Dynamic Extension of Typed Functional Languages

Don Stewart

PhD Dissertation

School of Computer Science and Engineering
University of New South Wales

2010

Supervisor: Assoc. Prof. Manuel M. T. Chakravarty
Co-supervisor: Dr. Gabriele Keller

THE UNIVERSITY OF NEW SOUTH WALES

SYDNEY · AUSTRALIA

Abstract

We present a solution to the problem of dynamic extension in statically typed functional languages with type erasure. The presented solution retains the benefits of static checking, including type safety, aggressive optimizations, and native code compilation of components, while allowing extensibility of programs at runtime.

Our approach is based on a framework for dynamic extension in a statically typed setting, combining dynamic linking, runtime type checking, first class modules and code hot swapping. We show that this framework is sufficient to allow a broad class of dynamic extension capabilities in any statically typed functional language with type erasure semantics.

Uniquely, we employ the full compile-time type system to perform runtime type checking of dynamic components, and emphasize the use of native code extension to ensure that the performance benefits of static typing are retained in a dynamic environment. We also develop the concept of *fully dynamic* software architectures, where the static core is minimal and all code is hot swappable. Benefits of the approach include hot swappable code and sophisticated application extension via embedded domain specific languages.

We instantiate the concepts of the framework via a full implementation in the Haskell programming language: providing rich mechanisms for dynamic linking, loading, hot swapping, and runtime type checking in Haskell for the first time. We demonstrate the feasibility of this architecture through a number of novel applications: an extensible text editor; a plugin-based network chat bot; a simulator for polymer chemistry; and xmonad, an extensible window manager. In doing so, we demonstrate that static typing is no barrier to dynamic extension.

Acknowledgments

This thesis describes work carried out between 2004 and 2008 at the School of Computer Science, University of New South Wales in Sydney.

I am deeply indebted to my supervisor, Manuel Chakravarty, who drew me into the functional programming world, and encouraged the exploration of its uncharted corners. I am grateful to my co-supervisor, Gabriele Keller, for ongoing feedback and insight.

Three groups of people influenced my work and thinking over this time, and I would like to thank them directly. Firstly, the members of the Programming Languages and Systems (PLS) group at the University of New South Wales – Roman Leshchinskiy, André Pang, Sean Seefried, Stefan Wehr, Simon Winwood, Mark Wotton, and Patryk Zadarnowski – who, along with my supervisors, built an energetic culture of innovation and exploration in functional programming that shaped the direction of this work from the beginning.

Secondly, my colleagues at Galois, Inc. in Portland, for giving me the resources and motivation to complete my research, and for the opportunity to apply functional programming techniques to solve difficult problems, all in an environment of talent and fun. In particular, I wish to thank John Launchbury, for his mentorship, Adam Wick, for support and motivation, and Jason Dagit, for constructive feedback.

Thirdly, the Haskell community provided willing feedback and suggestions for many of the projects described in this thesis, motivating me to pursue ideas I might have passed by. Shae Erisson, in particular, encouraged and inspired this work in its early days, helping to ensure the success of several of the projects. I am glad to be a citizen of such a community of artists and hackers.

Finally, this work would not have been possible without Suzie Allen, and her patience, encouragement, and love.

Portions of the text of this thesis were originally published in [122, 146, 147, 148], and due acknowledgment is due to André Pang, Sean Seefried, Manuel Chakravarty, Gabriele Keller, Hugh Chaffey-Millar and Christopher Barner-Kowollik, for their contributions.

Don Stewart. Portland Oregon, June 2010.

Contents

List of Figures

Chapter 1

Introduction

Static typing is no barrier to extension.

1.1 Motivation

There is a tension between static checking of programs to ensure safety, and dynamic modification of programs to gain flexibility. The goal of this thesis is to explore and resolve this tension – to have robust, safe software with the flexibility to modify and add new features dynamically. We seek a middle way, and employ two main tools to this end: software architecture design and static type systems.

Programming languages play a large role in how easy it is to build robust software. The right language can make many classes of errors unlikely, while the wrong language will obscure mistakes, causing errors to go undetected. Languages vary wildly in what support they provide for building safe and secure code. We will focus on languages with a serious attitude towards safety and assurance: statically typed, purely functional languages. Specifically, we will use the type system to prevent errors wherever possible.

Good software architecture – the design of structures for combining software components and organizing their interactions – is also important for building flexible, scalable software. Good architectures make modification and extension of code easy, and

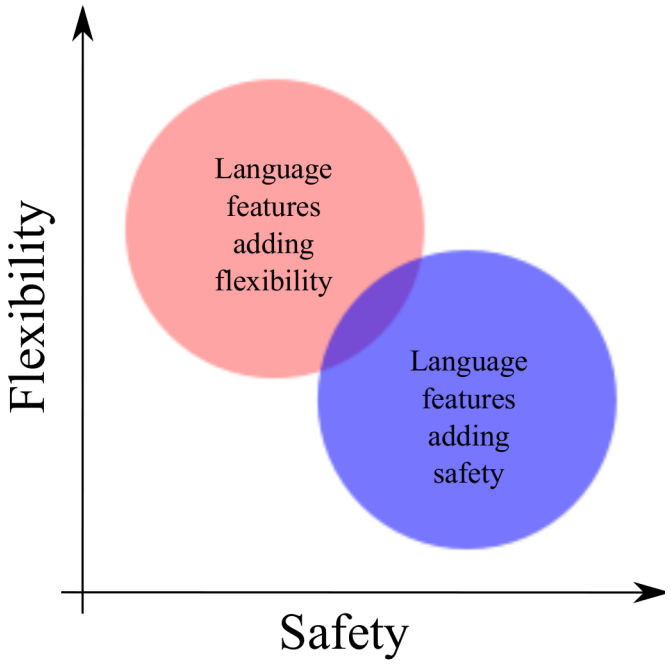

Figure 1.1: Flexibility versus safety in languages

can make it simpler to repair systems that go wrong. Poor architectures can make it challenging to replace broken components.

Ideally, we should be able to build software systems that satisfy both concerns, combining mechanisms for safety *and* flexibility in an integrated and complementary fashion, as illustrated in Figure 1.1. That is, integrate language design and powerful type systems with approaches to software architecture design, in order to produce code that is safe and flexible.

This thesis focuses specifically on the problems that arise when we attempt to reconcile two key technologies for software safety and flexibility:

- Modern Hindley-Milner-style static type systems with type erasure; and
- Dynamic code loading with hot swapping of native code components.

We reconcile these two different technologies – types and dynamic architectures – combining them in a unified framework to establish that, ultimately, we need not sacrifice one for the other.

1.1.1 Safety via types

A key concern of this thesis is software safety. The first tool we reach for to make programs safer is modern static type checking – a powerful technique for the construction of reliable systems. Hindley-Milner-style (HM) type systems [38, 68] such as those found in Haskell [125, 72, 119] or ML [109] and extensions to such systems – including type classes [161], generalized algebraic data types [127], functional dependencies [80], and type families [136] – make it easier for significant invariants to be stated and checked automatically as part of the regular development process. Indeed, a range of new languages are based on this observation, designed explicitly to support property statements in types, including Agda [117] and Epigram [105]. This thesis assumes a development style focused on safety via rich types.

Such a type-directed development model encourages *co-design* of code and properties [98]. Properties are asserted via type information, and freely intermixed with code, rather than existing somehow apart. The use of types as machine-checked assertions allows us to build more complex software, with higher assurance of correctness, and with little additional cost [134, 121, 86, 50, 26]. Our assertion is that type systems are a foundational technology for a future of correct-by-design software.

Typically, languages supporting strong HM type systems use *type erasure semantics* [36] in their implementation. Due to progress and preservation properties of the type system, the program cannot "go wrong" with a type error once checked at compile time [128], removing any need for additional runtime type checking. This fact allows the compiler to discard all type information from the program after code generation. Erasure of type information, however, makes it very difficult to write type safe programs whose type is not known until runtime [31].

Taken at face value, this emphasis on static type safety rules out a range of powerful extensible programming practices, including those based on forms of runtime code loading. The argument is, broadly, that static typing limits programmer expression[1],

[1]For a representative argument from this viewpoint, see "Is Weak Typing Strong Enough?", Steve Yegge, http://sites.google.com/site/steveyegge2/is-weak-typing-strong-enough

and that we face mutually exclusive options: either we have static safety, but lose flexibility; or have dynamic flexibility, and pay the tax of runtime type checking and exposure to runtime type errors.

1.1.2 Flexibility via runtime code loading

Dynamic linking and loading of code is a technique long associated with the C programming language [85]. Its origins are in Multics [118], via operating system support for dynamic linking (e.g. `dlopen()` [156]). Rather than linking all library code statically into an executable, shared components are instead loaded at program startup time, avoiding code duplication. An extension to this is the ability to relink code dynamically, making it possible to replace parts of a program on the fly.

Many runtimes built for languages that are primarily implemented via bytecode interpretation (such as Lisp-like languages, Erlang or Java [56]) support the ability to insert new (byte)code at runtime. Programming practices that exploit these features – including forms of `eval()` [106] and other meta-programming techniques (e.g. "monkey patching" [171]) – are commonplace. Even in language implementations without specific support for full meta-programming, or even dynamic linking, there are often ways to dynamically generate code under programmer control, either via reflection mechanisms or code generation support libraries (such as LLVM [155] or Harpy [57]).

Generating new code dynamically is an extremely powerful (and potentially dangerous) feature, and makes entirely new programming techniques possible. A primary example of this is *code hot swapping* [7, 8, 64, 46], where faulty software components are replaced at runtime with corrected code, without loss of service. Such techniques have been used in operating systems, for example, to allow the kernel to be patched without requiring a reboot [11, 12].

Hot swapping, in the limit, allows a programmer to seamlessly update code without reducing availability of the software service to its users: a powerful mechanism that hands flexibility to the programmer.

The downside of such powerful language features is the difficulty of reasoning about dynamic code prior to execution. Static types can not describe code that simply doesn't exist statically. The loss of static type information, and the deferral of type checking to the last possible moment, greatly hinders analysis for safety and correctness, as it becomes impossible to state properties of code without running the code and observing its behavior.

Such extreme programming flexibility appears to be fundamentally in tension with compile-time approaches to correctness and safety property checking.

1.1.3 The middle way

So how can we gain flexibility without sacrificing type safety? What language features and software architectures do we need to retain strong types, while allowing dynamic repair and extension to our programs? This thesis explores this problem specifically as seen in the interaction of dynamic linking, loading and code generation with strong static typing and type erasure – types of the kind seen in Haskell. To tackle the problem, we look at systems and language techniques for combining these tools, and new software architectures to ultimately, unambiguously provide both safety and flexibility.

We investigate how to integrate dynamic language features into languages that have a strong phase distinction between compile-time and runtime, and how to ensure type correctness when code is replaced at runtime, in order to retain the significant benefits of compile-time type checking, while still allowing for programming practices that enable code extension.

Concretely, this thesis proposes mechanisms for safely integrating dynamic linking, plugin loading, hot swapping and runtime meta-programming into statically typed purely functional programming languages. We develop a range of practical uses for the combination of these features in real applications, including server and client systems, software configuration and code replacement models, and runtime meta-programming with type sandboxing. We *can* have flexibility without sacrificing safety.

1.2 Approach

1.2.1 Dynamic extension

Many successful software systems support some form of extension capability, whether via plugins, "mods", or scripts, allowing users to expand the functionality of the application without requiring modification to source code. The pressure to provide extension capabilities in order to succeed is famously satirized as "Zawinski's Law"[2]:

> Every program attempts to expand until it can read mail. Those programs which cannot so expand are replaced by ones which can.

Most commonly these extensions are loaded at application startup time as plugins. In some designs, extensions must be explicitly specified at compile-time, while a more flexible approach is to allow plugins to be loaded at any point during process execution. Applications that support such extensions typically require specialized library or operating system features to support loading code into the process' address space.

The languages most commonly associated with dynamic extension are usually dynamically typed – such as Lisp, JavaScript, Python or Ruby – where runtime extension via meta-programming is a widespread, if controversial [171], programming technique. There is a natural fit between dynamic typing and dynamic extension: since clearly we *must* defer some type checking until runtime, as the code to check is simply not available at compile-time. However, to respond by deferring *all* type checking to the last possible moment is a drastic move, as now any type error becomes a runtime error.

We argue for a compromise approach: as much static checking as possible, but with the use of dynamic type checking where required. We maintain the invariant that any *component* in the system will be type checked before executing code in that component. We only execute well-typed code.

[2] From the Unix Jargon File, http://www.catb.org/~esr/jargon/.

Extension via plugins

The first technique for dynamic extension we will investigate is the most fundamental: dynamic linking of program components as *plugins*. Applications that support plugins are significantly more flexible than ones that do not, as new features may be added or removed from the program under user control, without requiring code be integrated into the source of the application itself.

However, plugins pose a difficulty for whole-program type checking – they are typically implemented as modules that are developed separately from the application code. Plugins force us to give up on the idea of type checking *all* source before running any of it – leading inevitably to more flexible component-wise type checking, where modules are type checked in isolation, and their interfaces are exposed for later checking against use sites.

In Chapter 3, we will develop an approach to type safe plugins in Haskell that retains component-wise static checking (plugins are internally statically type checked), while deferring the checking of use sites of plugin code until runtime. Only once a plugin is loaded will its interface into surrounding code be checked, and only then will the plugin's code be executed. Type checking thus proceeds in stages, interleaved with some execution.

We demonstrate this approach via a dynamic linking framework for the Glasgow Haskell Compiler [154], providing dynamic typing of interface components, and dynamic linking and loading as a runtime service. We also develop pragmatic applications of plugins in Haskell.

Runtime code generation and meta-programming

Once support for dynamic loading of native code components is available, and a method to type check the interfaces between components is in place, runtime *meta-programming* becomes feasible. We can generate code in multiple phases, and incorporate the new code into an executing process.

In Chapter 4 we will explore techniques for runtime meta-programming in Haskell via native code compilation. We will show that by combining dynamic linking, dynamic typing and compile-time reflection we can add support for meta-programming to a language.

Applications for runtime meta-programming with static typing are developed, including type-based sandboxing of untrusted code, and optimizing simulators for scientific computing. Language support for meta-programming is investigated, along with library approaches to simplify the use of runtime meta-programming.

Code Hot Swapping

An extension of plugin architectures is the ability to load and unload components at any time during execution, and to support *hot swapping*: where a component is seamlessly replaced with a modified version, without loss of application state.

Full support for hot code swapping requires many of the techniques developed for plugin support, and encourages the dynamic loading interface to be under direct programmer control (as loading points are no longer constrained to be solely at program startup). Code unloading requires extensions to the dynamic linking interface to support the removal of (possibly executing) code without failure.

The key challenge, however, of code hot swapping involves program state. Any state – data generated during program execution not fully determined by the arguments to the component, including system and kernel data structures – must be preserved during the unload $\rightarrow$ reload cycle.

Preservation of state through hot swapping is complicated by the fact that the shape of the data may change during upgrade as the code that operates on the data is modified. We must provide a way for existing code to continue to work even as the shape of the data changes. In a statically typed setting such type evolution must not break code. The challenge of transparently supporting state persistence and migration in a Hindley-Milner type system quickly leads to "the expression problem" [160] – which

describes the difficulty of extending both the variants of a data type, and the functions on that data type, without modifying *existing* code.

In Chapter 5 we will develop approaches to dynamic hot swapping in a typed, purely functional language, propose language support techniques and application architectures that can handle simple and radical degrees of hot swapping, and explore approaches to typed state preservation between module reloads, and more generally across application instances. In doing so we will show that it is possible to fully support hot code reloading in the context of a statically typed language with a rich type system, preserving type safety, and gaining flexibility in the process.

1.2.2 Static typing

In many software systems it is unacceptable to ship code that has not been checked for correctness. Software errors can be dangerous, and even fatal [101, 143]. Particular invariants may be argued to hold via testing, or established via type systems and formal methods, putting a bound on the complexity of software by automating the difficult task of verifying properties throughout the source.

In this thesis we use a particular notion of static typing – extensions to Hindley-Milner-style type systems, as supported by languages like Haskell – where it becomes feasible for programmers to assert correctness and safety properties of code directly via types. More generally, we expect sound type systems – where well-typed programs don't go wrong.

In order to retain the benefits of rich static typing, we attempt to maintain full compatibility with the compile-time type system, when used dynamically. That is, we seek solutions that allow any type-based programming technique to continue to work when applied dynamically. This will mean that safety properties in types that were used at compile-time will remain valid at runtime.

Specific language support that makes dynamic extension easier is covered in Chapter 6, where we investigate domain specific languages designed for writing extension components.

1.2.3 A framework approach

Resolving the tension between dynamic extension and static typing brings together a wide range of techniques and approaches: linking; compilation; type systems; meta-programming; language design; and software architectures.

The solution we propose in this thesis introduces a *framework for dynamic extension*. This framework is a structure for the techniques we describe, where each technique adds a specific capability to the overall framework. The framework provides a set of capabilities that ensure the full range of dynamic extension techniques are available in a compiled, statically typed language. The structure of this framework is described in Chapter 2, and is the unifying concept of this thesis.

The framework for dynamic extension is built around three core techniques: dynamic linking; dynamic typing; and runtime compilation, which are further supported by first-class modules, extension languages and hot swapping. Each technique may be used in combination with the others to construct different solutions to the dynamic extension problem, allowing a range of more flexible programming techniques. Throughout the thesis we develop examples and instances derived from this unifying framework.

1.3 Contribution

This thesis provides a constructive solution to the problem of dynamic extension of modern statically typed languages. This problem has received relatively little study, despite the widespread use of dynamic extension capabilities. We contribute a framework where language techniques and software architectures combine to provide dynamic extension capabilities to a language implementation, while retaining the benefits of strong static type checking.

The consequence and impact of these features on the design of typed functional programs is explored via a wide range of real applications developed through the course of this work. Software architectures for cleanly combining dynamic and static techniques are described.

We show that a statically typed language can elegantly and efficiently support dynamic code modification, allowing us to gain flexibility without sacrificing safety, via our framework. By using dynamic linking, runtime code generation, and crucially, ensuring that the full type checker is available at runtime to type check dynamic components, type safe dynamic extension of executing software is possible.

The framework for dynamic extension will cover the following features:

- Dynamic linking;
- Runtime type checking;
- First class modules;
- Runtime compilation;
- Code hot swapping; and
- Embedded extension languages.

1.4 Structure

To establish the feasibility of our approach we develop a framework for dynamic extension of typed functional languages, implement it for Haskell, and demonstrate practicality through a wide range of applications.

- In Chapter 2 we develop the notion of a framework for dynamic extension, based on linking, loading, runtime type checking, hot swapping and language support, and introduce the approaches we take for each component.
- Chapter 3 investigates dynamic linking in detail and approaches to programmatic control of linking. We explore techniques for typed dynamic linking and typed plugins. We introduce a new approach to polymorphic dynamic types.
- In Chapter 4 the notion of runtime compilation is explored, and multi-phase compilation in general. Specific applications for runtime compilation, and the benefits of full optimizing native code based runtime compilation are developed.
- Chapter 5 looks at the integration of hot swapping into a language, and how to tackle the state preservation and migration problem. We look at *fully dynamic* architectures, where any component may be replaced dynamically, and develop applications that instantiate this model.
- Chapter 6 develops techniques for supporting dynamic extension by users, applying features from functional languages to make dynamic extension easier. In particular, we investigate the use of embedded domain specific languages for user-defined application extension.
- Finally, Chapter 7 summarizes the contribution of the thesis.

Each chapter is self-contained, referring to distinct fields of work. As such, related work is deferred to the end of each chapter.

Chapter 2

A framework for dynamic extension

To address the question of how to safely dynamically extend programs written in typed functional languages, we propose a *framework for dynamic extension*. This framework sets in place the required infrastructure needed to support forms of dynamic extension in a statically typed, compiled language with type erasure. Each component of the framework provides a particular capability that enables one or more distinct dynamic extension features. This chapter introduces the framework, defines each component of the framework, and then describes our realization of each component.

2.1 Components of the framework

The framework consists of six parts:

- Dynamic linking;
- Dynamic type checking;
- First class modules;
- Runtime code generation;
- Code hot swapping; and
- Embedded extension languages.

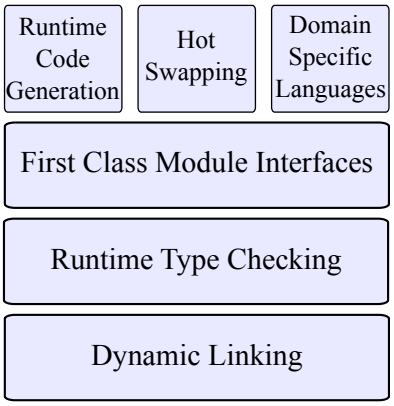

Figure 2.1: A framework for dynamic extension

The framework sets in place approaches to improve the state of dynamic extension in typed functional languages at all layers of abstraction: from low level linking and loading of untyped code, through runtime meta-programming and code hot swapping, and to improvements in user-facing extensible systems via domain specific language support. We will investigate the utility of each in turn. The layering of components to form the entire framework is illustrated in Figure 2.1.

We now briefly define each component of the framework, what forms of dynamic extension it allows, and the specific technical approach we will take to design and implement that component, before a detailed technical examination of the framework components that form the proposed solution.

2.1.1 Dynamic linking

Dynamic linking of native code is the foundational mechanism for extension on which the framework proposed in this thesis is based.

> **Definition** Dynamic linking is the ability to import program fragments into a running process, bringing new code into scope in a program at runtime.

Throughout this work we use the notion of a *module* or *component* – a set of program fragments forming a unit that is separately compilable. Program fragments are (following Cardelli [24]) "any syntactically well-formed program term, possibly containing

free variables". Names bound in such fragments are referenced from the running process, and the process of dynamic linking resolves those references, closing the name environment.

Traditionally, linking is done statically and on the assembly level. Symbolic references in an object file to names in other files are replaced with actual machine addresses. The linker combines sets of such modules into a single executable with all symbolic addresses resolved. Dynamic linking delays the resolution of symbolic addresses until after the code is loaded into a running process' address space (either resolving all symbols at load time, or lazily, as symbols are required). The loaded code may not have been available during the compile-time of the main program – and we assume that this is the case.

Informally, dynamic linking allows a set of data types and functions collected into a single unit – the *module* – to be introduced into a running program's environment by *loading* a reference to the module, usually the module's name, at runtime. This operation may be under user control. Binding of the data types and functions within the module to names in the executing program happens late, at linking time. This is the form of "plugin" we are seeking in the context of a statically typed functional language such as Haskell, namely, true dynamically loaded modules with late binding.

Once loaded, those functions and data types are available for use within the scope of their binding, possibly by accessing them through an indirect interface. When the local scope exits, the dynamically loaded code is no longer available for use[1]

Code loaded may either be compiled (and optimized) native code, or interpreted code (or both, if the underlying runtime supports the mixture, as the GHC runtime does [139]). In this thesis we emphasize dynamic loading of optimized, native code.

To demonstrate feasibility of our framework, we implement a dynamic linking system for native code Haskell modules, as an extension to the Glasgow Haskell Compiler. This library, `hs-plugins` [122]), provides direct support for type safe plugins in Haskell

[1]In a lazy language, in particular, this process is dangerous, as unevaluated *thunks* may contain references to dynamically loaded code. We side-step this issue in our implementation by retaining the code in memory, but preventing new references from being constructed to it.

for the first time. We present the dynamic linking component of the framework in Section 2.3, and dynamic linking mechanisms in detail in Chapter 3. The library forms the basis for the entire framework described in this thesis.

2.1.2 Runtime type checking

The separation of runtime-loaded code from code available at compile-time, and the related technique of separate compilation, imposes a fundamental difficulty for type systems with a strong phase distinction [23, 59], where all type checking is performed at compile-time. In order to ensure type safety when combining program fragments produced at different program phases, a form of type checking at phases other than the initial "compile-time" phase is necessary. *Runtime type checking* is thus the second major component of our framework.

> **Definition** Runtime type checking is the ability to interleave execution of a program with type checking of the program. Execution may begin before the complete program has been type checked, however, all program fragments will be type checked before they are executed.

In particular, we are interested in runtime type checking of the *interfaces* between components – sets of signatures of the types and functions exported by modules – and the surrounding program that loads them.

Our framework requires that type information for component interfaces be preserved until the time the component is linked against the running program – and not erased at compile-time. At that point, the interfaces at each *splice* point in the program – points where old and new code initially interact – are type checked using a variant of the language's type checker.

Our model for adding forms of runtime type checking to a statically typed language proceeds in three steps:

- Firstly, type checking of dynamic components of the program is delayed. This is achieved via the use of a *universal* static type. Such a type wraps code for

which type checking would not succeed until a later phase. Rather than reject the program as ill-typed, the compiler passes it on for runtime checking. We use dynamic types in the style originated by Abadi et al. [2], and developed for Haskell by Cheney and Hinze [31] as a wrapper to represent all dynamic types. In previous work in Haskell, dynamics have been used to add forms of generics, however, we apply it to the problem of typing dynamically loaded code.

- The second step is to preserve enough type information so that a type check is possible at runtime. The first mechanism we propose suffices for monomorphic types (and polymorphic types with some restrictions). We use `Data.Typeable`, originally developed for generic programming (Lämmel and Peyton Jones [88]), to reify types to values (represented as integers) via a type class. We then develop a more flexible approach where the compiler's internal type data structures are serialized with the object code for the dynamic components, that ensures all type information may be preserved without restriction.

- Thirdly, runtime type checking requires a mechanism to actually perform type checking at runtime. We initially consider forms of *typecase* [167], which have restrictions on the flexibility of types that may be checked. We then discard such approaches, and replace it with a novel contribution: building a form of typecase directly by using full language type checking subsystem at runtime, and, in doing so, supporting all types in the language (including arbitrary rank polymorphism, GADTs and type classes). Our design ensures there are *no* restrictions on the strength of types used in dynamic components.

Once dynamically loaded code is ready for type checking, that type information is reflected back to the type level and passed to the type checker, in the form of a type assertion between code on either side of a splice point. If type checking is successful, the component code may be executed, if not, it is discarded. We present the solution we develop in Section 2.4. Detailed mechanisms for preserving type information through to runtime are described in Section 3.3.

2.1.3 First class modules

The third feature of the framework we propose is a requirement to support *first class modules*:

> **Definition** A first-class module is a set of program fragments forming a compilable unit, represented as data at runtime.

Such modules may be manipulated programmatically in convenient units. By doing so we allow the programmer to inspect and modify loaded components without stepping outside the language. The loaded components may be referenced by the programmer directly. Modules, in this sense, may be seen as reified interfaces, available as values. We lean on existing language features to build in support for first class modules, using existential types and type classes to describe the abstract interface a module provides and its set of methods.

The ability to reference dynamic components as first class modules allows for a range of flexible software architectures to be implemented: applications can be structured as loose collections of services, which may or not be available at any given point during program execution, and any of which may be repaired on the fly. The approach to modules we take is described in Section 2.5, and we describe software architectures based on first class dynamic modules in Chapter 5.

2.1.4 Runtime code generation

To support the full range of dynamic extension capabilities we provide mechanisms for the programmatic generation of new native code at runtime, that is, *runtime code generation*.

> **Definition** Runtime code generation is the ability to interleave execution with compilation.

Code will be represented as data in one stage, and reflected into native code in the next, using compilation at runtime. The generated code is then dynamically linked back into the application.

Runtime code generation, with dynamic linking and type checking makes possible a multi-stage meta-programming model, including `eval()`-like mechanisms and staging operators. We illustrate the uses of runtime meta-programming via online evaluation engines for Haskell, an implementation of a dynamically generated `printf` function via meta-programming, along with other applications.

A novel contribution of our approach is to reuse the compile-time meta-programming facilities of Haskell (Template Haskell [141]) – specifically the ability to transform code to data – combining this with dynamic linking, to yield a *runtime* meta-programming implementation.

Additionally, the use of native code runtime meta-programming allows for specialization of modules at runtime. We describe the implementation of a plugin-based program specialization mechanism, applied to Monte-Carlo simulation of polymer chemistry, that improves performance and interactivity over one-shot static compilation approaches through runtime code generation. This application is described in Section 3.6.3.

The approach we use to add support for runtime code generation in the typed dynamic linking framework is to allow the use of *the full language compiler at runtime* – not a restricted language subset, or interpreter. The compiler's code generator is made available to libraries via an API, and distributed with the application. The use of the compiler for runtime code generation (over, for example, a bytecode interpreter) ensures runtime-generated code suffers no performance penalty in comparison to statically-generated code (modulo the impact on cross-module optimizations).

We implement a runtime code generator interface for Haskell as a library based on GHC, and in turn, support forms of runtime meta-programming. Our solution is presented in Section 2.6 and details of the approach we take are described in Chapter 4.

2.1.5 Module hot swapping

Applications that must be tolerant to faults, or which have changing requirements, but nonetheless require high availability of service, are often built with plugin architectures that support *code hot swapping*.

> **Definition** Hot swapping is the replacement of existing code fragments at runtime without loss of the environment (state) associated with that code.

This is a generalization of dynamic linking to support *replacement* of code and *restoration* of state. The goal of code hot swapping is to allow for upgrades to components without observable loss in availability.

We describe and implement a mechanism for hot swapping in Haskell, based on loosely-coupled application architectures, dynamic linking and state serialization, in Chapter 5, and build a number of applications that use hot swapping to maintain service, including a network application that reloads features under user control, and a text editor able to replace any component within the editor, without loss of performance over a statically compiled application.

We address the type safety issues introduced by hot swapping via a range of mechanisms, including open data types modeled via type classes, and dynamically typed extensible data. In doing so, we ensure preservation of data, despite types changing during application upgrades. The architecture we propose allows for all components in the application to be replaced, relying only on a small static core.

The specific solution presented for module hot swapping is described in Section 2.7, and further developed in Chapter 5.

2.1.6 Embedded extension languages

With support for plugins, hot swapping, runtime code generation and runtime type checking, our framework for dynamic extension makes a wide range of features and capabilities possible, as illustrated in Figure 2.2.

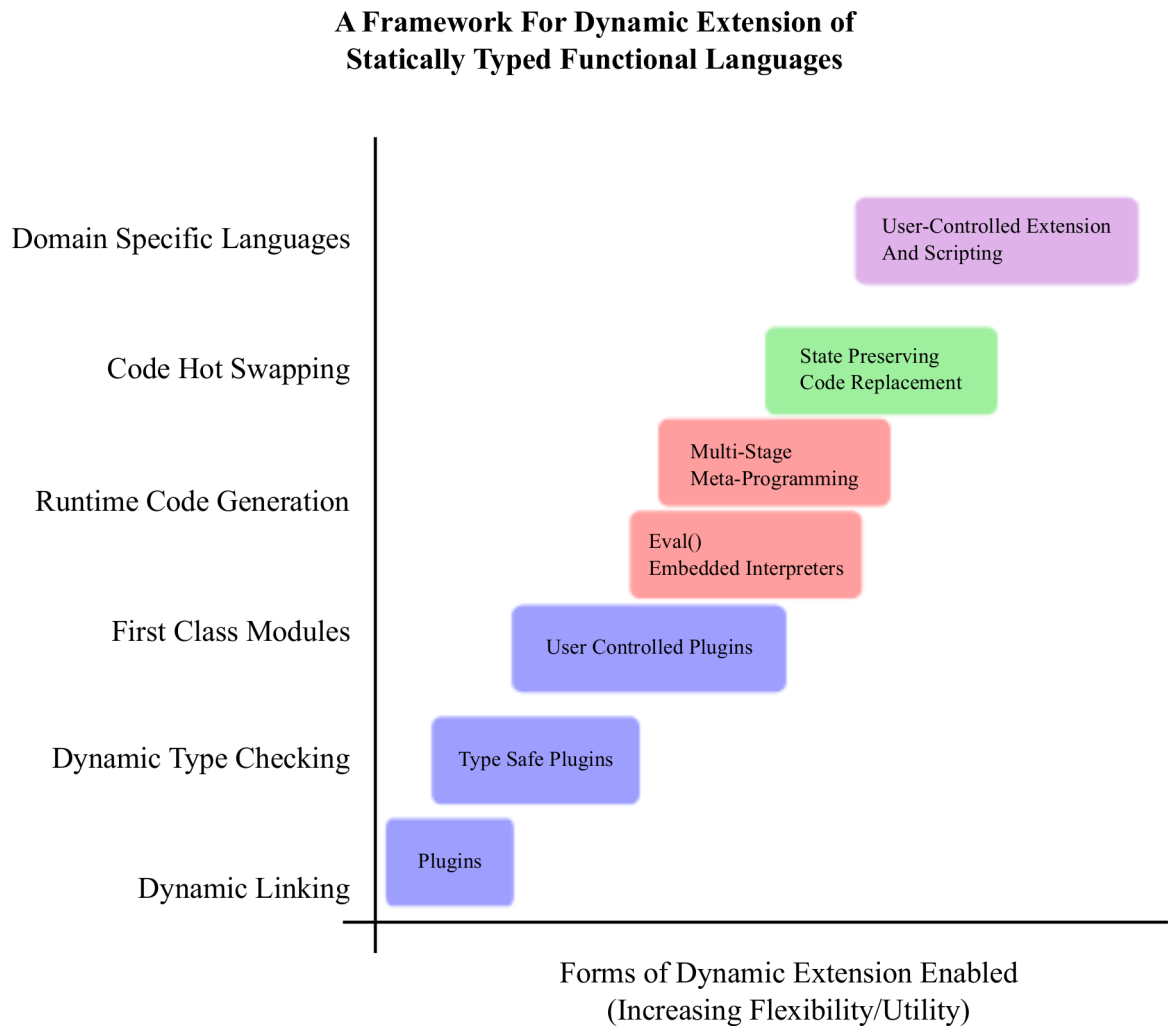

Figure 2.2: Dynamic extension forms enabled by each component, by expressive power

To put a handle on the complexity of programming in such an environment, we propose the use of embedded domain specific languages (EDSLs) [70, 71] for extensible components, making it possible to use the development language of the application as its extension language.

The use of EDSL-based extension languages avoids the common tendency to add a second language implementation (such as Lua or JavaScript) to an application, specifically to support extension by users. An added benefit is that "throw-away" scripts and configuration files written in the extension language may be statically checked, optimized, and compiled to native code, avoiding the typical engineering penalties of custom extension languages.

We describe a range of applications that use the application language as the extension language, hiding complexity via EDSLs. In particular, we describe the implementation of an extensible text editor in Haskell, scriptable in Haskell and a window manager for Unix configured in Haskell, in Chapter 6. Section 2.8 describes our approaches to domain specific embedded extension languages, based on the framework described in this thesis. Our battle cry is that when applications are written in sufficiently expressive languages, the extension language *is* the application language.

