Plugging Haskell In

André Pang
Information & Communication Technologies
CSIRO, Australia
Andre.Pang@csiro.au

Don Stewart Sean Seefried Manuel M. T. Chakravarty
Programming Languages and Compilers
School of Computer Science and Engineering
University of New South Wales
{dons,sseefried,chak}@cse.unsw.edu.au

Abstract

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 domain-specific languages, which support runtime *plugins* via dynamic loading of components. We show that Haskell can be comfortably used as a statically typed extension language for both Haskell and foreign-language applications supported by the Haskell FFI, and that it can perform type-safe dynamic loading of plugins using dynamic types. Moreover, we discuss how plugin support is especially useful to applications where Haskell is used as an embedded domain-specific language (EDSL). We explain how to realise type-safe plugins using dynamic types, runtime compilation, and dynamic linking, exploiting infrastructure provided by the Glasgow Haskell Compiler. We demonstrate the practicability of our approach with several applications that serve as running examples.

Categories and Subject Descriptors

D.3.m [**Programming Languages**]: Miscellanous

General Terms

Design, Languages

Keywords

Extension languages, Dynamic loading, Dynamic typing, Staged type inference, Functional programming, Plugins

1 Introduction

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 recompiling its source code. Indeed, extension languages have been advocated as an alternative to writing an application in a single language [25]. They are so convenient that some applications, most notably Emacs [32], 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; such components are often called *plugins*.

In this paper, we discuss the use of Haskell as a *typed* extension language and outline an approach to the dynamic loading of Haskell plugins based on *dynamic types* [1, 2, 4, 22, 37]. We illustrate the broad scope of applications for Haskell plugins and 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 paper.

We have implemented a library, called *hs-plugins*, capable of dynamically compiling and loading Haskell code into a running application's address space, utilising the object loading capabilities of the Glasgow Haskell Compiler's runtime system [30, 33]. The functions 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: this enables any application that can interface with C to easily support Haskell as an extension language.

In summary, our main contributions are the following:

- We describe an architecture for type-safe plugins in Haskell, using dynamic typing.
- We integrate extensions languages and EDSLs.
- We introduce Haskell plugins in foreign languages and lightweight parsers for more user-friendly EDSLs.

The paper consists of two main parts. First, Sections 2, 3, and 4 illustrate the usefulness of Haskell plugins with increasingly challenging application scenarios. Second, Sections 5, 6, and 7 discuss the principles behind the implementation of plugins in Haskell.

Haskell'04, September 22, 2004, Snowbird, Utah, USA.

2 Plugins in Haskell

A convenient way of extending an application dynamically is by way of *plugins*; i.e., software components that the application loads and links while already running. Plugins present two distinct challenges: (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*. In the remainder of this section, we outline an architecture that addresses these challenges for the case of a Haskell program extended by plugins also written in Haskell. We do so by running through a simple example.

2.1 Defining an Interface

Many extension languages (e.g., Lisp and Python) lack strong static typing. Then, a plugin interface is characterised 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 a plugin.

In Haskell, we need to ensure 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 5, but have a closer look at the specification of interfaces here.

To avoid arbitrary dependencies of plugin code on application internals, the application programmer defines plugin *interfaces* in extra modules. We collect interface functions in a data structure, called `Interface` in the example. Here it merely contains a single string processing function named `stringProcessor`; a more complex plugin might have more.

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

data Interface = Interface {
       stringProcessor :: String → String }

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

In addition to defining the interface signature, this 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 behaviours which a user may optionally override via plugins.

2.2 Implementing an Interface

We compile the application against `StringProcAPI`. The plugin must also import and implement the interface type declared in that module. Additionally, the application expects the plugin to bind the implementation of the interface to a specific identifier, named `resource` in our example. 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:

```
module StringProcPlugin (resource) where
import StringProcAPI (plugin)

resource = plugin { stringProcessor = reverse }
```

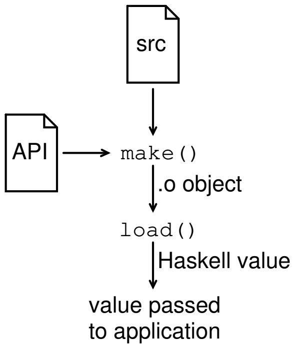


Figure 1. Compiling and loading plugins

2.3 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,
2. loading of plugin(s) and resolving of symbols, and
3. calling of the plugin's functions.

