*⟩

⟨*valueCorrespondenceBase*⟩ ::= ⟨*valuePattern*⟩ ⟨*infixStubExpression*⟩ ⟨*leftToRightCorrespondenceOperator*⟩
  ⟨*infixStubExpression*⟩ ⟨*stubNonSequencingExpression*⟩
  | ⟨*stubNonSequencingExpression*⟩ ⟨*infixStubExpression*⟩ ⟨*rightToLeftCorrespondenceOperator*⟩
  ⟨*infixStubExpression*⟩ ⟨*valuePattern*⟩
  | ⟨*valuePattern*⟩ ⟨*infixStubExpression*⟩ ⟨*bidirectionalCorrespondenceOperator*⟩
  ⟨*infixStubExpression*⟩ ⟨*valuePattern*⟩
  | ⟨*singleOrNamedMultiValueDescription*⟩ ⟨*bidirectionalCorrespondenceOperator*⟩
  ⟨*singleOrNamedMultiValueDescription*⟩

⟨*singleOrNamedMultiValueDescription*⟩ ::= ⟨*memberNameExpr*⟩
  | ⟨*namedMultiValueDescription*⟩

⟨*correspondenceOperator*⟩ ::= ⟨*bidirectionalCorrespondenceOperator*⟩
  | ⟨*leftToRightCorrespondenceOperator*⟩
  | ⟨*rightToLeftCorrespondenceOperator*⟩

⟨*infixStubExpression*⟩ ::= '(' ⟨*stubNonSequencingExpression*⟩ ')'
  | ⟨*empty*⟩

⟨*bidirectionalCorrespondenceOperator*⟩ ::= '<–>'

⟨*leftToRightCorrespondenceOperator*⟩ ::= '–>'
  | '–>?'

⟨*rightToLeftCorrespondenceOperator*⟩ ::= '<–>'
  | '<–?'

⟨*valueCorrespondenceRefinement*⟩ ::= '{' ⟨*valueCorrespondence*⟩* '}' ';'

C.6 The stub language

This is a fairly conventional C-like grammar, with a few Cake additions.

⟨*stubLangExpression*⟩ ::= ⟨*sequencingExpression*⟩ /* lowest precedence operator */

⟨*stubLiteralExpression*⟩ ::= ⟨*STRING_LIT*⟩
  | ⟨*INT*⟩
  | ⟨*FLOAT*⟩
  | 'void'
  | 'null'
  | 'true'
  | 'false'

⟨*stubPrimitiveExpression*⟩ ::= ⟨*stubLiteralExpression*⟩
  | ⟨*setExpression*⟩
  | ⟨*IDENT*⟩
  | 'this'
  | 'that'

```
    |   'success'
    |   'out' ⟨IDENT⟩
    |   'out' '_'
    |   '(' ⟨stubNonSequencingExpression⟩ ')'
    |   '{' ⟨sequencingExpression⟩ ';'? '}'

⟨memberSelectionOperator⟩ ::=  '.' | '->' | '...'

⟨memberSelectionSuffix⟩ ::=  ⟨memberSelectionOperator⟩ IDENT

⟨postfixExpression⟩ ::=  ⟨stubPrimitiveExpression⟩ ( ⟨suffix⟩ )* '...'?

⟨suffix⟩ ::=  '(' ( ⟨stubLangExpression⟩ (',' ⟨stubLangExpression⟩ )* )? ')'
    |   ⟨memberSelectionOperator⟩ IDENT
    |   '[' ⟨stubLangExpression⟩ ']'

⟨unaryOperatorExpression⟩ ::=  (' ' | '!' | '-' | '+' | '*' | 'delete' | '&')* ⟨postfixExpression⟩
    |   'new' ⟨memberNameExpr⟩ ⟨namedMultiValueDescription⟩?

⟨tieExpression⟩ ::=  ⟨unaryOperatorExpression⟩ ( 'tie' ⟨postfixExpression⟩ )?

⟨castExpression⟩ ::=  ⟨tieExpression⟩ ⟨valueInterpretation⟩?

⟨multiplicativeOperatorExpression⟩ ::=  ⟨castExpression⟩ ( ( '*' | '/' | '%' ) ⟨castExpression⟩ )*

⟨additiveOperatorExpression⟩ ::=  ⟨multiplicativeOperatorExpression⟩ ( ( '+' | '-' ) ⟨multiplicativeOperatorExpression⟩ )*

⟨shiftingExpression⟩ ::=  ⟨additiveOperatorExpression⟩ ( ('<<' | '>>' ) ⟨additiveOperatorExpression⟩ )*

⟨magnitudeComparisonExpression⟩ ::=  ⟨shiftingExpression⟩ ( ( '<' | '>' | '<=' | '>=' ) ⟨shiftingExpression⟩ )?

⟨equalityComparisonExpression⟩ ::=  ⟨magnitudeComparisonExpression⟩
        ( ( '==' | '!=' ) ⟨magnitudeComparisonExpression⟩ )?

⟨bitwiseAndExpression⟩ ::=  ⟨equalityComparisonExpression⟩ ( '&' ⟨equalityComparisonExpression⟩ )*

⟨bitwiseXorExpression⟩ ::=  ⟨bitwiseAndExpression⟩ ( '^' ⟨bitwiseAndExpression⟩ )*

⟨bitwiseOrExpression⟩ ::=  ⟨bitwiseXorExpression⟩ ( '|' ⟨bitwiseXorExpression⟩ )*

⟨logicalAndExpression⟩ ::=  ⟨bitwiseOrExpression⟩ ( '&&' ⟨bitwiseOrExpression⟩ )*

⟨logicalOrExpression⟩ ::=  ⟨logicalAndExpression⟩ ( '||' ⟨logicalAndExpression⟩ )*

⟨conditionalExpression⟩ ::=  ⟨logicalOrExpression⟩
    |   'if' ⟨stubNonSequencingExpression⟩
        'then' ⟨stubNonSequencingExpression⟩ 'else' ⟨stubNonSequencingExpression⟩

⟨optionalBindingExpression⟩ ::=  ⟨binding⟩
    |   ⟨conditionalExpression⟩

⟨optionalLambdaExpression⟩ ::=  'fn' ⟨bindableIdentSet⟩ '=>' ⟨optionalLambdaExpression⟩
    |   ⟨optionalBindingExpression⟩

⟨stubNonSequencingExpression⟩ ::=  ⟨optionalLambdaExpression⟩

⟨sequencingExpression⟩ ::=  ⟨stubNonSequencingExpression⟩
        ( ⟨stubSequenceOperator⟩ ⟨stubNonSequencingExpression⟩ )*

⟨stubSequenceOperator⟩ ::=  ';'
    |   ';&'
    |   ';|'
```

Appendix D

Case study code

This Chapter includes full source for the case studies described in Chapter 5.