2.2 Implementation

To validate our approach to dynamic extension, each subsystem of the framework is implemented for the Haskell language, a statically typed functional language with no prior support for dynamic extension. In turn, we show feasibility by implementing a range of applications in Haskell that exhibit different dynamic extension features.

The core of the system we develop is the `hs-plugins` library for Haskell (originally published as part of this effort in "Plugging Haskell In" [122]), which provides dynamic linking for Haskell under programmer control. In addition, we provide runtime type checking, hot swapping and meta-programming facilities. The hot swapping extensions to `hs-plugins` were originally described in "Dynamic Applications From The

Component	Implementation
Dynamic linking	`hs-plugins:System.Plugins.Load`
Runtime type checking	`hs-plugins:System.Plugins.Load`
First class modules	Existential types and type classes
Runtime code generation	`hs-plugins:System.{Eval, Plugins.Make}`
Hot swapping	`hs-plugins:System.Plugins.Load`
Language support	`Yi` lexer DSL, `xmonad` configuration DSL, et al.

Figure 2.3: Implementation of the framework for dynamic extension

Ground Up" [148], and support for loading languages other than Haskell in a series of papers on specializing simulators [147, 84, 28], all as part of this work.

The effectiveness, and generality, of the techniques are explored through a range of applications developed during the course of this work:

- a network service, *Lambdabot* [122];
- an extensible text editor, *Yi* [148];
- a scriptable window manager, *XMonad* [149]; and
- a specializing polymer chemistry simulator, *polysim* [147, 84].

Each component of the framework is instantiated in the modules and applications illustrated in Figure 2.3.

We now provide illustrative examples of each subsystem of our framework, and an introduction to how each part contributes to a synthesized approach to dynamic extension.

2.3 Dynamic linking

The first layer of the framework for dynamic extension is dynamic linking and loading of compiled plugin components. Our approach to dynamic loading of plugins loads Haskell *modules* into a running application's address space. The symbol resolution and linking support is provided as a runtime service by the Glasgow Haskell Compiler. Once linked, code and data in the module are transparently available to the application as regular Haskell values.

Modules may have dependencies on other modules. In order to resolve symbols in such cases, the dynamic loading system implements automatic dependency tracking and supports cascading loading of modules in their dependency order. Closed sets of modules forming an *archive* may be loaded in a single operation. Loaded code may be used by the Haskell application, or in turn passed through to code written in other languages, such as C. We support loading Haskell code modules into foreign languages, and loading of C object files into Haskell.

The contribution of the dynamic linking component of the overall framework is to:

- define support for loading and linking of native code modules;
- implement a comprehensive dynamic linking framework for Haskell; and
- support the sharing of dynamic code between C and Haskell.

A dynamic linker system presents two distinct requirements: (1) an interface and communication protocol between the host application and its plugins must be defined and (2) the plugins' object code needs to be integrated into a running application by a *dynamic link loader*.

We will now outline an architecture that addresses these challenges for Haskell. We will begin by working through an example of implementing a plugin for a Haskell program, and loading it dynamically, using our linker.

2.3.1 Defining a plugin interface

In languages without strong static typing, plugin systems are implemented with an interface characterized by two sets of *symbol names*: those that the plugin can access from the host application and those that the host application expects to be defined by the plugin. Moreover, there is an informal agreement about the data structures that are passed between the host application and the plugin.

In a statically typed language, such as Haskell, however, we need to ensure precisely that the host application and plugin agree on the *types* of data structures and functions that they share. Moreover, we want the application to ensure that any violation of these types on either side is detected and reported by the system. We defer the details of how we guarantee type safety until Section 2.4, and here explore the specification of interfaces for plugin modules.

To avoid arbitrary dependencies of plugin code on application internals, in our model the application programmer first defines plugin interfaces to be used by plugins. We collect interface functions in a data structure, called `Interface` in the following example, corresponding to component "A" in Figure 2.4. In this simple example it merely contains a single string processing function named `stringProcessor`; a more complex application will have a richer API.

```
module StringProcAPI (Interface(..), plugin) where

data Interface = Interface { stringProcessor :: String -> String }

plugin :: Interface
plugin = Interface { stringProcessor = id }
```

In addition to defining the interface signature, the interface (or API) module provides a default implementation named `plugin`. Default implementations can provide sensible values in the absence of any dynamically loaded plugins and are useful in many contexts, such as providing default runtime behaviors which a user may optionally override.

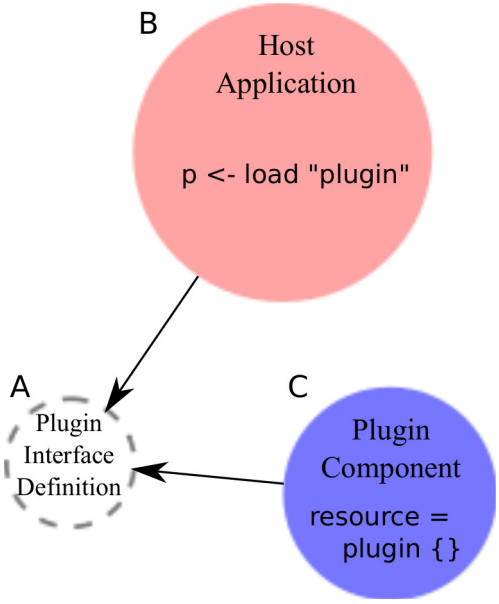

Figure 2.4: Defining a plugin against an interface

2.3.2 Implementing an interface

The next step to enable application support for plugins is to compile the application against the interface we defined, `StringProcAPI`. Any plugins must then also import and implement the interface type declared in that module. The shared use of the plugin API are illustrated by arrows from the application and plugin, to the interface, in Figure 2.4. Finally, the application requires that plugins bind their implementation of the plugin interface to a specific name, identified as `resource` in our example.

We can now implement a plugin component that may be loaded into the application. In the following plugin implementation, we define the `stringProcessor` function to reverse its input string and wrap it up in the plugin's `resource` interface. The plugin itself is a single Haskell module, exporting a data structure corresponding to the plugin's expected interface:

```
module StringProcPlugin (resource) where
import StringProcAPI (plugin)
resource = plugin { stringProcessor = reverse }
```

2.3.3 Using a plugin

Now we can discuss how a plugin is used from an application. There are generally three steps involved:

1. Compilation of the plugin's source code (possibly offline);
2. Loading of plugin(s) and resolving of symbols; and
3. Calling of the plugin's functions through the plugin interface.

Plugins may be compiled by the plugin author or by the host application; this choice influences the amount of trust the application can put into the type safety of a plugin.

A simple application using the `StringProcPlugin` defined above might read strings from the user, line by line, and output the result of applying the `stringProcessor` to each of these strings. For our example, let us assume the plugin object code is loaded by a function named `load`, which accesses the dynamic loader and linker. An application then uses the dynamic plugin as follows:

```
(mod, rsrc) <- load "StringProcPlugin.o" ["."] [] "resource"
```

This code fragment loads the object and obtains a handle to the plugin symbol named `resource`. More precisely, it obtains a handle, `mod`, to the Haskell module contained in `obj`, as well as the value bound to `resource` in that module, which we name `rsrc`. Note that both functions take extra parameters that we do not use here; details concerning the signatures of these and other functions of the plugin library are deferred to Chapter 3.

The `load` function invokes the dynamic loader subsystem of our framework, and brings the plugin's object code into the application's address space, loads any necessary dependencies, and resolves all symbols. It then returns the value exported by the plugin as a normal Haskell value for use by the host application.

To use the new code provided by the plugin, the `stringProcessor` function from `rsrc` can simply be applied to a `String` (e.g., `stringProcessor rsrc "Hello World"`) as `rsrc` has type `StringProcAPI.Interface`.

In our framework, an application is free to load plugins at any time, not only during startup, and it may even reload plugins that change during program execution. For example, we might allow the string processor to change from one line of user input to the next. The following function applies the plugin's `stringProcessor` function to user input, reloading the plugin (and thus giving different behavior) if its source code has changed:

```
loop mod rsrc = do
      str   <- getLine
      rsrc' <- if isNew "StringProcPlugin.o"
               then reload mod "resource"
               else return rsrc
      putStrLn (stringProcessor rsrc' str)
      loop mod rsrc'
```

A session with this application, where we change the plugin source during its execution, may look like the following (with verbose messages included for clarity):

```
% ./a.out
Loading package base ... linking ... done
Loading object StringProcAPI
             StringProcPlugin ... done
> abcdEFGH1234
4321HGFEdcba

... Changing reverse to map Char.toUpper in the plugin source and recompiling.

> abcdEFGH1234
Reloading object StringProcPlugin ... done
ABCDEFGH1234
```

The first string is reversed, whereas the second is converted to upper case, after the code has been reloaded.

2.3.4 Loading plugins from other languages

Since Haskell has a standardized foreign function interface (FFI) [30], we can use our plugin infrastructure from any language supported by the FFI. This includes directly

supported languages, such as C, but also languages that can interoperate with C, such as C++ and Objective-C.

To use plugin functions such as `load` from C, function parameters and results must be marshaled between languages. If anything, this is more convenient in Haskell than other extension languages, such as Python or Lua, as the Haskell FFI emphasizes data marshaling on the Haskell side, which results in less marshaling code written in C for the host application. Marshaling of common data types, such as a C `(char *)` to and from a Haskell `String`, can even be performed automatically by the plugin infrastructure, as described in Chapter 3.

2.4 Runtime type checking

An operating system's dynamic loader (e.g. `dlopen` on UNIX, or `LoadLibrary` on Windows) is inherently unsafe. To find a value or function in a loaded object, the dynamic loader searches for a symbol of that name, and returns a reference to the value associated with that symbol. This is unsafe because a user may generate a plugin with the correct symbol name but the wrong type, which will, in all likelihood, lead to a crash. This lack of safety of operating system loaders implies that any safety guarantees have to be provided by higher-level software. The dynamic loader is thus part of the trusted computing base provided by the language, and its safety determines the safety of any software that uses it.

Unfortunately, GHC's dynamic loader – originally implemented to support loading of compiled code into the interpreter interface [139] – although it performs a few extra checks, is also unsafe. Moreover, type safe alternatives to the `dlopen` library that are based on typed assembly language [67] are not feasible for use in Haskell – our language under scrutiny – as GHC loads objects compiled via a standard C compiler.

In this section we describe the approach to dynamic type safety we develop in the context of Haskell plugins, so they are as safe as statically linked components, meeting our overall need for flexibility, without compromising safety.

As we don't want to simply trust the authors of plugin code to provide a value of correct type, we consider two techniques for ensuring type safety in a framework that ultimately employs an unsafe dynamic loader to do the low-level work:

- Firstly, we explore the use of dynamic typing to check plugin values as they are loaded; and
- Secondly, we describe a novel form of polymorphic dynamics, using the full type checker at runtime, in a form of staged type inference, to overcome limitations of the first technique.

2.4.1 Dynamically checked plugins

Plugins compiled separately from the host application must import and implement the application's plugin API, as described in Section 2.3.1. However, due to separate compilation of the plugin, we cannot guarantee that the plugin imports the correct API and provides a symbol of the correct type, prior to loading the plugin.

Hence, we need to be able to annotate the compiled object code with type information so that the plugin infrastructure can perform type checks when loading the plugin. Haskell provides a suitable `Dynamic` type, with a long history [2, 31, 88, 100, 167], in its standard libraries, which we employ for this purpose. A `Dynamic` encapsulates a value with a representation of its type in the form of a tag, which can be checked against its use. Unlike regular disjoint union types with tags, the set of types that may be stored in a `Dynamic` is not fixed. Values stored in a `Dynamic` have their type checking delayed until runtime, where a *typecase* is performed, to check the unwrapped value is of the expected type.

The plugin infrastructure requires that a plugin wraps its interface in a `Dynamic` value. The host application uses the `fromDynamic` function to perform the type cast from the `Dynamic` type to the interface's expected type, returning an error value if the cast fails. We then extend the dynamic linker's raw `load` interface with a version that performs a dynamic type check. This `dynload` function performs a conventional `load`,

but also unwraps the plugin interface as a dynamically typed value and performs a type check. The implementation of `dynload` is described in Section 3.3.

In the following example, the application expects a plugin to export a value of type `API.Interface` wrapped in a `Dynamic` type; however, the plugin author instead provides something of dynamic type `Int`. The `dynload` function catches this type error when it loads the object file, displaying an appropriate error message:

```
Loading package base ... linking ... done
Loading object API Plugin ... done
Fail: type <Int> doesn't match <API.Interface>
```

This approach gives us as much type safety as the underlying dynamics implementation permits. However, dynamically typed object code is not a defense against malicious code: while we can check whether the types are correct, there is no way to check whether the value associated with the type is indeed of `Dynamic` shape – we must trust the object file was produced by a correct Haskell compiler.

There is no reason, however, why the dynamic type check cannot be replaced with an alternative that implements more rigorous object code checking: `dynload` effectively provides hooks for adding verification to the underlying, unsafe dynamic loader. An application may even be able to choose between a number of trusted verification hooks depending on what its needs are.

2.4.2 Safety and flexibility

`Dynamics` in the Haskell standard libraries provide runtime type checking by way of the following function:

$$fromDynamic :: Typeable\ a \Rightarrow Dynamic \rightarrow Maybe\ a$$

In other words, `fromDynamic` expects to be applied to a value of type `Dynamic`. Unfortunately, this leads to a weakness in `dynload`: neither the operating system's nor GHC's dynamic loader can guarantee at runtime that an arbitrary symbol obtained from an object file has type `Dynamic`.

If a plugin provides the correct symbol name, but is bound to a non-`Dynamic` value, `fromDynamic` will simply crash – it cannot perform a type case on a value that doesn't provide a type tag. So, for this form of dynamic type checking to be safe in the presence of separate compilation, we need to be able to differentiate `Dynamic` values from others in the plugin's object code. Such a type would need to be distinguished by a special runtime tag, making it distinct from all other Haskell types. With our broad goal to avoid significant runtime modifications where possible, we set this option aside.

The standard `Dynamics` library also fails to deduce type equalities correctly in the presence of dynamic loading. It uses integer keys to represent types for fast comparison. However, under separate compilation, the key created for the plugin's type is not the same as the key generated for the application's version of that type, due to the implementation of hashing in the Haskell standard library. Luckily, this can readily be remedied, and we provide an alternative dynamics implementation in our plugin infrastructure library, which uses string comparison on the canonical name of a type, rather than on hash keys of the type representation. An alternative implementation, supported by GHC more recently, is a global type hashing scheme directly in the runtime, preventing collisions between `Dynamics` generated in different phases.

There is another problem in the use of the standard `Dynamics` library: it requires values which are instances of the `Typeable` class. This restricts the `Dynamic` type to a monomorphic value, as this mechanism cannot deal with type quantification; consequently, it cannot deal with any form of polymorphism. Fortunately, we can work around this to some extent using GHC's rank-N polymorphism. For example, a data type declaration such as:

$$\mathit{data\ Interface} \;=\; \mathit{Interface}\,\{\ \mathit{rsrc} \;::\; \forall\, a.\, \mathit{Eq}\, a \;\Rightarrow\; a \;\rightarrow\; a \;\rightarrow\; \mathit{Bool}\ \}$$

enables us to wrap polymorphic values such that the data type itself, namely `Interface`, remains `Typeable`, by hiding its polymorphic internals, ensuring that we can use such types dynamically.

This technique works well for plugin interfaces. Since we advocate the use of a single type representing the plugin API, storing polymorphic values wrapped inside

the API type is not an additional restriction. However, the need to wrap and unwrap the polymorphic value in another type can be tedious in some contexts, e.g. in an online Haskell interpreter, where the generated plugins—expressions entered at the prompt—may often be polymorphic.

2.4.3 Improving runtime type checking

A number of researchers have proposed forms of dynamics that support polymorphic values by some form of runtime type unification [3, 100, 129, 130]. We present a novel approach with lower implementation costs, which stays within the limits of the Haskell type system; nevertheless, we can dynamically check plugin values without placing any artificial limits on a plugin's interface type, while remaining as safe as statically linked code.

When loading a plugin, we need to check that the value we retrieve from the object is compatible with the type of the API; i.e., we need the type of `Plugin.resource` to be compatible with the `API.Interface`. This is the case if the following test module type checks:

```
module APITypeConstraint where

import qualified API
import qualified Plugin

_ = Plugin.resource :: API.Interface
```

Instead of using any form of userland dynamic types for runtime type checking, *we simply invoke the regular type checker on the test module* at runtime. This is not unlike the idea of *staged type inference* [142]. This approach has the advantage that it is entirely independent of any extensions of the type system supported by the underlying implementation. It does not require any extension, but it also does not inhibit the use of any features of the type system in plugin APIs; in particular, types involving type variables do not pose any additional difficulties—much in contrast to dynamic types. However,

invoking the full type inference machinery at runtime may sound expensive; and we quantify the costs in Chapter 3.

In our plugin library, we generate a temporary module, corresponding to the test module above, to check the conformance of plugin code to plugin APIs when the function `pdynload` ('p' for 'polymorphic') is used to load the plugin. If GHC's type checker accepts the `APITypeConstraint` module, it is safe to load the plugin, as long as this code use pattern is followed. More precisely, `pdynload` is implemented as follows:

```
pdynload obj incs pkgs ty symbol =
     do tmp <- newTyConstraint ty symbol
        err <- unify tmp
        if null err
            then do v <- load obj incs pkgs symbol
                    return (Just v)
            else do mapM_ putStrLn err
                    return Nothing
```

The function `pdynload` is closely related to `fromDynamic` or an arm of a *typecase* expression, as it is predicated on a type equality. It checks that the value exported by the plugin, referred to by `symbol`, has the type determined by `ty`. It does so in two steps: first, `newTyConstraint` generates a temporary module similar to `APITypeConstraint` above, and second, `unify` invokes GHC's type checker on this temporary module. If the test is successful, we can safely load the plugin; otherwise, we return an error.

This leads to the question of how type information is propagated from the API and plugin source to the `APITypeConstraint` module implementing the type constraint. Luckily, no further mechanism is needed, GHC's standard support for separate compilation is sufficient. In particular, GHC generates an *interface file* ("`.hi` file") with type information for each compiled module's exported values. This type information is sufficient for the purposes of checking plugins, as the values we are interested in must already be in the export list. Hence, staged type inference for the restricted case of plugins is more easily implemented than is generally the case of arbitrary program fragments.

As an example, consider the following plugin source:

```
module Plugin where

resource :: (Num t) => t
resource = 0xBAD
```

This module on its own is well typed. However, it leads to a type error when compiling the `APITypeConstraint` module, as `resource` is clearly not of type `API.Interface`. Since `resource` is exported, its type will be in `Plugin`'s interface file, which—in the case of GHC—has the following form:

```
interface Plugin where
resource :: forall t. (GHC.Num.Num t) => t
```

In contrast, the interface file generated from the application's API module might be the following:

```
interface API where

data Interface = Interface
     (GHC.Base.String -> GHC.Base.String)

plugin :: Interface
```

Given these two interface files, the type checker will reject the type expression:

```
Plugin.resource :: API.Interface
```

when checking the test module with the following error message, indicating that the `Num` class restriction on the plugin value is not compatible with the API type:

```
Checking types ... done
<typecheck>:1:4:
 No instance for (Num API.Interface)
  arising from use of ‘Plugin.resource’
  at <typecheck>:1:4-18
 In the expression:
  Plugin.resource :: API.Interface
```

Note that none of this requires access to the plugin source. All that is required is that plugin object files are accompanied by their interface files, which the compiler generates automatically when it creates the object files.

In summary, we considered two fundamentally different mechanisms for type safe plugins: (1) limited runtime type checking using dynamic types and (2) full runtime type checking via type representation serialization and employing the existing type checker to perform the type check, overcoming the limitations of the first approach. The latter allows flexibility, ensuring any type discipline used at compile-time will remain valid for runtime-loaded components.

2.5 First class modules

The third component of our framework for dynamic extension is a requirement to represent dynamically loadable modules as first class values. This is essential if we are to support manipulation of the loaded object as a unit, for example, to hot swap it with a replacement.

2.5.1 A type class for first class modules

As described in Section 2.3.1, on the plugin side, we require the interface between the loaded component and the application to be specified via an interface dictionary structure, listing the methods and types to be shared between the plugin and the application. For simple extensible applications this interface will be a data structure containing a small set of functions to be provided by the plugin code. For more complex applications the interface will be represented via a Haskell type class, capturing precisely the notion of a required interface, with a phantom type to represent each plugin.

To illustrate this model for interface types, we show an example from Lambdabot [2], a network chat service built around a suite of services implemented via plugins. Firstly, the plugin interface is defined as a typeclass, `Module`, here simplified for illustrative purposes:

[2]Lambdabot, http://www.cse.unsw.edu.au/~dons/lambdabot.html

```
class Module m where
       moduleHelp      :: m -> String -> String
       moduleInit      :: m -> ...
       moduleExit      :: m -> ...
       process         :: m -> String -> String -> ...

       moduleInit _ = return ()
       moduleExit _ = return ()
```

Each plugin will define a new phantom type, `m`, to be used to index the appropriate class instance representing the code stored in that plugin. A set of functions required by plugins are specified as type class methods, in this case, functions to provide help documentation, initialize and exiting functions (to set up and tear down the plugin), and a `process` function, providing the core functionality of the plugin. Additionally, the type class mechanism smoothly incorporates the notion of default implementations. The use of type classes to capture a notion of module interfaces, and the connection to abstract types was noticed by Wadler and Blott [161], further developed by Läufer and Odersky [92], and continues to be a topic of interest [166, 157].

2.5.2 Existential types

With such a type class in place, we can uniformly manipulate plugins in the host application, with a type guarantee that they will implement the above interface. A basic implementation of such a plugin would be:

```
module Hello where

data Hello

instance Module Hello where
    moduleHelp _ _ = "The simplest plugin"
    process _ _    = return "Hello, world!"
```

The host application can load plugins implementing this interface, using mechanisms described in the previous sections, however, the application now needs a way to manipulate objects containing code instantiating this class uniformly.

Our approach, from Haskell folklore, is to represent such values via an existential type [25, 124, 90] in the style of Läufer, [91], with quantification in "the negative position":

$$data\ MODULE\ =\ \exists\, m\, \exists\, s.\, Module\ m\ s\ \Rightarrow\ MODULE\ m$$

The host application can now treat all dynamically loaded plugins as values of type `MODULE`, with a static guarantee the interface will be supported via the `Module` type class. We also support hidden state representations, and other abstract type components, via this interface. These loaded modules may be stored in data structures, combined and composed, and unloaded.

The use of *existential types* is supported by many existing Haskell implementations, including GHC, Hugs[3], and nhc98 [133]. Existential types are commonly associated with dynamic types [44, 45], and have been previously used to provide *dynamic dispatch* in Haskell [90]. Indeed, since plugins are often used to dynamically dispatch functions, more research is needed on possible interactions between existential types, plugins and module systems.

The contribution of approaches to first class modules in our framework is to ensure flexible manipulation of loaded objects as first class values, making possible powerful manipulation of modules as units by the programmer.

2.6 Runtime compilation

Along with dynamic linking, runtime type checking and access to modules, an ability to generate and compile new code dynamically is a component of the framework for dynamic extension described in this thesis.

The capabilities for dynamic code generation are implemented in a *compilation manager* subsystem of the plugin library infrastructure. It arranges for source code to be conditionally compiled, by determining whether it has changed since its previous

[3]Hugs, http://haskell.org/hugs

compile. It also implements the merging of extra code into a plugin's abstract syntax at compile-time, allowing compulsory definitions to be included automatically, and functions for `eval()`-like features.

2.6.1 Invoking the compiler

Our plugin library provides a `make` function that accepts the path to a Haskell source file as an argument, and invokes GHC to compile the source to an object file (whose output path can be controlled by the host application), which can then be dynamically loaded. It is important for the performance of plugin reloading that a call to `make` does not unnecessarily recompile the source code. Invoking GHC as an external process adds significant overhead (even when GHC detects that the source has not changed), so we detect modification times on the source before calling GHC. This makes `reload` a cheap operation: it only invokes GHC to recompile a plugin when compilation is actually required.

2.6.2 Generating new code at runtime

The ability to load and reload plugins during the execution of a program enables applications that go beyond the standard idea of programs that are extended with plugins. For example, we can have programs loading plugins that are generated (at runtime) by the same program, and illustrate this idea with the example of an interactive Haskell shell.

As an example of dynamically reloadable plugins, we present an application named *plugs* (**Pl**ugin **U**ser's **G**HCi **S**ystem), an interactive Haskell shell reminiscent of *hi* (**h**make **i**nteractive) [163] or GHCi, that instantiates the notion of runtime evaluation for Haskell.

A user enters strings of Haskell expressions at a prompt, which `plugs` immediately evaluates and prints to the screen. It achieves this by writing each expression (together with a wrapper) into a plugin module, which is then compiled, loaded, executed, and

```
          __
    ____ / /_  ______ ______
   / __ \/ / / / / __ `/ ___/
  / /_/ / / /_/ / /_/ (__  )
 / .___/_/\__,_/\__, /____/
/_/            /____/

Type :? for help
Loading package base ... linking ... done

> case 7 * 8 of x -> x + 1
57
> :load Data.List
> sort [7,3,8,6,4,2,0,1,9]
[0,1,2,3,4,6,7,8,9]
```

Listing 2.1: Haskell interpreter, plugs, example session

then unloaded. This process is closer to *hmake interactive* than to GHCi, as the code is compiled and not interpreted. An example session is displayed in Listing 2.1.

The system works as follows: together with `import` declarations corresponding to all in-scope modules, it dumps user input into a Haskell source file constituting a plugin. For example, if the user types in `sort [7,3,8,6,4,2,0,1,9]`, the plugin essentially contains:

```
module UserCodePlugin (resource) where
import UserCodeAPI (plugin)
import Data.List


resource = plugin {
  code = show $ sort [7,3,8,6,4,2,0,1,9]
}
```

The actually generated plugin is slightly more complex: it contains line number hints in the source code, which enable the compilation manager to produce informative error messages in the case of malformed user input, and mechanisms to intercept exceptions thrown by the system.

The system then proceeds to compile, load, and execute this plugin as before, using the `make` and `load` functions of our plugin library. The entire system is only 70 lines of Haskell. Here is the core of `plugs`:

```
source      <- writeOut userString
(obj,_,err) <- make source []
if not (null err)
  then mapM_ putStrLn err
  else do (m,v) <- load obj ["."] [] symbol
          v'    <- Exception.evaluate (code v)
          putStrLn v'
          unload m
```

The `writeOut` function generates the plugin from user input, while `Exception.evaluate`, which is part of GHC's exception library, controls runtime exceptions.

The benefit of using dynamically loaded plugins to implement plugs is clear: we can reuse an existing Haskell compiler and runtime system to add an interactive Haskell environment to any application, requiring only a few lines of code. Compared with *hmake interactive*, plugs omits the linking phase, which leads to significantly faster turn-around times (about a factor of two). Moreover, it is only a small step from plugs to an embeddable Haskell engine that provides an application with Haskell scripting capabilities.

2.6.3 Cross-language extension

The linking and loading mechanisms presented are not restricted to just Haskell code. As an example, we have extended the `eval` engine on the Haskell side by exporting it as a C function. It evaluates a Haskell expression stored as a C string in the IO monad, with a similar code generation strategy as used by `plugs` in the previous subsection. The C string is used to dynamically generate a plugin that is loaded back into the application, and the resulting Haskell value is returned to the caller. Variants of `hs_eval`

are provided for the basic Haskell types, allowing the plugin result type to be checked against its usage. Here, we present a C program that loads a Haskell plugin to evaluate a fold:

```
#include <stdio.h>
#include "RunHaskell.h"

int main(int argc, char *argv[]) {
  int *p;
  hs_init(&argc, &argv);
  p = hs_eval_i("foldl1 (+) [0 .. 10]");
  printf("%d\n", *p);
  hs_exit();
}
```

When executed, this C program calls the Haskell runtime system, which arranges for our Haskell plugin library to compile, load and execute the C string passed to `hs_eval_i()`, returning a pointer to the integer result to the caller. Dynamic typing on the Haskell side checks that the compiled Haskell value matches the type expected by the C program, yielding an error (indicated by a null pointer) if the C program expects the wrong type from the Haskell string.

```
% ./a.out
Compiling plugin ... done
Loading package base ... linking ... done
Loading object Plugin ... done
55
```

2.7 Hot swapping

With the ability to load, link, check and recompile native code on the fly, the next aspect of the overall framework to tackle is support for true code hot swapping – where components of a running application may be replaced without loss of state or service. In the limit case, all code is replaceable dynamically, yielding extremely flexible architectures. Dynamic applications are able to add, remove, and exchange code at runtime. This increases application flexibility by facilitating runtime configuration, user exten-

sion, hot code swapping, runtime meta programming, and application configuration by EDSLs.

Stallman's Emacs editor [145] is a widely known example of a dynamic application from the Lisp world—Stallman calls it an *on-line extensible system*. Emacs consists of a small core, implemented as a conventional C program, which is extended to a fully-fledged editor by a large body of Lisp code, which is byte-compiled and dynamically loaded. The application core contains the Lisp engine as well as those editor primitives that would not be sufficiently efficient in interpreted byte code. Such an application is *mostly dynamic,* but the code of the static core cannot change dynamically; thus, some functionality cannot change either. In the case of Emacs, the editor primitives and any other primitives provided by the Lisp engine are fixed. This includes the choice of user interfaces, which can be text-based or depend on one or more GUI toolkits (such as the X Window System).

An entirely different example of a mostly dynamic application is the Linux kernel. It can load and unload *kernel modules* at runtime using an in-kernel module loader. Just like Emacs, Linux contains fixed functionality in its static core; some of this functionality (such as the process scheduler, the I/O scheduler, and the virtual memory subsystem) would benefit from being dynamically configurable. In contrast to Emacs, both the core as well as the modules of Linux are implemented in C. As a consequence, kernel configuration is achieved via mechanisms independent of dynamically loaded code—which is in contrast to Emacs, where configuration files are just stylized Lisp files.

We move beyond these partially-dynamic applications by exploring a *fully dynamic* software architecture whose static core is minimal; i.e., the core contains only what is needed to load the rest of the application at runtime.

We experimented with two fully dynamic applications: the extensible editor *Yi* and the IRC[4] robot *Lambdabot*. Such fully dynamic applications enable hot code swapping. We demonstrate this by presenting a method for replacing the *currently executing*

[4]Internet Relay Chat (IRC) is a standardized online chat system.

application code by a variant without losing the application state. Dynamic code replacement is always a tricky business; in a purely functional language, the semantic consequences are even more subtle, particularly in the presence of laziness and thunks. We achieve it via an aggressive strategy: re-doing the dynamic boot process while preserving a handle to the application state, destroying all traces of the obsolete code. This process is fast in practice and avoids a large range of semantic issues that other approaches to dynamic software updates incur. Dynamic software updates, and also dynamic extension and configuration, are generally convenient, but they are of special value in high availability applications.

The contribution of our code reloading system is:

- Fully dynamic software architectures; and
- Hot code swapping that preserves application state;

2.7.1 Dynamic architectures

The foundation for extensibility in our framework is a dynamic software architecture based on runtime code loading. Applications are partitioned into three main components: (1) a static bootstrap core; (2) a support services library; and (3) a suite of plugins. Under this framework, the binary itself consists solely of a static core of a few dozen lines of Haskell. The purpose of the static core is to provide an interface to the dynamic linker and to assist in hot swapping. The static core has *no* application-specific functionality.

The static core, with the help of the dynamic linker, loads any configuration data (as plugins) followed by the main application code. Control is then transferred to the main application's entry point—its `main` function—and the program proper starts up.

During execution the user may invoke the Haskell dynamic linker library to recompile and reload any configuration files, or to reload the application itself. The state of the application is preserved during reloading (hot swapping), with the result that

new code can be integrated or existing code be replaced, without having to quit the application.

2.7.2 State preservation

Once an application is running we would like to be able to hot swap application code. This is a profound procedure: we want to replace application code while the application is running, without losing state—allowing us to "fix-and-continue" or to incorporate new functionality on the fly. To do so, we must determine how to preserve state.

When updating code dynamically, we wish to continue execution from exactly the point at we reached prior to the dynamic update. To do this, we must reconstruct any relevant application state built by the code before the update. Reconstructing an application's state in Haskell is usually simple. By default entire Haskell programs are purely functional, so simply re-entering the application via its normal entry point is enough to recreate the application's state from a previous run, assuming all environmental parameters are the same. There is no need to actually store state between runs, as we are always able to reconstruct it.

However, some applications require large amounts of mutable state, and reproducing the mutations to that state may be infeasible. A text editor, like Yi, is such an application. Editor buffers are potentially very large flat byte arrays, constructed over the entire running time of the application. Any copying is prohibitively expensive. In our design, buffers are stored in a single global state component of the editor. What we need to do is capture this state during the update, and then inject it into the new updated code, continuing exactly where we left off.

This brings us to a central reason for using a dynamic application structure centered around a minimal static core; i.e., a reason besides the desire to modify all functionality of the application at runtime. We use the static core to keep all global state while the dynamic linker reloads plugins. For this to work it is crucial that all requests to load or re-load plugins go via the static core, as mentioned previously. Ultimately,

the static core is the only safe place to keep the global state during reloading, as the whole dynamic application code may be replaced. In other words, we structure the entire dynamic application so that it takes its state as an argument, making it purely functional again, and hence, enabling state-preserving hot swapping.

At this point, Haskell offers significant advantage over languages which encourage the use of global state. Unrestricted use of global state leads to a multitude of values scattered throughout the program; hence, bookkeeping of such a dispersed state becomes difficult and potentially infeasible. Hot swapping is only practical if the application already has disciplined use of global state—as is the case by default in Haskell programs. Without restricted state, runtime system or operating system support seems necessary to deal with the state problem.

2.7.3 State preservation as types change

Replacing code dynamically while still preserving state works well when extending functions. New functionality, bug fixes and extensions can all be integrated without having to quit the application. However, reinjection of global state is only safe if the definition of the state data type is unchanged in the new code. If, for example, the buffer type changes to include an extra field and we restore the old state containing an old buffer value lacking this field, the new code will crash—it simply is not compiled to handle a value of the old type. What we need is a safe way of injecting an old state value into a new, different state type. This problem, state transfer, has been considered in work relating to operating systems supporting hot swapping [144, 13] as well as in the work of Hicks [64].

We tackle this problem by reducing it to a form of parsing: we define an error-handling parsing function to inject values of the old type into the new type. To transfer values between types we convert the state value into a binary representation, via serialization. Prior to rebooting the application, we encode the old state value in this binary format. After injecting this binary state value into the new code, the value is parsed to construct a state value of the new type, where missing, or extra, fields are

either ignored or replaced with default values. In this way global state values can be preserved over data type changes, and the state value is effectively immortal. We illustrate this via *XMonad*, a tiling window manager for Unix.