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, as we will see in Section 5.

A simple application using the `StringProcPlugin` 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 source is compiled by a function named `make`, and the resulting plugin object code is loaded by a function named `load`. An application could use the plugin as follows:

```
do
  (obj, _, _) ← make "StringProcPlugin.hs" []
  (mod, rsrc) ← load obj ["."] [] "resource"
```

This code fragment first compiles the plugin source using `make`, yielding the location of the object file in `obj`. It then 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 call `rsrc` here. 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 in Appendix A.

This process of object code generation and loading is illustrated in Figure 1. The two functions `make` and `load` invoke two different subsystems of our plugin infrastructure called the *compilation manager* and *dynamic loader*, respectively. The compilation manager checks whether the plugin source needs recompiling, and if so, calls the Haskell compiler with the appropriate flags. The dynamic loader 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. We will discuss both subsystems in more detail in Sections 6 and 7.

The `stringProcessor` from `rsrc` can simply be applied to a `String` (e.g., `stringProcessor rsrc "Hello World"`) as `rsrc` has type `StringProcAPI.Interface`. 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 behaviour) if its source code has changed:

```
loop mod rsrc = do
  str ← getLine
  (_,isNew,_) ← make "StringProcPlugin.hs" []
  rsrc′ ← if isNew
           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
Compiling StringProcPlugin ... done
Loading package base ... linking ... done
Loading object StringProcAPI
               StringProcPlugin ... done
> abcdEFGH1234
4321HGFEdcba
... Change reverse to Char.toUpper in plugin source
> abcdEFGH1234
Compiling StringProcPlugin ... done
Reloading object StringProcPlugin ... done
ABCDEFGH1234
```

The first string is reversed, whereas the second is converted to upper case.

3 Beyond Plugins

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.

3.1 A Haskell Evaluator

Using 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) [35] or GHCi. A user enters strings of Haskell expressions at a prompt, which plugs immediately evaluates and prints to the screen. It achieves this by dumping each expression (together with a wrapper) into a plugin module, which is then compiled, loaded, executed, and 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 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]) }
```

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

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 1. Haskell interpreter, plugs, example session

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.

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, only a few lines long:

```
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, handles 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.

Plugs is a simple Haskell environment and doesn't provide the full feature set of, for example, GHCi. However, there is no reason why these features cannot be added to plugs if desired. Its simplicity is also a virtue: plugs is easy to modify and extend. For example, we have recently embedded a modified version of plugs in an IRC 'bot', providing a Haskell interpreter for IRC users—which would have been a far more difficult task with GHCi.

3.2 Plugging Haskell Into C and Objective-C

Since Haskell has a standardised Foreign Function Interface (FFI) [3], 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 `make` and `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, Tcl, or Lua, as the Haskell FFI emphasises data marshalling on the Haskell side, which results in less marshalling code written in C for the host application. Marshalling 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 Section 6.3.

In addition, we have complemented the basic `make` and `load` API with foreign functions that enable a host application in C to communicate with Haskell plugins on a more symbolic level. For example, our plugin library implements an `hs_eval` function, resembling a scripting language's `eval` operator. 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();
}
```

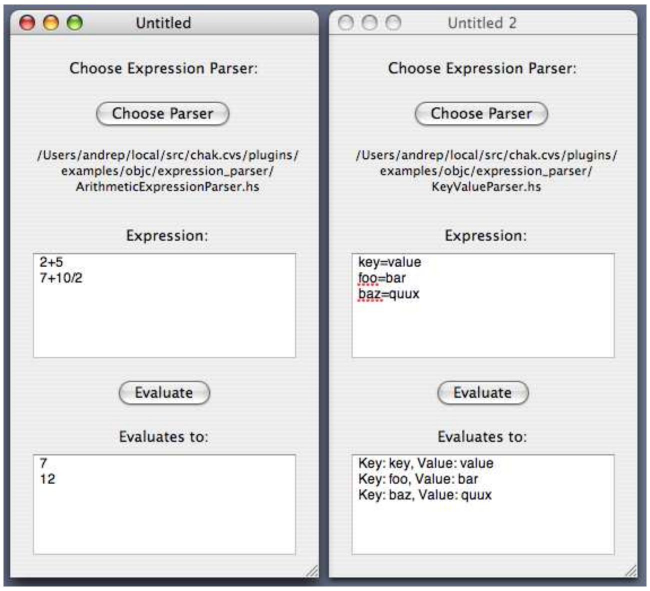
When executed, this C program calls the Haskell runtime system, which arranges for our Haskell plugin library to compile, load and execute the C string passed to `hs_eval_i()`, returning a pointer to the integer result to the caller. Dynamic typing on the Haskell side checks that the compiled Haskell 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.

Any language with a Haskell binding, possibly going via a common interface to C, can take advantage of plugins to dynamically load Haskell code at runtime. As an example, consider an Objective-C program that evaluates expressions entered by the user. While the GUI is implemented in Objective-C, the expression evaluator is a Haskell source plugin that the user can choose and alter while running the application. Figure 2 shows a screenshot of the application running with an arithmetic expression evaluator plugin (left side) and a key/value parser plugin (right side). The entire application and plugin code combined is just over 100 lines.

4 Extension Languages and EDSLs

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).



/Users/andrep/local/src/chak.cvs/plugins/
examples/objc/expression_parser/
ArithmeticExpressionParser.hs