D.1 p2k

```
exists elf_archive("rump.a") kfs;
exists elf_archive("puffs.a") puffs;
derive elf_archive puffs_inst = instantiate(puffs, puffs_ops, pops, "puffs");
/* instantiate args: (component, structure type, structure name, symbol prefix) */
puffs_inst
{
    declare
    {
        puffs_fs_fhtonode : (_, _, _, out puffs_newinfo as puffs_full_newinfo) ⇒ _;
        puffs_node_lookup : (_, _,    out puffs_newinfo as puffs_full_newinfo, _) ⇒ _;
    }
}

derive elf_archive("user_kfs.a") fs = link[
    puffs,
    kfs
]
{ puffs ⟷ kfs {
    pattern /puffs_fs_(.*)/ { names (mount: _) } ⟷rump_vfs_\\1
                                                  { names (mount: _) };

    /* The above pattern generates (at most):
     * puffs_fs_{umount,statvfs,sync,fhtonode,nodetofh,suspend} */

    /* This one generates
     * puffs_node_{lookup,create,mknod,open,close,access,getattr,setattr,
     *   fsync,mmap,seek,remove,link,rename,mkdir,rmdir,symlink,readdir,
     *   readlink,read,write,inactive,reclaim}.
     */
    pattern /puffs_node_(.*)/ { names (mount: _, cookie: _) }
                         (/*.*/cookie as vnode_unlocked ptr)
        ⟷ RUMP_VOP_\\U\\1\\E { names (cookie: _) };

    values puffs_usermount (puffs_getspecific(this)) ⟶ mount;

    values puffs_cred ({
      puffs_cred_getuid(this, out uid) ;| let uid = 0;
      puffs_cred_getgid(this, out gid) ;| let gid = 0;
      puffs_cred_getgroups(pcr, out groups[NGROUPS], out ngroups)})
```

```
  ⟶(rump_cred_create(uid, gid, ngroups, groups)) kauth_cred;
values puffs_cred ⟵(rump_cred_destroy(this)) kauth_cred;

values vnode_unlocked ⟶({RUMP_VOP_LOCK(that, LK_EXCLUSIVE); that}) vnode;
values vnode_unlocked ⟵(RUMP_VOP_UNLOCK(that, 0)) vnode;

values puffs_cn ⟶(rump_makecn(that↪pcn_nameiop, that↪pcn_flags,
  that↪pcn_name, that↪pcn_namelen, that↪pcn_cred, curlwp)) component_name;
values puffs_cn ⟵(rump_freecn(this, RUMPCN_FREECRED)) component_name;

values vnode_bump_no_unlk ⟶({RUMP_VOP_LOCK(that, LK_EXCLUSIVE); rump_vp_incref(that); that}) vnode;
values vnode_bump_no_unlk ⟵({assert(RUMP_VOP_ISLOCKED(that) == 0); that}) vnode;

puffs_node_create(mount, vn as vnode_bump_no_unlk, ni, cn, vap)
⟶ RUMP_VOP_CREATE(vn, ni, cn, vap);
puffs_node_mknod(mount, vn as vnode_bump_no_unlk, ni, cn, vap)
⟶ RUMP_VOP_MKNOD(vn, ni, cn, vap);
puffs_node_remove(mount, vn as vnode_bump_no_unlk, targ_vn as vnode_bump_no_unlk, cn)
⟶ RUMP_VOP_REMOVE(vn, targ_vn, cn);
puffs_node_link(mount, vn as vnode_bump_no_unlk, targ_vn as void ptr, cn)
⟶ RUMP_VOP_LINK(vn, targ_vn, cn);
puffs_node_rename(mount, srcdir_vn as vnode_bump_no_unlk, src_vn as vnode_bump_no_unlk, src_cn,
                         targdir_vn as vnode_bump_no_unlk, targ_vn as vnode_bump_no_unlk, targ_cn)
⟶ RUMP_VOP_RENAME(srcdir_vn, src_vn, src_cn, targdir_vn, targ_vn, targ_cn);

puffs_node_mkdir(mount, vn as vnode_bump_no_unlk, ni, cn, vap)
⟶ RUMP_VOP_MKDIR(vn, out ni, cn, vap);
puffs_node_rmdir(mount, vn as vnode_bump_no_unlk, targ_vn as vnode_bump_no_unlk, cn)
⟶ RUMP_VOP_RMDIR(vn, targ_vn, cn);
puffs_node_symlink(mount, vn as vn_bump_no_unlk, ni, src_cn, vap, linktgt)
⟶ RUMP_VOP_SYMLINK(vn, out ni, cn, vap, linktgt);

// override name-matching rules
puffs_fs_fhtonode(mount, cookie, fid , _, _, out newvp as puffs_full_newinfo) ⟷
  rump_vfs_fhtovp(mount, fid, out newvp);
puffs_node_lookup(mount, cookie, out newvp as puffs_full_newinfo, cn)
    ⟷ RUMP_VOP_LOOKUP(cookie, out newvp, cn);

values puffs_newinfo ({puffs_newinfo_setcookie(this, that); this})
                    ⟵(RUMP_VOP_UNLOCK(this, 0)) vnode;
// Some calls return a fuller set of newinfo
values puffs_full_newinfo ({puffs_newinfo_setcookie(this, that);
                            puffs_newinfo_setvtype(this, vtype);
                            puffs_newinfo_setsize( this, vsize );
                             puffs_newinfo_setrdev(this, rdev); this})
    ⟵ ({let (vtype, vsize, rdev) = rump_getvninfo(this); this}) vnode;

values vnode_lkshared ⟶({RUMP_VOP_LOCK(that, LK_SHARED); that}) vnode;
values vnode_lkshared ⟵({RUMP_VOP_UNLOCK(that, 0); that}) vnode;

uio_outbuf:
values (buf: uint8_t[] ptr, resid : size_t, off : const off_t)
 ⟷ uio
{ void ⟶(rump_uio_setup(that↪buf, that↪resid, that↪offset, RUMPUIO_READ)) void; };

values uio_outresult ⟵(rump_uio_free(this)) uio;
values uio_outresult_subtract (*this - that)⟵(rump_uio_free(this)) uio;
values uio_outres_len_off
    ⟵({rump_uio_getresid(that↪resid);
        rump_uio_getoff(that↪readoff);
        rump_uio_free(this)}) uio;

puffs_node_read(mount, vn as vnode_lkshared,
    uio as uio_outbuf(buf, *resid , offset ),
    _, inout resid out_as uio_outresult, cr, ioflag ) // directional "as": {in,out}_as
⟶ RUMP_VOP_READ(vn, uio, ioflag, cr);
```

```
    puffs_node_readlink(mount, vn, cr, uio as uio_outbuf(linkname, linklen , 0),
        inout linkname out_as uio_outresult_subtract)
    ⟶ RUMP_VOP_READLINK(vn, uio, cr);

    puffs_node_readdir(mount, vn as vnode_lkshared, uio as uio_outbuf(dent, ∗reslen , ∗readoff ),
        /∗ readoff ∗/ _ out_as uio_outres_len_off(readoff, reslen ), /∗ reslen ∗/ _,
        cr , inout eofflag ,
        out cookies as ( invalid off_t )[ncookies ], /∗ ncookies ∗/ _)
        // TODO: check strange bug in p2k code: no size given for "cookies" buffer!
    ⟶ RUMP_VOP_READDIR(vn, uio, cr, eofflag, cookies, if cookies == null then null else out ncookies);
    // NOTE: Cake notices the mismtach between
    // (out _)[] and caller_frees(free) out(_ [])
    // and insert the necessary memcpy/free!

   uio_inbuf: values /∗uio_inbuf∗/ (buf: /∗(invalid ∗/uint8_t/∗)∗/[] ptr, resid : size_t , off : const off_t)
   ⟷ uio
   { void ⟵(rump_uio_setup(that↪buf, that↪resid, that↪offset , RUMPUIO_WRITE)) uio; };

    values uio_inresult ⟵(rump_uio_free(this)) uio;

    puffs_node_write(mount, vn, uio as uio_inbuf(buf, ∗resid , offset ),
        _, _ out_as uio_inresult, cr , ioflag )
    ⟶ RUMP_VOP_WRITE(vn, uio, ioflag, cr);

    // "inactive" notification requires special action in rump
    puffs_node_inactive(mount, vn as vn_no_lk) ⟶{
        rump_vp_interlock(vn);
        RUMP_VOP_PUTPAGES(vn, 0, 0, PGO_ALLPAGES);
        RUMP_VOP_LOCK(vn, LK_EXCLUSIVE);
        RUMP_VOP_INACTIVE(vn, out recycle) }--
     ⟵
     --{ if recycle then puffs_setback(
          puffs_cc_getcc(mount, PUFFS_SETBACK_NOREF_N1
        )) else void };

    // reclaim maps to call with non-analogous name
    puffs_node_reclaim(mount, vn as vn_no_lk) ⟶{ rump_vp_recycle_nokidding(vn); void };

    // unmount requires special action
    puffs_fs_unmount(mount, flags) (let rvp in_as vn_no_lk = puffs_getroot(mount)↪pn_data)⟶ {
        rump_vp_recycle_nokidding(rvp);
        rump_vfs_unmount(mount, flags) ;|
            { rump_vfs_root(mount, out rvp2, 0); assert(success && rvp == rvp2); } };

    // puffs sync needs two calls in rump
    puffs_fs_sync(mount, waitfor, cr) ⟶ { rump_vfs_sync(mount, waitfor, cr); rump_bioops_sync(); };

    // fhtonode and nodetofh map to non-analogous names
    puffs_fs_fhtonode(mount, fid, _, out ni as puffs_full_newinfo) ⟶
        rump_vfs_fhtovp(mount, fid, ni);

    puffs_fs_nodetofh(mount, vn as vnode_nolk, fid, fidsize ) ⟶ rump_vfs_vptofh(vn, fid, fidsize );

} };
```