A more complex solution could use a version tag in the state data type and a class of injection functions to achieve binary compatibility between arbitrary versions of the state type.

2.7.4 Persistent state

The problem of preserving state during dynamic code reloading is tightly connected to the general problem of persistence. Applications that are able to extract and inject their entire state can be extended to support persistence via the usual mechanisms—e.g. serialization of values to binary representations [164]. When passing the state value from dynamic code to the static core, the core can arrange to write the state to disk, and on rebooting, reread this state, passing it back to the dynamic main.

The problem of preserving state across code loading effectively forces us to separate and sanitize use of state in the application. Extending the state preservation to full persistence is relatively simple once we reach that point, with the caveat that very large state components will remain expensive to serialize.

2.8 Embedded languages

Plugins extend a host application at predefined points, which some extension frameworks call *hooks*. In our case, the symbols that the base application loads from a plugin are hooks that can be redefined by extension code written in Haskell (independent of whether the base application is written in Haskell or in another language using Haskell plugins via the FFI). Extension languages generally simplify application extension by virtue of two properties:

1. Extensions at well defined places based on a well defined API are conceptually simpler than arbitrary additions to the application source. They are also more robust with respect to changes in the application base.

2. Extension languages are often higher-level than the language in which the application base is implemented (e.g., Lisp in Emacs, whose core is written in C).

A variant of the second property is where the extension language is not a general purpose language, but *domain specific*: one tailored to the application domain. This can simplify the complexity of extensions to the level where they can be implemented by non-programmers.

Work over the past fifteen years has demonstrated that Haskell is an excellent host language for embedding *domain specific languages* (DSLs): implementing domain specific languages as a library in a general-purpose host language, which saves the overhead of implementing a new language from ground up [34, 48, 70, 73, 116, 138, 165, 37, 5, 18, 96, 32, 9, 135, 53, 81, 62, 146] etc.). Domain specific languages implemented in this fashion are called *embedded domain specific languages* (EDSLs). In this context, when Haskell is used as an extension language, it seems particularly desirable to provide plugin authors with a domain specific notation. In other words, Haskell plugins make EDSLs more applicable, which in turn makes Haskell an especially attractive extension language.

We use domain specific languages, embedded in Haskell, for a number of applications described in later chapters, including:

- Typed application configuration files;
- Lightweight parsers; and
- An extension language for a text editor.

2.9 Summary of the framework

We have presented a framework for adding dynamic extension capabilities to a statically typed functional programming language. The framework combines techniques for dynamic linking of native code modules, runtime type checking, first class modules, module hot swapping, and embedded languages to support a range of extension models and approaches. The framework is implemented for Haskell, and feasibility is shown through a set of diverse applications.

We introduced approaches to *full* runtime type checking of dynamics, representations of Haskell modules via existentially typed instances, fully dynamic application architectures capable of code hot swapping, and novel domain specific languages for application configuration.

In Chapters 3, 4 and 5 we will examine the realization of these systems in detail, while exploring language approaches to improving the ease of extension in Chapter 6.

Chapter 3

Dynamic linking

Extension languages enable users to expand the functionality of an application without touching its source code. Commonly, these languages are dynamically typed languages, such as Lisp, Python, or Lua [75, 77], or, alternately, domain specific languages, which support runtime *plugins* via dynamic loading of components.

One of our goals is to develop the infrastructure necessary to ensure Haskell can be comfortably used as a statically typed extension language for both Haskell and foreign-language applications supported by the Haskell FFI. To do this we must ensure *type safe* dynamic loading of native code plugins is possible.

3.1 Overview

In this chapter we describe two approaches to dynamic loading of Haskell plugins, based on *dynamic types* [2, 3, 31, 88, 100, 167] for type safety. The maturity of our approach is illustrated with a broad selection of applications that use Haskell plugins based on this framework.

The library we have implemented, *hs-plugins*[1], is capable of dynamically loading object code into a running Haskell application's address space, utilizing the object loading capabilities of the Glasgow Haskell Compiler's runtime system [139]. The functions

[1] Available online at http://www.cse.unsw.edu.au/~dons/hs-plugins/

provided by the plugin are transparently available to the application, appearing as standard Haskell values. In addition, the plugin library tracks module dependencies, enables manipulation of the plugin source as abstract syntax, and supports different levels of trust between application and plugin authors with respect to the type safety of plugins. Applications written in foreign languages may use the plugin library's C language bindings, enabling any application that can interface with C to easily support Haskell as an extension language.

In Section 2.1.1 we introduced the notion of dynamic linking which is used in this thesis to build a general framework for dynamic extension via native code plugins, and in Section 2.3 we described our approach to dynamic linking in Haskell with an introduction to linker use via plugins. Unchecked dynamic linking is inherently unsafe, so in Section 2.1.2 we defined the concept of runtime type checking of components, and outlined our solution in Haskell in Section 2.4.

In this chapter we will describe mechanisms for dynamic linking in detail, covering our implementation, as well as our approach to runtime type checking. We will also describe the applications that type safe dynamic linking enables, including module-based network servers and a computational chemistry application, before covering related work.

Our primary contribution in this chapter, then, is the following:

- We describe the design of a dynamic linking library for Haskell;
- We describe two mechanisms for adding type safety to dynamically loaded code, based on dynamic types;
- We extend the architecture to support Haskell plugins in foreign languages, and plugin loading of foreign languages into Haskell;
- We demonstrate feasibility through a series of practical applications.

We now discuss the principles behind the implementation of a dynamic linker for native code plugins in Haskell.

3.2 Dynamic linking in Haskell

The *dynamic loader's* purpose is to load object code into an application's address space, and to arrange for that code to be available to the application. To achieve this, it needs to interact with the Haskell runtime system, which typically uses the operating system's dynamic loader for some of its work (such as parsing and inspecting object files). The dynamic loader also requires a high-level mechanism for resolving module-level dependencies between plugins and libraries, as any dependent libraries or modules must be loaded prior to loading the plugin itself – the GHC runtime system cannot do this. Chasing dependencies and finding libraries is a significant proportion of our dynamic loader's code.

3.2.1 Runtime loading

The object loading facilities of the GHC runtime system, implemented in C, provide single module loading and searching with appropriate symbol relocation and resolution, but without dependency chasing. Object loading comprises part of the linker created for the Haskell Execution Platform [139], and is used by GHCi to load Haskell libraries during an interpreter session. Following GHCi's approach, our plugin library implements a foreign function interface to the object loader and linker of the runtime system, making them available in Haskell. We then extend the low-level functionality by providing full library and module dependency searching and loading, in a similar style to GHCi.

The GHC runtime provides the following primitive services for loading object files, presented here as Haskell bindings to `Linker.h`, the system in the GHC runtime responsible for integrating new code into a process:

```
foreign import ccall unsafe "initLinker"
   initLinker :: IO ()

foreign import ccall unsafe "loadObj"
   c_loadObj :: CString -> IO Bool
```

```
foreign import ccall unsafe "resolveObjs"
   c_resolveObjs :: IO Bool

foreign import ccall threadsafe "lookupSymbol"
   c_lookupSymbol :: CString -> IO (Ptr a)

foreign import ccall unsafe "unloadObj"
   c_unloadObj :: CString -> IO Bool

foreign import ccall unsafe "addDLL"
   c_addDLL :: CString -> IO CString
```

Several primitive actions are supported: initializing the linker (an idempotent operation), which allocates symbol table structures; loading a static object file (populating a symbol table with addresses – possibly by memory mapping the object file); resolving symbols (finding the actual address corresponding to the symbol); looking up a symbol (which returns the address of where the code corresponding to the symbol is loaded in memory); unloading (which removes references to symbols) and the ability to load dynamic link libraries (such as a `.so` object archive on Unix). Notice how a symbol table in a dynamic linker corresponds approximately to an *environment*, when abstractly considering a programming language's execution model.

3.2.2 Basic plugin loading

The most basic interface to object loading in our framework is via the `load` function. A call to `load` imports a single object file into the caller's address space, returning the value associated with a specific symbol requested. Libraries and modules that the requested module depends upon are loaded and linked in turn via dependency chasing (described in Section 3.2.3). The following Haskell type describes this operation:

$$\begin{aligned} \mathit{data}\ \mathit{LoadStatus}\ a\ = &\ \mathit{LoadSuccess}\ \mathit{Module}\ a \\ \mid &\ \mathit{LoadFailure}\ \mathit{Errors} \end{aligned}$$

$$
\begin{aligned}
\mathit{load} \quad & :: \mathit{FilePath} \\
& \rightarrow [\mathit{FilePath}] \\
& \rightarrow [\mathit{PackageConf}] \\
& \rightarrow \mathit{Symbol} \\
& \rightarrow \mathit{IO}\ (\mathit{LoadStatus}\ a)
\end{aligned}
$$

The first argument is the path to the object file to load; the second argument is a list of directories to search for dependent modules (such as modules describing interfaces to the application). The third argument is an optional list of paths to user-defined Haskell package databases. The `Symbol` argument is the symbol name of the value to retrieve. `load` returns a pair of an abstract representation of the module (for later calls to `reload` or `unload`) and a Haskell value associated with the symbol.

The value returned must be given an explicit type signature, or provided with appropriate type constraints so that the Haskell type system can determine the expected type returned by `load`, as the return type is polymorphic.

An example of loading a single object file using this framework:

```
do mv <- load "Plugin.o" ["api"] [] "resource"
   case mv of
        LoadFailure msg -> print msg
        LoadSuccess _ v -> return v
```

We also provide convenient functions for unloading and reloading objects, and unloading an object with all its dependencies:

$$
\begin{aligned}
\mathit{unload} &:: \mathit{Module} \rightarrow \mathit{IO}\ () \\
\mathit{reload} &:: \mathit{Module} \rightarrow \mathit{Symbol} \rightarrow \mathit{IO}\ a \\
\mathit{unloadAll} &:: \mathit{Module} \rightarrow \mathit{IO}\ ()
\end{aligned}
$$

`unload` takes a module and unloads the corresponding object from the address space. `reload` takes a module and a symbol. It unloads the module, re-compiles its source if specified in source form, and then reloads the resulting object, returning the new

code associated with the symbol. It is similar to performing `unload` and recompiling the source of the object, followed by a `load`. We introduce a compilation manager for source plugins in Chapter 4, which describes applications for source plugins, and their safety is discussed in Section 3.5.2.

3.2.3 Dependency chasing

The problem with dependency chasing is that object files do not contain the information necessary to determine which other object files a given plugin depends upon. GHC has the same problem, and it solves the dependency issue by storing module and library dependency information for an object in the corresponding interface (`.hi`) file.

Consider the following Haskell module:

```
module M where

import Data.List

n :: Int
n = 42

f :: Int -> a -> [a]
f n a = replicate n a
```

Which, when compiled to an object file, is accompanied with an interface object containing the following information (summarized):

```
Magic:   33214052.
Version: [6, 1, 2, 1].

interface main:M 6121
  interface hash:   00034c6556404f86d55450389df23ce2
  ABI hash:         bcb3ae1a0ea4893c956b38da5325589d
  export-list hash: 325831d70dbe43dd26e5608822c9553a

where
    export main:M f n

    package dependencies: base ghc-prim integer-gmp
```

```
  import base:Data.List 8cda3d6f394921d3576148f5bf16f6fe
  import base:GHC.List  63b8ccecc5c0a4d90983b6029ded9540
  import base:Prelude   60a025ec08e8cd16b30877bd031a20f9

f :: forall a. Int -> a -> [a]
     Arity:      2
     Strictness: U(L)L
     Unfolding:  (GHC.List.replicate)

n :: Int
     HasNoCafRefs
     Strictness: m
     Unfolding: (GHC.Types.I# 42)
```

The interface distributed with the object contains a lot of useful information for the dynamic linker, including hashes of the object's source and binary interfaces; a list of exported functions (`f` and `n`); a list of external packages the object depends upon (and thus must be loaded prior to evaluating code in the module) along with hashes of the modules in those packages that are required; and, finally, full serialized representations of the types of the values exported from the object, and unfolding and strictness information for the optimizer. We will take advantage of this type representation in Section 3.4.

As all plugin objects are compiled with GHC, this interface file is always created with the object, and we are able to use it for our own purposes. Interface files are stored in a compressed binary format that makes them quite difficult to parse, so we have extracted GHC's binary `hi` file parser as a library module.

When an application makes a call to `load` an object file, we first parse the interface file associated with that object, extracting module and library dependencies. We then load these dependencies in their dependency order, before finally loading the plugin itself, resolving its symbols, and returning the requested value to the application.

Unfortunately, there exists another problem with dependencies in Haskell: library dependencies are noted in an interface file by the canonical name of that library. For example, the library `/usr/local/lib/ghc-6.2.1/HSunix.o` will be stored in the interface file as the simple string '`unix`'. Complicating this further are libraries that can

depend on other libraries (including C libraries) – these additional dependencies do not appear in the interface file.

Fortunately, the necessary information required to find the full path and dependency information of a library is stored in GHC's library configuration database, to which we have access. To parse this information, we have implemented a package file parser for this database, enabling us to reconstruct all the information required to find and load a library.

Once the interface file is read and all the package information is resolved, we are finally able to find and load a plugin's dependent libraries and modules. We maintain some mutable state in the dynamic loader to keep track of libraries and modules that have been loaded, mirroring the interior state of the GHC runtime, allowing us to skip loading components when they are needed repeatedly. We also use this state to store a map from the libraries' canonical names to their full paths. Internally, the state is locked using `MVars`, so that concurrent access is synchronized and safe.

With direct object loading, unloading, and support for packages, along with cascading dependency chasing, we have designed and implemented the core of a dynamic linker for Haskell, in the `hs-plugins` library. We now turn to the question of how to ensure dynamically loaded code is type correct.

3.3 Typing dynamic linking

In the previous section we introduced plugin loading in Haskell, deferring the question of how to ensure plugin loading is type safe. Earlier, in Section 2.1.2, we defined the concept of runtime type checking of components, and outlined our solution for Haskell in Section 2.4. We now describe the mechanisms of our approach to type safe plugins in detail, and quantify the costs and benefits of the two approaches we develop.

Recall the type of the plugin loading function:

$$
\begin{aligned}
\mathit{load} \;::\; & \mathit{FilePath} \\
& \rightarrow [\mathit{FilePath}] \\
& \rightarrow [\mathit{PackageConf}] \\
& \rightarrow \mathit{Symbol} \\
& \rightarrow \mathit{IO}\ (\mathit{LoadStatus}\ a)
\end{aligned}
$$

Our first contribution to type safety of this function is to employ dynamic types, in the style of Cheney and Hinze [31] and Lämmel and Peyton Jones [88] (Baars and Swiestra have a similar solution [10]), and apply them to dynamic loading. We define a type safe `dynload` function with the following type:

$$
\begin{aligned}
\mathit{dynload} \;::\; & \mathit{Typeable}\ a \\
& \Rightarrow \mathit{FilePath} \\
& \rightarrow [\mathit{FilePath}] \\
& \rightarrow [\mathit{PackageConf}] \\
& \rightarrow \mathit{Symbol} \\
& \rightarrow \mathit{IO}\ (\mathit{LoadStatus}\ a)
\end{aligned}
$$

`dynload` has the same interface as `load`, with an additional constraint that the value we return must be `Typeable`. This implies that the type defined in an API must derive `Typeable` for dynamic typing to succeed. `dynload` performs a `load`, which returns a value of type `Dynamic`, and then performs a type check to ensure that the value in the

object file is the type that the application requires, otherwise returning `LoadFailure` with a type error message.

The implementation of `dynload` depends on `load`, with the addition of a `typecase` construct following a successful (unsafe) load:

```
dynload obj paths pkgs sym = do
    s <- load obj paths pkgs sym
    case s of
         e@(LoadFailure _)   -> return e
         LoadSuccess m dyn_v -> return $
            case fromDynamic (unsafeCoerce# dyn_v :: Dynamic) of
                Just v' -> LoadSuccess m v'
                Nothing -> LoadFailure ["Mismatched types"]
```

As described in Section 2.4.1 our model for dynamically typed plugins requires the plugin author to wrap their type as a `Dynamic` value. The plugin loader, when extracting a symbol for the user, retrieves it as a Haskell value of polymorphic type that is required to be wrapped in a `Dynamic`. A type cast is used to cast the generic value to a `Dynamic`, without changing the underlying representation. The `Dynamic` is then unwrapped, using `fromDynamic`, which converts a dynamically typed Haskell value back into an ordinary value of the correct type, as determined by the surrounding type context, or `Nothing` in case of an error:

```
fromDynamic :: Typeable a => Dynamic -> Maybe a
fromDynamic (Dynamic t v) =
      case unsafeCoerce# v of
        r | t == typeOf r -> Just r
          | otherwise     -> Nothing
```

The key step is the type equality expressed as value equality on the name of the type, implemented via the `typeOf` function, which returns a name representation of a Haskell type (this approach to type cast is a form of *nominal* type analysis, rather than *structural*).

A `Dynamic` in Haskell is a pair of a polymorphic Haskell value and its type representation:

$$data\ Dynamic\ =\ Dynamic\ TypeRep\ (\forall\ a.\ a)$$

To construct values of this type, we wrap them, discarding any knowledge the compiler has of the value's type via a coerce. This is only a type-changing operation. The representation of `v` does not change.

$$toDyn\ ::\ Typeable\ a\ \Rightarrow\ a\ \rightarrow\ Dynamic$$
$$toDyn\ v\ =\ Dynamic\ (typeOf\ v)\ (unsafeCoerce\#\ v)$$

The type representation of the value is constructed, and will be used as evidence in a later stage to `fromDynamic` that the type of `v` is known. For a full treatment of the implementation of the `Dynamic` type in Haskell, and type safe cast in this manner, see Lämmel and Peyton Jones in [88, sections 8 and 9.3], and in Cheney and Hinze [31].

3.3.1 Limitations

While this approach is convenient and direct, it suffers from a number of safety and flexibility concerns, as described in Section 2.4.2.

Firstly, the approach based on `Dynamic` suffers from inflexibility – type representations can only be constructed for monomorphic types, significantly limiting the expressivity of loaded object code. Many features of Haskell's type system cannot be reflected into type representations that can be matched at runtime using equality of types' names. To meet our overall objective of allowing the full range of Haskell types to be used in dynamic extension, we must provide *polymorphic dynamics*.

Secondly, there is a safety concern: we must trust the loaded object indeed provides a symbol with `Dynamic` type, otherwise the `unsafeCoerce` in `fromDynamic` will fail with a crash. Alanko [4] described the approach in Section 3.3 (in contrast with proposals for typed assembly language) as:

> "... probably safe enough for most practical purposes, but ... can be broken by malice or otherwise unfortunate circumstances."

To overcome this constraint, we develop an approach to polymorphic dynamics, where type equality of representations is replaced with unification of a type constraint between plugin and application code, based on type information stored with the object file.

Thirdly, we modify the trust argument for type checking in Chapter 4, where we ensure only valid Haskell objects are loaded, removing threats from arbitrary object files by systematically requiring source code plugins.

3.4 Polymorphic dynamics

To overcome the restriction `Dynamic` places on monomorphic types for dynamic components, we introduce a form of polymorphic dynamics. Rather than relying on an approach to constructing type representations based on ad hoc polymorphism [88], we extract the actual type representation the compiler uses and preserve it through to runtime. That is, we reuse the complete type information stored in interface meta-data that accompanies compiled object code as our serialized type representation. This effectively pushes the type representation construction problem into the compiler, where we can ensure that all Haskell types may be represented on the value level.

However, having a valid representation of any Haskell type is not enough. We must also address the use of pattern matching to perform `typecase`. The general problem with previous approaches to dynamics based on pattern matching or equality testing on type representations is that it can only approximate true type *unification*, which is actually what we expect when matching types. In doing so, we unnecessarily restrict the expressivity of dynamic types, in favor of the easy implementation of type checking at runtime.

We propose that instead of a `fromDynamic` with an equality check on type representations, we replace the equality with actual type unification using the Haskell type checker *at runtime*. In doing so we obliterate any limitations on the forms of types that may be checked dynamically.

Section 2.4.3 previously introduced this approach to polymorphic dynamics, describing how a type constraint of the form `Plugin.resource :: API.Interface` is passed to the compiler at dynamic load time. To support dynamic type checking of this sort, the type of `load` is changed as follows:

$$
\begin{aligned}
\mathit{pdynload} \;::\; & \mathit{FilePath} \\
& \rightarrow [\mathit{FilePath}] \\
& \rightarrow [\mathit{PackageConf}] \\
& \rightarrow \mathit{Type} \\
& \rightarrow \mathit{Symbol} \\
& \rightarrow \mathit{IO}\ (\mathit{LoadResult}\ a)
\end{aligned}
$$

`pdynload` is a replacement for `dynload` that uses the full Haskell type checker to check the value returned by `load`. It has the same interface as `load`, along with an additional `Type` argument, which is a value-level representation of the expected type of the loaded component. This type can be constructed either via Template Haskell's support for type reification [141]:

```
[t| forall a . Int -> [a] |] :: Type
```

or by the application programmer directly writing the AST term representing the type they demand.

The type is then passed to the Haskell type checker, with a Haskell expression asserting that the plugin's value has this type. That is, we script the type checker at runtime with a small type program to test that the type exported by the plugin can be unified with the type expected by the application. If the type check succeeds, then the expected type unifies (under all of GHC's type system) with the plugin's interface type, and the plugin is safe to use. We then return the resulting value from the plugin. If the type check fails, `LoadFailure` is returned.

The implementation is described in Section 2.4.3, which requires a library function that wraps the underlying compiler, creating a temporary module importing both the application's API and the plugin, generating the required type equality. Type checking

then proceeds, and either the plugin unifies with its expected type in the application, or it fails with a runtime type error.

The key steps are to (1) use the compiler's compile-time type AST as our dynamic type representation on the value level, and (2) instead of pattern matching on type representations, reflect the types back as a type equivalence, and require the Haskell type system unify the types that cross the plugin boundary. To our knowledge, this is the first such use of the complete type system to replace pattern matching in dynamics, thus allowing arbitrarily complex types in dynamic components.

Our approach to polymorphic dynamics based on `pdynload` and runtime type unification requires that the interface files generated by GHC are retained with the loaded code, as they contain the serialized type representations needed for runtime type checking. As all Haskell code is shipped with its interface file, this is a light burden. In addition, interfaces conveniently contain a hash of the expected types, allowing us to check for unexpected changes in interfaces (perhaps due to malicious modification of interface files). This approach removes any need to employ fragile `Dynamic` values, which rely on required type representations, removing one source of unsafety. Instead, we must trust that the interface files provided with the plugin are not malicious or corrupt – returning us to the same trusted code base as for static linking.

Finally, our approach neatly side-steps any flexibility concerns with `Dynamic` types. Any Haskell type may be used in the plugin interface, including any future type system extensions. All we require is that the type must be checkable via GHC. We thus achieve a core goal of this thesis – there are no limitations on the strength of types that may be used to describe dynamic components, returning safety to the level of statically linked components.

The main constraint now on types that can be checked dynamically is that any value used at runtime must be checkable in a closed type environment, as required by the Haskell module system [41]. Types shared between application and plugin must be available at plugin compilation time via a declared, shared interface definition – there is no access to the application's internal type environment. We consider the restriction

load	*dynload*	*load + ghc*	*pdynload*
1	1.07	1.22	1.46

Figure 3.1: Cost of checked runtime loading, relative to an unchecked load

to module-level granularity, disallowing partial code fragments – a trivial cost for the majority of extension applications, though it does rule out some forms of inherited environments and type application in multi-stage meta-programming.

3.5 Comparing approaches

3.5.1 Performance of dynamic type checking

The question that remains is whether invoking the full Haskell type checker—in our case, actually invoking all of GHC—is sufficiently fast to be feasible. In fact, the overhead is less than might be imagined, as no code generation is required. Table 3.1 shows the comparative performance of the various load strategies. We use the runtime loading of a plugin without any type checking as a reference value (named *"load"* in the table). Runtime type checking using dynamic types from the standard library is only 7% slower (*"dynload"* in the table). In contrast, invoking GHC adds 22% to the base load time, with another 24% if we use it to type check the `APITypeConstraint` module (*"pdynload"* in the table). Overall, staged type inference of plugins using `pdynload` provides maximum flexibility, but is 46% more expensive than an unchecked load. These ratios are averaged over several different architectures, running different versions of the compiler, and different operating systems.

We believe the additional overhead of `pdynload` versus an unchecked `load` is not a significant issue, for two reasons. First, plugins are typically loaded only once, so this cost is amortized over the running time of the application. Second, there is plenty of room for optimization if performance is a problem. For example, Shields, Sheard and Peyton Jones [142] discuss several mechanisms for limiting the amount of type

inference that needs to be performed at runtime, some of which might be adapted to our framework. Additionally, we speculate that linking the type checker statically into the application—rather than invoking GHC as an external process—will greatly reduce type checking times. This could be based on GHC's recent support for linking compiler components as a library.

3.5.2 Type safety and source code plugins

Despite `pdynload` overcoming the limitations of runtime type checking by dynamic types, it remains vulnerable to attack. Since the type checker trusts the type information contained in the interface file, the user can provide a malicious interface file. Nothing short of proof-carrying code [114, 33] will give full type safety with dynamically loadable objects, although hashes of interfaces, now provided by GHC, help pragmatically. However, we do not need to trust interface files when we have access to the source code of plugins.

In this case, we can generate the object code with a matching interface file using the plugin library's compilation manager (`make`), in combination with the dynamic loader (`pdynload`). If this process is successful, the application can be sure that the plugin is internally well-typed and also has a matching interface. It still needs to trust the full compiler, but this is no different from when the plugin is statically linked into the application. Approaches to runtime compilation are described in Chapter 4.

We considered two fundamentally different mechanisms for type safe plugins: (1) limited runtime type checking using dynamic types and (2) full runtime type checking via the compiler, to overcome the limitations of the first approach. We demonstrated that the added flexibility of the latter implies a roughly 50% penalty with respect to the time needed to load a plugin. Independent of the mechanism used to check the plugin interface, we may compile the plugin source at load time to ensure that it is internally type safe. Table 3.2 summarizes the different levels of type safety achieved when only *object code* is available, or when *source code* is compiled at load time.

	load	*dynload*	*pdynload*
Only object code	No safety	No safety	Safe Interface
Source code available	Internal only	Full; type variable free interface	Full safety

Figure 3.2: Type safety of plugins using different approaches

3.6 Applications

We now describe the integration of dynamically loaded plugins, using the interfaces defined above, with a range of practical applications.

3.6.1 Lambdabot and Riot

Lambdabot is an online chat service, built around a suite of plugins. Riot is "a tool for keeping (textual) information organized" [2] written by Tuomo Valkonen. Both were modified as part of this work to load new functionality via plugins, using our plugin framework.

In the case of Lambdabot, the core of the application provides essential network services, as well as the ability to load additional features as plugins. Plugins are represented in the application as existentially typed values implementing the `Module` interface defined in Section 2.5.1. A more extensible version of Lambdabot, where all components of the system may be reloaded – preserving state – is described in Chapter 5.

At startup, Lambdabot connects to an IRC server, and starts serving user requests. A suite of plugin names are read from a configuration file, corresponding to a directory of compiled plugin objects. These objects are topologically sorted by dependencies,

[2] Riot, an Information Organization Tool, Tuomo Valkonen, 2008, http://hackage.haskell.org/package/riot

and dynamically loaded using the dynamic loading capabilities described earlier. Each plugin provides a distinct name, and a *phantom type* [132] is used to index the dictionary of services provided by the plugin.

Inside the application a table of references to loaded modules is maintained, and may be added to dynamically. During execution, a user may invoke the dynamic linker to load any new plugins, ensuring the application need not be disconnected from the network to upgrade its functionality.

The Riot text organization tool was modified in a similar manner, with the dynamic linker initialized at application loading time, and then a configuration plugin loaded, adding user-specified functions and data. The users' configuration is a Haskell plugin, and written as a Haskell source file. In this way plugin loading enables Haskell to be used as an extension language for the host application, by loading new configuration information at startup.

3.6.2 Haskell Server Pages

Haskell Server Pages (HSP) is a domain specific language, based on Haskell, for writing dynamic web pages [107]. Its main features are concrete XML expressions as first class values, pattern matching on XML, and a runtime system for evaluating dynamic web pages. Broberg [21] reimplemented the original HSP to run instead on the `hs-plugins` dynamic linking architecture introduced in this work. We include a brief discussion of this work as an interesting application of dynamic linking infrastructure in Haskell, building directly on our results.

HSP built on dynamic server pages replaces an interpreter model for serving up dynamic page content written in an EDSL in Haskell with on-demand compilation of dynamic pages as plugins, using our `hs-plugins` framework. Each page is then cached as a compiled object, in case it is needed later, by our framework.

In this way, dynamic server pages written in an EDSL in Haskell can be competitive with native code in performance, yet as flexible as interpreted language content, typically used for dynamic pages. Page source stubs are wrapped in boilerplate to al-

low dynamic type checking to succeed – guaranteeing that pages provide well-formed symbols – using the architecture described in Chapter 4.

3.6.3 Specializing simulators for computational chemistry

A novel application of plugins we describe is in the design of a Monte-Carlo simulator for polymer chemistry, based on program specialization. Material in this section was originally published in a series of papers [147, 84, 28]. Full details of the approach to Monte-Carlo simulator design and optimization are described there, and are not germane to this thesis.

In the specializing simulator approach [84], a simulator for a polymer chemistry reaction is built based on program specialization of the simulator to its arguments. The use of a simulator generator implies runtime code generation and compilation. In our case, the latter consists of invoking a standard C compiler, such as GNU's `gcc` or Intel's `icc` to compile the generated specialized simulator. Given the long running times of typical Monte-Carlo simulations, we can easily amortize time-consuming compilation with costly optimizations enabled. In addition to the simplification of data and control structures explicitly performed by the specializing simulator generator, the C compiler can exploit the fact that many of the variables of the generic simulator are now embedded as constants. This leads to additional constant folding, loop unrolling, and other optimizations.

After code generation, the simulator is dynamically loaded into the main application. The approach is attractive for interactive applications that, for example, animate the simulation graphically.

A foreign language interface for plugins

The amount of interaction between the main application and the generated simulator varies among domains, but loading the executable code of the specialized simulator into the main application usually simplifies matters. In particular, it encourages a tight integration between the running simulator and a graphical frontend displaying

the evolving simulation. While exploring a design space with many, very imprecise, but comparatively short-running simulations, users often want to see the continuous progress of the simulation.

In the polymerization kinetics application, we use a small, custom dynamic linker in Haskell to dynamically load and link the compiled C code back into the main Haskell program. It is based on the core of the dynamic linker described in this chapter, and provides a simple binding to the GHC Haskell runtime, implementing a bare-bones linking system for C objects, using the four functions:

$$
\begin{aligned}
&\mathit{pluginInit} && :: \mathit{IO}\ () \\
&\mathit{pluginLoad} && :: \mathit{CString} \rightarrow \mathit{IO\ Bool} \\
&\mathit{pluginResolve} && :: \mathit{IO\ Bool} \\
&\mathit{pluginSym} && :: \mathit{CString} \rightarrow \mathit{IO}\ (\mathit{Ptr}\ a)
\end{aligned}
$$

When passed an object handle through the compilation manager, a simulator object is loaded and resolved, and the linker returns a C function pointer to the C simulator, wrapped as a Haskell value. We achieve this by the following function, which uses the conversion function `withCString` from the Haskell foreign function interface (FFI).

```
loadAndGetSym :: String -> String -> IO (Ptr a)
loadAndGetSym objFile sym =
  withCString objFile $ \objFileC ->
  withCString sym     $ \symC -> do
    pluginInit
    pluginLoad objFileC
    pluginResolve
    pluginSym symC
```

To load a simulator, we might call this with

```
loadAndGetSym "simulator.o" "doSimulate"
```

Foreign evaluation

Once the C pointer associated with the given symbol is retrieved by the linker, control can then pass from Haskell to C by evaluating the function pointed to by the obtained

C pointer. However, we cannot directly coerce the C function pointer to a normal Haskell function value, as normal Haskell functions are represented differently. We can, however, throw the function pointer to the C runtime, which can then execute the function. We do this by calling a C wrapper `apply` function, with the function pointer as an argument. The C function `apply` is imported into Haskell via the Haskell FFI:

```
foreign import ccall unsafe "apply.h apply"
   apply :: Ptr () -> IO ()
```

where `apply` itself is a function to evaluate its argument:

```
void apply(void (*f)(void)) {
    f();
}
```

Now, we can evaluate a C function by passing it to `apply`. Using the FFI, C functions can also call into Haskell and thus realize a two-way connection between the simulator and the renderer.

3.7 Discussion

3.7.1 A standalone type checker

The dynamic type checking mechanisms provided by `pdynload` require that we call the compiler at runtime to type check a code fragment. This is currently achieved by writing the code fragment to a temporary file and invoking the compiler with appropriate flags to stop compilation after type checking, hence preventing code generation. We treat the compiler simply as a type checking function. If errors are produced they can be returned to the caller. Without code generation, type checking is relatively fast, but it could be further improved by linking the compiler directly into the application as a library. Some facilities to do this already exist in GHC.

One caveat of using `pdynload` to type check object-code plugins (discussed in Section 3.4), is that the plugin's interface file must accompany object files in the same directory. Hence a plugin consists of two files rather than just one, which makes them

a bit more unwieldy. In the future, we hope to add support for GHC to store the interface information in the compiled object file (or shared library), which alleviates the need for an extra file. Moreover, we plan to link parts of GHC, such as the type checker, into the host application to save the overhead of forking an external process (via the GHC API).

3.7.2 Loading packages and archives

We sometimes wish to load sets of objects as convenient archives. In Haskell, such archives are represented as *packages* of Haskell and C object files, treated as a single unit. The Cabal package management tool [79] constructs such archives from source bundles, using meta-data supplied by the user. Sets of core libraries are distributed together as the Haskell Platform [35].

To load package sets we provide the following function:

$$loadPackage :: String \rightarrow IO\ ()$$

This function, `loadPackage`, consults the package database maintained by the compiler of all packages registered on the system, extracting a list of dependent libraries and C objects. These are then sorted and loaded in dependency order. Unloading of packages as a unit is also supported. Support is also provided to load packages from local databases of registered libraries. Section 5.3.3 describes an application for whole-package loading, namely, efficient hot swapping of an entire application's code.

3.7.3 Loading C objects into Haskell

As described in Section 3.6.3, it is also possibly to load generic C object files, either static or shared, into a Haskell process using our library via the `loadShared` and `loadRawObject`. No additional safety checks are possible, due to the lack of type information provided by the C compiler.