```
2+5
7+10/2
```

```
7
12
```

/Users/andrep/local/src/chak.cvs/plugins/
examples/objc/expression_parser/
KeyValueParser.hs

```
key=value
foo=bar
baz=quux
```

```
Key: key, Value: value
Key: foo, Value: bar
Key: baz, Value: quux
```

Figure 2. Using Haskell plugins to parse expressions in an Objective-C application

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.

Recent work 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 [6, 10, 16, 17, 24, 29, 36]. 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.

4.1 Typed Application 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, our plugin framework offers a *merge* facility 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.

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. 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. The exact merging process and type checking during `make` is described in more detail in Section 7.

Overall, the configuration file is compiled and loaded into the application at start-up, enabling user-defined behaviour 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.

4.2 PanTHeon

PanTHeon [29] is an implementation of Elliott's language *Pan* for continuous image/animation synthesis [10, 11]. It consists of a library implementing the EDSL Pan and a client which is capable of displaying the images and animations.

In-between the library and client are `effect` plugins that, using the EDSL, describe images and animations that the client displays. The Pan EDSL models images as functions from continuous Cartesian co-ordinates to colours; animations are modelled as functions from continuous time to images. Interactive effects can be defined by using a user interface monad, `UI`, which enables the user to attach GUI widgets (such as radio boxes and sliders) to variables that affect the appearance of an effect (e.g., a circle's radius or the amount to stretch an image horizontally).

As a simple example, consider the following plugin whose `effect` declaration implements a blue circle on a white background, and whose radius can be adjusted by a slider widget:

```
circle :: Frac → ImageC
circle r = condR (circleReg r) blueI whiteI

circleReg :: Frac → Image Bool
circleReg r (x,y) = distO (x,y) < r

circleAnim :: Frac → AnimC
circleAnim rad t = circle rad

effect :: UI AnimC
effect =
  let v = makeSlider "Radius" (10,100) 30
  in liftM circleAnim v
```

Without getting into the details of Pan, `circleReg` creates a black and white image of the region covered by a circle with radius `r`. The function `circle` turns this into a blue and white image and `circleAnim` makes the radius variable over time—resulting in an animation. The animation is hooked up to a slider in `effect`, which the PanTHeon client expects to have type `UI AnimC`.

In PanTHeon, as in the configuration example of the previous subsection, error messages during plugin compilation are provided to the user, with the remaining challenge of making them clear for the EDSL user. PanTHeon benefits enormously from the plugins library. From the beginning, one of its design goals was that effects described in it should be interactive. Although this goal is satisfied in part by the `UI` monad, the ability for users to interactively view effects, make minor alterations and load new versions within a single runtime session, is much more convenient than forcing a re-compile of the PanTHeon client every time an effect is changed. In addition multiple effects can be open and displaying simultaneously, implemented by multiple plugins.

4.3 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. Mixfix syntax cannot be defined, for example, nor could one define a `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 *redefine* syntax, even though this may be desirable in a DSL context. 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 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.

5 Type Safety

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.

Unfortunately, GHC's dynamic loader, 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 [15] are not feasible for use by GHC, as GHC loads objects compiled via a standard C compiler. 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 gory low-level work: we first discuss the use of dynamic typing to check plugin values as they are loaded; second, we discuss the use of a full type checker at runtime, providing a form of staged type inference, to overcome limitations of the first technique.

5.1 Dynamically Checked Plugins

Plugins compiled separately from the host application, must import and implement the application's plugin API (see Section 2.2). 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.

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 [1, 4, 22, 37] in its standard libraries; `Dynamic` encapsulates a value with a representation of its type, which can be checked against its usage at runtime. The `Dynamic` type thus enables some type checking to be delayed until runtime. The plugin infrastructure now requires that a plugin wraps its interface in a `Dynamic` value. The host application uses the `fromDynamic` function to coerce (or cast) the `Dynamic` type to interface's expected type, returning an error value if the cast fails.

We thus complement our dynamic linker's `load` function with a `dynload` function: `dynload` performs a conventional `load`, but also unwraps a dynamically typed value and performs a type check. 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 defence 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 valid. 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.

5.2 Dynamic Types and Dynamic Loading

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