D.2 ephy

```
derive elf_archive ("ephy+.a") ephy = instantiate(
     instantiate ( elf_archive ("ephy.a"), EphyCommandManagerIface, man_impl, ""),
         EphyEmbedIface, embed_impl, "");
/∗ instantiate args: (component, structure type, structure name, symbol prefix) ∗/
exists elf_external_sharedlib ("webkit") webkit
{
    declare
    {
         webkit_web_back_forward_list_get_forward_list_with_limits:
             (_) ⇒ GList_of_WebKitWebHistoryItems;
         WebKitEmbedLoadState: class_of enum {
             WEBKIT_EMBED_LOAD_STARTED;
             WEBKIT_EMBED_LOAD_REDIRECTING;
             WEBKIT_EMBED_LOAD_LOADING;
             WEBKIT_EMBED_LOAD_STOPPED;
         };
         WEBKIT_BACK_FORWARD_LIMIT: const int = 100;
         WebKitHistoryType: class_of enum {
             WEBKIT_HISTORY_BACKWARD;
             WEBKIT_HISTORY_FORWARD;
         };
    }
};
derive elf_exec ephy_webkit = make_exec(
    link [ephy, webkit]
    {
         ephy ⟷ webkit
         {
             /* Summary of history item handling.

              * webkit_construct_history_list is an embed function
              * which gets a webkit back_forward_list from webkit
              * then calls webkit again to get either a forward_list or a back_list
              * and then constructs a webkit_history_item for each element.
              * webkit_history_item is an ephy-specific class simply wrapping the underlying gobject.
              * Its getters use strdup.
              * A back_forward_list provides methods to get_ either a forward GList or
              * a back GList, containing WebKitWebHistoryItems */
             values GList_of_EphyHistoryItem ⟷GList_of_WebKitWebHistoryItem
             {
                 data as EphyHistoryItem ptr ⟷ data as WebKitWebHistoryItem ptr;
             };

             values
             {
                 /* history item handling */
                 // update rule
                 EphyHistoryItem (this↪data)⟶ WebKitWebHistoryItem;
                 // init rule only: we never send updates to webkit
                 EphyHistoryItem (*(webkit_history_item_new(that) tie this))⟵? WebKitWebHistoryItem;
                 /* raw (user-typed) URL handling */
                 raw_url ⟷ raw_url
                 {
                     pattern "(about:|(http[s]?| file |ftp ://)).*" ⟶(g_strdup(that) tie that) void;
                     pattern ".*" ⟶(g_strconcat("http://", that) tie that) void;
                 };

             }
             ephy_embed_factory_new_object(EPHY_TYPE_EMBED) ⟶g_object_new(EPHY_TYPE_BASE_EMBED);

             g_type_interface_peek(_, EPHY_TYPE_EMBED) ⟶{ ephy_impl };
             g_type_interface_peek(_, EPHY_TYPE_COMMAND_MANAGER) ⟶{ man_impl };

             values (embed: EphyBaseEmbed,
                     history : EphyHistory)  ⟷    (web_view: WebKitWebView,
                                                   scrolled_window: GtkScrolledWindow,
                                                   load_state: WebKitEmbedLoadState,
                                                   loading_uri: char [])
             {
                   void // history is a singleton, so always add it when forming the association
                  (let history = ephy_embed_shell_get_global_history(ephy_embed_shell_get_default()))
                  ⟶? ({ // from webkit_embed_init
                      let web_view = webkit_web_view_new();
                  let sw = gtk_scrolled_window_new(null, null);
                      gtk_scrolled_window_set_policy(sw,
                          GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);

                      gtk_container_add(sw, web_view);
                      gtk_widget_show(sw);
                      gtk_widget_show(web_view);
                      gtk_container_add(that, sw);
                     // set callbacks
                      g_object_connect(web_view,
                          "signal::load-committed",
                              fn (view, frame, embed) ⇒{
                                  ephy_base_location_changed(embed,
                                      webkit_web_frame_get_uri(frame))
                                  }, embed,
                          "signal::load-started",
                              fn (view, frame, embed) ⇒{
                                  if loading_uri != null then
                                      ephy_history_add_page(history, loading_uri, FALSE, FALSE)
                                  else void;
                                  ephy_base_embed_update_from_net_state(
                                      EPHY_EMBED_STATE_UNKNOWN | EPHY_EMBED_STATE_START
                                      | EPHY_EMBED_STATE_NEGOTIATING | EPHY_EMBED_STATE_IS_REQUEST
                                      | EPHY_EMBED_STATE_IS_NETWORK);
                                  }, embed,
                          "signal::load-progress-changed",
                              fn (view, progress, embed) ⇒{
                                  if load_state == WEBKIT_EMBED_LOAD_STARTED then
                                      { set load_state = WEBKIT_EMBED_LOAD_LOADING } else void;
                                  ephy_base_embed_set_load_percent(embed, progress)
                                  }, embed,
                          "signal::load_finished",
                              fn (view, frame, embed) ⇒{
                                  set load_state = WEBKIT_EMBED_LOAD_STOPPED;
                                  ephy_base_embed_update_from_net_state(
                                      EPHY_EMBED_STATE_UNKNOWN | EPHY_EMBED_STATE_STOP
                                      | EPHY_EMBED_STATE_IS_DOCUMENT | EPHY_EMBED_STATE_IS_NETWORK)
                                  }, embed,
                          "signal::title-changed",
                              fn (view, frame, title, embed) ⇒{
                                  ephy_base_embed_set_title(embed, title);
                              }, embed,
                          "signal::hovering-over-link",
                              fn (view, frame, embed) ⇒{
                                  ephy_base_embed_set_link_message(embed, location);
                                  }, embed, s,
                          null);

                      // associate preferences somehow
                      webkit_web_view_set_settings(web_view, settings) // "settings" is a global

                      }) void;
             };

             ephy_manager_do_command(man, "cmd_copy") ⟶webkit_web_view_copy_clipboard(man...web_view);
             ephy_manager_do_command(man, "cmd_cut") ⟶webkit_web_view_cut_clipboard(man...web_view);
             ephy_manager_do_command(man, "cmd_paste") ⟶webkit_web_view_paste_clipboard(man...web_view);
             ephy_manager_do_command(man, "cmd_selectAll") ⟶webkit_web_view_select_all(man...web_view);

             ephy_manager_do_command(man, "cmd_copy") ⟶webkit_web_view_can_copy_clipboard(man...web_view);
             ephy_manager_do_command(man, "cmd_cut") ⟶webkit_web_view_can_cut_clipboard(man...web_view);
             ephy_manager_do_command(man, "cmd_paste") ⟶webkit_web_view_can_paste_clipboard(man...web_view);
             ephy_manager_do_command(man, _) ⟶{ false };

             ephy_load_url(embed, url) ⟶ webkit_web_view_open(embed...web_view, url);

             ephy_load(embed, url as raw_url, flags, preview_embed)
                 ⟶ {     set embed...loading_url = url; // hmm, is "let" the right behaviour?
```

```
                    webkit_web_view_open(embed...web_view, url) };

        ephy_stop_load(embed) ⟶webkit_web_view_stop_loading(embed...web_view);
        ephy_can_go_back(embed) ⟶webkit_web_view_can_go_back(embed...web_view);
        ephy_can_go_forward(embed) ⟶webkit_web_view_can_go_forward(embed...web_view);
        ephy_can_go_up(embed) ⟶{ false };
        ephy_get_go_up_list(embed) ⟶{ set [] }; // should really be list...
        ephy_go_back(embed) ⟶webkit_web_view_go_back(embed...web_view);
        ephy_go_forward(embed) ⟶webkit_web_view_go_forward(embed...web_view);
        ephy_go_up(embed) ⟶{ void };
        ephy_get_js_status(embed) ⟶{ "" };

        // toplevel is ignored
        ephy_get_location(embed, toplevel) ⟶ {
            g_strdup(webkit_web_frame_get_uri(webkit_web_view_get_main_frame(embed...web_view))); };

        ephy_reload(embed, force) ⟶ webkit_web_view_reload(embed...web_view);

        // some functions are no-ops
        pattern /ephy_(set_zoom|scroll_lines|scroll_page| scroll_pixels |shistory_copy)/
             (...)  ⟶ { void }; // silent ignore
        ephy_get_zoom(embed) ⟶{ 1.0 }; // no zoom support in webkit

        ephy_get_security_level(embed, out level,  description )
            (let unknown = EPHY_EMBED_STATE_IS_UNKNOWN)⟶ { out level = unknown; };

        // more silent ignore
        pattern /ephy_(show_page_certificate|print|.∗print_preview.∗|set_encoding|get_encoding)/
             (...)  ⟶ { void }; // silent ignore: "return 0" for preview_n_pages is inferred
        ephy_has_automatic_encoding(_) ⟶{ false };
        ephy_has_modified_forms(_) ⟶{ false };

        pattern /ephy_get_(((back)|(forward))((ward)?))_history/ (embed) ⟶ {
            // get a pointer to the WebKit-internal list
            let full_bf_list = webkit_web_view_get_back_forward_list(embed...web_view);
            // copy out the portion we want into a GList
            let copied_sublist = webkit_web_back_forward_list_get_\\1\\4_list_with_limits(
                full_bf_list ,
                WEBKIT_BACK_FORWARD_LIMIT)
            } ;
        ephy_get_next_history_item(embed) ⟶{
            webkit_web_back_forward_list_get_forward_item(
                webkit_web_view_get_back_forward_list(embed...web_view)
                )
        };
        ephy_get_previous_history_item(embed) ⟶{
            webkit_web_back_forward_list_get_back_item(
                webkit_web_view_get_back_forward_list(embed...web_view)
                )
        };
        ephy_go_to_history_item(embed, item) ⟶
            webkit_web_view_go_to_back_forward_item (embed...web_view, item);
    } // end ephy ⟷ webkit
  } // end link
); // end make_exec
```