3.7.4 Loading bytecode objects

We speculated on a design for a plugin system based on bytecode interpretation in earlier work [122]. The Glasgow Haskell Compiler provides support for mixing compiled and interpreted code in the same process [139], and the compiler itself is packaged as a library that may be called from a Haskell process [6] (originally introduced to provide access to syntax highlighting of full Haskell in a developer environment).

More recently, Daniel Gorin has developed the `hint` package[3] which wraps up GHC's interpreter in a convenient programmatic interface, making it possible to load Haskell source modules, compile them to bytecode objects, and evaluate the bytecode. This has since been used to provide bytecode interpreted online evaluation and sandboxing, and forms a complement to the work on native code object loading described in this work. Future work would be to unify the interfaces to both compiled and interpreted extension, as the underlying type safety mechanisms described in this work remain the same. Additionally, *hint* should provide a performance boost to the cost of runtime polymorphic type checking, as the type checker is already linked into the running application. An additional exciting possibility would be to serialize bytecode objects to disk, and thus provide a form of function serialization in Haskell for the first time, in the manner of Clean [129, 130].

3.7.5 Executable size

Currently, the use of our plugin infrastructure library produces rather large binary sizes, due to statically linking large parts of GHC and many Haskell libraries into the resulting application. Modifying the plugin infrastructure—and thus the GHC runtime system and linker—to use the operating system's native shared library mechanism (i.e. `.so`, `.dll` or `.dylib` files) would produce much smaller application binary sizes. This work has recently been completed, and GHC is now able to produce system shared objects for packages and archives. Extending our plugin loader to support programmatic manipulation of shared Haskell packages this way is future work.

[3] hint: runtime Haskell interpreter, Daniel Gorin, 2007, http://hackage.haskell.org/package/hint

This is especially important for embedding Haskell support in applications written in other languages, where the larger binary size may be a barrier to using Haskell as an extension language. Initial results with GHC[4] indicate that this produces executable sizes comparable to an application embedding support for languages such as Perl and Python, whose supporting libraries and runtime systems are usually stored in shared libraries.

3.8 Related work

The *dynamic loader* described in this chapter is strongly related to the Glasgow Haskell Compiler's dynamic loader, which is used extensively by GHCi—the compiler's interactive environment—to enable the mixing of compiled and interpreted code. We reuse GHC's dynamic loader in a more extensible fashion, enabling it to be used from Haskell code by binding it to Haskell via the FFI [30]. The first binding to GHC's dynamic loader was developed by André Pang, and directly led to the work described here [122]. We have also implemented dependency resolution similarly to GHCi, using the interface files generated by GHC during compilation, and we re-use its interface and package parser. GHCi does not, however, use the techniques we propose for ensuring the type safety of the objects that are loaded. GHCi emerged from the HEP [139] project, whose full goals have not yet been realized. In a sense, we are adding some of the missing elements of the HEP with our dynamic linker.

3.8.1 Type safe linking

Foundational work in type safe linking was conducted by Cardelli [24], introducing a calculus for linking and its interaction with separate compilation and module systems. Cardelli concentrates on module systems and formalizing a notion of linkage with type safety, with the aim to ensure link compatibility with module systems supporting separate compilation. Glew and Morrisett [55] build on this work combining module

[4] "Building plugins as Haskell shared libs", http://www.well-typed.com/blog/30, May 2009

constructs from Cardelli's link calculus with typed assembly language [111] to define type safety rules for static linking and loading. Key properties of the system linker and its impact on module systems and type safety are developed, to ensure linked objects are sound. They advocate a view of module systems as linker scripts.

Hicks, Weirich and Crary [67] provide a framework for formalizing and making type safe dynamic linking in the style of the Unix `dlopen` method: *DLpop*. They extend a typed assembly language with a `load` primitive, and develop a variety of dynamic linking strategies. In particular, type checking of dynamic components in DLpop implies type checking of typed object files, along with a form of dynamics.

They provide a term-level representation of types, so that interfaces can be checked in untrusted code. Notably, the `load` function, taking a pair of a type representation and an object file, corresponds directly to our `dynload` function, where the type representations are supplied by the `Typeable` dictionary. However, our approach, with the benefit of working directly on regular object files, cannot verify those object files for type safety, and must trust the interface files and compiler. A more extensive discussion of TAL approaches to dynamic linking is given by Alanko [4].

3.8.2 Dynamic typing

Abadi et al. introduced forms of dynamic typing for functional languages [2], introducing the `Dynamic` type, its introduction forms, and a `typecase` elimination, along with an extensive discussion on the history of dynamic types in a static setting. Initial extensions for polymorphic dynamic types are discussed, including matching of polymorphic dynamics up to renaming of bound variables. The interaction of dynamics with forms of polymorphism and abstract data types is developed by Abadi et al. [3], where the problem of binding type variables is raised. Either we require polymorphic types to be closed, or require type application at runtime (which is possible under our framework). Notably, they describe a typed `eval` function from abstract syntax to a dynamically typed value.

Amber [22], an ML-like language with an emphasis on persistence and serialization, provided good dynamic typing support, using an early form of the dynamic value encapsulated with its type. Persistence in Haskell also encouraged the development of forms of dynamic loading. In other work [40], type classes are used to define a `Persistent` class, for values which may be retrieved from persistent storage for checking.

Leroy and Mauny [100] describe an approach to combining dynamics with polymorphism, presenting two forms of polymorphic dynamics for ML. They employ "dynamic patterns" to pattern match on type representations paired with the actual value. Their pattern match allows firstly, for equivalence of polymorphic types up to renaming of bound type variables, and secondly, when the type is more general than the expected type. Our implementation, based on a full type representation of all types supported by the compiler, and runtime unification via the full type system shares some similarities with their approach, but comes with much cheaper implementation costs.

Approaches to dynamic typing in Haskell [31, 88, 10] have been discussed earlier, and are based on pairing an abstract Haskell value with a term level representation of its type. As described in this chapter, our approach is unique in that it removes all limitations on both term level type representations, and employs full unification, rather than pattern matching, to perform the dynamic check. Weirich [167] introduces a characterization of type safe cast via the heterogeneous symbol table problem, and explores solutions in ML and Haskell using forms of `typecase`.

Baars and Swierstra [10], in particular, seek to link the type representation with the actual type it represents, in order to avoid type coercions. They employ forms of universal and existential quantification to construct a data type that serves as a witness that two types are equal when supplied with a dynamic. Like other approaches to dynamics in Haskell, it doesn't support polymorphic dynamics.

An alternative approach to dynamics is via intensional type analysis [60], which permits approaches to runtime inspection of the structure of types, rather than on the

name. This was originally developed to allow forms of polymorphism, where reliance on a uniform representation is removed, however, the operations for type analysis at runtime can be used to implement forms of dynamic checking, as described by Alanko [4], section 9.2.

Separately, work has been undertaken to bring the benefits of static typing to dynamically typed languages. This approach is known as *soft typing* [51, 27, 169], and allows the compiler to accept what would otherwise by ill-typed programs, inserting runtime type checks to handle the error dynamically. Our emphasis on as much static checking as possible, but with limited runtime type checking for dynamic components, has a similar view point.

3.8.3 Clean

Many other functional programming languages more or less support dynamic typing and dynamic loading. In particular, *Clean* [131] has good support for a form of polymorphic dynamic types. Clean's dynamics have been used in a variety of scenarios, including type safe distributed communication [63] and an interactive Clean interpreter [159], similar in spirit to plugs and GHCi. In particular, it describes a safe file I/O system for storing arbitrary objects (including functions) to disk. This system requires a form of dynamic linking combined with polymorphic dynamic typing [129, 130]. While Clean's use of dynamics is similar to ours, we concentrate on the topics of extension languages and plugin infrastructure rather than the use of dynamics themselves. The primary difference with their approach is the use of a restricted subset of the type checker at runtime, while we overcome the obstacles this imposes by reusing the existing language type checker.

3.8.4 ML

Adding dynamic typing and dynamic loading to the ML family of languages has also received a significant amount of attention [45, 47, 54, 87, 100]. In particular, Leroy and Mauny describe an `eval_syntax` function from abstract syntax to dynamically typed

values, and briefly discuss how such a function may be used to embed CAML within a program. This work extends Abadi et al. with a form of polymorphic type pattern matching. Objective Caml provides a `dynlink` library [99], which supports dynamic loading and linking of bytecode objects. Type safety is ensured by checking that the object has been compiled against the same interface as the application.

3.8.5 Java and .NET

Both the Java Virtual Machine (JVM) and the .NET runtime support dynamic loading via *class loaders* [102, 1]. Since JVM bytecode is a typed language, class loaders are able to perform type safe dynamic loading; the JVM contains a verifier that ensures the type safety of dynamically loaded code, and a similar method is implemented for the .NET Common Language Runtime. Languages run on top of .NET, including functional languages such as F# [151], are able to use the runtime's features to support Dynamic Link Libraries (DLLs).

While statically typed, Java, in particular, supports forms of reflection well. Programmers are encouraged to inspect the class (i.e. type) of an object dynamically, and can obtain other information about objects at runtime. New class objects may be created dynamically, via a reflection interface, with little checking on the reflection allowed (leading to the possibility of runtime exceptions).

The core innovation of the JVM is that all code is loaded and checked at runtime. A class loader loads class objects as required, resolving names, and accessing bytecode for objects that are lazily required. Bytecode is verified dynamically for safety properties, and then is available for the user. Applications for the JVM's dynamic loading abilities include Java Server Pages, similar in spirit to the Haskell Server Pages design.

One primary difference between the Java and .NET approach to type safe dynamic loading and ours is in the power of the type system – Java's is extremely restricted – for example, parametric polymorphism is a relatively new innovation in Java – so the safety properties we seek to allow in types in dynamic code are simply not expressible in Java. This lack of expressivity greatly reduces the attractiveness of Java's dynamic

typing mechanism, although the ground breaking work in Java for allowing statically typed, but dynamically extensible, verified bytecode loading is impressive.

The second, and more fundamental difference, is that the JVM and .NET rely on bytecode objects that are JIT-compiled, while our approach relies on dynamic linking of native code objects, making the problem of verifying object code harder in our case.

Chapter 4

Runtime compilation

In the previous chapter, we looked at approaches to type safe dynamic linking and loading of object code. In this chapter, we investigate runtime compilation and code generation as part of our framework for dynamic extension. Runtime code generation adds significant expressive capabilities, including *meta-programming* functionality, program specialization techniques, and, when combined with type safe dynamic linking, sandboxing of untrusted native code.

In Section 2.6, we introduced our approach to runtime compilation and illustrated its utility with a native code command line for evaluating Haskell code. In this chapter, we explore the design of a runtime *compilation manager* for Haskell in detail as part of our `hs-plugins` library. We develop several applications, including source plugins, runtime meta-programming, type safe sandboxing, and optimizing embedded domain specific code.

4.1 A compilation manager

The compilation manager is the second major subsystem of the dynamic extension library infrastructure. It arranges for source code to be conditionally compiled to native code by determining whether it has changed since its previous compile. It also implements utilities for analyzing and manipulating the abstract syntax of the source.

We provide support for combining template source files with user-supplied source, making it easier for domain specific code fragments to be supplied by users with little overhead.

4.1.1 Invoking the compiler

The compilation manager exposes an interface to invoke the compiler on a plugin source, the suite of `make` functions:

$$
\begin{aligned}
&\mathit{make} && :: \mathit{FilePath} \\
&&& \to [\mathit{Arg}] \\
&&& \to \mathit{IO\ MakeStatus}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{makeAll} && :: \mathit{FilePath} \\
&&& \to [\mathit{Arg}] \\
&&& \to \mathit{IO\ MakeStatus}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{recompileAll} && :: \mathit{Module} \\
&&& \to [\mathit{Arg}] \\
&&& \to \mathit{IO\ MakeStatus}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{makeWith} && :: \mathit{FilePath} \\
&&& \to \mathit{FilePath} \\
&&& \to [\mathit{Arg}] \\
&&& \to \mathit{IO\ MakeStatus}
\end{aligned}
$$

`make` takes the path to a Haskell source file as its first argument. The second argument is a list of directories to search for additional modules to link against (such as the application's API), and `make` returns a value containing the path to the newly created object file, a flag indicating whether the object file had to be rebuilt, and a list of errors produced by the compiler if failure occurs.

The `makeAll` function recursively compiles any dependencies it can find using GHC's own dependency chasing framework, via the `- -make` flag. Dependencies will be recompiled only if they are visible to GHC – this may require passing appropriate "include-path" flags in the `Arg` parameter list. `makeAll` takes the top-level file as the first argument.

`recompileAll` is similar to `makeAll`, but operates only on previously loaded modules. It bypasses using GHC `- -make`, and instead traverses its own graph of the dependency chain. By maintaining its own dependency list, it is able to detect and load changes much faster than GHC's general purpose dependency chaser.

`makeWith` is a little different. It takes the path to a source file and the path to a template file (described in the following section), along with a list of directories to search in as its arguments. It returns the same values as `make`. The difference is that `makeWith` merges the abstract syntax of its two file arguments to create a third source file, which is then compiled using `make`. The merge algorithm we use for combining syntax is described in Section 4.1.2.

Status of compilation is indicated with the following types:

$$\begin{aligned} \mathit{data\ MakeStatus} &= \mathit{MakeSuccess\ MakeCode\ FilePath} \\ &\mid \mathit{MakeFailure\ Errors} \end{aligned}$$

$$\begin{aligned} \mathit{data\ MakeCode} &= \mathit{ReComp} \\ &\mid \mathit{NotReq} \end{aligned}$$

The `MakeStatus` type represents success or failure of compilation. Compilation can fail for the usual reasons: syntax errors, type errors and so forth. The `MakeFailure` constructor returns any error messages produced by the compiler. `MakeSuccess` returns a `MakeCode` value, and the path to the object file produced.

The `MakeCode` type is used when compilation is successful, to distinguish two cases:

- The source file needed recompiling, and this was done; or
- The source file was already up to date and recompilation was skipped.

We also provide a convenience function for testing whether a plugin needs to be recompiled:

$$hasChanged :: Module \rightarrow IO\ Bool$$

This function returns `True` if the module or any of its dependencies have older object files than source files.

All together, these functions form the core of a compilation manager for Haskell source modules. We now turn to facilities to support the manipulation of the plugin source.

4.1.2 Manipulating abstract syntax

One other feature that is governed by the compilation manager is the manipulation and addition of syntax to the plugin's source code to facilitate simplified plugins written in languages embedded in Haskell (EDSLs). The use case we wish to tackle is the addition of boilerplate EDSL imports, types and definitions to an end-users extension file so they may program directly in the extension language, avoiding the need to expose them to the surrounding Haskell infrastructure. We describe large scale applications of this model in Section 4.4.

Our approach is to allow the application author to write a Haskell source file *template* that contains any declarations the application requires of a plugin. The application can, for example, specify a module name and export list in the stub that is used in preference to the plugin's, ensuring that a compiled plugin will have the correct module name. Or, to ensure safety, the application can define a required top level definition and type that the dynamic loader will expect to find in every plugin module.

The plugin library uses a Haskell parser, such as `haskell-src`[1] or `haskell-src-exts`[2], to construct the abstract syntax trees of the two sources. The type information of the

[1] `haskell-src-exts`: Facilities for manipulating Haskell source code: an abstract syntax, lexer, parser and pretty-printer. http://hackage.haskell.org/package/haskell-src

[2] `haskell-src-exts`: Haskell-Source with Extensions (HSE, haskell-src-exts) is an extension of the standard haskell-src package, and handles most registered syntactic extensions to Haskell. http://hackage.haskell.org/package/haskell-src-exts

plugin is then erased from the AST, and type information, module declarations, and import/export lists from the template AST are merged into the source of the plugin. The resulting AST contains the union of the type information provided by the application author, and any declarations from the plugin author. This AST is written out as a Haskell source file, and is compiled via `make`; any compilation errors which occur are returned to the host application, which can display them to the user.

The syntax manipulation functions have the following interface:

$$
\begin{aligned}
&\mathit{merge} && :: \mathit{FilePath} \\
&&& \to \mathit{FilePath} \\
&&& \to \mathit{IO\ MergeStatus}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mergeTo} && :: \mathit{FilePath} \\
&&& \to \mathit{FilePath} \\
&&& \to \mathit{FilePath} \\
&&& \to \mathit{IO\ MergeStatus}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{mergeToDir} && :: \mathit{FilePath} \\
&&& \to \mathit{FilePath} \\
&&& \to \mathit{FilePath} \\
&&& \to \mathit{IO\ MergeStatus}
\end{aligned}
$$

The first function, `merge`, attempts to take the union of two source files and place the result in a temporary file. If the function has previously tried to merge these stub files during this session, then the temporary name for the result module is reused in order to help with recompilation checking.

The merge status type is defined as:

$$
\begin{aligned}
\mathit{data\ MergeStatus} = {} & \mathit{MergeSuccess\ MergeCode\ Args\ FilePath} \\
& \mid \mathit{MergeFailure\ Errors}
\end{aligned}
$$

type MergeCode = MakeCode

The function `merge` will include any import and export declarations written in the stub, as well as any module name, so that plugin authorś need not worry about this compulsory syntax. It is important to note that the module and types from the template are used to override any of those that appear in the first source (that the user writes). That is, if the user supplies the following code:

```
module A where
    a :: Integer
    a = 1
```

and the application author requires the following module and type:

```
module B where
    a :: Int
```

Calling `makeWith "A" "B" []` will merge the module name and types from module B into module A, generating a third file:

```
{-# LINE 1 "A.hs" #-}
module MxYz123 where
{-# LINE 3 "B.hs" #-}
a :: Int
{-# LINE 4 "A.hs" #-}
a = 1
```

Additionally, if a plugin requires some non-standard library which must be provided as a `-package` flag to GHC, they may specify this using a new pragma, `GLOBALOPTIONS`. Options specified in the source this way will be added to the command line used to compile the plugin. This is useful for users who wish to use GHC flags that cannot be specified using the conventional `OPTIONS` pragma.

The complete list of union operations performed by the merge operation are:

- use the source file location pragmas;
- use the template file module name;

- use the template file export list;
- take the set union of top level declarations;
- overwrite any type definitions the source supplies for template types;
- remove any reference to the template name from the source file; and
- take the set union of the import lists.

The other functions, `mergeTo` and `mergeToDir`, let the application programmer specify a particular output file or output directory to place compiled objects.

With these functions, `makeWith` is able to combine application-specified AST templates with user program fragments to produce complete plugins. In Section 4.4.4, we will revisit HSP, exploring Broberg's use of the merge functionality to support concise domain syntax.

4.2 An *eval* for Haskell

We have reached an interesting point in our framework development. We can link and load code, type check its use, and now compile new code on the fly. In addition, we can elaborate code fragments with additional definitions. There is nothing stopping us from taking Haskell code fragments from the user, wrapping them in a template module, compiling them dynamically, and then linking that code before executing it. And so we do just that, with the first library-supplied `eval` function for Haskell:

$$\begin{aligned} \mathit{eval} \;::\; & \mathit{Typeable}\ a \\ & \Rightarrow \mathit{Exp} \\ & \rightarrow \mathit{IO}\ (\mathit{Maybe}\ a) \end{aligned}$$

This `eval` function builds on our type safe `load` and compilation management via `make` to provide a type safe form of runtime evaluation for Haskell – that is, step one towards runtime meta-programming. The `Exp` argument is a fragment of Haskell code, with

may be a `String` or fragment of abstract syntax constructed via Template Haskell's support for Haskell building syntax. We will compile this fragment, and evaluate it dynamically.

The value returned by `eval` is constrained to be `Typeable` – meaning we can perform a limited runtime type check, using the `dynload` function described in Chapter 3. One consequence of this is that the code must evaluate to a monomorphic value (which will be wrapped in a `Dynamic`). In addition, we provide forms of `eval` with additional arguments for controlling the compiler environment (such as optimization flags, library paths, and specifying source pre-processors).

If the evaluated code type checks under the `Typeable` constraints, `Just v` is returned. A value of `Nothing` indicates type checking failed. Type checking may fail at two places: when compiling the program fragment, or when type checking the splice point. In this way `eval` resembles a meta-programming `run` operator for closed source fragments – closed in the sense that the code fragment must be closed over by the typing environment inside a Haskell module. To evaluate fragments of code that have polymorphic type, we can wrap the underlying value in a rank-*N* type, as described in Section 2.4.2.

To compile a user's code fragment safely, we wrap it in a closed module with the following code:

```
dynwrap :: String -> String -> [Import] -> String
dynwrap expr nm mods =
    withFreshName $ \x ->
        "module "++nm++ "( resource ) where\n" ++
         concatMap (\m-> "import "++m++"\n") mods ++
        "import Data.Dynamic\n" ++
        "resource = let { "++x++" = \n" ++
        "{-# LINE 1 \"<eval>\" #-}\n" ++ expr ++ ";} in toDyn "++x
```

This function generates a new Haskell source module, with a top level `resource` value, wrapped in a dynamic type, bound to the user's code fragment. The fragment is provided with a line pragma, so that error messages are local to the user's code, and

make no mention of the surrounding module. We will use a similar approach, with modified wrappers, for type safe sandboxing.

The generated module is then compiled and loaded back into the running application, before type checking it against its use site. Finally, the `resource` is accessed, returning a freshly compiled Haskell code fragment to the application.

Sometimes the user can provide evidence that they know the type is sound – via some side condition. In that case, the restriction on types `Dynamic` imposes is tedious, and so we provide an `unsafeEval` with no type additional checking, primarily for application authors:

$$\begin{aligned} &\mathit{unsafeEval} :: \mathit{Exp} \\ &\qquad \rightarrow \mathit{IO}\ (\mathit{Maybe}\ a) \end{aligned}$$

The obligation here is on the programmer to provide other evidence for safety (for example, the application might control what values are passed to `unsafeEval`, and can ensure that all generated values are well-typed).

This evaluation framework will form the basis for some applications described in later sections, including online Haskell evaluators.

4.2.1 Evaluating Haskell from other languages

We can also expose these Haskell evaluation functions to other languages via the foreign function interface, like so:

$$\begin{aligned} &\mathit{hs\_eval\_b} :: \mathit{CString} \rightarrow \mathit{IO}\ (\mathit{Ptr}\ \mathit{CInt}) \\ &\mathit{hs\_eval\_c} :: \mathit{CString} \rightarrow \mathit{IO}\ (\mathit{Ptr}\ \mathit{CChar}) \\ &\mathit{hs\_eval\_i} :: \mathit{CString} \rightarrow \mathit{IO}\ (\mathit{Ptr}\ \mathit{CInt}) \\ &\mathit{hs\_eval\_s} :: \mathit{CString} \rightarrow \mathit{IO}\ \mathit{CString} \end{aligned}$$

The `hs_eval_*` functions are foreign language interfaces to `eval`. It is available to foreign languages who include a `"RunHaskell.h"` header provided with the library. The foreign bindings use dynamic types to check that the compiled value has the type

expected by the caller, yielding a null pointer to C otherwise. That is, we perform the type checking on the Haskell side, propagating minimal chances for errors to the more weakly typed languages. In Section 2.6.3 we provide an example of evaluating a Haskell code fragment dynamically from a C program.

4.3 Runtime meta-programming

We have already used Template Haskell [141] to provide convenient construction of abstract syntax, both values and types. In this section we look at how a compile-time meta-programming model, such as Template Haskell, when combined with a runtime evaluation mechanism based on linking and loading, yields a novel implementation technique for runtime meta-programming, *for free*.

We take as our starting point the primitives introduced by Taha and Sheard [153] in their MetaML language, a statically typed multi-stage programming language based on ML. Their language is based on four staging operators:

- `delay`, which takes a code fragment and delays its evaluation by one *stage*;
- `escape`, which lets us splice fragments of syntax together;
- `lift`, which reifies code into abstract syntax; and
- `run`, which takes a code fragment and runs it.

Template Haskell, a compile-time meta-programming language, already provides forms of the first three primitives: `delay`, `lift` and `splice`, all for operating on abstract syntax as first class values at compile-time. Our contribution is an implementation of `run` which works at runtime, rather than compile-time. Template Haskell, a compile-time model, gives us a level of cross-stage persistence which we reuse in this work. For a full treatment of TH's meta-programming model see Sheard and Peyton Jones [141].

To instantiate the programming model, we provide a `Eval.Meta` module, which exports a `run` function, with the following type:

$$run \;::\; (Printable\, t,\; Typeable\, u) \Rightarrow Q\, t \;\rightarrow\; u$$

That is, it takes any quotable, printable syntax fragment, and generates a typeable Haskell value from it – at *runtime*. The behavior is as follows, where we combine Template Haskell's lifting and delay mechanisms, with our implementation of `run`:

```
-- Lift a code fragment, then run it.
*Eval.Meta> run [| 1 + 2 ::Int |] + 3 :: Int
6

-- Bind a fragment, lift it, then run it
*Eval.Meta> let a = 1 + 2 in run [| a::Int |] :: Int
3

-- Bind the result of running a fragment
*Eval.Meta> let a' = run a :: Int
*Eval.Meta> a'
3
*Eval.Meta> a' + 2
5

-- Lift an abstraction, then apply it:
*Eval.Meta> (run [| \x → x::Int |] a') :: Int
3
```

The implementation is surprisingly easy, but has limitations. In particular, as our `run` values, when reduced to base values, are compiled objects, we cannot get access to the environment maintained in previous stages, other than top level names. Closures don't work, for example, as our persistent values must be closed Haskell modules, which do not expose internal environments from previous stages. For example,

```
-- A lifted computation that refers to an 'a' bound in a previous stage
*Eval.Meta> let a' = let a = 1 + 2 in run [| a::Int |]::Int

-- Which can be typed adequately
*Eval.Meta> :t a'
a' :: Int
```

```
-- But not then evaluated.
*Eval.Meta> run [| a' |] :: Int

<eval>:1:0: Not in scope: `a'
*** Exception: source failed to compile
```

We thus consider this a novel meta-programming model based on compiled native code fragments, that has extremely cheap implementation costs. Our slogan: compile-time meta-programming, plus dynamic linking and dynamic type checking equals a runtime meta-programming model.

Our implementation of `run` is as follows:

```
run :: (Printable t, Typeable u) => Q t -> u
run e = unsafePerformIO $ do
            s <- runQ e
            v <- eval s imports
            case v of
                Nothing -> error "Run fragment failed to compile"
                Just v' -> return v'
```

Our `run` function accepts a delayed code fragment (constructed by Template Haskell). The `runQ` method then fully reduces the delayed fragment to a chunk of abstract syntax, which in our case, must be renderable. The rendered syntax fragment provided by Template Haskell is then evaluated dynamically using our `eval` function provided earlier. Our use of `unsafePerformIO` to conceal the effect of running the compiler we feel is morally sound, as compilation is a referentially transparent action.

4.3.1 The heterogeneous symbol table problem

Using the primitives provided by Template Haskell, and the `run` function above, we can encode a new solution to a classic dynamic typing problem: heterogeneous tables [167] – a map of strings to values of any type – which also illustrates the deep connection between dynamic typing and multi-stage programming.

First, some preliminaries:

```
{-# LANGUAGE TemplateHaskell #-}
```

```
module M where

import System.Eval.Haskell
import Language.Haskell.TH
import System.IO.Unsafe
```

We enable Template Haskell, and import our `eval` function. In addition, as compilation is a pure function, we can dispense with IO errors in compilation by casting them to `Nothing` – so we import `unsafePerformIO` to achieve this.

We then introduce a (remarkable) `delay` construct:

```
delay :: Printable a => Q a -> b
delay e = fromMaybe (error "Type error") $
          unsafePerformIO $ do
                s <- runQ e
                unsafeEval s []
```

This delay operator takes a code fragment expression (produced via TH operators) and persist it from compile-time to runtime, and then reduces it via `runQ` to a closed fragment of abstract syntax. We then *suspend evaluation* of the fragment by our compiled `eval` function by wrapping it in a lazy thunk – produced by `unsafePerformIO`. This delay means that compilation will occur only when the value is forced! Only when we require the value will it actually be compiled. Compilation of this Haskell program will be radically and unpredictably interleaved with its execution, and laziness will also mean that fragments never used are never compiled.

We will use this construct to build a symbol table of values, that when demanded, will be compiled and returned as a regular Haskell values. An asynchronous exception is thrown at runtime if there is a type error. That is, we reuse the static type system to provide a delayed dynamic type check, by suspending type checking of the fragment until runtime.

We define, in the style of Weirich, a heterogeneous symbol table, where the elements all have different polymorphic type – not statically typeable in Haskell98.

```
table :: [(String, a)]
table =  [("a", delay [|'a'|]           )
         ,("b", delay [| 42 |]          )
         ,("c", delay [| \x -> (x,x) |] )
         ]
```

And introduce a lookup function for the symbol table:

```
lookupH :: String -> [(String, a)] -> a
lookupH s t = case lookup s t of
                Nothing -> error "Symbol not found"
                Just e  -> e
```

A session with this new form of data type for Haskell follows (notice how a type must be given to resolve the polymorphism of the symbol table) is as follows:

```
*M> lookupH "a" table  :: Char
'a'

*M> lookupH "b" table  :: Integer
42

-- Polymorphic functions still work...
*M> let fn :: forall a. a → (a,a) = lookupH "c" table
*M> fn 1
(1,1)

*M> fn 'c'
('c','c')

*M> fn 1
(1,1)

-- We can add elements dynamically:
*M> let table' = ("x", delay [| \x → x |]) : table

*M> let fn :: forall x. x → x = lookupH "x" table'

*M> :t fn
fn :: forall x. x → x

-- And the new table element is only compiled now:
```

```
*M> fn 1
1
*M> fn ()
()

-- Looking up the element again doesn't recompile the table element:
*M> let fn' :: forall x. x → x = lookupH "x" table'
*M> fn' "foo"
"foo"
```

We have shown a surprising result: compile-time meta-programming, along with a straight-forward implementation of runtime compilation, when combined with dynamic linking, allows for a form of multi-stage programming, sufficiently rich to encode classic dynamic typing problems, such as the heterogeneous symbol table.

4.3.2 Type safe printf

Another classic problem in typed meta-programming is how to construct a type safe printf. Shields, Sheard and Peyton Jones [142] introduce an approach to an interpreter for `printf` format strings, where the types of the arguments are checked dynamically using a typecase.

We present an implementation of type safe printf for Haskell, based on runtime compilation of a generated pretty printing function constructed by the printf interpreter. This is a full-scale implementation, with the novel advantage that the printf function is compiled, optimized native code, with the type:

$$printf :: String \rightarrow Dynamic$$

We lex and parse the string format argument, building a Haskell code fragment on the fly that represents a Haskell implementation of the printing function specified by the printf argument. This is then compiled with our `eval` mechanism. Type checking is delayed until the plugin is type checked. The result is a just-in-time compiled Haskell implementation of `printf`. Our implementation is a compiled version of the `run`-based implementation of printf. The full implementation is provided in Appendix A.

4.4 Applications

We will now explore some end-user facing applications of our framework of linking, type checking and compilation. The first widely used applications are evaluators for Haskell, including an online evaluator for untrusted code, using type safe sandboxing.

4.4.1 A Haskell interactive environment

In Section 2.6.2 we outlined the `plugs` application for native code read-eval-print loops in Haskell. Here we present the entire source of such a Haskell "interpreter":

```
import Eval.Haskell
import Plugins.Load

import System.Exit              ( ExitCode(..), exitWith )
import System.IO
import System.Console.Readline  ( readline, addHistory )

symbol = "resource"

main = do
        putStr "Loading package base" >> hFlush stdout
        loadPackage "base"
        putStr " ... linking ... " >> hFlush stdout
        resolveObjs
        putStrLn "done"

        shell []

shell :: [String] -> IO ()
shell imps = do
        s <- readline "plugs> "
        cmd <- case s of
                Nothing          -> exitWith ExitSuccess
                Just (':':'q':_) -> exitWith ExitSuccess
                Just s           -> addHistory s >> return s
        imps' <- run cmd imps
        shell imps'

run :: String -> [String] -> IO [String]
```

```
run ""   is = return is
run ":?" is = putStrLn help >> return is

run ":l" _             = return []
run (':':'l':' ':m) is = return (m:is)

run (':':'t':' ':s) is = do
        ty <- typeOf s is
        when (not $ null ty) (putStrLn $ s ++ " :: " ++ ty)
        return is

run (':':_) is = putStrLn help >> return is

run s is = do
        s <- unsafeEval ("show $ "++s) is
        when (isJust s) (putStrLn (fromJust s))
        return is

help = "\
\Commands :\n\
\  <expr>               evaluate expression\n\
\  :t <expr>            show type of expression (monomorphic only)\n\
\  :l module            bring module in to scope\n\
\  :l                   clear module list\n\
\  :quit                quit\n\
\  :?                   display this list of commands"
```

This example shows how powerful the combination of dynamic loading, runtime type checking and runtime compilation, can be, especially when in the form of a library. In only a few lines a complete language frontend can be crafted.

We make use of a new function, `typeOf`, provided by the `hs-plugins` library, to implement type inspection. This function compiles the user-supplied Haskell fragment, and shows the resulting `Dynamic`, printing its type.

```
typeOf :: String -> [Import] -> IO String
typeOf src mods = do
    pwd                <- getCurrentDirectory
    (cmdline,loadpath) <- getPaths
    tmpf               <- mktemp dynwrap src mods
    (obj,_,err)        <- make tmpf cmdline
    ty <- if null err
```

```
            then do (_,v::Dynamic) <- load obj [pwd] loadpath symbol
                    cleanup tmpf obj
                    return $ (init . tail) $ show v -- as a string
            else do mapM_ putStrLn err
                    removeFile tmpf
                    return []
    return ty
```

4.4.2 Source plugins for compiled programs

In Section 3.5.2 we considered the trust problem with dynamic type checking of regular object code. Without a form of proof-carrying code [114] or typed assembly language [111] we are forced to trust that object code was derived from well-typed source code. An alternative is to move the compilation pipeline into our trusted path, and only accept source plugins, so that we may inspect and type check the untrusted code ourselves, using the Haskell type checker. We support source plugins via the compilation framework described in this chapter.

4.4.3 Type-based sandboxing of untrusted code

In Section 2.6.2 we introduced a Haskell command line interface based on dynamic loading and our compilation manager. We now illustrate how to embed such an interpreter into a Haskell application. We modified Lambdabot to provide evaluation of Haskell code fragments as a service to users. The danger in doing so is that users may type any Haskell code fragment, and, as long as it type checks and compiles, the engine will attempt to execute it. This is clearly unsafe, as any IO action will be well-typed. So our policy is to ensure only pure Haskell code can be executed

Pure code is the subset of Haskell that provides referentially transparent total and partial functions, non-termination and imprecise exceptions [126]. It is safe as the type system guarantees it can have no effect on the world, other than returning a value. We may also choose to allow memory effects [94], but no IO. Template Haskell may be

allowed, but without support for running constructed code fragments, only building them (to prevent malicious IO actions at compile-time).