D.3 xcl

```
exists elf_archive ("rxvt.a") client_of_xlib
{
    declare {
        XFillRectangle : (dpy: Display_unlocked ptr,  ...) ⇒ _;
        XGetGeometry: (dpy: inout Display, d: Drawable, root: Window ptr,
            x: out int, y: out int, width: out unsigned, height: out unsigned,
            borderWidth: out unsigned, depth: out unsigned) ⇒ _;
        XTranslateCoordinates:  (...,  child : out Window) ⇒same_screen : _; // named return value!
        XQueryTree: (_, _, root: out Window, parent: out Window,
            children : out Window[nchildren], out nchildren ) ⇒ _;
    }
};

exists elf_archive(" libxcb.a ") xcb_library;
exists elf_archive(" xcl_util.a ") xcl_util
{
    declare {
        _XFlushGCCache: (dpy: Display_locked ptr, ...) ⇒ _;
        XCBPolyFillRectangle: (dpy: Display_locked ptr,  ...) ⇒ _;
        XCBPolyRectangle: (dpy: Display_locked ptr,  ...) ⇒ _;
        encap_string: class_of struct {
            len: size_t;
            bytes: char ptr;
        };
    }
};
alias any [xcb_library, xcl_util] xcb;

derive elf_exec(" rxvt_static_xlib ") output = link[
    client_of_xlib, xcb
    ]
{
    client_of_xlib ⟷ xcb
    {
        values
        {
            Display_unlocked  ⟶({LockDisplay(that);  that}) Display_locked;
            Display_unlocked ⟵  ({UnlockDisplay(that);  that}) Display_locked;

            Display_unlocked  ⟶({LockDisplay(that);
                                 XCBConnectionOfDisplay(that)}) XCBConnection;
            Display_unlocked ⟵  ({UnlockDisplay(that);  void}) XCBConnection;

            Display           ⟶(∗XCBConnectionOfDisplay(that)) XCBConnection;

            pointer_to_chars ⟶ length_prefixed_string
            {
                void ⟶( if ∗that then strlen(that) else 0) len;
                void ⟶(that) bytes;
            };

            pattern /Window|Pixmap|Cursor|Font|GContext|Colormap|Atom/ ⟷\\U\\1\\E
            { void ⟶(∗that) xid };
            pattern /VisualID|Keysym|Keycode/ ⟷\\U\\1\\E
            { void ⟶(∗that) id };
            Time ⟷ TIMESTAMP
            { void ⟶ (∗that) id };
            CARD8 ⟷BUTTON
            { ∗this ⟷ id };
            Drawable ⟷ DRAWABLE
            { void ⟶ (∗that) window.xid };
            Fontable ⟷ FONTABLE
            { void ⟶ (∗that) font.xid };

        } // end values

        cvtINT16: values unsigned ⟵  INT16 // cvtINT16toInt
        { val ⟵( if ((val) & 0x00008000) then ((val) | 0 xffffffffffff0000 ) else (val)) val };

        XSync(dpy, discard)      ⟶    {     XCBSync(dpy, 0);
                                            if discard then XCBEventQueueClear(dpy) else void;
                                            true
                                         };
```

```
_XFlush ⟶ XCBFlush;
XFlush(dpy) ⟶ { XCBFlush(dpy); true };

values _ ⟵ pattern /XCB(.*)Rep/ // Cake compiler will define an Xlib-side struct...
  // ... with an arbitrary name
{
  // Xlib calls sometimes have a "root" Window output parameter...
  // typedef CARD32 XID; typedef XID Window;
  // ... while in XCB..Rep structs, it's a WINDOW
  // typedef struct WINDOW { CARD32 xid; } WINDOW;
  root ⟵ root.xid ;

  // similar
  child ⟵ child.xid ;

  // identifier styles differ
  borderWidth ⟵ border_width;
};

/* This is the pattern that should capture the "common case" XCB calls.
 * In general, this sort of pattern should be upgraded into a style. */
pattern /X(.*)/(dpy, ...) ⟶
            { let reply = XCB\\1Reply(dpy, XCB\\1(dpy, in_args ... )) ;&
                // our output parameters are packed into a struct
                // which we would like to treat like a normal struct, say:
                { out out_args... = *reply;
                    free(r); true }
                ;| false ;
            };

/* Our QueryTree handling omits the hacky free-avoidance
 * "reuse the memory chunk the reply came in" done by XCL. I don't
 * see why this matters... our "free" is generated. */
XLoadFont(dpy, name as pointer_to_chars) ⟶{ let f = XCBFONTNew(dpy);
                                            XCBOpenFont(dpy, f, name.length, name.bytes);
                                            f.xid };

XRecolorCursor(dpy, cursor, fg, bg) ⟶ { XCBRecolorCursor(dpy, cursor,
                                            fg↪red, fg↪green, fg↪blue,
                                            bg↪red, bg↪green, bg↪blue); true };

pattern /X((Draw)|(Fill))Rectangle/(dpy, d, gc, x, y, w, h) ⟶
                                                { let p_r = new rectangle (x: x, y: y, w: width, h: height);
                                                FlushGC(that.dpy, dpy);
                                                XCBPoly\\2Rectangle(dpy, d, gc↪gid, 1, p_r);
                                                delete p_r;
                                                true };


XFree(p) ⟶ { Xfree(p); true };                          // vvv array constructor
XRaiseWindow(dpy, w) ⟶{ XCBConfigureWindow(dpy, w, CWStackMode, Above); true};
XLowerWindow(dpy, w) ⟶{ XCBConfigureWindow(dpy, w, CWStackMode, Below); true};

XAllocColor(dpy, cmap, def) ⟶ { let r = XCBAllocColorReply(dpy, XCBAllocColor(dpy, cmap,
                                            def↪red, def↪green, def↪blue), 0);
                                out def = r; // def is inout; invoke the value corresp
                                free(r); // we need to do this because
                                true }; // the reply *doesn't* flow back

XFreeGC(dpy, gc) ⟶ {     LockDisplay(that.dpy); // need to unlock before exit
                     for_each(that.dpy↪ext_procs,
                       fn ext ⇒ if (ext↪free_GC)
                                then (*ext↪free_GC)(dpy, gc, &ext↪codes)
                                else void
                     );
                     UnlockDisplay(dpy);
                     XCBFreeGC(dpy, gc↪gid);
```

```
                                  _XFreeExtData(gc↪ext_data);
                                  Xfree(gc); true };

     XmbTextListToTextProperty(...) ⟶ { XLocaleNotSupported };

     XChangeGC(dpy, gc, mask, vals) ⟶ {      let new_mask = mask & (1 << (GCLastBit + 1)) -1;
                                          if new_mask then _XUpdateGCCache(gc, new_mask, vals) else void;
                                          if (gc↪dirty & (GCFont | GCTile | GCStipple))
                                             then _XFlushGCCache(dpy, gc)
                                             else void;
                                         true };
     XStoreName(dpy, w, name as pointer_to_chars) ⟶
        { XCBChangeProperty(dpy, PropModeReplace, w, XA_WM_NAME, XA_STRING, 8, name.length, name.bytes); true };

     XChangeProperty(dpy, w, prop, type, format, mode, data, nelements) ⟶
        {  let legal_args = nelements < 0 && (format == 8 || format == 16 || format == 32);
           let new_nelements = if legal_args then nelements else 0;
             let new_format = if legal_args then format else 0;
           XCBChangeProperty(dpy, mode, w, prop, type, new_format, new_nelements, data);
             true };

     // xcl does this by call-around to its own just-now-defined XChangeProperty,
     // but we can't do this in Cake, so we have to expand it directly.
     XSetIconName(dpy, w, icon_name as pointer_to_chars) ⟶
        {     XCBChangeProperty(dpy, mode, w, XA_WM_ICON_NAME, XA_STRING, 8, icon_name.bytes, icon_name.length);
             true };

     pattern /XGrab(Pointer|Keyboard)/ (dpy, w, ownerEvents, ...) ⟶
        {     let ret = XCBGrab\\1Reply(dpy, XGrab\\1(c, ownerEvents, w, in_args...), 0);
            let status = if ret then ret↪status else GrabSuccess;
             free(r); status };

     // another case of "was call-around"
     XSetWMProtocols(dpy, w, protocols, count) ⟶ { let prop = XInternAtom(dpy, "WM_PROTOCOLS", False);
                                                   if prop == None then False else {
                                                      XCBChangeProperty(dpy, PropModeReplace,
                                                          w, prop, XA_ATOM, 32, count, protocols);
                                                      True
                                                    }
                                                  };

     XSetTextProperty(dpy, w, tp, property) ⟶ XCBChangeProperty(dpy, PropModeReplace,
                                               w, property, tp↪encoding,
                                               tp↪format, tp↪nitems, tp↪value);
     XSetWMName(dpy, w, tp) ⟶XCBChangeProperty(dpy, PropModeReplace,
                                               w, XA_WM_NAME, tp↪encoding,
                                               tp↪format, tp↪nitems, tp↪value);
     XSetWMIconName(dpy, w, tp) ⟶XCBChangeProperty(dpy, PropModeReplace,
                                               w, XA_WM_ICON_NAME, tp↪encoding,
                                               tp↪format, tp↪nitems, tp↪value);
     XSetWMClientMachine(dpy, w, tp) ⟶XCBChangeProperty(dpy, PropModeReplace,
                                               w, XA_WM_CLIENT_MACHINE, tp↪encoding,
                                               tp↪format, tp↪nitems, tp↪value);
   } // end pairwise
}; // end link
```

Appendix E

Simple dynamic points-to analysis

This chapter describes the dynamic points-to analysis used by the Cake runtime. As described in Chapter 4, this is a simple implementation designed to avoid both explicit annotation and the need for binary data-flow analysis. However, it is an imprecise analysis, in that it occasionally fails to find a unique solution, and was created essentially for proof-of-concept purposes only. It is also specialised for compiled C code, in ways we will describe.

E.1 The problem

Recall that the Cake language has dynamic binding (§2.4.3). This means that it must apply rules that are appropriate to the actual run-time objects that are passed around between components. It is not sufficient to use a static approximation to this, such as the static type information that is used by C compilers and translated into DWARF information.

Uncertainty about which rules might bind to an object is limited to objects that are reached through pointers. We do not worry about undiscriminated unions or obscure packing practices, following earlier exclusions and "well-behavedness" assumptions which effectively ruled those cases out (§2.4.3).

Given a pointer into some memory in a *well-behaved* program, we therefore wish to determine the "type" of that memory. This must be sufficient to describe not just the memory directly pointed to, but other memory from which some well-behaved code receiving that pointer might legitimately *adjust* the pointer to access, by making *admissible reinterpretations.*

Here we use "type" to refer to the abstract view of that memory which was defined by the abstract data type for which the memory was allocated. Our chief difficulty is that for C code, this is never recorded anywhere at run time, because the memory allocation function, malloc(), is generic.

(Unlike much of the rest of this dissertation, here we use "type" rather than "class" or "description", to better fit in with DWARF terminology.)

E.2 Approach

We generalise the problem into discovering a precise type for the entire allocation into which the pointer points. This means the malloc()-allocated heap block. Note that this question yields more information than the original question, since cases exist where admissible reinterpretations do not allow exploring the whole heap block. However, for our purposes, the more general question is usually no more difficult to answer in practice.

As described in §4.3.2, discovering precise types for static- and stack-allocated memory is straightforward. These can be distinguished by their characteristic address ranges on any given system. We therefore only concern ourselves with heap allocations here.

We make a number of further assumptions.

- Heap allocations will only use the array layouts and object containment structures permitted in the C language. This reflects that C is the most commonly used language to have an untyped interface to memory allocation. Other languages with similar treatment of allocation can likely be accommodated with minimal changes, but we do not consider these here.

- *Imprecise* static type information is available for the pointer. This means type information of the kind generally present in C compiler output. This may be lacking in other languages, or in use of void pointers. We assume void pointers have been annotated with a sufficiently precise type that only admissible reinterpretations are subsequently done on the pointer. This is reasonable for C code, and will be a source of valuable constraints to our analysis.

- Each allocated block holds exactly one abstract data type, or an array of elements of the same type (and hence the same size). This is occasionally violated where, say, pairs of structures are allocated together, in which case a C programmer might do, say, malloc(sizeof(struct X) + sizeof(struct Y)). This can be overcome by an annotation declaring a new aggregate struct type in the relevant compilation unit, consisting of struct X and struct Y fields at the relevant offsets.

- The allocation site of the object determines its precise type. That is, only one type of object is allocated by any particular malloc() call, although we allow for arrays of any length. (To deal with discriminated unions, we may relax this by allowing the object's *contents* to affect the type, i.e. by inspecting a discriminant field. This need not be limited to data types actually declared as unions: padded structs such as sockaddr in the Unix sockets API [IEEE POSIX, 1988] also fit this discriminated pattern, and could be captured by simple annotations.) Although C does not support explicitly annotating that a particular field in an enclosing structure serves as a discriminant, the Cake programmer can annotate this separately by re-describing the whole structure as a Pascal-style variant record, which is supported by DWARF.

- There *is* a well-defined precise type. Even in complex cases, such as temporally discriminated unions, we hold (philosophically speaking) that there is always a well-defined type, i.e. a distinction between memory accesses that respect the abstraction and ones that do not. This definition may be arbitrarily complex, but in practice only a few patterns are in use.

- The precise DWARF type is defined in the relevant compilation unit's DWARF information. This is usually true, but again, violations are possible, in which case the Cake programmer is expected to provide an annotation. For example, in our experience, the GNU C compiler will always emit debugging information for a type T if a variable, field or parameter of type T or T* is used within the compilation unit. However, if the code does only sizeof T, the information will not be emitted unless special compiler options are used. In practice, this means that "factory functions" which call malloc() but simply pass the returned pointer onwards under an imprecise static type, without accessing the memory, may not contain the relevant information and may require annotation.

Given these assumptions, our approach has three parts.

Firstly, we instrument (or extend) the allocator so that given a pointer pointing anywhere within any heap block, we can discover the start address, size and *allocation site* of that block. This is a strong requirement, and brings some time and space overhead.

Secondly, we enumerate a set of candidate typings of that block, using arithmetic constraints and the set of data types defined in the DWARF compilation unit containing the allocation site.

Thirdly, we eliminate candidate typings other than the correct one, using additional constraints from various sources. The primary source is the imprecise static type attached to the input pointer, and its offset from the start of the heap block—these must be consistent with the typing. Other sources of constraints supplement this in some cases.

We now describe each of these parts in more detail.

E.2.1 Allocator instrumentation

Our key requirement of allocator instrumentation is that a heap block's size can be discovered given a pointer anywhere into that block. By "heap block size" we mean the size of an individually allocated user data region, i.e. the exact size passed to malloc().

Since a typical high-performance malloc() implementation keeps an index only of free regions, and not used regions [Wilson et al. 1995], this is nontrivial. Pointers to the start of a block can reveal the block size, usually stored in the preceding word, but even this is not sufficient, since this size may have been rounded up to satisfy the allocator's alignment. In any case, since we require an analysis that works given a pointer to *anywhere* in the block, we require additional metadata.

We could opt to extend the data structures kept by the host malloc() implementation so that precisely-sized records are kept on each allocated block (preferably indexed by address). This might enable more optimisations than a purely instrumentation-based approach, since synergies with existing bookkeeping data structures can be exploited.

For now, we do not pursue this option, partly because performance is not an explicit goal, and partly because *custom* allocators also require analogous modifications. Our purely instrumentation-based approach allows this instrumentation to be re-used.

Our current implementation stores $(address, size)$ pairs in a separate structure.[1] This also makes it straightforward for custom allocators (i.e. other than the host allocator) to be instrumented in the same way (a requirement discussed previously in §4.3.2).

Note that only malloc() call-sites allocating objects which might be passed across a Cake-composed interface need to use the instrumented allocator. For others, perhaps allocating internal object structures within a performance-critical portion of a library, the original higher-performance malloc() can be used directly.

E.2.2 Enumerating possible typings

Any DWARF type defined in the allocating compilation unit, and whose size is a *factor* of the heap block size, is initially a candidate. In other words, the block may be either a singleton or an array of the type.

E.2.3 Constraints

We begin with a list of typings derived from factorisations of the block size s. Each factorisation is a pair, (n, τ), meaning that the block could represent an array of n instances of τ. If s has many factors, there will be more candidate types left in consideration. In practice, s will be a multiple of the word size, so at least any word-sized types or fractions thereof (bytes, half-words etc.) will remain.

The imprecise static type attached to the input pointer, and its offset from the start of the heap block together define a *containment constraint*: if the heap block starts at address A, a candidate type is T and the input pointer is $A + o$ with imprecise static type τ, then if T does not contain a τ at offset o, T may be ruled out. This often sufficient to determine a unique precise type.

Pointers to arrays are often accompanied by a *length* value for that array. Relevant Cake annotations (§2.3.8) can link the array with the length. Therefore, when following pointers that have been annotated in this way, the Cake runtime has valuable additional constraint which is usually sufficient to precisely determine the type of the block. This is because we assume pointers into arrays are precisely typed if they are typed at all (§4.3), which is reasonable since without knowing the precise element size, correct pointer arithmetic is not possible.

Fig. E.1 shows an array of three FooWindow objects allocated in a heap block, and three different input pointers. In the first case, the pointer is precisely typed except for lacking the array bound. Since the size of FooWindow divides the size of the heap block

[1]Currently we use a Google Sparse Hash, http://code.google.com/p/google-sparsehash/, but a custom data structure could yield a better time–space trade-off.