However, we can use the fact we have control over the evaluated code's environment to provide a form of type-based *sandboxing* of untrusted code. The core architecture is, first, to place the untrusted code into a separate address space, by forking the evaluator once per request. This allows us to use the operating system to terminate long-running computations – a property which cannot be determined by the type system.

Secondly, we then specify a safe set of imports that may be used by user code – that is, we define a trusted environment for Haskell, where no code that can defeat the type system – such as `unsafeCoerce` or `unsafePerformIO` – is allowed[3]. We then wrap the user supplied code in an exception handler, so it can't throw exceptions that contain values that could do harm, and `show` the value the user provides. Finally, we truncate the output produced by any dynamic expression so that it can't overwhelm the network with a deluge of data.

The key question then is: how do we render harmless any well-typed IO actions the user specifies? Our safety solution relies on intercepting the user's code via the `Show` typeclass, giving us a chance to inspect the code for its type, prior to executing it. For IO code, we dispatch to a custom instance of `Show` for IO that turns the nasty IO action into a harmless value:

```
instance Typeable a => Show (IO a) where
    show e = '<' : (show . typeOf) e ++ ">"
```

And that's it. Any action of IO type specified by a user to our service will be printed as "`<IO a>`", rendering it harmless.

In summary, with access to source, we can use the type checker as a runtime verification tool to rule out dangerous code prior to execution, and rely on the specification of a small, safe trusted base of Haskell libraries to make the evaluation engine worthwhile.

[3]Attempts to standardize this safe subset of the core libraries are underway at the time of writing, http://hackage.haskell.org/trac/ghc/ticket/1380

4.4.4 Dynamic server pages, revisited

In Section 3.6.2, we described Broberg's work building a form of *dynamic server pages* on top of our framework. Here, we review his use of the syntax merging and manipulation support provided by our framework.

Broberg [21] builds a evaluation engine for dynamic content on the web, where each web page corresponds to a Haskell plugin, that is compiled and evaluated when that page is accessed. Previously generated pages are cached by our `hs-plugins` framework, for later use.

Users write fragments of Haskell code, and the `mergeWith` and `makeAll` functions are used to generate the actual code used to produce the dynamic content. The initial page template imports the domain specific language definition, and defines a top level symbol the engine expects the user's code to be bound to. `mergeWith` then generates the actual plugin code into a cache directory, and compiles it with `makeAll`, chasing any dependencies. The resulting object is loaded into the server, serving up a new dynamic page.

4.4.5 Optimizing embedded DSLs

The use of our approach to plugin compilation and dynamic loading has seen significant adoption in projects using embedded domain specific languages in Haskell. We describe some of the recent work using our plugin system as the basis, to illustrate the maturity of the approach.

Conal Elliott [49] uses our plugin loading system as the backend for a domain specific language for graphics programming. Users manipulate values in a graphical interface, generating Haskell code fragments, that are compiled via `hs-plugins` to produce optimized native code implementations of the graphical algorithms, in order to improve the performance of online-generated code.

Similarly, Seefried et al. [138] use `hs-plugins` to import code fragments written by the user of a graphics system, `PanTHeon`, transforming the code via domain specific

optimizations, before compiling and loading the user's code fragment as a plugin. In this case the plugin extends the user-facing application with new functionality.

Seefried also describes work on *compiler plugins* for GHC in his PhD thesis [137], where new code transformations and optimizations not feasible in Template Haskell may be imported dynamically into GHC, extending its range of optimizations with domain specific approaches. Seefried's work co-evolved with the work described in this thesis, and reuses much of the low-level infrastructure developed here.

A third example is that of GPU kernels embedded in Haskell. Lee et al. [96] introduce a language for GPU programming directly in Haskell. Their approach generated fragments of CUDA code, compiled with the CUDA compiler, producing CUDA object files that are loaded back into the Haskell application, in a similar manner to that which we described in Section 3.6.3. In later work they bind directly to the CUDA runtime, loading objects on the CUDA side.

Other projects that have adopted our library and approach for plugin loading of Haskell DSLs include CAMILA [108], a formal methods toolset, and Pivotal [58], an IDE for Haskell where documents may be written as Haskell programs.

4.5 Related work

The closest related work to the compilation manager we describe in this project is Malcolm Wallace's `hmake` system [163], a compilation-based read-eval-print loop and dependency chasing build tool for Haskell programs. `hmake`, notably, can perform its own dependency analysis on source, while we consult GHC's package system. Additionally, `hmake` is compiler neutral. GHC's dependency management system, the GHC compilation manager[4], is used as part of this work.

The direct inspiration of the approach to meta-programming based on Template Haskell was Taha and Sheard's MetaML language [153], and Weirich's work on the heterogeneous symbol table problem [167].

[4]The GHC Compilation manager, http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler#OverallStructure

4.5.1 Template Haskell and staged type inference

Shields, Sheard and Peyton Jones [142] introduce an approach to staged computation and dynamic typing, via a small call-by-value polymorphic lambda calculus extended to support staged execution and staged type inference. They introduce three operators for splicing, deferring and running code at various stages, that lead to MetaML and Template Haskell. Their `defer`, `run` and `typecheck` operations have analogues in our implementation. Notably, the staged type inference approach demands the use of the type checker at multiple program stages, along with a type scheme for polymorphic types that can be preserved over stages. We have implemented many of the ideas they propose, and for optimized native code, while they experimented with an interpreter for a small core calculus. Interestingly, they suggest that `run` may make it possible to write a form of dynamic linking – their `getCode` operator is similar to `dynload` in our framework – while we developed dynamic linking, leading to a form of the `run` operator. One of their motivating examples is an `printf` interpreter that uses dynamic types to represent the argument list. The types are checked at runtime using typecase. We implement a similar version, using native code compilation, in Appendix A.

Template Haskell itself [141] – a compile-time meta-programming extension to Haskell – is directly reused in our compilation framework to provide code construction and manipulation support, and the original motivation for our `printf` design. We make the novel move of reusing the compile-time meta-programming facilities at runtime – essentially providing a new dynamic backend to Template Haskell. We also provide a form of multi-stage persistence, though our use prevents references to bound variables in previous stages, due to the closed-module restriction.

Template Haskell's reification facilities allow us to build syntax trees representing code and types in our program, providing many of the features of MetaML described below. Lynagh describes an implementation of compile-time `printf` via Template Haskell [103], Section 4.

4.5.2 Multi-stage programming

There is a significant body of work on multi-stage programming, which we touch on here, including extensive work formalizing MetaML [152, 123, 110].

Wickline, Lee, Pfenning and Davies [168] develop a type system based on modal logic as a basis for specifying and analyzing staged computation, in the context of a typed, modal lambda calculus. They give an embedding of a two-level staged language in MiniML. In particular, they define a "modal restriction" on variables that ensures they cannot refer to worlds in which they are unavailable.

Other work in ML has focused on the optimization opportunities presented by runtime code generation [95], with an emphasis in online specialization of programs through dynamic compilation. Foundational work on two-level staged programming was by Nielson and Nielson [115]. Staged computation has its origins in meta-circular interpretation. Läufer and Oderksy [93] construct a meta-circular interpreter for a statically typed language, without dynamic types.

Finally, approaches to type safe printf in statically typed functional languages have yielded several solutions, including Danvy's [39], where the printf format string is used as the concrete syntax for an interpreter, suggesting the use of abstract syntax for a statically well-typed printf. Hinze responds [69] by using singleton types and multi-parameter type classes to produce a `Format` type class in Haskell, leading, almost, to a pretty printing domain specific language. They push on the boundaries of Haskell types, describing how to simulate restricted forms of dependent types in Haskell.

Chapter 5

Hot swapping

Some Lisp programs such as Emacs, but also the Linux kernel (when fully modularized) are *mostly dynamic;* i.e., apart from a small static core, the significant functionality is dynamically loaded. In this chapter, we explore *fully dynamic* applications in Haskell where the static core is minimal and all code is *hot swappable*. Material in this chapter appeared in Stewart and Chakravarty [148].

Hot swapping raises interesting issues for typed languages, including persistence and serialization questions, and the specter of the "expression problem" when the shape of data types change during an application's lifetime. We describe our approach to these issues, and demonstrate the feasibility of our architecture through a number of novel applications: Yi, an extensible editor; Lambdabot, a plugin-based chat bot; and XMonad, a popular extensible window manager. Benefits of the approach include hot swappable code and sophisticated application configuration and extension via embedded DSLs.

5.1 Overview

In Section 2.7 we introduced our approach to hot swapping, describing partially extensible applications, such as Emacs or the Linux kernel, where significant application functionality cannot be extended dynamically. We then move beyond such applica-

tions by introducing a fully dynamic software architecture, where the core contains only what is needed to load the rest of the application at runtime.

In the applications we modified to support hot swapping: Lambdabot and Yi, we found two main advantages over applications that are only mostly dynamic. Firstly, with a smaller static core, we gain more flexibility and extensibility from dynamic code loading and code swapping, as fewer features depend on the static core. Cases in point are the user interfaces in Emacs and Yi. As already mentioned, the user interfaces supported by Emacs depend on the primitive operations realized in Emacs' static core. In contrast the minimal static core of Yi is independent of the choice of user interface, which enables completely dynamic interface configuration.

Secondly, fully dynamic applications enable a straight-forward implementation of hot code swapping, where *currently executing* application code can be replaced by a variant without losing the application state. Dynamic code replacement is always a tricky business; in a purely functional language the semantic consequences are even more subtle. We achieve it by redoing the dynamic boot process while preserving a handle to the application state. This process is fast in practice and avoids a large range of semantic issues that other approaches to dynamic software updates incur. Dynamic software updates, and also dynamic extension and configuration, are generally convenient, but they are of special value in high availability applications like servers that are in continual use.

In summary, this chapter contributes:

- a fully dynamic software architecture;
- hot code swapping that preserves application state;
- practical experience with the new software architecture;
- a design for state serialization and persistence.

The source code for Yi[1], Lambdabot[2] and XMonad[3] is publicly available.

[1] http://www.cse.unsw.edu.au/~dons/yi.html

[2] http://www.cse.unsw.edu.au/~dons/lambdabot.html

[3] http://xmonad.org

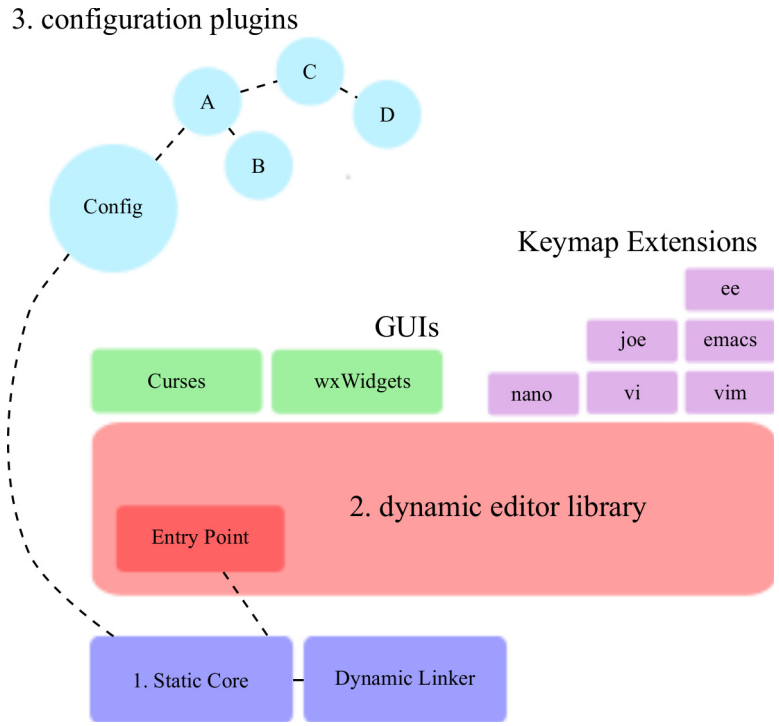

Figure 5.1: Structure of Yi

5.2 A dynamic architecture

The foundation for extensibility in Yi and Lambdabot is a dynamic architecture based on runtime code loading. Figure 5.1 illustrates this architecture, as implemented in the Yi editor. It is partitioned into three main components: (1) a static bootstrap core; (2) an editor library; and (3) a suite of plugins. Under this framework, the binary itself consists solely of a small static core. The purpose of the static core is to provide an interface to the dynamic linker and to assist in hot swapping. The static core has *no* application-specific functionality. Lambdabot is structured similarly, with the difference that the dynamic application is a tree of dependent modules, rather than a single archive of objects like the Yi main library.

The static core, with the help of the dynamic linker, loads any configuration data (as plugins) followed by the main application code. Control is then transferred to the main application's entry point—its `main` function—and the program proper starts up.

In Yi files are opened, buffers allocated and the user may begin editing. Lambdabot connects to an IRC server node and begins serving user requests. Note that the Yi configuration plugins and the main application are dynamically linked against each other so that plugins may call application code (and vice versa). For clarity we omit these details from Figure 5.1

During execution the user may invoke the Haskell dynamic linker library, described in Chapters 3 and 4, to recompile and reload any configuration files, or to reload the application itself. The state of the application is preserved during reloading (hot swapping), with the result that new code can be integrated or existing code be replaced, without having to quit the application.

In the rest of this section, we discuss the process of booting in more detail; in the following section, we discuss hot swapping.

5.2.1 Dynamic bootstrapping

The single purpose of the minimal static core is to initiate the dynamic bootstrapping process, by arranging the dynamic linking of the application proper. Hence, we seek a structure that it is generic in its functionality (so that the structure is reusable), efficient and type safe. The following simplified code illustrates such a structure:

```
import System.Plugins

main = do
   status <- load "Yi.o" [] [] "main"
   case status of
     LoadFailure e       -> return ()
     LoadSuccess m main' -> main'
```

Here `Yi.o` is the main module of the editor library, which we `load` using the plugin infrastructure described previously. `Yi.o` is the root of a tree of modules determined by module dependencies. The plugin infrastructure takes care of recursively loading all required modules and packages, such that the entire program is loaded and linked. The final argument to `load`—i.e., `"main"`—is the symbol that represents the application's entry point. If loading is successful, the plugin infrastructure presents us with

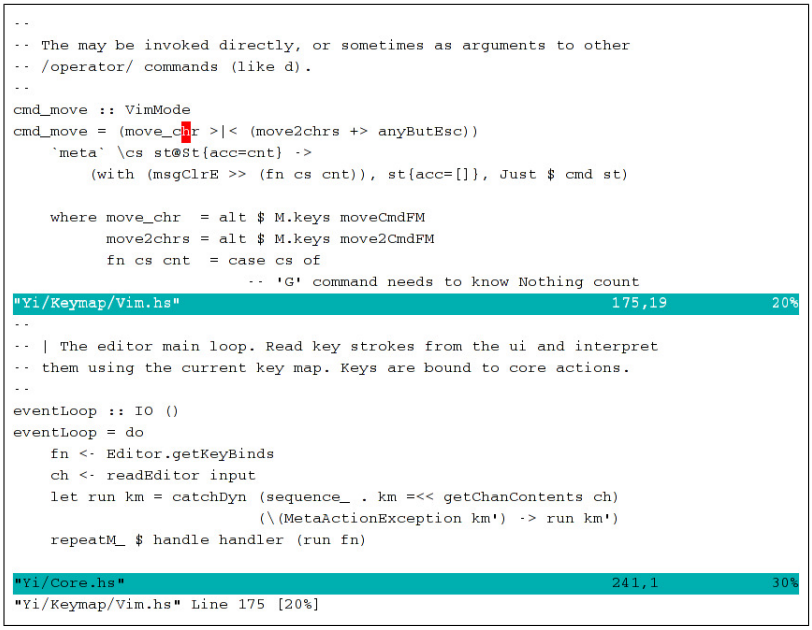

Figure 5.2: Screenshot of Yi's ncurses interface

the Haskell value represented by the given symbol. Hence, we transfer control to the dynamic portion of the application by evaluating that value.

On the dynamic side, the entry point is a conventional main function:

```
main = do
    setupLocale
    args    <- getArgs
    mfiles  <- do_args args
    ...
```

In this case, the Yi editor entry point is called, and execution continues as if the dynamic `main` had been invoked directly at startup. The application has now been booted, and only dynamically loadable code is executing.

A screenshot of Yi running under its ncurses interface[4] is in Figure 5.2 and the GTK interface in Figure 5.3 (courtesy Jean-Philippe Bernardy). The full sources for the application booter, configuration and entry points are listed in Appendix B.

[4] An interface based on wxHaskell [97] is also available

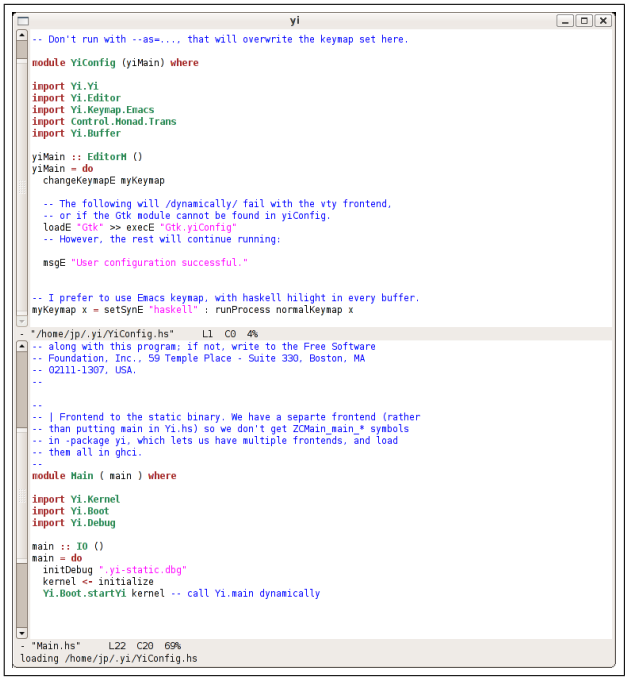

Figure 5.3: Screenshot of Yi's GTK interface

5.3 Hot swapping

Once the application is up and running there are two operations that we would like to be able to perform:

1. reloading of plugins; and

2. hot swapping of application code.

The first operation means that changes to configuration files (implemented as plug-ins) can affect the running application without having to restart the program. The second is more profound: we want to replace application code while the editor is running, without losing state—allowing us to "fix-and-continue" or to incorporate new functionality on the fly. We will first consider how to reload configuration files in a dynamic architecture, which then leads us to the second, more difficult problem.

5.3.1 Dynamic reconfiguration

We follow the Lisp tradition of expressing configuration files as source code in the extension language, in our case Haskell. The reasons for using the extension language rather than special purpose configuration languages are very much the same as those for the use of embedded domain specific languages (EDSLs) over standalone DSLs. However, we also inherit the drawbacks of the EDSL approach, such as error messages that are harder to comprehend for end users – hence we use techniques such as line pragma insertion, to localize error messages (see Section 4.2). On the positive side, we shall see in Section 6.2 how configurations can naturally and conveniently grow into EDSLs in our approach.

In any case, users specify their own configuration settings by writing Haskell code in configuration files. These files are compiled and dynamically loaded. The values they provide are used to update a default configuration record type; in other words, we follow the scheme for application configuration via plugins described in our earlier work, and we will see this approach used again in XMonad.

Configuration files are dynamically loaded when the static core of Yi begins executing. The user-defined configuration values are retrieved and passed to the dynamic editor core, which uses these values to set initial states for the various components of the editor. It is important to remember that it is the static core that performs all dynamic linker operations, including plugin loading, in our architecture. All invocations of the dynamic linker by dynamic code go via the static core; i.e., the dynamic code calls into the static core, which in turn forwards the request to the dynamic linker. As we discuss soon, this structure is crucial for the reloading process.

To be able to pass configuration values through to the dynamic code, we need to extend the simple dynamic main function illustrated in Section 5.2.1 slightly. Rather than jumping to a constant main function, we instead pass the configuration values as arguments to a modified entry point function:

$$main' :: Config \rightarrow IO\ ()$$

To initiate reloading of configurations—or any other plugins—from the dynamic application, we need to interact with the dynamic linker. As already explained, the linker is not visible from the dynamic code: it is linked to the static core. This structure is crucial to enable the hot swapping of the entire dynamic application code, which we will discuss shortly. Hence, the static core needs to provide the dynamic application the required functions of the dynamic linker as function arguments to the dynamic entry point.

The dynamic entry point has the following type:

$$
\begin{aligned}
&\mathit{type}\ \mathit{DynamicT} = (\mathit{Maybe}\ \mathit{State},\\
&\qquad\qquad\qquad\quad \mathit{Maybe}\ \mathit{Config},\\
&\qquad\qquad\qquad\quad \mathit{Maybe}\ \mathit{State} \rightarrow \mathit{IO}\ (),\\
&\qquad\qquad\qquad\quad \mathit{IO}\ (\mathit{Maybe}\ \mathit{Config}))\\
&\\
&\mathit{dynmain} \qquad :: \mathit{DynamicT} \rightarrow \mathit{IO}\ ()
\end{aligned}
$$

The first component is an optional editor state value for when we wish to preserve state over hot code swapping (Section 5.3.2). The second field is a configuration record retrieved from the configuration plugins. The final two components constitute dynamic linker functions, which we call `reboot` and `reconf`, respectively.

The function `reconf` is a simple wrapper around the dynamic linker's reloading primitive, `reload`, which checks for changes to the configuration files. If there are changes, `reload` triggers recompilation and reloading of the configuration modules. The reconfiguration function has the following type:

$$\mathit{reconf} :: \mathit{IO}\ (\mathit{Maybe}\ a)$$

The type is polymorphic so that we can enforce statically that `reconf` does not depend on the representation of the values extracted from configuration files—that is the concern of the consumers of configuration values.

When `reconf` is called from the dynamic code a new config value is retrieved by the static core, which passes this value back to the caller in the dynamic application. The

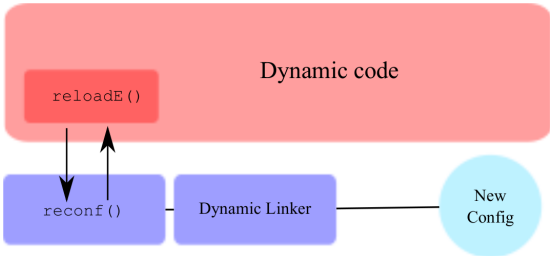

Figure 5.4: Reloading configuration files from dynamic code

dynamic code then uses the new value to update the application's state. For example, Yi calls `reconf` by defining an editor action in the dynamic code:

```
reloadE :: Action
reloadE = do
  modifyEditor_ $ \e -> do
    conf <- reconf
    return $ case conf of
      Nothing -> e
      Just (Config km sty) ->
        e { curkeymap = km, uistyle = sty }
  UI.initcolours
```

The function `reloadE` atomically (with respect to the dynamic application's global state) invokes the static linker's `reconf` function. Then, control is passed back to the static core where the actual recompilation checks and dynamic reloading takes place. Once the new configuration data is returned `reloadE` uses it to set the current user interface style and key mappings. In this way we can inject new (stateless) code dynamically, enabling changes in configuration files to be immediately reflected in the running application. This process is illustrated in Figure 5.4.

We use dynamic reconfiguration for similar purposes in Lambdabot. Lambdabot was originally developed as a static binary of some 1500 lines of Haskell; in addition, it used dynamically loaded plugins as an extension mechanism, as described in Section 2.5.1. We refactored Lambdabot to use a minimal static core which loads the actual Lambdabot application dynamically.

5.3.2 State preservation

In the discussion so far, we neglected an important problem: during reloading a plugin, be it the main body of the dynamic application or an extension, we lose any state maintained in that plugin. In our experience, it is not unusual for plugins to require private state (for example, in the form of IORefs), which is reinitialized on reloading. The loss of this state may be acceptable for small plugins or configuration files, but is unacceptable if we attempt to dynamically reload large amounts of code, or indeed the application itself. Hence, we will now turn to discussing an approach to maintain plugin state across reloads by *state injection* from the static core.

When updating code dynamically, we wish to continue execution from exactly the point that we reached prior to the dynamic update. To do this, we must reconstruct any relevant application state built by the code before the update. Reconstructing an application's state in Haskell is usually simple. By default entire Haskell programs are purely functional, so simply re-entering the application via its normal entry point is enough to recreate the application's state from a previous run. There is no need to actually store state between runs, as we are always able to reconstruct it. *Referential transparency makes hot swapping easy!*

However, some applications require large amounts of mutable state, and reproducing the mutations to that state may be infeasible. Yi is such an application. Editor buffers are potentially very large flat byte arrays, constructed over the entire running time of the application. Any copying is prohibitively expensive. Buffers in Yi are stored in a single global state component of the editor. What we need to do is capture this state during the update, and then inject it into the new updated code, continuing exactly where we left off.

This brings us to a central reason for using a dynamic application structure centered around a minimal static core; i.e., a reason besides the desire to modify all functionality of the application at runtime. We use the static core to keep all global state while the dynamic linker reloads plugins. For this to work it is crucial that all requests to load or re-load plugins go via the static core, as mentioned previously. Ultimately,

the static core is the only safe place to keep the global state during reloading, as the whole dynamic application code may be replaced. In other words, we structure the entire dynamic application so that it takes its state as an argument, making it purely functional again, and hence, enabling state-preserving hot swapping.

5.3.3 Reloading the application

So far, we established that hot swapping needs to be via the static core, while passing any global state from the old to the new instance through the static core. We realize this by the `reboot` component of the configuration argument of type `DynamicT` passed to the dynamic entry point, as discussed in Section 5.3.1. The static core will pass the following function, with an unusual type, as a concrete value for `reboot`:

$$remain :: a \rightarrow IO\,()$$

It takes a state value and 'returns' a nominal `()` value—in fact it *never* returns to the caller, it instead reloads all the application code (which is fast, cf. Section 5.4). The input state is then passed back to the new main function we just loaded. The new dynamic entry point then restarts the editor and uses the state parameter to restore the previous environment. The state value lets us keep any files and buffers open in Yi, for example.

We utilize polymorphism in `remain`, as with other interfaces to the static core. This simplifies state transfer, as we can be sure that the static core can not depend on the representation of the state component. Parametericity will enforce this invariant.

The complete sequence of operations performed by `remain` is:

1. the dynamic code calls `remain` with the current state as an argument;
2. static core unloads the dynamic application;
3. the (new) dynamic application is loaded; and
4. the core calls the application's entry point, passing the old state as an argument.

In the static core, we implement this as follows:

```
remain :: a -> IO ()
remain st = do
   unloadPackage "yi"
   status <- load "Yi.o" [] [] "main"
   case status of
     LoadFailure e       -> return ()
     LoadSuccess m main' -> main' st
```

The function `unloadPackage` unloads the entire dynamic application (packaged as an archive). We then reload the code from object files, and re-enter the code via its normal entry point, supplying the old state value as an argument – making the application, in a sense, purely functional, as it is able to reconstruct its previous instance via parameters only. The dynamic code then inspects this value, and does a quick restart—avoiding the initial application start up—and instead directly entering the editor main loop.

In Yi we invoke the hot swap event by getting a handle to the current state, shutting down the editor (which involves terminating some threads, and exiting the user interface). We then jump into the static code by calling `reboot` (i.e. calling into the static core), and passing down the state.

```
rebootE :: Action
rebootE = do
    st <- getEditorState
    Editor.shutdown
    reboot (Just st)
```

When the new application code is entered after the reload, the old state is injected back into the application, and we short circuit the usual initialization steps.

We can increase the performance of dynamic reloading by structuring the dynamic code as a single library, rather than a tree of separate modules. Doing so means that the dynamic linker doesn't have to read interface files for each module it needs to load—to chase dependencies—and instead loads the entire dynamic linker in one load. In Section 5.4 we discuss the performance of this operation in detail.

5.3.4 Upgrading state types

Replacing code dynamically while still preserving state works well when extending functions. New functionality, bug fixes and extensions can all be integrated without having to quit the application. However, reinjection of global state is only safe in a statically typed language with type erasure, such as Haskell, if the definition of the state data type is unchanged in the new code.

If, for example, the buffer type changes to include an extra field and we restore the old state containing an old buffer value lacking this field, the new code will crash—it simply is not compiled to handle a value of the old type. What we need is a safe way of injecting an old state value into a new, different state type. This problem, state transfer, has been considered in work relating to operating systems supporting hot swapping [144, 13] as well as in the work of Hicks [64], and is an instance of the *expression problem* [160, 172]. A comprehensive introduction to the expression problem in the context of Haskell is given by Seefried [137], Chapter 5, in particular, describes the four requirements of a solution:

- Extensibility via new functions, and new data variants;
- Strong static type safety;
- No modification to existing code is allowed; and
- Separate compilation must be retained.

We tackle this problem by reducing it to a form of parsing, in both Yi and XMonad: we define an error-handling parsing function to inject values of the old type into the new type. To transfer values between types we convert the state value into a binary representation, via serialization. Prior to rebooting the application, we encode the old state value in this binary format. After injecting this binary state value into the new code, the value is parsed to construct a state value of the new type, where missing, or extra, fields are either ignored or replaced with default values. In this way global

state values can be preserved over data type changes, and the state value is effectively immortal.

A more complex solution could use a version tag in the state data type and a class of injection functions to achieve binary compatibility between arbitrary versions of the state type.

5.3.5 Persistent state

The problem of preserving state during dynamic code reloading is tightly connected to the general problem of persistence. Applications that are able to extract and inject their entire state can be extended to support persistence via the usual mechanisms—e.g. serialization of values to binary representations [164][5]. When passing the state value from dynamic code to the static core, the core can arrange to write the state to disk, and on rebooting, reread this state, passing it back to the dynamic main.

The problem of preserving state across code loading effectively forces us to separate and sanitize use of state in the application. Extending the state preservation to full persistence is relatively simple once we reach that point, and we do so in the large, in Lambdabot, where extensible modules define a state component which may be serialized when the application is shut down, automatically.

5.4 Performance

We discuss the performance of our dynamic architecture by way of results obtained with the Yi editor. All measurements were made on a Pentium-M 1.6Ghz laptop running OpenBSD 3.7, with 256M RAM.

Startup time. We first evaluate the startup cost of the dynamic architecture by comparing the startup time for the Yi editor with and without the dynamic architecture in Figure 5.5. Caches were primed with a dummy run before each run.

[5]Or using our `Data.Binary` library, http://hackage.haskell.org/package/binary

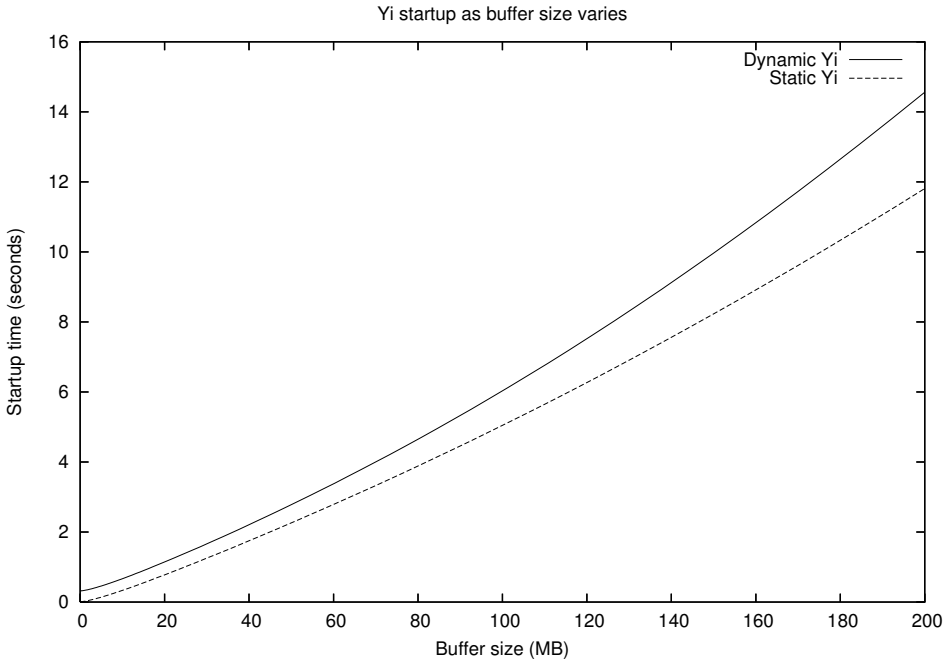

Figure 5.5: Dynamic architecture startup cost

We instrumented Yi to measure the time from entering the static main until the editor main loop, located in the dynamic application, commences. The startup time increases linearly with the buffer size (from the cost of loading the buffer into memory). The results demonstrate that the dynamic architecture has a *startup cost of around 30 milliseconds* on the benchmark platform for typical editing jobs.[6] In other words, the cost of loading buffers by far exceeds the startup costs of the dynamic architecture. As the buffer size increases, factors unrelated to dynamic linking come into play, in particular, after around 100M the amount of real memory has an impact as the machine begins to swap (graduate student laptops...).

Hot code swapping. We display the costs of state-preserving hot code swapping in Figure 5.6. The displayed time includes hot swap the entire application code, reinjecting the preserved state, and returning to the dynamic main loop.

[6]We define *typical* as file under 20M in size. The average source file in the Haskell hierarchical libraries is 10k; so, 20M is a very generous limit.

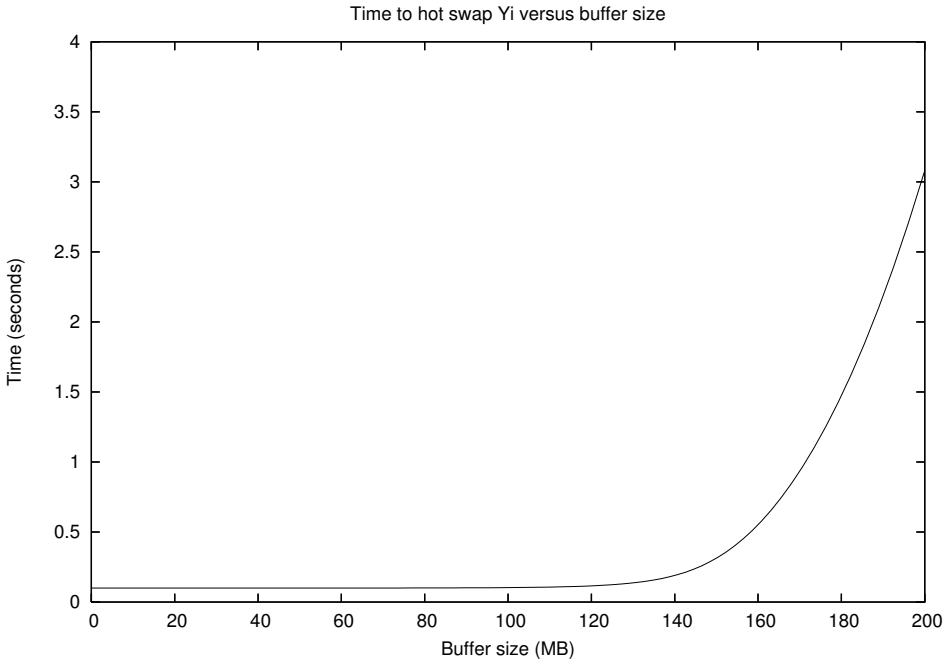

Figure 5.6: Performance of hot swapping

The results indicate *a constant cost of 10 milliseconds for hot swapping*, independent of the Yi state size (the state increases as the buffer size increases), up to buffer sizes of around 120M, at which point the test machine starts to run out of *physical* memory, and performance rapidly deteriorates as hot swapped code competes for space with the large buffer state. The cost of hot swapping is independent of the size of Yi buffers as we pass references to buffers to the static core, not buffers themselves.