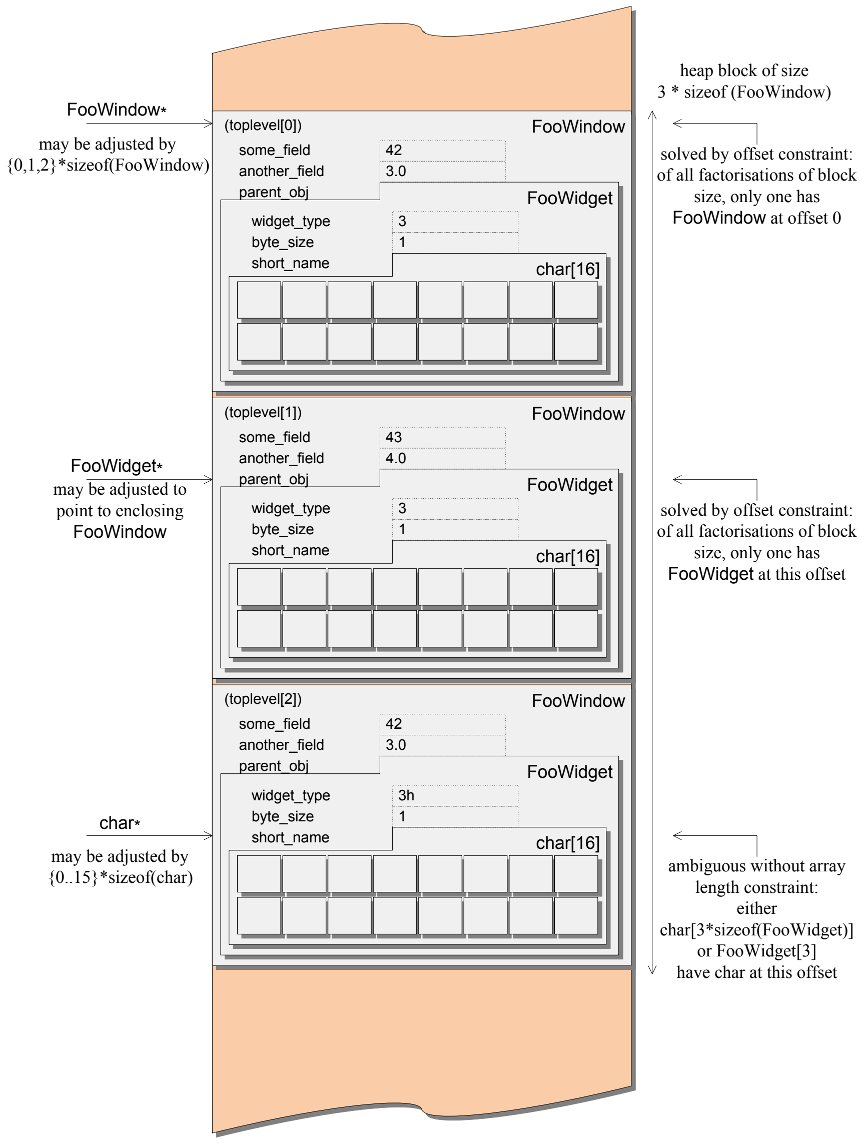

FooWindow*
may be adjusted by {0,1,2}*sizeof(FooWindow)
FooWidget*
may be adjusted to point to enclosing FooWindow
char*
may be adjusted by {0..15}*sizeof(char)
(toplevel[0])
FooWindow
some_field 42
another_field 3.0
parent_obj
FooWidget
widget_type 3
byte_size 1
short_name
char[16]
(toplevel[1])
some_field 43
another_field 4.0
(toplevel[2])
widget_type 3h
heap block of size 3 * sizeof (FooWindow)
solved by offset constraint: of all factorisations of block size, only one has FooWindow at offset 0
solved by offset constraint: of all factorisations of block size, only one has FooWidget at this offset
ambiguous without array length constraint: either char[3*sizeof(FooWidget)] or FooWidget[3] have char at this offset

Figure E.1: Examples of a heap allocation with complex internal structure

in exactly one way, this is trivially solved *provided* that no type *containing* a FooWindow at offset zero has a size which also divides the block size. In the second case, a FooWidget pointer points at the relevant offset within the second element of the array. This will be correctly solved so long as no other structures than a FooWindow[3] *both* contain a FooWidget at this offset *and* have overall size matching the heap block size. Like the previous case, this condition is highly likely. In the third case, we are out of luck: both the actual structure and the candidate typing char[m], where m is the size in bytes of three FooWindows, satisfy the containment constraint. In these cases we must rely on an array constraint. We discuss this case in §E.4.

A trivial but useful kind of constraint is the "previously seen" constraint: once we deduce an unambiguous typing for a given heap block, it is stored in our heap metadata for later retrieval. This means that if we encounter a less precise pointer subsequently, we are nevertheless sure to have a precise solution.

A tentative idea for addressing difficult cases is exploiting "previously seen" in an expensive but powerful manner. By scanning the stack and static storage, we can potentially discover other pointers into the heap object than the input pointer or those previously seen. It is likely that one of these is precisely typed. Therefore, if the analysis fails, we can force such an analysis—perhaps piggy-backed on top of a garbage collector, as proposed for various other dynamic analyses [Reichenbach et al. 2010]—until it hits such a pointer. Although expensive, this cost is potentially greatly amortised, because the collector may discover precisely-typed pointers for many other objects at the same time.

E.3 Cake's tolerance of imprecision

The analysis we have shown so far is clearly imprecise. In many cases, however, the Cake language can tolerate imprecision.

When using array length constraints, the analysis is effectively redundant: the Cake programmer has given us both a precise element type and a precise array size. Therefore, it doesn't matter if the analysis doesn't yield a unique answer—all the Cake runtime needs to know is how much of the heap block to apply correspondences to, which is a given. Note, however, that in practice it may also yield a precise typing for the whole block—e.g. in the example, where perhaps few or no other types contain an array of 16 chars, hence allowing the deduction that the block is a FooWidget[3]—and this should be remembered for future use. This will not succeed in some cases, however: for example, if the programmer only wants to pass the first 8 characters to the called function, out of a larger string, the analysis will not discover any typings.

Another case is where the pointed-to object is of an opaque or shareable type—in this case, the runtime does not care whether the analysis is ambiguous. Rather, the pointer can be given to the receiving component as-is, with no need to copy and convert any data. For example, if a char* is passed and this yields an ambiguous typing for the block (say, it might be one big array of chars or a pointer into some smaller, contained array of chars), this does not matter so long as char is shareable with the receiving component. Similarly, uncorresponded types are effectively opaque so the runtime need not apply any correspondences.

Similarly, if no admissible reinterpretation could reach a more specific type of object for which *different* Cake correspondences are defined than for the imprecise type, then the imprecise type suffices. For example, if no Window rules are defined, nor rules for any other type containing Widget, then knowing that the pointed-to object is a Widget suffices for making the appropriate rule selection..

If the analysis fails, Cake issues a run-time warning and proceed with an imprecise result. Even in cases beyond those enumerated above, this does not always result in incorrect behaviour. (In short, this is because even if it would be admissible to reinterpret the pointer to some containing type, this *need not* happen in any given execution.)

E.4 Difficult cases

Our analysis is prone to quirks of factorisation. As a simple example, imagine passing the bottom (char) pointer, in Fig. E.1, *without* knowing the length of the array to which it points. There is no way to distinguish a whole block of char from simply the contained array of char. We rely on explicit passing of array size, with appropriate Cake annotations, to catch these cases.

More generally, this problem occurs where we have a subobject whose size divides a superobject's size, and whose size also divides the subobject's offset within the superobject. Our analysis cannot tell the difference between a heap block containing a single instance of the superobject, and an array of some number of the subobject type. For example, if a Widget is 48 bytes, a Window is 96 bytes, and the contained Widget lies at offset 48, then we cannot tell the difference between a pointer to the contained Widget and a pointer into the second element in an array of two Widgets.

Note that relaxing any part of this will yield a solution. If Widget or Window's sizes change so that the smaller no longer divided the larger, the difficulty does not occur. Similarly, if the size of Widget no longer divides its offset within Window, the problem does not occur. Therefore, this is rarely a problem except for small-sized subobjects, like the chars in the example. Fortunately, small-sized subobjects, being primitives, are very often shareable in Cake, making the ambiguity irrelevant. Meanwhile, in the case of larger contained objects, containment is often constrained to be at offset 0, or else (in the case of multiple inheritance) at a larger offset that is unlikely to be divisible by the subobject's size.

In the case of the Cake runtime, we resolve these situations by emitting a warning (as before) and guessing in favour of solutions involving contained objects and avoiding solutions involving arrays. This is because it is the more conservative option: it exposes less data to Cake correspondences, so minimises the likelihood of performing a meaningless operation. It is also arguably appropriate because array pointers are rarely passed at nonzero offsets.

E.5 Summary

The analysis presented in this section has proved sufficient for our needs, and in particular, does not cause issue with any of the examples in Chapter 5. However, a production-ready implementation of Cake would no doubt adopt the binary analysis approach described in §4.3.2. The approach described in this Appendix could be a useful complement for the minority of cases where such analysis remains imprecise.

Bibliography

F. Achermann and O. Nierstrasz. Applications = components + scripts. In *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proceedings of the 16th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 108–124, 2001. ISBN 1-58113-335-9.

F. Arbab. What do you mean, coordination. *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, 1998.

F. Arbab and F. Mavaddat. Coordination through channel composition. In *Proc. Coordination*, pages 21–38. Springer, 2002.