Comparison with other editors. To evaluate Yi's overall performance, we compare it to a range of common editors in Figure 5.7. The benchmark involves the execution of a specific editing sequence submitted to the keystroke interfaces of each editor. We then test performance as buffer size increases. The measured time includes editor start, GUI initialization, execution of the edit sequence, and editor shutdown. It might have been preferable to measure each operation in isolation, but the interface of some editors does not support such isolated measurements.

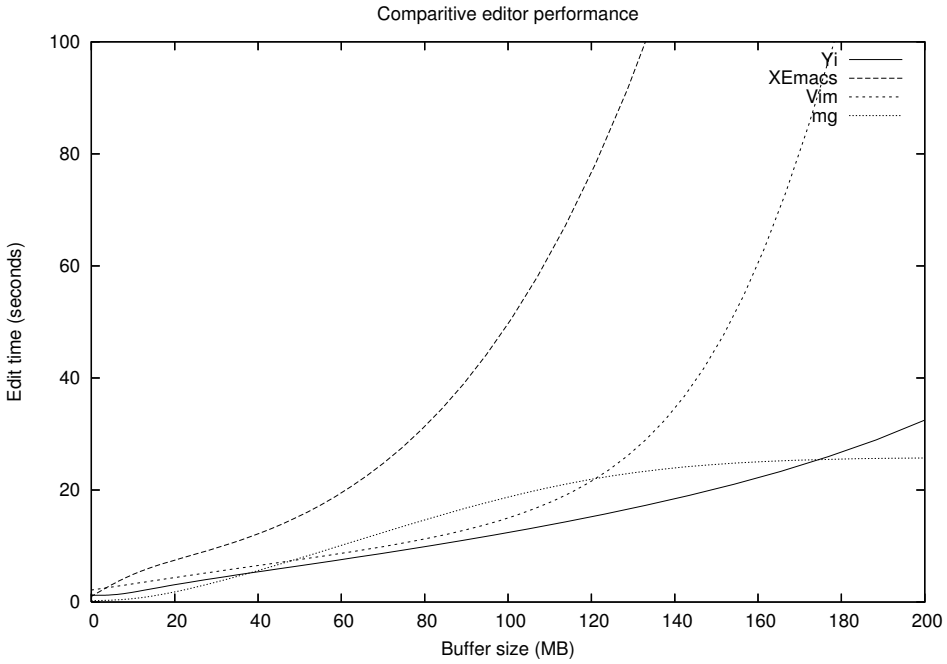

Figure 5.7: Comparative editor performance

It is interesting to observe that Yi (both dynamic and static) outperforms the mature extensible editors Vim and Emacs. It seems as if dynamic applications consisting of compiled functional code have a significant performance advantage over interpreted extensible systems. The small non-extensible editor `mg` also had good performance. We also tested several other small static editors (nano, ee and vi). They have good performance on small data sizes, but this advantage deteriorates rapidly as file sizes increase. We have not included them in the figure to improve readability.

5.4.1 Static applications

Dynamic linking is not supported in all environments that the compiler, GHC, runs in. For portability, it is convenient to be able to build a statically linked version of the application that provides all the same features and is usable in less supported environments. Such an application will lack the ability to dynamically load configuration files or new code.

Producing a static version of the dynamically loaded application can be achieved quite simply by adding a conventional `main :: IO ()` to the main module of the dynamic application, and arranging for the compiler to treat this as the normal entry point of the program, leaving the boot stub out of the compilation process. Both Yi and Lambdabot can be compiled as static-only applications in this way.

This is possible as the static core is minimal—it contains no application code—so even when it is left out, we still arrive at a working application. This clean separation of all dynamic linking-related code into a single static core is another reason for forcing the dynamic application to delegate all invocations of the dynamic linker to the static core.

5.4.2 Summary

We presented an architecture for *fully dynamic* applications in Haskell, based upon a minimal static core. We discussed a practical design and implementation of hot swapping for such dynamic applications, including state preservation as types change, and demonstrated the feasibility of this architecture with two applications, Yi and Lambdabot. We now describe the end-user impact of this design in a series of applications.

5.5 Applications

5.5.1 Lambdabot

Lambdabot is an Internet Relay Chat (IRC) robot described in previous chapters. IRC features messaging between individual users as well as *channels* where multiple users can meet to engage in conversation. In the chat network, bots appear as normal user clients that provide a wide range of services by reacting to commands sent to them directly or issued on a channel monitored by the bot. Some bots with special privileges authenticate users, award operator privileges to selected users, and stop network abuses. Other bots simply provide services to the users of a particular channel. Lambd-

abot belongs to the later category and is, for example, used on the `#haskell` channel of the `irc.freenode.net` network. Lambdabot services are implemented by a suite of dynamically loaded service plugins that include dictionary queries, weather forecasts, notification services, but also an online Haskell type checker and interpreter.

The first version of Lambdabot had a fairly large static core (about 1500 lines) implementing IRC protocol support. The service plugins amounted to another 5000 lines of Haskell. The main incentive for the use of plugins was ease of configuration and the ability to upgrade plugins without needing to shut down the bot. Channels, such as `#haskell`, almost always have active users who might want to use the bot's services. Hence, hot code updates are an attractive option. Unfortunately, the original architecture with the fairly large static core suffered from two drawbacks:

1. Patches to the core of Lambdabot required a complete restart—and the core is quite large (nearly 25% of the application's code); and

2. GHC's runtime system requires all libraries that are both used in statically and dynamically linked code to be in memory twice. This is clearly a waste of resources.

We refactored Lambdabot to use the dynamic architecture. Now, the static core contains a few dozen lines of Haskell, which are linked statically against the `hs-plugins` linker. This static core when invoked dynamically loads the Lambdabot main module, triggering cascading dynamic loading of the rest of what was formerly the static binary and is now the dynamic application code. After loading, control is transferred to the dynamic entry point of Lambdabot, passing a set of linker functions as an argument. These functions are then used to dynamically load Lambdabot's suite of service plugins. At this point the application connects to the server. Under this new architecture, we avoid both of the above problems.

Refactoring Lambdabot proved to be quite painless, despite Lambdabot having been developed without any thought of structuring it as a dynamic application. The responsibility of loading plugins was transferred from Lambdabot's simple object loader,

to the static core linked against the `hs-plugins` library. This was a matter of exporting linker functions in a manner similar to Yi, and in fact reduced and simplified the main application code, as all dynamic loading glue was factored into the static core.

Our experience with Lambdabot suggests that adapting other Haskell applications to the dynamic architecture we propose is feasible. Dynamically loading any Haskell application can be achieved with only small modifications to the application. In particular, the static entry point is simply hoisted to become the dynamic entry point.

5.5.2 XMonad

XMonad [149] is a tiling window manager for the X Window system, implemented, configured and dynamically extensible in Haskell. As for Yi, we choose Haskell as the configuration and extension language. Rather than developing an ad hoc extension language, XMonad is configured via a Haskell source module. This has enabled a large range of extensions to be built as library modules to XMonad[7], by hundreds of users, taking advantage of existing Haskell library and development tools. XMonad illustrates nicely that when the application language is sufficiently expressive, the configuration language may be written in the same language as a DSL, with benefits in productivity and safety for non-expert users.

XMonad is built around a reliable "micro-kernel" core application, of about 1000 lines of Haskell, enabling all the basic functionality of a window manager. It is then extended through hundreds of plugins, which users may import via their configuration file, `xmonad.hs`. Uniquely, it is the user's configuration file which defines the `main` method of the application, enabling radical extensibility – indeed, other window managers, such as Bluetile[8], are built on top of XMonad, treating the extensible application as a *library* for writing window managers.

At startup, the configuration file is compiled with GHC, and then executed. The user's configured XMonad code then hands control to the actual XMonad main routine,

[7] http://xmonad.org/xmonad-docs/xmonad-contrib

[8] Bluetile, www.bluetile.org

available via a library, passing in the configuration data structure to the application, in the style of Yi. The user's windows are then brought under management by XMonad.

At any point during execution, the user may edit the configuration file, modifying settings for the window manager. Hitting a key combination, `mod-q`, triggers recompilation of the user's configuration file, and the cycle starts over, with the configuration main executed, and new xmonad code loaded into the application's address space. Any state generated by the window manager is serialized and passed to the new configuration's `main` function, ensuring smooth upgrades as configurations change, in the manner described earlier in this chapter.

This state upgrade model is used by thousands of XMonad users daily, and has been shown to be a reliable way to ensure availability of a critical application – such as a window manager – while supporting extension via rich configuration – even through major version upgrades. New releases of XMonad trigger a recompile of the configuration file, and a reload, to the point that users may not know they have upgraded their window manager!

5.6 Discussion

There still remains much work to be done in the area of extensible, statically typed applications. Extensions to Yi and Lambdabot, for example, must currently be loaded as compiled plugins. In the case of Yi, the time taken to compile quick scripting jobs—such as evaluating a simple Haskell expression—is tedious when the code is immediately discarded once a result is produced. An alternative would be to embed a Haskell interpreter in the application and use that for small scripting jobs, while retaining compiled code for larger, more permanent application extensions. Haskell users already follow this pattern, writing throw-away code in an interpreter such as GHCi, but resorting to compilation for more long-lived code. Ideally we would like to be able to *mix* compiled code with interpreted code.

When extending the state of a dynamic application, consumers of the state, in particular the state transfer functions (which must preserve state over changes in the state type) need to be modified to handle the new state type. We speculate that a state design based upon extensible generic programming may reduce the cost of state extension and allow extensible state transfer functions. Work on open, extensible, functions [89] seems promising.

When extending the state type while hot swapping, we are currently forced to step outside of the type system—by translating the state value into a binary representation—and then reconstructing this value after the swap. Ideally, we would like to be able to automatically extend the state type, without requiring the binary encoding or requiring a new state type. Extensible records seem suitable in this regard [82]. The transfer of values as the state type is extended is achieved via a transform from a type τ to the same type augmented with new record fields. The result would be that preserving state over extensions to the state type can be typed inside the language without resorting to untyped binary intermediaries.

We avoid a large number of semantic concerns by hot swapping all dynamic code, which obviates the need to worry about references into old code persisting in hot swapped code. However, more fine grained hot swapping may be desirable in some applications.

5.7 Related work

Since the work on Yi described in this chapter, Jean-Philippe Bernardy has further developed the editor code base, increasing the range of embedded DSLs used in the application [14], and developing a new approach to syntax highlighting based on incremental parsing [15]. Karlsson further developed the parsing incremental infrastructure for Yi [83]. A detailed JavaScript mode for Yi was developed by Dogan [43].

Emacs. Emacs [145] was the first editor whose design revolved around facilitating extensibility. This implied an extension language beyond simple editing macros and the

ability to dynamically load extension code that can add new or override existing functionality. Limitations of the hardware and operating systems of the time, encouraged interpretation of extension code, and thus, required the most performance-sensitive functionality to be included in the static core (which is implemented in C in current flavors of Emacs). Stallman [145, Section 4], in fact, explicitly lists dynamic linking as an alternative approach, but dismisses it as impractical due to lack of operating system support. Moreover, Emacs pre-loads the most essential Lisp byte-code at build time and dumps the resulting runtime image into a binary file for faster startup.

Java and Eclipse. The architecture of the Eclipse workbench revolves around a hierarchy of dynamically loaded plugins [19]. At the core is the *Eclipse platform* together with a number of core plugins, which are always available. In recent versions of Eclipse, the core is close to minimal, permitting the use of the Eclipse plugin architecture for non-IDE applications. However, in contrast to Lisp applications and our proposal, Eclipse configuration is via XML meta data. Moreover, there seems to be no simple facility for hot swapping code. Hot swapping has also been used in the context of Java to implement dynamic profiling [42].

Erlang. Hot code swapping (aka dynamic software updating) is an important feature of Erlang [7, 8, 61, 78], which has been part of the language for several years. Hot swapping in Erlang is used, for example, in telephony applications that must have very high availability. Recent work has looked at formalizing the hot code swapping capabilities of Erlang [17]. Hicks [64] cites Erlang's functional nature as contributing to the simplicity of its hot swap implementation, making reasoning about global state much simpler. Sewell et al. have described more recent work on typing updates in distributed systems, focusing on marshalling and type equality concerns, via the Acute language [140].

Linux and other operating systems. The Linux kernel can be compiled such that most of its functionality is in kernel modules than can be loaded and unloaded at

runtime. When fully modularized, only functionality that is crucial during booting the kernel itself, such as memory management and process scheduling, are in the static core. Everything else, including support for the file system used for the root partition, is loaded at runtime; the root file system module comes from an initial RAM disk. Linux' level of modularization is limited as it began as a monolithic operating system with support for kernel modules added much later. In contrast, *micro kernel* operating systems provide a higher degree of modularization and, in some cases, hot swapping capabilities [12, 13, 144].

Hicks describes approaches to dynamic software updating in C-like languages [64, 65, 66], using an approach based on dynamic patching of application code, to support migration. The work is based on typed assembly language for type safe object code. In later work, a practical implementation of dynamic software update for C is developed [113], allowing large-scale C programs to be patched dynamically. Separately, Duggan develops type systems to model hot swapping [46].

Chapter 6

Language support for extension

The success of applications, such as Emacs, command shells, web browsers, Microsoft Word, Excel, and The GIMP, is to a significant degree due to them providing users with an *extension language*. An extension language – whether general purpose (e.g., Emacs's Lisp) or application specific (e.g. Word macros) – enables users to add new functionality to an application without understanding or re-compiling its source code. Indeed, extension languages have been advocated as an alternative to writing an application in a single language [120]. They are so convenient that some applications, most notably Emacs [145], consist of only a small core with almost all higher-level functionality being implemented in the extension language. Code components written in an extension language can usually be loaded at runtime, so that an application can be extended dynamically via plugins.

Lisp applications pioneered the idea of configuration files based on the extension language itself—in this case, Lisp. This approach is lightweight as it avoids a parser and analyzer for yet another language. Moreover, configuration languages have the tendency to slowly grow to include ad hoc abstraction facilities and control structures. These features are available in a systematic way right from the beginning if the extension language is used for configuration. This idea is even more powerful when combined with the concept of *embedded domain specific languages* (EDSLs). Then, configuration files can be tailored to the task and be used for complex configurations. For

example, our Yi editor uses self-optimizing lexer combinators (i.e., an embedded language of regular expressions) to define editor command sequences such that different configuration files emulate the interface of various existing editors (e.g., Nano, Vim, Emacs). This approach is more lightweight than, e.g., Emacs' keymaps. It has the added benefit that configuration files are statically type checked, and it illustrates how configurable components can be structured to achieve flexible, modular extensions.

In this chapter, we discuss the use of Haskell as a *typed* extension language. We argue that plugins can be viewed as a further step in the development of embedded domain specific languages (EDSLs) – that is, domain specific languages (DSLs) that are embedded in a host language (such as Haskell) instead of being implemented from scratch. In particular, when a DSL is used as an extension language, an implementation of that DSL as an EDSL gets plugin support for free by using the framework discussed in this thesis. Material in this chapter has been previously published [148, 146].

6.1 Domain specific languages for extension

In Section 2.8 we introduced the notion of plugins as extension modules loaded at predefined hooks in an application, and extension languages that make it simpler for non-expert users to extend applications. In the context of this work, when Haskell is used as an extension language, it seems particularly desirable to provide plugin authors with a domain specific notation. In other words, Haskell plugins make EDSLs more applicable, which in turn makes Haskell an especially attractive extension language.

6.1.1 Typed configuration files

As a first simple example of an application-specific EDSL, consider textual application configuration files, which are commonly called *resource* or *rc* files on UNIX-like systems (e.g. `.bashrc`, `.vimrc` or `.muttrc`). As an application becomes more complex language features are added to these configuration files, usually in an ad-hoc manner.

As a result the application is configurable in its own DSL, which typically lacks type safety, expressiveness, and a coherent syntax. Haskell plugins allow for a much more principled approach as we demonstrate with the example of a simple, typed EDSL for configuring a mail user agent.

First, the application describes the parameters available for user configuration, along with an interface instance with default values:

```
module MailConfigAPI where

data Interface = Interface {
      editor          :: IO String,
      attribution     :: String -> String,
      header_color    :: Color,
      colorize        :: [String],
      include_source  :: Bool }

data Color = Black | Cyan | Yellow | Magenta

mail :: Interface
mail = Interface {
      editor = return "vi",
      attribution = \ person -> person ++ ":",
      header_color = Black,
      colorize = [],
      include_source = False }
```

The structure clearly follows the plugin API modules that we considered before. Since the configuration API is written by the application programmer, it can be of the usual complexity. However, configuration files will be implemented by normal users who may not have any programming background. Hence, it is imperative to avoid clutter where possible. To help with this goal, we use the plugin framework's syntax *merge* facilities (see Chapter 4) that combines two Haskell source files, so that plugins can be partitioned into configuration information provided by the application user, and an application-supplied *stub* file with extra syntax (containing the module name, export list, and import declarations, and so on). The user's configuration file and the stub are merged together by the compilation manager to create the actual plugin source.

Haskell Server Pages provides a full-scale implementation of this model of syntactic extension, as described in Section 4.4.4.

In our mail configuration example, the stub may be:

```
module MailConfig (resource) where

import MailConfigAPI

resource :: Interface
resource = mail
```

Then, a user's configuration file might be:

```
import System.Directory

resource = mail {
  include_source = True,
  attribution = \n -> "Today, " ++ n ++ " wrote:",
  editor = do
    b <- doesFileExist "/usr/bin/emacs"
    return (if b then "emacs" else "vi") }
```

When the compilation manager merges the configuration file with the stub, declarations in the configuration file replace those of the same name in the stub. In our example, this has two useful effects: (1) if the configuration file is empty, `resource = mail` will provide a default configuration; and (2) the type signature in the stub file will be used to ensure that `resource` is of the right type. Our model of overriding default values via user-supplied record updates is used in XMonad, for example, as the primary configuration mechanism.

In the context of user-friendly EDSLs, the problem of how to achieve clear error messages arises. This is a standard problem of EDSLs, and we do not claim to have found a new solution here. However, the application can inspect and modify any error messages due to a `make` invocation before forwarding them to the user. In the case of XMonad, unnecessary verbosity is removed, and the concise error message is displayed in a new window, directing users to a line and column number in their configuration file, which appears to be a reasonable approach in practice.

Overall, the configuration file is compiled and loaded into the application at start-up, enabling user-defined behavior based on values from the plugin's `resource` data structure. As this example shows, the application author can re-use language infrastructure for the application's configuration system without having to resort to an ad-hoc language implementation, and gains additional features, such as type checking, that are often omitted from DSLs.

6.1.2 Lightweight parsers

The primary attractiveness of EDSLs lies in the reuse of the host language's features—however, one pays a price for this inheritance. While Haskell has some facilities for syntactic control (precedence and fixity can be controlled to some degree), it is not perfect. Mix-fix syntax cannot be defined, for example, nor could one define a new form of `do` notation without some form of preprocessing. These have to be specified as extensions to Haskell's syntax despite the fact that they are both merely syntactic sugar.

Additionally, it is not possible to easily *redefine* syntax, even though this may be desirable in a DSL context[1]. One of the most attractive features of DSLs is that they are easier for non-programmers to use, as the target audience already has familiarity with the notation of their domain. Ensuring the syntax matches that of the domain is not merely convenient; it can be crucial to the success of the language.

Providing that the DSL can be *semantically* embedded in Haskell, a plugin architecture with runtime compilation provides a solution: a *lightweight parser* can be written to perform a DSL-to-Haskell syntax transformation. This parser can then be called by the host application to transform the DSL into code suitable for compilation with `make`. The resulting Haskell plugin can then be compiled and dynamically loaded into the application. In other words, a lightweight parser is essentially a pre-processor of EDSL

[1]There is some support via the `-XNoImplicitPrelude` option, which provides a level of automatic syntax rebinding: numeric literal construction; equality; `do`-notation; and arrow notation may be re-bound to user-specified constructors.

code into vanilla Haskell. It can also produce syntax errors in a form that is easy to understand for the end user, as opposed to the Haskell-centric error messages that `make` produces. The situation is more tricky for errors relating to static semantics, as comprehensive error checking by the lightweight parser will make the parser significantly less lightweight, although it will be beneficial to usability.

By defining the syntax of the DSL purely as a frontend to the existing compiler infrastructure, we do away with compiler modules that are normally needed when writing a DSL, such as abstract syntax definitions, name supplies, and symbol table utilities. Not only is the compiler infrastructure cheaply available, but by using plugins, one could also plug in the DSL parsers themselves. Different syntax could be provided by different parser plugins, and users could then choose whichever syntax they prefer to write in. Clearly, in the context of extension languages, a plugin-based architecture affords extra flexibility to EDSLs, and makes them even more attractive as an implementation technique.

6.1.3 Layout extension in XMonad

In the XMonad window manager, users may customize the window manager by specifying layout algorithms for their workspace using Haskell functions. Layout algorithms may be arbitrarily complex, supporting custom internal state, and dynamic messages passing from the core to the extension components.

User-supplied layout extension functions take a stack of windows, and the screen geometry, and returns a list of geometries for each window. A type class is used to allow type-based indexing of layout functions in each extension, via a phantom. The type of layouts users supply are:

$$layout :: phantom\ a \rightarrow Rectangle \rightarrow Stack\ a \rightarrow [(a, Rectangle)]$$

The window manager then requests the X Window server to redraw all windows with the requested geometries. Dozens of useful algorithms have been written by users, enabling all sorts of custom workspace designs for different workflows[2].

In order to select between layouts, and deliver messages to extension modules, when the state of the server changes (for example, when a window is deleted), a form of dynamically extensible message architecture is implemented, following Marlow [104], adapted for an application-level message passing system, where the message type is *open*, and can be extended dynamically.

The core message type is specified via an open type class:

$$\mathit{class\ Typeable\ a\ \Rightarrow\ Message\ a}$$

Actual messages delivered by the system are existentially typed, in the following form:

$$\mathit{data\ SomeMessage\ =\ \exists\, a.\, Message\ a\ \Rightarrow\ SomeMessage\ a}$$

We then provide a dynamic cast of the message to each desired type, using the dynamic type cast provided by `Data.Dynamic`:

$$\begin{aligned}&\mathit{fromMessage\ ::\ Message\ m\ \Rightarrow\ SomeMessage\ \rightarrow\ Maybe\ m}\\&\mathit{fromMessage\ (SomeMessage\ m)\ =\ cast\ m}\end{aligned}$$

This infrastructure allows us to define new messages on a per-extension basis, and support new handler code in each component, without the need to recompile the core application to support the new extension types.

All the core message routing infrastructure sees are existentially typed values supporting the `Message` interface. We can thus extend the application's support for typed messages with both new message variants, and new message functions, without recompilation.

[2]Extending XMonad Layouts, http://xmonad.org/xmonad-docs/xmonad-contrib/XMonad-Doc-Extending.html#5

To make a particular type into a dynamic message, we add it to the class of messages. Here, for example, for X Window system events:

instance Message Event

To receive a message, a new extension component specifies a type-class method, `pureMessage`, which takes `SomeMessage` and casts it to the desired type. If the delivered message is not of the type the extension is looking for, `fromMessage` returns `Nothing`, and the message is passed on to other extensions to handle.

In this way we support open, extensible message handling by user extensions, via language support for open data types. XMonad includes other domain specific language support, including a monoidal language for specifying placement rules for client applications, and a concise keymap language for specifying keybindings, along with the core configuration language design, based on record update describe earlier in this thesis.

6.2 An extension language for Yi

As already mentioned, the Lisp experience shows that it is attractive to realize configuration files in a dynamic environment as dynamically loaded code. After that, it is a small step from configuration files to extension modules and the use of embedded domain specific languages (EDSLs) [70, 71] for both application configuration as well as extension. Languages that are used to extend applications dynamically are called *extension languages.* Many programs written in conventional languages opt to implement extension languages by embedding interpreters for new application-specific languages—Vim can be configured and extended in *Vim script*, for example. An alternative is to embed an existing scripting or extension language in the application, for example Perl [162], Lua [75, 77], or Scheme.

Lisp programs, such as Emacs, allow configuration and extension of the application in the application language itself. Yi, Lambdabot and XMonad are built following this model, as dynamically reloadable plugins directly provide extension and configuration

via compiled code. Moreover, we use EDSLs for more complex configurations and extensions.

To illustrate our approach to creating modular, flexible extensions, we use as a running example the self-optimizing lexer combinators used by Yi. These are an embedded regular expression language for implementing editor keystroke interfaces (or keymaps) based upon a lexer combinator library. When combined with dynamically reloadable configuration files and hot swapping, the use of a configuration and extension EDSL allows us to rapidly extend keystroke interfaces for Yi. Indeed, with very little programming effort, Yi is able to emulate, to differing degrees of completeness, several existing editors, including Vi [150] and Vim [20] (different editors of the Vi family) as well as Emacs [145], Nano [112] and Ee [74].

Configuration files in Yi are conventional Haskell modules which Yi arranges to compile and dynamically load, bringing user configuration data and code into the editor. The user may thus implement small interface extensions, or indeed completely new interfaces, by extending their configuration files. We use this EDSL to construct new lexers emulating existing editor interfaces, and make use of advanced lexer features, such as threaded lexer state storing nested lexer histories, and lexer table elements that trigger monadic lexer switching, to develop sophisticated interfaces.

6.2.1 The lexer language

The interface extension language of Yi is an embedded lexer combinator language based on the self-optimizing combinators described in our earlier work [29]. We use these as the foundation of a novel EDSL for the construction of *key bindings*. User-written lexer fragments are written in configuration files which are then appended to, or replace, default key bindings at runtime. The use of a DSL for specifying key bindings allows full interfaces to be constructed by those unfamiliar with the implementation of the application. Additionally, interfaces written in a combinator style can be constructed from fragments spread over multiple files, allowing code reuse. This enables users to conveniently specify their own custom mappings.

The use of a domain specific language for key bindings has enabled us to create specifications for the often obscure grammars of existing editor interfaces with little effort. We suspect that key binding construction via DSLs will allow cleaner and more intuitive interfaces to be constructed, as the gap from formal specification to implementation is less.

6.2.2 Overview of key bindings

Most editors have a fixed mapping of character sequences to editor actions. Users interact with the editor by typing character sequences which trigger specific functions in the editor. This method of interaction is analogous to an interpreter—keystrokes are lexed and parsed producing phrases in the *editor language* which specify editor state changes. These phrases are then executed, causing the corresponding changes to occur.

It is important to note that the input to the lexer is possibly infinite, making it essential to use a lexer that consumes input lazily (lexer generators that consume input strictly are not suitable). We can model a lazy keymap lexer as a function:

$$keymap :: [Char] \rightarrow [Action]$$

where `Action` is the type for editor actions. The return type of `keymap` is a list of such actions, as each key sequence generates a discrete action event, and there are an infinite number of such events.

In Yi the keybinding lexers have precisely this type. Multiple functions of this type can be loaded as a plugins to the editor, and lexer definitions can be implemented as separate combinator fragments spread across multiple modules. Different editor interfaces are emulated by writing different lexers. The result is an elegant combinator language for writing `[Char] -> [Action]` editor interfaces.[3]

[3]The original lazy lexer combinators described in Chakravarty [29] have been modified in two ways to support key binding programming: firstly, lexers immediately return tokens once they are uniquely determined (not waiting till the next input character). Secondly, lexer composition with overlapping bindings is permitted, with the new bindings overriding previous bindings.

Regular expressions	Construction and control
`epsilon`	`action`
`char`	`meta`
`string`	`>|<`
`star`	`>||<`
`plus`	`execLexer`
`quest`	
`alt`	

Figure 6.1: Lexer combinators in Yi

We briefly summarize the lexer combinator language in Figure 6.2.2. It consists of a set of regular expression combinators, and a number of combinators for binding regular expressions to actions, to produce lexer fragments (`action`, `meta`), joining regular expressions (`>|<`), combining lexer fragments (`>||<`), and running lexers over input (`execLexer`). The semantics of these operations are described by Chakravarty [29].

6.2.3 A simple example

Here is a simple example of how Yi users would dynamically extend the editor interface via code fragments in configuration files. The user writes a configuration file, `Config.hs`. This file will be loaded by the static core when the application is invoked, and reloaded on demand as the user extends or modifies the code:

```
module Config where

import Yi.Yi
import Yi.Keymap.Vim

yi = settings { keymap = keymapPlus bind }

bind = char 'n' `action` \_ ->
     Just $ mapM_ insertE "--"
```

In the above example, `settings` is a set of default configuration values, and `yi` is the distinguished value the dynamic loader expects to find in any config file. The `keymap` field specifies which lexer the editor is to use as its keystroke handler—in this case the Vim interface augmented with a binding for the character 'n', that when triggered inserts the Haskell comment token "-" into the buffer at the current point. The function `keymapPlus` augments the default binding with new lexer fragments, allowing us to compose the Vim lexer with new user-supplied code.

Users may thus write their own lexer bindings in configuration files using the EDSL, gaining the safety and expressiveness of Haskell in the process. Configuration files that fail type checking are rejected, and default values are substituted in their place.

6.2.4 Lexer table elements

Key bindings, under our scheme, are lexers from strings to action *tokens*. Keystroke interfaces use regular expressions to construct self-optimizing lexer tables specifying what editor actions to invoke given a particular series of keystrokes. The elements of the lexer tables are functions combining primitive editor operations. These are built from a set of around 80 primitives providing the following functionality:

- Movement commands: left, right, up, down, goto, ...
- Buffer editing actions: insert, delete, replace
- Reading portions of the buffer: read, read line
- Multiple-buffer actions: next, prev, focus, unfocus, close, split
- Undo/redo, yank/paste, search/replace
- File actions: new, write
- Meta actions: reboot, reload, quit, suspend
- Higher-level actions: map, fold

These functions return `Action` type, a synonym for `IO ()`. Our extension language thus allows us to construct bindings from character input to monadic expressions, specifying state changes to the editor. These actions values can be composed with `>>=` (or other monadic combinators), in the usual way.

When passed input, keymaps return a series of (unevaluated) editor actions. For example, in Vi or Vim emulation mode, user input of "`jl2x`" generates the following list of actions:

$$[downE,\ leftE,\ replicateM\_\ 2\ deleteE]$$

These actions are self-explanatory. As the user types characters a lazy list of these editor actions is produced. These actions in turn need to be forced, to generate their effects, so the main editor loop body is as follows:

$$sequence\_\ .\ keymap\ =<<\ getChanContents\ ch$$

The character input reader runs in a separate thread, returning a lazy list of keystrokes via a channel to the main thread. The list of keystrokes is passed as an argument to the current keymap, our (pure) key binding lexer. The resulting list of actions are evaluated as they are produced, causing immediate state changes in the application.

6.2.5 A complete interface

We now present a basic keybinding definition for the `ee` [74] editor, implemented in our combinator lexer extension language. This interface can be written to a configuration file, replacing the default interface, and may be extended while the editor is running. The brevity of the code provides an indication of the power of domain specific extension languages for this task, and we believe the lexer combinator EDSL to be of general utility for programming application keystroke interfaces:

```
keymap cs = fst3 $ execLexer lexer (cs, ())

lexer = insert >||< command

insert  = any `action` \[c] -> Just (insertE c)
```

```
command = cmd `action` \[c] -> Just $ case c of
    '\^L' -> leftE
    '\^R' -> rightE
    '\^U' -> upE
    '\^D' -> downE
    '\^B' -> botE
    '\^T' -> topE
    '\^K' -> deleteE
    '\^Y' -> killE
    '\^H' -> deleteE >> leftE
    '\^G' -> solE
    '\^O' -> eolE
    '\^X' -> quitE
    _     -> undefined
```

Where `any` and `cmd` are patterns that match any character, and the set of command characters, respectively. At the top level, a key binding takes a lazy list of input characters, and runs the lexer on this list. This causes the lexer table to be dynamically constructed. The lexer is built from two lexer fragments: one for self-inserting characters, and another for a small set of control characters.

Partially defined extensions, such as those including `undefined`, or code that throws other exceptions [126], may be caught by the application, and dealt with in the usual manner. For example, Yi catches and prints exceptions via the message buffer, before resuming execution in the main loop. Furthermore, the effect of malicious extensions can be mitigated somewhat using techniques described in Chapter 3.

We now consider more sophisticated interfaces utilizing threaded recursive state, lexer switching and finally monadic lexer switching.

6.2.6 Threaded state

A powerful feature of the lexer combinator language upon which we base our extension language is the ability to thread state through the lexer as it evaluates input. This has proved to be invaluable, and we use the state, for example, to communicate results between different regular expressions, to implement command history and line editing, as well as stackable dynamic key mappings.

We take as a simple example the task of emulating the prefix repetition arguments to commands used in the Vi family of editors, including Vim and Vi. Many commands can be optionally prefixed with numerical repetition arguments. For example `3x` is the sequence to delete three characters. We need a way to parse any numerical prefix n in a digit lexing fragment, but make that value available later on, once we've decided which action to perform.

We implement this by threading an accumulator as a state component through the lexer. Digit key sequences can be appended to this lexer state by the digit lexer fragment (as well as, perhaps, echoed to a message buffer), until a non-digit key is pressed. Control is then transferred to a command character lexer. Once the full command has been identified, the digit state value is retrieved and the command replicated by that value. The digit lexing code is

```
nums :: Lexer String Action
nums = digit `meta` \[c] s ->
  (msgE (s++[c]), s++[c], Just lexer)
```

The lexer state now consists of a `String` value. Rather than using the `` `action` `` combinator for binding regular expressions to actions, we instead access a lexer state component when the lexer table element is retrieved. This is achieved via the `` `meta` `` combinator, which allows us access to the lexer state component, as well as specifying (1) any token to return, (2) a new state value, and (3) a lexer to continue execution with.

In the above code, `s` is the state component of the lexer. When a digit is matched in the input stream, we extract the existing lexer state. We immediately echo the input value, and any previous input digits to the message buffer (via `msgE`), append the current character to the state, and continue execution with the default lexer. In this way digits will be accumulated, as well as being echoed to the screen each time they are pressed.