K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley Professional, 4th edition, 2005. ISBN 0321349806.

U. Assmann, T. Genssler, and H. Bar. Meta-programming grey-box connectors. In *Proceedings of 33rd International Conference on Technology of Object-Oriented Languages (TOOLS 33)*, pages 300–311, 2000.

G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *Proceedings of 14th International Conference on Compiler Construction*, pages 250–254, 2005.

G. Banavar, G. Lindstrom, and D. Orr. Type-safe composition of object modules. Technical Report UUCS-94-001, University of Utah, Salt Lake City, Utah, USA, 1994.

D. Beazley. Swig: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.

K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN 0321278658.

A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Classboxes: controlling visibility of class extensions. *Computer Languages, Systems & Structures*, 31:107–126, 2005.

A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2:39–59, 1984.

A. P. Black. An asymmetric stream communication system. In *Proceedings of the ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 4–10, New York, NY, USA, 1983. ACM. ISBN 0-89791-115-6.

S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML query language. W3C working draft, 2002.

B. Boehm and B. Scherlis. Megaprogramming. In *Proceedings of the DARPA Software Technology Conference*. Meridien Corp., Arlington, VA, USA, April 1992.

H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988. ISSN 0038-0644.

J. Bosch. Superimposition: a component adaptation technique. *Information and Software Technology*, 41:257–273, 1999.

A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74:45–54, 2005.

G. Bracha, C. Clark, G. Lindstrom, and D. Orr. Module management as a system service. In *OOPSLA Workshop on Object-oriented Reflection and Metalevel Architectures*, 1993.

G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-oriented Programming and International Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM. ISBN 0-201-52430-X.

J. Callahan. *Software packaging*. PhD thesis, University of Maryland, 1993.

J. Callahan and J. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering*, 17:626–635, 1991.

N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32:444–458, 1989.

M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, and S. Panne. The Haskell 98 foreign function interface 1.0: an addendum to the Haskell 98 report. URL http://www.cse.unsw.edu.au/%7echak/haskell/ffi/. Retrieved on 2010/12/13.

C. Chambers. Object-oriented multi-methods in Cecil. In O. Madsen, editor, *ECOOP '92 European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer Berlin / Heidelberg, 1992.

S. Chiba. Load-time structural reflection in Java. In *Proceedings of the European Conference on Object-Oriented Programming*, 2000.

C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 130–145, New York, NY, USA, 2000. ACM. ISBN 1-58113-200-X.

G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX Annual Technical Conference*, page 14, 1998.

B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, 9 2005.

R. DeLine. Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering*, 27:124–143, 2001.

R. DeLine. Avoiding packaging mismatch with flexible packaging. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 97–106, New York, NY, USA, 1999. ACM. ISBN 1-58113-074-0.

C. Dellarocas. The SYNTHESIS environment for component-based software development. In *Proc. 8th International Workshop on Software Technology and Engineering Practice*, page 434, Los Alamitos, CA, USA, 1997. IEEE Computer Society. ISBN 0-8186-7840-2.

L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 151–170, London, UK, 1987. Springer-Verlag. ISBN 3-540-18353-1.

F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software*, pages 114–121, 1975.

S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. United States Patent 6397242, May 2002.

D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: a tool for generating binary adapters for evolving Java libraries. In *Proceedings of the 30th International Conference on Software Engineering*, pages 963–964. ACM, 2008.

E. Dijkstra. *A discipline of programming.* Prentice Hall, 1976.

A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD 2001, pages 509–520, New York, NY, USA, 2001. ACM. ISBN 1-58113-332-4.

D. Dougherty and A. Robbins. *Sed and Awk.* O'Reilly Media, Inc., 1997.

E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: a flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 44–56, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6.

S. Eisenbach, C. Sadler, and D. Wong. Component adaptation in contemporary execution environments. In *Proceedings of 7th IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer-Verlag, 2007.

B. Ellis, J. Stylos, and B. Myers. The Factory Pattern in API design: A usability evaluation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7.

M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-51459-1.

Y. Eterovic, J. Murillo, and K. Palma. Managing components adaptation using aspect oriented techniques. In C. Canal, J. Murillo, and P. Poizat, editors, *Proceedings of the First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT '04)*, 2004. ISBN 84-688-6782-9. Technical Report of the Universities of Málaga, Extremadura and Évry.

S. I. Feldman. Make: a program for maintaining computer programs. *Softw: Pract. Exper.*, 9, 1979. ISSN 1097-024X.

R. Filman and D. Friedman. *Aspect-oriented programming is quantification and obliviousness*, chapter 2. Addison-Wesley, 2005.

M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. patch (1) considered harmful. In *HOTOS'05: Proceedings of the 10th Conference on Hot Topics in Operating Systems*, page 16, Berkeley, CA, USA, 2005. USENIX Association.

M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

G. H. Fletcher, C. M. Wyss, E. L. Robertson, and D. V. Gucht. A calculus for data mapping. *Electronic Notes in Theoretical Computer Science*, 150(2):37–54, 2006. ISSN 1571-0661.

J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X.

C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

Free Software Foundation, 2009. *GNU ld manual.* Free Software Foundation, version 2.20 edition, October 2009.

Free Standards Group, 2005. *DWARF Debugging Information Format version 3.* Free Standards Group, December 2005.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

D. Garlan and M. Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, 1994.

D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35:97–107, 1992.

R. Glass. Reuse: what's wrong with this picture? *Software, IEEE*, 15:57–59, 1998.

GNOME Developers 2002. Changes from 1.2 to 2.0. GNOME developer documentation, 2002. URL http://developer.gnome.org/gtk/2.24/gtk-changes-2-0.html. Retrieved on 2012/4/30.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, The.* Addison-Wesley Professional, 3rd edition, 2005. ISBN 0321246780.

E. Gunnerson and N. Wienholt. *A Programmer's Introduction to C# 2.0.* Apress, 2005. ISBN 1590595017.

C. Haack, B. Howard, A. Stoughton, and J. Wells. Fully automatic adaptation of software components based on semantic specifications. In *Proc. 9th Int'l Conf. Algebraic Methodology & Softw. Tech.*, 2002.

D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Softw. Pract. Exper.*, 20:5–12, January 1990. ISSN 0038-0644.

T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 388–402, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5.

W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. *ACM SIGPLAN Notices*, 28:411–428, 1993.

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978. ISSN 0001-0782.

U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 36–56, London, UK, 1993. Springer-Verlag. ISBN 3-540-57120-5.

U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9.

S. Holzner. *Eclipse: A Java Developer's Guide.* O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2004. ISBN 0596006411.

G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007. ISSN 0163-5980. doi: 10.1145/1243418.1243424.

J. Hunt and M. McIlroy. An algorithm for differential file comparison. Technical report, Bell Laboratories, 1976.

IEEE POSIX, 1988. Standard portable operating system interface for computer environments. IEEE Standard 1003.1-1988, 1988.

J. Järvi, M. Marcus, and J. Smith. Library composition and adaptation using C++ concepts. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, pages 73–82. ACM, 2007.

T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 417–430, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9.

C. F. Joerg. *The Cilk system for parallel multithreaded computing.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

B. Johnson, M. Young, and C. Skibo. *Inside Microsoft Visual Studio .NET.* Microsoft Press Redmond, WA, USA, 2002. ISBN 0735618747.

T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4.

A. Kantee. Rump file systems: Kernel code reborn. In *Proceedings of the 2009 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2009. USENIX Association.

R. Keller and U. Holzle. Binary component adaptation. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 307–329. Springer, 1998.

W. Kent. The many forms of a single fact. In *Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers*, pages 438–443. IEEE, February 1989.

S. Kepser. A simple proof for the turing-completeness of XSLT and XQuery. In *Proceedings of the Extreme Markup Languages 2004 Conference, 2-6 August 2004, Montréal, Quebec, Canada*, 2004. URL http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2004Kepser01.html.

B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming.* Springer-Verlag, 2001.

G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th European Conference in Object Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Conference.* USENIX Association, 1986.

G. Kniesel, P. Costanza, and M. Austermann. JMangler: a framework for load-time transformation of Java classfiles. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 98–108, 2001.

A. Krause. *Foundations of GTK+ development.* Springer, 2007. ISBN 1590597931.

D. Kuhn. On the effective use of software standards in systems integration. In *Proceedings of the First International Conference on Systems Integration*, pages 455–461. IEEE, 1990.

D. Lamb. Specification of iterators. *IEEE Transactions on Software Engineering*, 16(12): 1352 –1360, December 1990. ISSN 0098-5589.

B. Lampson. Software components: Only the giants survive. In A. Herbert and K. Jones, editors, *Computer systems: theory, technology, and applications: a tribute to Roger Needham.* Springer-Verlag New York Inc, 2004.

X. Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 177–188, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8.

S. Liang. *The Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley Professional, 1999.

T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, 1977. ISSN 0001-0782.