The state can be used later by the command lexer fragment:

```
command = cmd `meta` \[c] s ->
  (msgClrE >> fn c (read s), [], Just lexer)
```

```
where fn c i = case c of
        '\^L' -> replicateM_ i leftE
        '\^R' -> replicateM_ i rightE
        ...
```

Here we first clear the message buffer, then construct a new editor action using the digits stored in the state to specify the repetition. Finally, we return this action, along with a new empty state, and continue with the default lexer.

The state is convenient for other values we extract from the key input stream. For example, suppose we wish to maintain a command history. We can keep track of all editor input in a list value inside the lexer state, and later retrieve or index this history, when we receive user input to do so.

These simple examples illustrate the close fit between our lexer combinators, and the domain of interface keybindings. We continue by describing how to model more difficult features of key binding syntax.

6.2.7 Modes

Most editors consist of a set of *modes*, or distinct sets of key bindings. Usually only one such set is in operation at any point. Even in *modeless* editors there are submodes introduced when certain keys are pressed.

There is a direct encoding of modes as independent lexer fragments in our extension language, with the result that the distinction between moded and modeless editors evaporates—modeless editors just consist of one large default lexer, with smaller lexer fragments for submodes. The main problem is actually how to achieve mode switching. That is, how to bind a keystroke to an action that causes control to pass to a new lexer.

The usual way to switch modes is via a distinguished mode switching sequence. There are often multiple ways to enter a new mode, the difference being that particular actions are performed prior to the mode switch. These rules are directly implementable in the lexer framework, via the `meta` action.

For example, we may specify a lexer fragment for the Vi and Vim editors that specifies mode switching into 'insert' mode (equivalent to the default self-insertion mode in Emacs):

```
switchChar `meta` \[c] st ->
  let doI a = (with a, st, Just insert)
  in case c of {
'i' -> doI nopE
'I' -> doI solE
'A' -> doI eolE
'a' -> doI $ rightOrEolE 1
'o' -> doI $ eolE >> insertE '\n'
'O' -> doI $ solE >> insertE '\n' >> upE
'C' -> doI $ readRestOfLnE >>= setRegE >> killE
}
```

These bindings all cause a mode switch to the insert mode lexer (`insert`) where keys are inserted into the buffer by default. The first binding (`'i' -> doI nopE`) causes a direct switch, whilst the other bindings all perform actions of various complexity prior to the switch. For example, `'A'` moves the cursor to the end of the line, and then enters insert mode.

We may also pass state from one mode to another via the state component of the lexer. State is sometimes useful over mode switches. For example, when exiting from a line editing mode, hitting `Ctrl-M` causes the final edited input to be passed from the line editor, to a sub-mode that can then interpret the edited string.

6.2.8 Line editing

Line editing is a useful feature when users need to enter more than a couple of keys to specify an command. It is typical for an editor to provide line editing features—the `vi` ex mode, for example. A clean way to implement a line editor in a pure lexer infrastructure is to add a line buffer to the lexer's state, and write a set of lexer fragments bound to editing actions on this buffer.

Line editing can be considered an extension to the model we described for lexing digits. As users type keys, we display them in a message buffer, and we must also

accumulate each keystroke in a local line buffer that we thread through the lexer. Most characters add themselves to the lexer line buffer, however, certain keys, `delete` for example, specify editing operations on the line buffer. Finally, after correctly entering a string, the user can hit the return/newline key, causing the contents of the edit buffer to be passed as input to a command interpreter (specified in another lexer). We assume for this example that the lexer state type is a `String`.

Insertion is as for digit lexing. Simple deletion of the most recently input character can be specified by the following lexer action:

```
lnEdit = delete `meta` \_ st ->
  let st' = case st of
              []     -> []
              (_:cs) -> cs
  in (msgE (reverse st'), st', Just editLexer)
```

where `delete` is the delete character, `editLexer` is the composition of the line editor lexer fragments for insertion, deletion, completion (and any other actions we provide) in line editor mode.

Finally, we need to specify how to exit the line editor mode: by hitting return. We can then parse the state value we accumulated:

```
lnEval = enter `meta` \_ st ->
  case reverse st of
    "w"     -> viWrite
    "q"     -> closeE
    "wq"    -> viWrite >> closeE
    ...
```

In `vi`, a user can hit the `escape` key at almost any point, causing a mode switch to command mode. We can add this functionality to the line editor mode:

```
lnEsc = char '\ESC' `meta` \_ st ->
  (msgClrE, [], Just command)
```

That is, clear the message buffer, empty the state, and return to command mode.

A more featured line editor might store a *point*, describing the current insertion point into the buffer, and binding the arrow keys to shift this insertion point. Such

features can be added as additional lexer fragments, possibly specified by a user in a configuration file loaded as a plugin.

6.2.9 Command history

Command history is a simple extension of the line editor model. Rather than store a single string accumulator, we modify the lexer state to store a list of such lines, along with an index into this list, that can be modified by an action bound to the up and down arrow keys.

On pressing return, the current line pushed onto the command history stack, and at any time during line editing we can scroll through the command history.

A lexer fragment to modify the index into a command history threaded through a lexer state:

```
lnHist = arrow `meta` \[k] st@(i,h) ->
  let (i',s) = msg k st
  in (msgE s, (i',s), Just editLexer)
 where
  msg k (i,h)
    | null h       = (0,[])
%     | k == keyUp   = if i < length h - 1
                     then (h !! i, i+1)
                     else (last h, length h - 1)
    | k == keyDown = if i > 0
                     then (h !! i, i-1)
                     else (head h, 0)
```

We now consider how to add new bindings that dynamically extend editor key bindings.

6.2.10 Dynamic mappings

Many applications allow users to dynamically extend the table of key mappings by binding keys to a new sequence of keystrokes. In Vi or Vim this is achieved by `:map` and `:unmap` commands. We may also remap previously defined mappings, shadowing the former definition with a new definition.

In order to implement dynamically extensible mappings cleanly we would like to be able to update the application's lexer table at runtime. This is not possible if our lexer is either a hand-written parse function (i.e. the lexer table is compiled code) or if our lexer is an statically generated lexer table. One solution is to use dynamic reloading to recompile the lexer on the fly, however this is a rather sledgehammer approach when we simply want to update a lexer table.

An alternative is to construct new lexer combinators at runtime. This approach—implementing dynamically extensible key mappings via runtime constructed combinators—cleanly and elegantly allows us to implement dynamic mappings.

Let us consider user input in a `vi`-like editor of `:map zz d4j`. This binds the key sequence `zz` to the actions produced by typing `d4j`. This sequence deletes the next four lines down from the current line. How do we implement this using the lexer combinator extension language? What we need to be able to do is to inspect the current lexer table to find what actions the input `d4j` are bound to. We can then use the resulting actions to build a new lexer fragment dynamically.

To inspect the lexer table we need to run the current lexer, on the side, with `d4j` as input. This will produce a list of `Actions`. We can then use a fold to join the list of actions into a single action. This action we then use on the right hand side of a newly constructed lexer fragment (i.e. a new lexer table entry), bound to zz, which we use to augment the current lexer.

Assuming we wish to bind our `zz` to the command mode lexer, we can inspect the lexer table elements bound to an arbitrary input string, `inp`, like so:

```
lookupC inp = fst3 $ execLexer command (inp, [])
```

This builds a new lexer on the fly, with `inp` as input, returning a list of actions that are associated to the given input. We can construct a single action from the list of actions `as`. This can be achieved by folding the monadic » over this list, with a no-op to start the fold off. We then construct a new lexer fragment binding "zz" to the resulting `Action`:

```
bind zzs = string zzs `action` \_ ->
  Just (foldl (>>) nopE as)
```

The last step is to transfer control from the currently executing lexer to a new lexer augmented with the bind we have just constructed. The right hand side of the meta binding for the `:map` command is thus:

```
(msgClrE, [], Just $ command >||< bind)
```

where new bindings in the right argument of `>||<` replaces any existing binds in command.

One issue remains, however. When we combine two lexers we create a new lexer value to which we pass control. If we ever call `:map` again, we will add any new binding to the *original* lexer, losing any dynamic bindings. The solution to this is to store the most recent lexer, with any dynamic mappings it has, in the lexer state, so that bindings will not be lost on later mappings. We can extend the lexer state to store the current lexer, along with the accumulator, like so:

```
data State = State {
  acc  :: String,
  cmd  :: Lexer State Action
}
```

Now new key mappings will be bound to a meta action as before, and we also record this augmented lexer in the lexer state. Rather than hard coding which lexer control passes to a meta action, we instead transfer control to the lexer found in the threaded state. Now, we are in a position to consider how to implement command *unmapping*.

6.2.11 Nested mappings

It is often desirable to allow users to unmap commands, and most extensible editors support this. Two main approaches may be considered. The first is to only allow one level of mapping to exist at any point. That is, if the sequence zz is mapped twice, the second mapping will overwrite the first mapping, and it will be forgotten – this is Vi's behavior. Unmapping zz will then revert to the mapping for zz found in the default lexer, i.e. no mapping at all.

We can implement this simple unmap similar to map. We run the default lexer with the sequence we wish to unmap as input. Three possible values will result, either:

- No actions are bound to the sequence
- Exactly one action
- Multiple actions are bound to the sequence

If no actions were bound to some sequence we are unmapping, then we can unmap this sequence from the current mapping by binding it to the empty action: `nopE`. If exactly one action is bound to the sequence, then that input did have a previous binding, and we restore this by binding the input to the action we have just retrieved. The difficult case is if some input string *s* results in multiple actions—this implies that components of the string are individually bound. In this case we really need to delete the entry for *s* from the table entirely (mapping *s* to `nopE` won't do, as it adds a binding). Direct deletion of table elements is not yet possible, however we can emulate deletion by rebinding dynamically added bindings to the original lexer, filtering out the entry we wish to delete.

With this model it is possible to implement *stackable* dynamic mappings. Rather than new bindings overwriting previous dynamic bindings, we can instead revert to the last bind (which may have been the result of a `:map`) for a particular sequence. To do this we extend the lexer state to carry a table whose keys are input character sequences, and whose elements are lists of bindings. We can then implement an `:unmap` that reverts to the innermost binding for a particular sequence, by popping the top element of the list of bindings for that sequence. It would also be possible to browse the history of key bindings or to write the current set of bindings to a file, so that dynamic mappings can persist beyond the current editor session.

6.2.12 Higher order keystrokes

A number of existing editor interfaces have commands that take other commands as arguments. For example, it is common for simple editing commands to be parameterised by movement commands that when executed define a region of the buffer for the editing command to operate over. We can implement such semantics in a similar

way to dynamic mappings: we run a new lexer with the input key sequence to find the sequence of actions an input sequence maps to, and then use that to construct a composite action.

Extracting actions from the table is as before. For example, if we take the Vi or Vim key sequence `d4j` (delete four lines down), we then run a new lexer with `d4j` as input, returning a list of actions bound to `d4j`.

We can now take these actions and use them to construct a new expression of type `Action` to find the range of the buffer a command is to operate on. The following expression uses `as`, the actions bound to `d4j`, to calculate the start and end position in the buffer of a movement command:

```
do p <- getPointE
   foldr (>>) nopE as
   q <- getPointE
   when (p < q) $ gotoPointE p
   return (p,q)
```

We save the current point, and force the actions retrieved from the lexer. This yields a new point in the buffer. Then, we move back to where we started, and return a pair defining a buffer range. Finally, we can call the editor's deletion primitive with the range as an argument:

```
deleteNE (max 0 (abs (q-p) + 1))
```

We are thus able to implement parametrized bindings via lexer table elements that are defined via the results of running other elements of the lexer table.

6.2.13 Monadic lexer switching

One problem that we have not approached yet is how to implement lexer switching where the switch is based on the result of evaluating an element of the lexer table—that is, an `Action`.

Switching lexers based on the lexer input is easy—it's a pure operation, and we can use lexer `meta` actions to transfer control (and even pass control to lexers threaded

through the lexer state). However, there are some features that require a mode switch based on the outcome of an editor action. That is, we need to pass control to a specific lexer based on the value returned when evaluating an element found in the current lexer table.

An example of such an action is text searching. Applications often provide a search function. If a user types '/' in many editors they enter a line editor submode for regular expression searching. On entering a complete search phrase, a buffer search takes place. If a search successfully finds a match, the editor displays a prompt and switches to a sub lexer where the user can answer *yes* or *no* to trigger individual text replacements. If there is no match, an error message is displayed, and control continues as normal.

The problem, then, is how to specify that if we find a match (the result of an IO event), control should be transferred to the search key binding lexer. We solve this by introducing the `metaM` combinator, an editor action for lexer continuations. It has the type

```
metaM :: ([Char] -> [Action]) -> IO ()
metaM km = throwDyn (MetaActionException km)
```

where the dynamic exception is defined as

```
newtype MetaActionException =
  MetaActionException ([Char] -> [Action])
    deriving Typeable
```

The `metaM` function connects the elements of the lexer table to the currently running lexer. Such an action allows a table element to cause control to be passed to a new lexer. When a `metaM` action is evaluated, it throws a dynamic exception which wraps the new lexer to which control is transferred.

To actually use this new lexer, we need to catch this exception in the main loop of the application, and replace the current keymap with the thrown keymap. The top-level key input code for Yi is thus:

```
do let run km = catchDyn
        (sequence_ . km =<< getChanContents ch)
```

```
        (\(MetaActionException km') -> run km')
    run dflt
```

The simple main loop code is now wrapped in a handler that passes control to a thrown keymap. We begin evaluation of user input with the default keybinding, `dflt`.

The `metaM` action is a powerful function—it allows elements of the lexer table to cause control transfers to lexers specified by the table element. We justify its inclusion by outlining some common editor behaviors only implementable in our framework via `metaM`.

6.2.14 Prompts and user interaction

A common editor action requiring monadic lexer switch are command prompts. In the Nano editor [112], a user can begin a search by typing `Ctrl-W`. This causes a mode switch to a submode for searching, providing a line editor and some related commands (for continuing, or canceling the search, for example). On hitting return, the search is performed.

After completing a search, any further searches behave as before, except that the previous search string is used to set a prompt, and a default search value. The default search value is a compiled regular expression produced as an intermediate result of a previous search. The problem is how to script this behavior.

When switching to search mode we need to check if a previous expression has been searched for. In Yi, the global editor state stores the most recent compiled regular expression, and the string that it was generated from. When we switch to search mode the switch *action* extracts any previous regular expression and uses it to generate a prompt, which then dynamically creates a new lexer with the prompt as its default state. When this action is evaluated, we use `metaM` to switch control to the custom lexer generated by the action.

```
char '\^W' `action` \_ -> Just (a >>= metaM)
  where
    a = do
      mre <- getRegexE
```

```
let prompt = case mre of
  Nothing      -> "Search: "
  Just (pat,_) -> "Search ["++pat++"]: "
msgE prompt
return (mkKM prompt)
```

Here, `mkKM` constructs a new search keymap via `execLexer`, with the prompt as its default state. In this way we are able to program lexer mode switches based on the result of monadic actions, enabling us to implement many common interface features. In effect we have lexers that build new lexers on the fly, whose elements are actions that trigger switches between lexers.

6.3 Related work

Extensible systems are often designed to be extended by relatively untrained users, who are not expected to be familiar with the language the application is written in. Yet there is a tension when power users seek to change any aspect of the application's behavior they wish. This leads us inevitably to domain specific extension languages where complexity is controlled through the restricted domain of action of the language; yet full power is available by using a real language. Embedded domain specific languages address the "programmability gap" of extensible systems, avoiding the need for ad hoc, poorly designed extension languages, and allowing us to reuse tools, libraries and infrastructure easily.

Deursen et al. [158] provide the following definition:

> "A domain specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

A good DSL, then, accurately models the problem domain. This enhances productivity, maintainability, portability and correctness by hiding irrelevant detail. In addition, such abstraction makes explorative coding easier.

Functional DSLs

DSLs and EDSLs have been successfully used in many problem domains. The following are some examples of related Haskell domain specific languages, that have been developed.

New architectures. Anand et al.[5] describe an EDSL that allows mathematicians to formulate novel high-performance SIMD-parallel algorithms for the Cell processor, using a Maple-like EDSL embedded in Haskell.

GPUs. Multiple groups have developed EDSLs for programming GPU hardware at a high level, including Chakravarty et al [96] and Claessen et al [32].

Cryptography. Galois has developed a DSL – Cryptol – for the specification of cryptographic algorithms, enabling cryptographic experts to write high performance VHDL, and verify functional correctness automatically. [50].

Real time control. Hawkins described an EDSL used at Eaton to intuitively describe the safety-critical behavior of embedded code for hydraulic hybrid vehicle control, lowering the risk of introducing bugs in the design phase. [62].

Financial modelling. EDSLs are perhaps most widely used in the financial services industry. Several groups use EDSLs to generate modeling software, along with supporting documentation and other artifacts [9, 135, 53].

Operating systems. The designers of Microsoft's "Barrelfish" manycore operating system used a number of Haskell-based DSLs to generate parts of the kernel [37]. Notably, Linspire [16] used a series of configuration DSLs in Haskell to *add* static type safety to an existing Linux distribution, increasing assurance.

Hardware design. Antiope employed a similar strategy for the design of ultra-low power radio chips. Their EDSL played two roles: it was the main language for simulation used to debug their protocol designs, and it was also the implementation language for their verification tools [170].

Chalmers University and Xilinx designed a hardware description language that provides high-level abstractions and aids in proving circuit equivalence [18], while

Ericsson has developed an embedded domain specific language for digital signal processing Feldspar [52].

Extension languages

An alternative approach to domain specific extension via embedded DSLs is to instead use special-purpose extension languages. Lua [75, 77], is specifically designed to support easy embedding in applications written in C and C++, and has become widely popular for extension programming of games [76]. Such extension languages are designed as general purpose programming languages with semantics similar to the application host language. The application is thus split between two (or more) different language toolchains. We advocate throughout this work that a sufficiently rich application language has no need of a separate scripting language, as more precise domain languages may be embedded without significant effort. In a similar manner, Scheme and Perl have been popular application scripting languages, including use in text editors such as Vim.

Chapter 7

Conclusion

In this thesis we have shown how to combine methods for dynamic extension in software with strong safety via types. To do so we developed a framework to bring together many disparate fields of work: linking; compilation; meta-programming; dynamic update and domain specific language design. We instantiated the framework for Haskell with a native code dynamic linker, compilation manager, and approaches to runtime meta-programming, hot swapping and dynamic update. Finally, we considered how to make the life of the user easier via embedded domain languages for extension.

We described the overall framework in Chapter 2, introducing our designs and implementations for each component, before turning to the subsystems in detail. Chapters 3, 4 and 5 developed our ideas for native code linking, compilation and dynamic update, while Chapter 6 showed that once this infrastructure is in place, a rich application language can shine as an embedded extension language.

At all times we sought pragmatic demonstrations of feasibility, and developed several full-scale, widely used applications that used this work: Yi, a text editor; Lambdabot, a network bot used in one of the largest developer IRC channels; and XMonad, a popular tiling window manager. All these applications are configured and extended through typed domain specific languages embedded in Haskell, providing concrete evidence for the success of our approach.

In turn, other groups have built on the framework developed here, with optimizing DSLs; specializing simulators for scientific computing; new extensible applications; and libraries for bytecode extension of Haskell. With the work of this thesis in place we now routinely use Haskell as a strongly, statically typed extension language, with relatively untrained users developing extensions to our software.

Our goal then to resolve the tension between static type safety, and dynamic extension has been achieved, and we see a bright future for robust, flexible applications in statically typed functional languages, like Haskell, where we need not compromise safety for flexibility.

Appendix A

Type safe printf via meta-programming

We present a type safe *printf* for Haskell, based on runtime meta-programming. Commentary on the formatting strings is based on the OpenBSD 3.5 man page for *printf*.

```
{-# OPTIONS -fglasgow-exts #-}

module Printf.Compile (
        printf,
        (!),
        ($>), ($<),
   ) where

import Printf.Lexer
import Printf.Parser
import Eval.Haskell             ( eval )
import Eval.Utils               ( escape )
import Plugins.Utils            ( (<>), (<+>) )
import Data.Dynamic
import Data.Typeable hiding     ( typeOf )
import Data.List
import Data.Maybe               ( isNothing, isJust )
import System.IO.Unsafe
```

Some convenient aliases:

```
type Type = String
type Code = String
```

Generate a new Haskell function as compiled native code, from a printf format string. It isn't applied to its arguments yet. The function will return a String, but we won't typecheck this till application.

```
printf :: String → Dynamic
printf fmt = run src ["Data.Char","Numeric"]
    where
          src     = compile · parse · scan' · escape $ fmt
          scan' s = either (error "lexer failed") (id) (scan s)

          run e i = case unsafePerformIO (eval e i) of
                       Nothing → error "source failed to compile"
                       Just a  → a
```

Shortcuts for function application of dynamically typed arugments.

```
infixr 0 $<
($<) :: Dynamic → [Dynamic] → String
f $< as = fromDynamic $! f `dynAppHList` as

infixr 0 $>
($>) :: Dynamic → [Dynamic] → IO ()
f $> as = putStr (fromDynamic $! f `dynAppHList` as)
```

A printf code generator. Compile a printf format syntax tree into a Haskell string representing a Haskell function to implement this printf.

```
compile :: [Format] → String
compile fmt =
     let (tys,src) = compile' fmt 0
     in "toDyn $ \\" <>
        spacify (map (λ(ty,i) → parens('x':show i <+> "::" <+> ty))
                     (zip tys [0..length src])) <+> "->" <+> consify src

      where spacify s = concat (intersperse " "  s)
            consify s = concat (intersperse "++" s)
```

Compile an individual format or string literal:

```
compile' :: [Format] → Int → ([String],[String])
compile' [] _ = ([],[])
```

```
compile' ((StrLit s):xs) i = ( ts, ( '"':s++"\"" ):ss )
    where (ts,ss) = compile' xs i

compile' ((ConvSp _ _ _ _ Percent):xs) i = (ts, "\"%\"":ss)
    where (ts,ss) = compile' xs $! i+1

compile' (c@(ConvSp _ _ _ _ t):xs) i =
        (typeOf t:ts, parens(
           (snd · plus · pad · alt · trunc · codeOf) c
        <+> ident i) : ss)
    where (ts, ss) = compile' xs $! i+1
```

Now, what argument type does each printf conversion specifier map to?

```
typeOf :: Conv → Type
typeOf x = case x of
     D       → "Int"
     O       → "Int"
     Xx      → "Int"
     XX      → "Int"
     U       → "Int"
     C       → "Char"
     S       → "String"
     F       → "Double"
     Ee      → "Double"
     EE      → "Double"
     Gg      → "Double"
     GG      → "Double"
     Percent → error "typeOf %: conversion specifier has no argument type"
```

And now we generate Haskell code for each particular format, on a case-by-case basis.

```
codeOf :: Format → (Format,Code)
codeOf c@(ConvSp _ _ p _ f) = case f of
```

diouxX. The int (or appropriate variant) argument is converted to signed decimal (d and i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x and X) notation. The letters abcdef are used for x conversions; the letters ABCDEF are used for X conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.

```
D  → (c,"(show)")
U  → (c,"(show)")
O  → (c,"(\\v -> showOct v [])")
Xx → (c,"(\\v -> showHex v [])")
XX → (c,"(\\v -> map toUpper (showHex v []))")
```

eE. The double argument is rounded and converted in the style [-]d.ddde+-dd where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An E conversion uses the letter E (rather than e) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

```
Ee → let prec = if isNothing p then "Just 6" else show p
     in (c,"(\\v->(showEFloat("++prec++")v)[])")

EE → let prec = if isNothing p then "Just 6" else show p
     in (c,"(\\v->map toUpper((showEFloat ("++prec++")v)[]))")
```

gG. The double argument is converted in style f or e (or E for G conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style e is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.

```
Gg → let prec = if isNothing p then "Just 6" else show p
     in (c,"(\\v->(showGFloat("++prec++")v)[])")

GG → let prec = if isNothing p then "Just 6" else show p
     in (c,"(\\v->map toUpper((showGFloat ("++prec++")v)[]))")
```

f. The double argument is rounded and converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is

explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

```
F  → let prec = if isNothing p then "Just 6" else show p
     in (c, "(\\v -> (showFFloat ("++prec++") v) [])")
```

c. The int argument is converted to an unsigned char, and the resulting character is written.

```
C  → (c,"(\\c -> (showLitChar c) [])")
```

s. The char * argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating NUL character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NUL character.

```
S  → (c,"(id)")
```

%. A '%' is written. No argument is converted. The complete conversion specification is '%%'.

```
Percent → (c,"%")
```

And an error case:

```
codeOf _ = error "codeOf: unknown conversion specifier"
```

Determine if we need a leading '+'. A '+' character specifying that a sign always be placed before a number produced by a signed conversion. A '+' overrides a space if both are used.

```
plus :: (Format, Code) → (Format, Code)
plus p@(StrLit _,_) = p
plus a@(c@(ConvSp fs _w _ _ x), code) = case x of
    D → prefix
    Ee→ prefix
    EE→ prefix
    Gg→ prefix
    GG→ prefix
    F → prefix
    _ → a

  where prefix = let pref | Signed `elem` fs   = "\"+\""
                          | Space `elem` fs    = "\" \""
                          | otherwise          = "[]"
                 in (c,parens("\\v ->"<+>pref<+>"++ v") <$> code)
```

Now work out the required padding.

A negative field width flag '-' indicates the converted value is to be left adjusted on the field boundary. Except for n conversions, the converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A '-' overrides a '0' if both are given.

A zero '0' character specifying zero padding. For all conversions except n, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (d, i, o, u, x, and X), the '0' flag is ignored.

```
pad :: (Format,Code) → (Format,Code)
pad (c@(ConvSp fs (Just w) p _ x),code)

        | LeftAdjust `elem` fs
        = (c, parens(parens("\\i c s -> if length s < i"<+>
                        "then s ++ take (i-length s) (repeat c) else s")
              <+>show w<+>"' '")<$>code )
        | otherwise
        = (c, parens(parens("\\i c s -> if length s < i"<+>
                        "then take (i-length s) (repeat c) ++ s else s")
              <+>show w<+>pad_chr)<$>code)

   where pad_chr | isNumeric x ¨ isJust p        = "' '"
                 | LeadZero `elem` fs             = "'0'"
```

```
                | otherwise                       = "' '"

pad (c@(ConvSp _ Nothing _ _ _),code) = (c,code)

pad ((StrLit _),_) = error "pad: can't pad str lit"

isNumeric :: Conv → Bool
isNumeric x = case x of
        D  → True
        O  → True
        U  → True
        Xx → True
        XX → True
        _  → False
```

And check the 'alternate' modifier

A hash '#' character specifying that the value should be converted to an "alternate form". For c, d, i, n, p, s, and u conversions, this option has no effect. For o conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero). For x and X conversions, a non-zero result has the string '0x' (or '0X' for X conversions) prepended to it. For e, E, f, g, and G conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For g and G conversions, trailing zeros are not removed from the result as they would otherwise be.

```
alt :: (Format,Code) → (Format,Code)
alt a@(c@(ConvSp fs _ _ _ x), code) | Alt `elem` fs = case x of
        Xx → (c,parens("\\v->if fst (head (readHex v)) /= 0"<+>
                       "then \"0x\"++v else v")<$>code)
        XX → (c,parens("\\v->if fst (head (readHex v)) /= 0"<+>
                       "then \"0X\"++v else v")<$>code)
        O  → (c,parens("\\v->if fst(head(readOct v)) /= 0"<+>
                       "then \"0\"++v else v")<$>code)
        _  → a

alt a = a
```

Handle precision. Involves truncating strings and decimal points An optional precision, in the form of a period '.' followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for d, i, o, u, x, and X conversions, the number of digits to appear after the decimal-point for e, E, and f conversions, the maximum number of significant digits for g and G conversions, or the maximum number of characters to be printed from a string for s conversions.

```
trunc :: (Format,Code) → (Format,Code)
trunc (c@(ConvSp _ _ (Just i) _ x), code) = case x of
        S → (c, parens("(\\i s -> if length s > i"<+>
                                  "then take i s else s)"<+>show i)<$>code)
        _ | isNumeric x → (c, code)
          | otherwise   → (c, code)

trunc c  = c
ident i  = 'x':show i
parens p = "("++p++")"

infixr 6 <$>
(<$>) :: String → String → String
[] <$> a = a
a  <$> b = a ++ " $ " ++ b
```

Construct a new dynamic:

```
infixr 6 !
(!) :: Typeable a => a → [Dynamic] → [Dynamic]
a ! xs = toDyn a : xs
```

We must also define a lexer for printf format strings. We use the Alex lexical analyzer. The following lexer for printf format strings is based on B1.2 Formatted Output, from Kernighan and Ritchie.

```
{
module Printf.Lexer ( scan, Token(..) ) where
}
%wrapper "monad"
```

```
$digit = 0-9
$conv  = [dioxXucsfeEgGpn\%]
$len   = [hlL]
$flag  = [\-\+\ 0\#]
$str   = [· # \%]

printf :-

<0> $str+                   { mkstr      }
<0> \%                      { begin flag }

<flag> $flag*               { mkflags `andBegin` fmt }

<fmt>  $digit+              { mkint     }
<fmt> \.                    { mkdot     }
<fmt> $len                  { mklength  }
<fmt> $conv                 { mkconv `andBegin` 0 }

{

mkflags, mkconv, mklength, mkint, mkstr, mkdot :: AlexInput → Int → Alex Token

mkflags  (_,_,input) len = return (FlagT (take len input))
mkconv   (_,_,(c:_)) _   = return (ConvT c)
mklength (_,_,(c:_)) _   = return (LengthT c)
mkint    (_,_,input) len = return (IntT (read (take len input)))
mkstr    (_,_,input) len = return (StrT (take len input))
mkdot    _ _             = return DotT

alexEOF = return EOFT

data Token
        = FlagT   [Char]
        | ConvT   Char
        | LengthT Char
        | IntT    Int
        | StrT    String
        | DotT
        | EOFT
      deriving (Eq, Show)

scan :: String → Either String [Token]
scan str = runAlex str $ do
```

```
        let loop tks = do
                tok ← alexMonadScan;
                if tok  EOFT then do return $! reverse tks
                                 else loop $! (tok:tks)
        loop []
}
```

And, finall, a parser for the token stream, using the Happy parser generator for Haskell.

```
{
module Printf.Parser where

import Printf.Lexer

}

%name parse
%tokentype { Token }
%token

       'h'  { LengthT 'h' }
       'l'  { LengthT 'l' }
       'L'  { LengthT 'L' }
       'd'  { ConvT 'd' }
       'i'  { ConvT 'i' }
       'o'  { ConvT 'o' }
       'x'  { ConvT 'x' }
       'X'  { ConvT 'X' }
       'u'  { ConvT 'u' }
       'c'  { ConvT 'c' }
       's'  { ConvT 's' }
       'f'  { ConvT 'f' }
       'e'  { ConvT 'e' }
       'E'  { ConvT 'E' }
       'g'  { ConvT 'g' }
       'G'  { ConvT 'G' }
       '%'  { ConvT '%' }

       '.'  { DotT }

        INT { IntT   $$ }
```

```
     STRING { StrT   $$ }
      FLAGS { FlagT $$ }

%%

printf :: { [Format] }
        :  {- epsilon -}   { []      }
        | format0 printf { $1 : $2 }

format0 :: { Format }
        : string        { $1 }
        | format        { $1 }

string :: { Format }
        : STRING        { StrLit $1 }

format :: { Format }
        : flags width '.' precision length conv { ConvSp $1 $2 $4 $5 $6 }
        | flags width               length conv { ConvSp $1 $2 Nothing $3 $4 }

flags  :: { [Flag] }
        : FLAGS         { mkFlags $1 }

precision :: { Maybe Prec }
        : INT           { Just $1 }
        |  {- epsilon -}  { Nothing  }

width  :: { Maybe Width }
        : INT           { Just $1 }
        |  {- epsilon -}  { Nothing }

length :: { Length }
        : 'h'           { Short  }
        | 'l'           { Long   }
        | 'L'           { Double }
        |  {- epsilon -}  { Default}

conv   :: { Conv }
        : 'd'           { D }
        | 'i'           { D } -- n.b
        | 'o'           { O }
        | 'x'           { Xx }
        | 'X'           { XX }
```

```
        | 'u'           { U }
        | 'c'           { C }
        | 's'           { S }
        | 'f'           { F }
        | 'e'           { Ee }
        | 'E'           { EE }
        | 'g'           { Gg }
        | 'G'           { GG }
        | '%'           { Percent }

{
```

Abstract syntax for printf format strings

```
data Format
        = StrLit  String
        | ConvSp { flags     :: [Flag],
                   width     :: (Maybe Width),
                   precision :: (Maybe Prec ),
                   length    :: Length,
                   conv      :: Conv }
        deriving (Show, Eq)

type Width  = Int
type Prec   = Int

data Flag
        = LeftAdjust    -- -
        | Signed        -- +
        | Space         -- ''
        | LeadZero      -- 0
        | Alt           -- #
        deriving (Show, Eq)

data Length
        = Short         -- h
        | Long          -- l
        | Double        -- L
        | Default
        deriving (Show, Eq)

data Conv
        = D
```

```
        | O
        | Xx | XX
        | U
        | C
        | S
        | F
        | Ee | EE
        | Gg | GG
        | Percent
        deriving (Show, Eq)

mkFlags :: [Char] → [Flag]
mkFlags [] = []
mkFlags (c:cs) = (case c of
        '-' → LeftAdjust
        '+' → Signed
        ' ' → Space
        '0' → LeadZero
        '\#' → Alt) : mkFlags cs

happyError :: [Token] → a
happyError []  = error "Parser" "parse error"
happyError tks = error $ "Parser: " ++ show tks

}
```

Appendix B

Launching a dynamic application

The boot loader for the editor described in Chapter 5 is as follows:

{-# OPTIONS -fglasgow-exts -cpp #-}

The boot loader for Yi. This is a small stub that loads the runtime Yi library, then jumps to it. It solves the problem of unnecessary code linking. As such, we only want to do plugin-related stuff here. So we don't even mess with the locale stuff yet. We should have no static dependencies on any other part of Yi. One of our jobs is to find a (`value :: Config`) to pass to `Yi.dynamic_main` (a dynamically loaded module in the Yi – To do this we need to check if the '-B' flag was on the command line.

Once we find our Config file we have to:

- "Config.o" if it exists
- `load -package yi`, and any dependencies, through "Yi.o"
- Jump to `Yi.dynamic_main` with `v:: Config` as an argument.

This is the only module that depends on the plugins package.

```
module Boot ( main, remain ) where
```

```
import Plugins
import Plugins.Utils          ( (</>), (<.>) )

import Data.Maybe             ( fromJust, isJust )
import Data.IORef
import Control.Monad          ( when )
import System.Directory
import System.Console.GetOpt
import System.IO.Unsafe       ( unsafePerformIO )
import System.Environment     ( getArgs, getEnv )
import System.Exit            ( exitFailure )

#ifndef NO_POSIX
import System.Posix.User
#endif
```

Yi uses config scripts stored in " /.yi"

```
config_dir, config_file, yi_main_obj :: FilePath
config_dir      = ".yi"                 -- /.yi/ stores Config.hs
config_file     = "Config.hs"           -- name of user-defineable Config file

yi_main_obj     = "Yi.o"                -- entry point into yi lib

config_sym, yi_main_sym :: Symbol
config_sym      = "yi"                  -- symbol to retrieve from Config.hs
yi_main_sym     = "dynamic_main"        -- main entry point
```

Where do the libraries live? This value can be overridden on the command line with the -B flag

```
libdir :: IORef FilePath
libdir = unsafePerformIO $ newIORef (LIBDIR :: FilePath)
 {-# NOINLINE libdir #-}
```

Now, squirrel away our config module, so `:reboot` and `:reload` can find the module to reuse.

```
g_cfg_mod :: IORef (Maybe Module)
g_cfg_mod = unsafePerformIO $ newIORef (error "no config data")
 {-# NOINLINE g_cfg_mod #-}
```

Save this module for reload as well.

```
g_yi_main_mod :: IORef Module
g_yi_main_mod = unsafePerformIO $ newIORef (error "no Yi.o loaded yet")
 {-# NOINLINE g_yi_main_mod #-}
```

Finding config files. Use 'Control.Exception.catch' to deal with broken or missing implementations of get functions.

```
get_home :: IO String
#ifndef NO_POSIX
get_home = Control.Exception.catch
        (getRealUserID >>= getUserEntryForID >>= (return ` homeDirectory))
        (~ _ ı getEnv "HOME")
#else
get_home = error "Boot.get_home not defined for this platform"
#endif

-- .yirc/
get_config_dir  :: IO String
```

```
get_config_dir = do
    home ˚ get_home
    return $ home </> config_dir

-- .yirc/Config.hs
get_config_file :: IO (String)
get_config_file = do
    home ˚ get_home
    return $ home </> config_dir </> config_file
```

Now, do some argv parsing to detect any -B args needed to find our runtime libraries. Any other args are ignored and passed through to "Yi.main". Think like GHC's "+RTS -RTS" options, except we don't remove -B flags – we expect them to be ignored by Yi.main

```
data Opts = LibDir FilePath

options :: [OptDescr Opts]
options = [
        Option ['B']  ["libdir"]
            (ReqArg LibDir "libdir")
            "Path to runtime libraries"
    ]

doArgs :: [String] ı (Maybe FilePath)
doArgs argv = case getOpt Permute options argv of
        (o, _, _) ı case reverse o of
                        []           ı Nothing
                        (LibDir d:_) ı Just d
```

Given a source file, compile it to a (.o, .hi) pair. First, jump to the /.yi/ directory, in case we are running in-place, to prevent bogus module dependencies.

```
compile :: FilePath → IO (Maybe FilePath)
compile src = do
    build_dir ° get_config_dir
    old_pwd   ° getCurrentDirectory
    setCurrentDirectory build_dir

    flags  ° get_make_flags
    status ° makeAll src flags

    setCurrentDirectory old_pwd

    case status of
        MakeSuccess _ obj → return $ Just obj
        MakeFailure errs  → do
            putStrLn "Errors in config file, using defaults"
            mapM_ putStrLn errs
            return Nothing
```

What packages do we need to build Config.hs?

```
packages :: [String]
packages = [ "yi" ]
```

Flags to find the runtime libraries, to help GHC out.

```
get_make_flags :: IO [String]
get_make_flags = do
    libpath ° readIORef libdir
```

```
    return $ concatMap (f libpath) packages
  where
    f l p = ["-package-conf", l </> p <.> "conf", "-package", p]

get_load_flags :: IO [String]
get_load_flags = do
    libpath ˚ readIORef libdir
    return $ map (˜p ı libpath </> p <.> "conf") packages
```

Now, load the config module

```
load_config :: Maybe FilePath ı IO (Maybe Module, Maybe ConfigData)
load_config m_obj = do
    paths   ˚ get_load_flags
    libpath ˚ readIORef libdir
    d       ˚ get_config_dir
    case m_obj of
        Nothing  ı return (Nothing,Nothing)
        Just obj ı do
           status ˚ load obj [d,libpath] paths config_sym
           case status of
                LoadSuccess m v ı return $ (Just m, Just (CD v))
                LoadFailure e   ı do
                    putStrLn "Unable to load config file, using defaults"
                    mapM_ putStrLn e ; return (Nothing, Nothing)
```

Find and load a config file. Load the yi core library. Jump to the real main in 'Yi.main', passing any config information we found.

The library structure in memory: Boot.o loads Yi.o, which depends on HSyi.o. So to reload the yi core, you need to unload Yi.o and HSyi.o, and then reload Yi.o.

```
main :: IO ()
main = do
    -- look for -B libdir flag
    argv ˚ getArgs
    let mlib = doArgs argv
    when (isJust mlib) $ writeIORef libdir (fromJust mlib)

    -- check if HOME/.yi/ exists
    d        ˚ get_config_dir
    d_exists ˚ doesDirectoryExist d
    when (˝ d_exists) $ createDirectory d

    libpath  ˚ readIORef libdir
    paths    ˚ get_load_flags

    -- look for HOME/.yi/Config.hs
    c        ˚ get_config_file
    c_exists ˚ doesFileExist c
    m_obj    ˚ if c_exists then compile c else return Nothing

    -- now load user's Config.o if we have it
    (mmod, cfghdl) ˚ load_config m_obj

    -- squirrel away the module, for reload() purposes
    writeIORef g_cfg_mod mmod

    -- now, get a handle to Main.dynamic_main, and jump to it
    status   ˚ load (libpath </> yi_main_obj) [] paths yi_main_sym
```

```
    yi_main ˚ case status of
        LoadSuccess m v ı  do writeIORef g_yi_main_mod m
                              return (v :: YiMainType)
        LoadFailure e ı do putStrLn "Unable to load Yi.Main, exiting"
                           mapM_ putStrLn e ; exitFailure

    -- jump to dynamic code
    yi_main (cfghdl, remain, reconf)
```

So, we want to reboot the editor. That is, recompile and reload configuration data. And reload the entire HSyi.o library.

```
remain :: IO ()
remain = do

    -- unload Yi.o
    yi_main_mod ˚ readIORef g_yi_main_mod
    unload yi_main_mod

    -- unloadPackage HSyi.o
    unloadPackage "yi"

    -- reload Yi.o, pulling in HSyi.o
    libpath ˚ readIORef libdir
    paths   ˚ get_load_flags
    status  ˚ load (libpath </> yi_main_obj) [] paths yi_main_sym

    yi_main ˚ case status of
        LoadSuccess _ v ı return (v :: YiMainType)
        LoadFailure e ı do putStrLn "Unable to reload Yi.Main, exiting"
```

```
                        mapM_ putStrLn e ; exitFailure
```

```
    -- reload config data. Crucial this comes after we reload HSyi.o,
    -- so that we link against the new version of the lib
    cfghdl ˚ reconf

    -- jump to dynamic code
    yi_main (cfghdl, remain, reconf)
```

Compile and reload configuration module

```
reconf :: IO (Maybe ConfigData)
reconf = do

    -- try to recompile
    c        ˚ get_config_file
    c_exists ˚ doesFileExist c
    m_obj    ˚ if c_exists then compile c else return Nothing

    -- reload out config modules
    cfg_mod  ˚ readIORef g_cfg_mod
    (mmod, cfghdl) ˚ case cfg_mod of
        Just m  ı do        -- was already loaded
            status ˚ reload m config_sym
            case status of
                LoadSuccess m' v ı return $ (Just m', Just (CD v))
                LoadFailure _    ı return (Nothing, Nothing)

        Nothing ı load_config m_obj
```

```
    writeIORef g_cfg_mod mmod
    return cfghdl
```

This is the type of the value passed from Boot.main to Yi.main. It must be exactly the same as the definition in Yi.hs. We can't, however, share the value in another module, without breaking ghci support (the same module would be linked both statically and dynamically).

It is an existential to prevent a dependency on ConfigAPI in Boot.hs. It gets unwrapped magically in "Yi.dynamic_main".

```
data ConfigData = » a` CD a

type YiMainType = (Maybe ConfigData,
                   IO (),
                   IO (Maybe ConfigData))
                ı IO ()
```

The entry point of the booted application, taking state passed from the bootloader, is as follows:

{-# OPTIONS -fglasgow-exts #-}

This is the real main module of Yi, and is shared between Main.hs (the static binary), and dynamically loaded by Boot.hs. We take any config arguments from the boot loader (if that is how we are being invoked) parse command line args, initialise the ui, before jumping into an event loop.

```
module Yi (static_main, dynamic_main) where

import Yi.Locale                        ( setupLocale )
import Yi.Version                       ( package, version )
```

```
import qualified Yi.Editor  as Editor
import qualified Yi.Core    as Core
import qualified Yi.Style   as Style

import qualified Yi.Map as M

import qualified Yi.Keymap.Vi    as Vi
import qualified Yi.Keymap.Vim   as Vim
import qualified Yi.Keymap.Nano  as Nano
import qualified Yi.Keymap.Emacs as Emacs

import qualified Yi.Curses.UI as UI

import Data.IORef

import Control.Monad            ( liftM, when )
import Control.Concurrent       ( myThreadId, throwTo )
import Control.Exception        ( catch, throw )

import System.Console.GetOpt
import System.Environment       ( getArgs )
import System.Exit
import System.IO.Unsafe         ( unsafePerformIO )
import System.Posix.Signals

import GHC.Base                 ( unsafeCoerce# )
import GHC.Exception            ( Exception(ExitException) )
```

Argument parsing. Pretty standard, except for the trick with -B. The -B flag is needed, and used by Boot.hs to find the runtime libraries. We still parse it here, but ignore it.

```
data Opts = Help
          | Version
          | Libdir String
          | LineNo String
          | EditorNm String
```

In case the user wants to start with a certain editor

```
editorFM :: M.Map [Char] ([Char] ı [Editor.Action])
editorFM = M.fromList $
    [("vi",     Vi.keymap)
    ,("vim",    Vim.keymap)
    ,("nano",   Nano.keymap)
    ,("emacs",  Emacs.keymap)
    ]


options :: [OptDescr Opts]
options = [
    Option ['V']  ["version"] (NoArg Version)
        "Show version information",
    Option ['B']  ["libdir"]  (ReqArg Libdir "libdir")
        "Path to runtime libraries",
    Option ['h']  ["help"]    (NoArg Help)
        "Show this help",
    Option ['l']  ["line"]    (ReqArg LineNo "[num]")
        "Start on line number",
```

```
    Option []      ["as"]      (ReqArg EditorNm "[editor]")
        "Start with editor keymap"
    ]

usage, versinfo :: IO ()
usage    = putStr   $ usageInfo "Usage: yi [option...] [file]" options
versinfo = putStrLn $ package++" "++version

do_opts :: [Opts] → IO ()
do_opts (o:oo) = case o of
    Help     → usage    » exitWith ExitSuccess
    Version  → versinfo » exitWith ExitSuccess
    Libdir _ → do_opts oo  -- ignore -B flag. already handled in Boot.hs
    LineNo l → writeIORef g_lineno ((read l) :: Int) » do_opts oo

    EditorNm e → case M.lookup e editorFM of
                    Just km → do
                        (k,f,g) ˚ readIORef g_settings
                        writeIORef g_settings (k { Editor.keymap = km }, f, g)
                        do_opts oo
                    Nothing → do
                        putStrLn $ "Unknown editor: "++show e++". Ignoring."
                        do_opts oo

do_opts [] = return ()

do_args :: [String] → IO (Maybe [FilePath])
do_args args =
```

```
    case (getOpt Permute options args) of
        (o, n, []) → do
            do_opts o
            case n of
                []   → return Nothing
                fs   → return $ Just fs
        (_, _, errs) → error (concat errs)
```

Set up the signal handlers.

```
initSignals :: IO ()
initSignals = do

    tid ← myThreadId

    -- to pass control-C to keymap (think emacs mode)
    installHandler sigINT (Catch (return ())) Nothing

    -- ignore
    sequence_ $ flip map [sigPIPE, sigALRM]
                         (~sig → installHandler sig Ignore Nothing)

    -- and exit if we get the following:
    -- we have to do our own quitE here.
    sequence_ $ flip map [sigHUP, sigABRT, sigTERM] $ ~sig → do
            installHandler sig (CatchOnce $ do
                    releaseSignals
                    UI.end
                    Editor.shutdown
                    throwTo tid (ExitException (ExitFailure 2))) Nothing
```

```
releaseSignals :: IO ()
releaseSignals =
    sequence_ $ flip map [sigINT, sigPIPE, sigHUP, sigABRT, sigTERM]
              (~sig 1 installHandler sig Default Nothing)
```

The "g_settings" var stores the user-configurable information. It is initialised with the default settings, so it works even if the user doesnt provide a HOME/.yi/x.hs, or stuffs up their config scripts in some way. The second component is our reboot function, passed in from Boot.hs, otherwise return ().

```
g_settings :: IORef (Editor.Config, IO (), IO Editor.Config )
g_settings = unsafePerformIO $
                newIORef (dflt_config
                         ,static_main
                         ,return dflt_config)
 {-# NOINLINE g_settings #-}
```

Default values to use if no HOME/.yi/Config.hs is found

```
dflt_config :: Editor.Config
dflt_config = Editor.Config {
        Editor.keymap = Vim.keymap,
        Editor.style  = Style.ui
    }
```

The line number to start on:

```
g_lineno :: IORef Int
g_lineno = unsafePerformIO $ newIORef (1 :: Int)
 {-# NOINLINE g_lineno #-}
```

Static main. This is the front end to the statically linked application, and the real front end, in a sense. 'dynamic_main' calls this after setting preferences passed from the boot loader. Initialise the UI getting an initial editor state, set signal handlers, then jump to UI event loop with the state.

```
static_main :: IO ()
static_main = do
    setupLocale
    args   ° getArgs
    mfiles ° do_args args
    config ° readIORef g_settings
    lineno ° readIORef g_lineno

    Control.Exception.catch
        (initSignals » Core.startE config lineno mfiles » Core.eventLoop)
        (˜e ı do releaseSignals
                 Editor.shutdown
                 UI.end
                 when (¨ $ isExitCall e) $ print e
                 throw e
        )
    where
      isExitCall (ExitException _) = True
      isExitCall _ = False
```

Dynamic main. This is jumped to from from Boot.hs, after dynamically loading HSyi.o. It takes in user preferences, sets a global variable if any settings were received, then jumps to static main.

```
dynamic_main :: YiMainType
```

```
dynamic_main v = dynamic_main' v

dynamic_main' :: YiMainType
dynamic_main' (Nothing, fn1, fn2) = do
    modifyIORef g_settings $
        ~(dflt,_,_) → (dflt,fn1, liftM unwrap fn2)
    static_main     -- No prefs found, use defaults
```

Preferences found by the boot loader, so write them into the global settings.

```
dynamic_main' (cfg, fn1, fn2) = do
        let cfg_ = unwrap cfg
        writeIORef g_settings (cfg_, fn1, liftM unwrap fn2)
        static_main
```

Unwrap a ConfigData value

```
unwrap :: Maybe ConfigData → Editor.Config
unwrap Nothing          = dflt_config
unwrap (Just (CD cfg)) = unsafeCoerce# cfg :: Editor.Config

data ConfigData = ∀ a. CD

type YiMainType = (Maybe ConfigData,
                   IO (),
                   IO (Maybe ConfigData))
                → IO ()
```

Bibliography

[1] Anders Aaltonen, Alex Buckley, and Susan Eisenbach. Flexible Dynamic Linking for .NET. *.NET Technologies*, 2006(4):1–8, June 2006.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, 1991.

[3] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Remy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, 1995.

[4] Lauri E. Alanko. Types and Reflection. Master's thesis, University of Helsinki, November 2004.

[5] Christopher Kumar Anand and Wolfram Kahl. A domain-specific language for the generation of optimized SIMD-parallel assembly code. Technical report, McMaster University, 2007. http://www.cas.mcmaster.ca/~kahl/Publications/TR/Anand-Kahl-2007a_DSL/.

[6] Krasimir Angelov and Simon Marlow. Visual Haskell: a full-featured Haskell development environment. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 5–16. ACM, 2005.

[7] Joe Armstrong. Erlang — a Survey of the Language and its Industrial Applications. In *INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog*, pages 16–18, 1996.

[8] Joe Armstrong. The development of Erlang. In *ICFP '97: Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, 1997.

[9] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: a two-stage DSL embedded in Haskell. *ICFP '08: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 43(9):225–228, 2008.

[10] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. *ICFP '02: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 37(9):157–166, 2002.

[11] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: challenges and solutions to updating an operating system on the fly. In *ATC'07: 2007 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[12] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Technical Conference*, pages 279–291. USENIX Association, 2005.

[13] Andrew Baumann, Jeremy Kerr, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Module hot-swapping for dynamic update and reconfiguration in K42. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, Australia, April 2005.

[14] Jean-Philippe Bernardy. Yi: an editor in Haskell for Haskell. In *Haskell '08: Proceedings of the ACM SIGPLAN Symposium on Haskell*, pages 61–62. ACM, 2008.

[15] Jean-Philippe Bernardy. Lazy functional incremental parsing. In *Haskell '09: Proceedings of the ACM SIGPLAN Symposium on Haskell*, pages 49–60. ACM, 2009.

[16] Clifford Beshers, David Fox, and Jeremy Shaw. Experience report: using functional programming to manage a Linux distribution. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 213–218. ACM, 2007.

[17] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE)*, April 2003.

[18] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *ICFP '98: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*. ACM SIGPLAN, 1998.

[19] Azad Bolour. Notes on the Eclipse plug-in architecture. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, 2003.

[20] Bram Moolenaar. The Vim Editor. http://www.vim.org/.

[21] Niklas Broberg. Haskell server pages through dynamic loading. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 39–48. ACM, 2005.

[22] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242/1986, pages 21–47. Springer-Verlag, 1986. LNCS 242.

[23] Luca Cardelli. Phase distinctions in type theory. Technical report, Digital Equipment Corportation, 1988.

[24] Luca Cardelli. Program fragments, linking, and modularization. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277. ACM, 1997.

[25] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[26] Bernard Carré and Jonathan Garnsworthy. SPARK—an annotated Ada subset for safety-critical programming. In *TRI-Ada '90: Proceedings of the Conference on TRI-ADA '90*, pages 392–402. ACM, 1990.

[27] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 278–292. ACM, 1991.

[28] Hugh Chaffey-Millar, Don Stewart, Manuel M. T. Chakravarty, Gabriele Keller, and Christopher Barner-Kowollik. A parallelised high performance Monte Carlo simulation approach for complex polymerisation kinetics. In *Macromolecular Theory and Simulation*, volume 16 (6), pages 575–592, 2007.

[29] Manuel M. T. Chakravarty. Lazy lexing is fast. *Fourth Fuji International Symposium on Functional and Logic Programming*, LNCS 1722:68–84, 1999.

[30] Manuel M. T. Chakravarty et al. *A Primitive Foreign Function Interface*, 2004. http://www.cse.unsw.edu.au/~chak/haskell/ffi/.

[31] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 90–104. ACM, 2002.

[32] Koen Claessen, Mary Sheeran, and Joel Svnesson. Obsidian: GPU programming in Haskell. http://www.cse.chalmers.se/~joels/writing/dccpaper_obsidian.pdf, 2009.

[33] Christopher Colby, Karl Crary, Robert Harper, Peter Lee, and Frank Pfenning. Automated techniques for provably safe mobile code. *Theor. Comput. Sci.*, 290(2):1175–1199, 2003.

[34] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[35] Duncan Coutts, Isaac Potoczny-Jones, and Don Stewart. Haskell: batteries included. In *Haskell '08: Proceedings of the 2008 ACM SIGPLAN Symposium on Haskell*, pages 125–126. ACM, 2008.

[36] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ICFP '98: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 301–312. ACM, 1998.

[37] Pierre-Evariste Dagand, Andrew Baumann, and Timothy Roscoe. Filet-o-Fish: Practical and dependable domain-specific languages for OS development. In *Workshop on Programming Languages and Operating Systems*, 2009.

[38] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '92: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[39] Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8:621–625, 1998.

[40] Tony Davie, Kevin Hammond, and Juan Quintela. Efficient persistent Haskell, 1998.

[41] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A formal specification for the Haskell 98 module system. In *Haskell '02: Proceedings of 2002 ACM SIGPLAN Workshop on Haskell*, pages 17–28, Pittsburgh, PA, USA, October 2002.

[42] Mikhail Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150. ACM, 2004.

[43] Deniz A. M. Dogan. A JavaScript Mode for Yi. Master's thesis, Chalmers University of Technology, June 2009.

[44] Dominic Duggan. Dynamic types have existential type. Technical report, University of Waterloo, December 1994.

[45] Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *Transactions on Programming Languages and Systems*, January 1999.

[46] Dominic Duggan. Type-based hot swapping of running modules. In *ICFP '01: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, pages 62–73. ACM, 2001.

[47] Dominic Duggan. Type-safe linking with recursive DLLs and shared libraries. *Transactions on Programming Languages and Systems*, November 2002.

[48] Conal Elliott. Functional Image Synthesis. In *Proceedings of Bridges 2001, Mathematical Connections in Art, Music, and Science*, 2001.

[49] Conal Elliott. Tangible functional programming. *ICFP '07: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 42(9):59–70, 2007.

[50] Levent Erkök and John Matthews. High assurance programming in Cryptol. In *CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*, pages 1–2. ACM, 2009.

[51] Mike Fagan. *Soft typing: an approach to type checking for dynamically typed languages*. PhD thesis, Rice University, Houston, TX, USA, 1991.

[52] Eotvos Lorand University Faculty of Informatics Feldspar group. Feldspar: Functional Embedded Language for DSP and PARallelism. http://feldspar.sourceforge.net/.

[53] Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Going functional on exotic trades. In *Journal of Functional Programming*, volume 19, pages 27–45, 2009.

[54] Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical report, The University of Edinburgh, 1997.

[55] Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250–261. ACM, 1999.

[56] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[57] Martin Grabmüeller and Dirk Kleeblatt. Harpy: run-time code generation in Haskell. In *Haskell '07: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 94–94. ACM, 2007.

[58] Keith Hanna. Pivotal: An interactive, document-centered presentation of Haskell. Online, http://www.cs.kent.ac.uk/projects/pivotal/index.html, September 2005.

[59] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 341–354. ACM, 1990.

[60] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141. ACM, 1995.

[61] B. Hausman. Turbo Erlang: Approaching the speed of C. In E. Tick and G. Succi, editors, *Implementations of Logic Programming Systems*, pages 119–135. Kluwer, Dordrecht, 1994.

[62] Tom Hawkins. Controlling hybrid vehicles with Haskell. In *Proc. ACM CUFP '08*. ACM, 2008.

[63] Hajnalka Hegedüs and Zoltán Horváth. Distributed computing based on clean dynamics. In *Proceedings of the 6th International Conference on Applied Informatics*, January 2004.

[64] Michael Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[65] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. *ICFP '01: Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, 36(5):13–23, 2001.

[66] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 13–23. ACM, 2001.

[67] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In *TIC '00: Selected papers from the Third International Workshop on Types in Compilation*, pages 147–176, London, UK, 2001. Springer-Verlag.

[68] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[69] Ralf Hinze. Formatting: a class act. *J. Funct. Program.*, 13(5):935–944, 2003.

[70] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[71] Paul Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[72] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: being lazy with class. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 12–1–12–55. ACM, 2007.

[73] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[74] Hugh Mahon. The Editor 'ee'. http://mahon.cwx.net/.

[75] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of an extension language: A history of Lua. In Martin A. Musicante and E. Hermann Hausler, editors, *V Simpósio Brasileiro de Linguagens de Programação*, pages B–14–B–28, Curitiba, May 2001.

[76] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *HOPL III: Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, pages 2–1–2–26. ACM, 2007.

[77] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua - an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.

[78] Erik Johansson, Mikael Pettersson, and Konstantinos Sagonas. A high performance Erlang system. In *PPDP '00: Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM, 2000.

[79] Isaac Jones. The Haskell Cabal: A common architecture for building applications and libraries. Presented at TFP 2005, Tallinn Estonia, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.9361&rep=rep1&type=pdf, September 2005.

[80] Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.

[81] Mark P. Jones. Experience report: playing the DSL card. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 87–90. ACM, 2008.

[82] Mark P. Jones and Simon Peyton Jones. Lightweight extensible records for Haskell. In *Proceedings of the 1999 Haskell Workshop*. Published in Technical Report UU-CS-1999-28, Department of Computer Science, University of Utrecht, September 1999.

[83] Anders Karlsson. Robust and Precise incremental parsing in Haskell. Master's thesis, Chalmers University of Technology, August 2009.

[84] Gabriele Keller, Hugh Chaffey-Millar, Manuel M. T. Chakravarty, Don Stewart, and Christopher Barner-Kowollik. Specialising simulator generators for high-performance Monte-Carlo methods. In *Proceedings of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL 08)*, volume 4902, pages 116–132. Springer-Verlag, 2008.

[85] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988.

[86] Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4: formally verifying a high-performance microkernel. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 91–96. ACM, 2009.

[87] Leif Kornstaedt. Alice in the land of Oz: An interoperability-based implementation of a functional language on top of a relational language. In *Proceedings of the First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01)*. Elsevier Science Publishers, 2001.

[88] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37. ACM, 2003.

[89] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ICFP '05: Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 204–215. ACM, September 2005.

[90] Konstantin Läufer. Combining type classes and existential types. In *Proceedings of the Latin American Informatic Conference (PANEL)*, September 1994.

[91] Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, May 1996.

[92] Konstantin Läufer and Martin Odersky. Type classes are signatures of abstract types. In *Seminar and Workshop on Declarative Programming*, pages 148–162, London, UK, 1992. Springer-Verlag.

[93] Konstantin Läufer and Martin Odersky. Self-interpretation and reflection in a statically typed language. In *OOPSLA Workshop on Reflection and Metalevel Architectures*, 1993.

[94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 24–35. ACM, 1994.

[95] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. *SIGPLAN Not. Best of PLDI 1979-1999*, 39(4):540–553, 2004.

[96] Sean Lee, Manuel M. T. Chakravarty, Vinod Grover, and Gabriele Keller. GPU kernels as data-parallel array computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009.

[97] Daan Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*. ACM, September 2004.

[98] Xavier Leroy. Programming with dependent types: passing fad or useful tool? Talk at WG2.8, IFIP Working Group 2.8 26th meeting, Frauenchiemsee, Germany, http://www.comlab.ox.ac.uk/ralf.hinze/WG2.8/26/slides/xavier.pdf, June 2009.

[99] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system 3.08. http://ropas.kaist.ac.kr/caml/ocaml/, July 2004.

[100] Xavier Leroy and Michel Mauny. Dynamics in ML. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 406–426. Springer-Verlag New York, Inc., 1991.

[101] Nancy Leveson and Clark Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[102] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 33, pages 36–44. ACM, 1998.

[103] Ian Lynagh. Template Haskell: A report from the field. URL: http://web.comlab.ox.ac.uk/oucl/work/ian.lynagh/papers/Template_Haskell-A_Report_From_The_Field.ps, May 2003.

[104] Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 96–106. ACM, 2006.

[105] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[106] John McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, 1978.

[107] Erik Meijer and Danny van Velzen. Haskell server pages: Functional programming and the battle for the middle tier. In *Haskell '00: Proceedings of the 2000 ACM SIGPLAN Workshop on Haskell*. ACM, 2000.

[108] Alexandra F. Mendes and João Ferreira. PURe CAMILA: A system for software development using formal methods. Online, http://wiki.di.uminho.pt/twiki/pub/Research/PURe/CAMILA/Relatorio.pdf, September 2005.

[109] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[110] Eugenio Moggi, Walid Taha, Zine E Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive. Technical report, 1998.

[111] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[112] Nano Core Development Team. GNU Nano Text Editor. http://www.nano-editor.org/.

[113] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–83. ACM, 2006.

[114] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM, 1997.

[115] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, 1992.

[116] Henrik Nilsson. Functional automatic differentiation with dirac impulses. In *ICFP '03: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 153–164, Uppsala, Sweden, August 2003. ACM Press.

[117] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, September 2007.

[118] Elliott I. Organick. *The multics system: an examination of its structure*. MIT Press, Cambridge, MA, USA, 1972.

[119] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Sebastopol, CA, USA, 2008.

[120] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31:23–30, March 1998.

[121] Bruno Pagano, Olivier Andrieu, Thomas Moniot, Benjamin Canou, Emmanuel Chailloux, Philippe Wang, Pascal Manoury, and Jean-Louis Colaço. Experience report: using Objective Caml to develop safety-critical embedded tools in a certification framework. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 215–220. ACM, 2009.

[122] André Pang, Don Stewart, Sean Seefried, and Manuel M. T. Chakravarty. Plugging Haskell In. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, pages 10–21. ACM, 2004.

[123] Emir Pasalic, Walid Taha, and Tim Sheard. Tagless staged interpreters for typed languages. In *ICFP '02: Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 218–229. ACM, 2002.

[124] Nigel Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1991.

[125] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[126] Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 25–36. ACM, 1999.

[127] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 50–61. ACM, 2006.

[128] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[129] Marco Pil. First class file I/O. In *Implementation of Functional Languages*, pages 233–246, 1996.

[130] Marco Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages*, pages 169–185, 1998.

[131] Rinus Plasmeijer and Marko van Eekelen. Concurrent clean language report version 2.1. November 2002.

[132] Morten Rhiger. A foundation for embedded languages. *ACM Trans. Program. Lang. Syst.*, 25(3):291–315, 2003.

[133] Niklas Röjemo. Highlights from nhc—a space-efficient Haskell compiler. In *FPCA '95: Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture*, pages 282–292. ACM, 1995.

[134] Curt J. Sampson. Experience report: Haskell in the 'real world': writing a commercial application in a lazy functional language. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 185–190. ACM, 2009.

[135] Cyril Schmidt and Anne-Elisabeth Tran Qui. The default case in Haskell: counterparty credit risk calculation at ABN AMRO. In *CUFP '07: Proceedings of the 4th ACM SIGPLAN Workshop on Commercial Users of Functional Programming*, pages 1–2. ACM, 2007.

[136] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 51–62. ACM, 2008.

[137] Sean Seefried. *Language Extension via Dynamically Extensible Compilers*. PhD thesis, University of New South Wales, June 2006.

[138] Sean Seefried, Manuel M. T. Chakravarty, and Gabriele Keller. Optimising Embedded DSLs using Template Haskell. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*, Lecture Notes in Computer Science. Springer Verlag, 2004.

[139] Julian Seward, Simon Marlow, Andy Gill, Sigbjorn Finne, and Simon Peyton Jones. Architecture of the Haskell Execution Platform (HEP). http://www.haskell.org/ghc/docs/papers/, 1999.

[140] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: high-level programming language design for distributed computation. In *ICFP '05: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 15–26. ACM, 2005.

[141] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, 2002.

[142] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302. ACM, 1998.

[143] Brian Cantwell Smith. The limits of correctness. *SIGCAS Comput. Soc.*, 14,15(1,2,3,4):18–26, 1985.

[144] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *Proc. of the Usenix Technical Conference*, 2003.

[145] Richard M. Stallman. Emacs the extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147–156, 1981.

[146] Don Stewart. Domain Specific Languages for Domain Specific Problems. *LACSS Workshop on Non-Traditional Programming Models*, 2009.

[147] Don Stewart, Hugh Chaffey-Millar, Gabriele Keller, Manuel Chakravarty, and Christopher Barner-Kowollik. Generative Code Specialisation for High-Performance Monte-Carlo Simulations. Technical Report 0710, University of New South Wales, 2007.

[148] Don Stewart and Manuel M. T. Chakravarty. Dynamic applications from the ground up. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 27–38. ACM, September 2005.

[149] Don Stewart and Spencer Sjanssen. Xmonad. In *Haskell '07: Proceedings of the 2007 ACM SIGPLAN Workshop on Haskell*, pages 119–119. ACM, 2007.

[150] Sven Verdoolaege and Keith Bostic. The Berkeley Vi Editor. http://www.bostic.com/vi/.

[151] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# (Expert's Voice in .Net)*.

[152] Walid Taha. A sound reduction semantics for untyped CBN mutli-stage computation. or, the theory of MetaML is non-trival (extended abstract). In *PEPM '00: Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 34–43. ACM, 1999.

[153] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 32(12):203–217, 1997.

[154] The GHC Team. The Glasgow Haskell Compiler (GHC). http://haskell.org/ghc, 2010.

[155] The LLVM Team. The LLVM Compiler Infrastructure. http://llvm.org/, 2010.

[156] The Open Group Technical Standard. IEEE Std 1003.1. standard for information technology - portable operating system interface (POSIX). Shell and utilities. Technical report, 2004.

[157] Peter Thiemann and Stefan Wehr. Interface types for Haskell. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 256–272, Berlin, Heidelberg, 2008. Springer-Verlag.

[158] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[159] Arjan van Weelden and Rinus Plasmeijer. A type safe interactive interpreter for a functional language using compiled code (draft). In *Draft Proceedings of the 15th International Workshop on Implementation of Functional Languages*, pages 363–378, 2003.

[160] Philip Wadler. The expression problem. Discussion on the Java Generic-ity mailing list, http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, 1998.

[161] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.

[162] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[163] Malcolm Wallace. hi — hmake interactive — compiler or interpreter? In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 339–345, 2000.

[164] Malcolm Wallace and Colin Runciman. The bits between the lambdas: binary data in a lazy functional language. In *ISMM '98: Proceedings of the 1st International Symposium on Memory Management*, pages 107–117. ACM, 1998.

[165] Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *ICFP '99: Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM.

[166] Stefan Wehr and Manuel M. T. Chakravarty. ML modules and Haskell type classes: A constructive comparison. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 188–204, Berlin, Heidelberg, 2008. Springer-Verlag.

[167] Stephanie Weirich. Type-safe cast: (functional pearl). In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 58–67. ACM, 2000.

[168] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Comput. Surv.*, page 8, 1998.

[169] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. In *LFP '94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 250–262. ACM, 1994.

[170] Gregory Wright. Functions to junctions: ultra low power chip design with some help from Haskell. In *Proc. ACM CUFP '08*. ACM, 2008.

[171] Nicholas C. Zakas. Maintainable JavaScript: Don't modify objects you don't own. Online posting. March 2, 2010, http://www.nczonline.net/blog/2010/03/02/maintainable-javascript-dont-modify-objects-you-down-own/, 2010.

[172] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. Technical Report 33, École Polytechnique Fédérale de Lausanne, 2004.