D. Liu, J. Peng, K. H. Law, and G. Wiederhold. Efficient integration of web services with distributed data flow and active mediation. In *Proceedings of the 6th international conference on Electronic commerce*, ICEC '04, pages 11–20, New York, NY, USA, 2004. ACM. ISBN 1-58113-930-6. doi: 10.1145/1052220.1052223.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6.

S. McConnell. *Rapid Development: Taming Wild Software Schedules.* Microsoft Press, Redmond, WA, USA, 1996. ISBN 1556159005.

S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: new-age components for old-fasioned Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 211–222, 2001.

M. McIlroy. Mass-produced software components. In *Proceedings of NATO Conference on Software Engineering*, pages 88–98. NATO Science Committee, 1969.

E. Meijer and J. Gough. Technical overview of the common language runtime, 2001. URL http://research.microsoft.com/en-us/um/people/emeijer/papers/CLR.pdf. Unpublished manuscript, retrieved on 2012/5/4.

L. Melloul, D. Beringer, N. Sample, and G. Wiederhold. CPAM, a protocol for software composition. In M. Jarke and A. Oberweis, editors, *Advanced Information Systems Engineering*, volume 1626 of *Lecture Notes in Computer Science*, pages 11–25. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-66157-3.

M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 52–67, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1.

R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 77–88, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1-55860-715-3.

R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100:1–40, 1992.

R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990. ISBN 0262132559.

J. Misra and W. Cook. Computation orchestration: a basis for wide-area computing. *Journal of Software and Systems Modeling*, 6:83–110, 2006.

M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. *SIGSOFT Softw. Eng. Notes*, 27:71–80, November 2002. ISSN 0163-5948.

I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24. ACM, June 2009.

I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2006.

N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007. ISSN 0362-1340.

O. Nierstrasz and F. Achermann. Separation of concerns through unification of concepts. In *Proceedings of the ECOOP 2000 Workshop on Aspects & Dimensions of Concerns*, 2000.

M. Nita and D. Notkin. Using Twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 205–214, Cape Town, South Africa, May 2010.

H. Nottelmann and U. Straccia. Information retrieval and machine learning for probabilistic schema matching. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, CIKM '05, pages 295–296, New York, NY, USA, 2005. ACM. ISBN 1-59593-140-6.

M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Inc, 2008. ISBN 0981531601.

S. O'Malley, T. Proebsting, and A. B. Montz. USC: a universal stub compiler. *SIGCOMM Comput. Commun. Rev.*, 24(4):295–306, 1994. ISSN 0146-4833.

W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of the ACM International Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 235–250, 1995.

J. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31: 23–30, 1998.

Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proc. 3rd ACM SIGOPS/EuroSys European Conference*, pages 247–260. ACM, 2008.

G. Papadopoulos and F. Arbab. Coordination models and languages. Technical Report SEN-R9834, CWI, Amsterdam, 1998.

D. L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd International Conference on Software Engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

R. Passerone, L. de Alfaro, T. Henzinger, and A. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE, 2002.

C. Peltz. Web services orchestration and choreography. *Computer*, 36:46–52, 2003.

B. Pierce. *Types and programming languages*. The MIT Press, 2002.

F. Pilhofer. *Design and Implementation of the Portable Object Adapter*. Sulimma, Frankfurt, 1999.

J. M. Purtilo. The POLYLITH software bus. *ACM Trans. Program. Lang. Syst.*, 16: 151–174, January 1994. ISSN 0164-0925.

J. Purtilo and J. Atlee. Module reuse by interface adaptation. *Softw. Pract. Exper.*, 21: 539–556, 1991.

C. Reichenbach, N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer. What can the GC compute efficiently?: a language for heap assertions at GC time. In *Proceedings of the ACM international conference on Object Oriented Programming: Systems, Languages and Applications*, OOPSLA '10, pages 256–269, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6.

A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)*, pages 347–360. Usenix Association, 2000.

D. J. Reifer, V. R. Basili, B. W. Boehm, and B. Clark. Eight lessons learned during COTS-based systems maintenance. *IEEE Software*, 20:94–96, 2003. ISSN 0740-7459.

D. Rine, N. Nada, and K. Jaber. Using adapters to reduce interaction complexity in reusable component-based software development. In *Proceedings of the 1999 Symposium on Software Reusability*, pages 37–43. ACM, 1999.

D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Commun. ACM*, 17: 365–375, July 1974. ISSN 0001-0782.

N. Sample, D. Beringer, L. Melloul, and G. Wiederhold. CLAM: Composition language for autonomous megamodules. In P. Ciancarini and A. Wolf, editors, *Coordinatio Languages and Models*, volume 1594 of *Lecture Notes in Computer Science*, pages 648–648. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-65836-8.

I. Savga, M. Rudolf, and S. Goetz. Comeback!: a refactoring-based tool for binary-compatible framework upgrade. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 941–942, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1.

M. Schaefer and O. de Moor. Specifying and implementing refactorings. In *Proceedings of the ACM International Conference on Object-Oriented Programming: Systems, Languages and Applications*, OOPSLA '10, pages 286–301, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6.

R. W. Scheifler and J. Gettys. The X window system. *ACM Trans. Graph.*, 5(2):79–109, 1986. ISSN 0730-0301.

D. Seeley. Shared libraries as objects. In *USENIX 1990 Summer Conference Proceedings*, pages 25–37, 1990.

A. Serra, N. Navarro, and T. Cortes. DITools: application-level support for dynamic extension and flexible composition. In *Proceedings of the USENIX Annual Technical Conference*, pages 19–19, Berkeley, CA, USA, 2000. USENIX Association.

J. Sharp and B. Massey. XCL: An Xlib compatibility layer for XCB. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. USENIX Association, 2002.

M. Shaw. Architectural issues in software reuse: It's not just the functionality, it's the packaging. In *Proc. IEEE Symposium on Software Reusability*. IEEE, 1995.

M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21:314–335, 1995.

M. Shaw. Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. Technical Report CMU/SEI-94-TR-002, Carnegie Mellon University, 1994.

A. Slowinska, T. Stancescu, and H. Bos. DDE: dynamic data structure excavation. In *Proceedings of the first ACM Asia-Pacific Workshop on Systems*, pages 13–18. ACM, 2010.

Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570, London, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6.

D. Solomon and H. Custer. *Inside Windows NT*. Microsoft Press Redmond, WA, USA, 1998. ISBN 1572316772.

F. Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 481–497, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4.

W. R. Stevens. *UNIX network programming, volume 2 (2nd ed.): interprocess communications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999. ISBN 0-13-081081-9.

C. Strachey. Fundamental concepts in programming languages, 1967. Reprinted in Higher-Order and Symbolic Computation, 13(1):11–49, Springer, 2000.

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12:157–171, January 1986. ISSN 0098-5589.

B. Stroustrup. Type-safe linkage for C++. *Computing Systems*, 1(4):371–403, 1988.

B. Stroustrup. *The C++ programming language*. Addison-Wesley, 1997.

System V, 1997. System V ABI specification, edition 4.1. The Santa Cruz Operation, Inc., 1997.

P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119. ACM, 1999.

S. Tatham. Coroutines in C. Web page, 2000. URL http://www.chiark.greenend.org.uk/%7esgtatham/coroutines.html. Retrieved on 2010/12/02.

D. Tidwell. *XSLT*. O'Reilly Media, Inc., 2008. ISBN 0596527217.

G. Van Rossum and F. L. Drake Jr. *The Python Language Reference Manual*. Network Theory Limited, 2003.

T. Veldhuizen. Software libraries and their reuse: Entropy, Kolmogorov complexity, and Zipf's law. In *Proceedings of the Workshop on Library-Centric Software Development*. Rensselaer Polytechnic Institute, 2005. Technical Report.

W3C, 1999. XSL transformations (XSLT) version 1.0. W3C Recommendation, 1999. URL http://w3.org/TR/xslt. Retrieved on 2012/5/2.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2.

J. Waldo and M. Clemsford. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6:5–7, 1998.

L. Wall and M. Loukides. *Programming Perl.* O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2000. ISBN 0596000278.

H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 1–16, Stevenson, Washington, USA, 2007. ACM. ISBN 978-1-59593-591-5.

A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 37–56, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4.

A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of European Software Engineering Conference held jointly with the ACM Symposium on the Foundations of Software Engineering. ESEC-FSE '07*, pages 35–44. ACM, 2007.

P. Wegner. Coordination as constrained interaction (extended abstract). In *Proceedings of the First International Conference on Coordination Languages and Models*, pages 28–33. Springer, 1996.

G. Wiederhold and M. Genesereth. The conceptual basis for mediation services. *IEEE Expert*, 12(5):38–47, September 1997.

G. Wiederhold. Mediation in information systems. *ACM Comput. Surv.*, 27(2):265–267, June 1995. ISSN 0360-0300. doi: 10.1145/210376.210390.

G. Wiederhold, P. Wegner, and S. Ceri. Toward megaprogramming. *Commun. ACM*, 35 (11):89–99, November 1992. doi: 10.1145/138844.138853.

P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proc. International Workshop on Memory management.* Springer Verlag, September 1995.

D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19:292–333, 1997.