```
fromDynamic :: Typeable a ⇒ Dynamic → 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 that a symbol obtained from a separately compiled and loaded plugin has type `Dynamic`. If a plugin provides the correct symbol name, but is bound to a non-`Dynamic` value, `fromDynamic` will simply crash. 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. Designing such a check is on-going work.

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 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.

There is another problem in the standard `Dynamics` library: it requires values which are instances of `Typeable`. 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; e.g., a data type declaration such as

```
data Interface = Interface {
   rsrc :: forall a. Eq a ⇒ a → a → Bool }
```

enables us to wrap polymorphic values such that the data type itself, namely `Interface`, remains `Typeable`, which means that we can use it as a `Dynamic` type.

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 a Haskell interpreter such as plugs, where the generated plugins—expressions entered at the plugs prompt—may be polymorphic.

5.3 Dynamically Checked Plugins Improved

A number of researchers have proposed forms of dynamics that support polymorphic values by some form of runtime type unification [2, 22, 26, 27]. We present an alternative 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 module type checks:

```
module APITypeConstraint where
import qualified API
import qualified Plugin

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

Instead of using any form of dynamic types to check types at runtime, we simply invoke the standard type checker on the above module; this is not unlike the idea of *staged type inference* [31]. 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; we quantify the costs below.

In our plugin library, we generate a temporary module, corresponding to `APITypeConstraint` 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. More precisely, `pdynload` is implemented as follows:

```
pdynload obj incs pkgs ty symbol =
 do tmp ← newTyConstraint ty symbol
    err ← typecheck 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, `typecheck` 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* (`.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 export list. Hence, staged type inference for the restricted case of plugins is more easily implemented than is generally the case.

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 expression `Plugin.resource :: API.Interface` when checking the `APITypeConstraint` module with the following error message, indicating that the `Num` class restriction on the plugin value is not compatible with the API type:

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

Table 1. Cost of checked runtime loading, relative to an unchecked load

```
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.

5.3.1 Type Checking Performance

The question that remains is whether invoking the full Haskell type checker—in our case, actually invoking all of GHC—is sufficiently fast. In fact, the overhead is less than might be imagined, as no code generation is required. Table 1 shows the comparative performance of the various load strategies. We use the runtime of loading 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 version 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 amortised over the running time of the application. Second, there is plenty of room for optimisation if performance is a problem. For example, Shields, Sheard and Peyton Jones [31] discuss several mechanisms for limiting the amount of type inference that needs to be performed at runtime, some of which might be adapted 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.

5.4 Type Safety of 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 [5] will give full type safety with dynamically loadable objects. 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 has also 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.

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

Table 2. Type safety of plugins using different approaches

5.5 Type Safety 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 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 2 summarises the different levels of type safety achieved when either only *object code* is available, or when *source code* is compiled at load time.

6 The Dynamic Loader

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. The dynamic loader also requires a high-level mechanism for resolving dependencies between plugins and libraries, as any dependent libraries or modules must be loaded prior to loading the plugin itself: the runtime system cannot do this. Chasing dependencies and finding libraries makes up a significant proportion of the dynamic loader's code.

6.1 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 [30] 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, much as is required by GHCi.

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: it solves it by storing module and library dependency information for an object in the corresponding interface (`.hi`) file. As plugins are compiled with GHC, this `hi` 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` is 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 file, `package.conf`, which we have access to. To parse this information, we have implemented a `package.conf` package file parser, 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 a mutable state in the dynamic loader to keep track of libraries and modules that have been loaded, so that we can skip loading components when they are needed repeatedly. We also use this state to store a map from the libraries' canonical names to their their full paths. Internally, the state is locked using `MVars`, so that concurrent access is synchronised and safe.

6.2 A Standalone Typechecker

The dynamic type checking mechanisms provided by `pdynload` requires 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 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.

6.3 Proxy Objects

One usually thinks of a dynamic loader as a tool to inject new code into the program's address space at runtime. However, the dynamic loader can also be thought of as an intermediary for the plugin: instead of directly exposing the plugin's functions to the host application, the dynamic loader can mediate communication between the host application and the plugin. This enables a complete separation of a plugin's implementation and location from the host application. For example, the dynamic loader can search a user-extensible plugin repository for a matching plugin that satisfies the interface needed by the application, or it transparently marshal data provided by a Haskell ESDL to a host application written in C. This gives the dynamic loader the same powers and features as *surrogate objects* or CORBA's *interceptors*, and can fulfil the *proxy*, *mediator*, and *adapter* design patterns [12].

This feature is particularly useful in Haskell since the dynamic loader maintains its own local state, separate from the application, which contains information about loaded plugins and their dependencies. If a plugin requires local state, the dynamic loader can supply this state, instead of the host application. This saves the host application from carrying around a plugin's state in its data structures and from requiring additional code simply to support this extraneous state (which only some plugins may make use of).

7 The Compilation Manager

Next to the dynamic loader, the compilation manager is the second major component of the plugin library infrastructure. It arranges for source code to be conditionally compiled, by determining whether it has changed since its previous 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.

7.1 Invoking the Compiler

Our plugin library's `make` function 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.

7.2 Merging Syntax

One other feature that is governed by the compilation manager is the addition of syntax to the plugin's source code, to facilitate simplified EDSL plugins. This is achieved by having the application author write a *stub* Haskell source file that contains any declarations the application requires of the 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.

The plugin library uses a Haskell parser to construct the abstract syntax trees of the two sources. The type information of the plugin tree is then erased, and type information, module, and import/export declarations from the stub tree are merged into the AST 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 existing Haskell parser library, `Language.Haskell`, is limited to parsing Haskell 98 with only a few extensions. If an EDSL file is to have syntax merged in, it is in turn limited to Haskell 98. In the future, we intend to remove this restriction by linking the GHC parser into our library, as a full Haskell parser.

8 Related Work

The *dynamic loader* described in this paper 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 [3]. 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 [30] project, whose full goals have not yet been realised. In a sense, we are adding some of the missing elements of the HEP.

Many other functional programming languages more or less support dynamic typing and dynamic loading. In particular, *Clean* [28] has good support for polymorphic dynamic types. Clean's dynamics have been used in a variety of scenarios, including type-safe distributed communication [14] and an interactive Clean interpreter [34], 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 [26, 27]. 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. However, it would be wise to research how Clean's dynamic types and dynamic loading work at a lower-level, to overcome some of the dynamics implementation problems discussed in Section 5.2.

Adding dynamic typing and dynamic loading to the ML family of languages has also received a significant amount of attention [9, 13, 18, 22]. 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. Objective Caml provides a `dynlink` library [21], 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.

The Java Virtual Machine (JVM) supports dynamic loading via *class loaders* [23]. 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. Due to the popularity of Java, the JVM is likely to be the most widely used implementation of a type-safe dynamic loader.

9 Future Work

This work has highlighted the need for a safer form of dynamic typing for Haskell; in particular, we need to be able to generate an exception if the value retrieved from a loaded object is not a `Dynamic` type. Currently, this value will be unsafely coerced to `Dynamic`, resulting in a crash. Unless we can handle this situation gracefully, using only dynamic types with object-code plugins remains as unsafe as not using them at all.

We envisage at least two approaches to solving this problem: (1) introduce a simple type system into the object file that can be understood by the runtime system or (2) provide a checksum of the type representation stored in the dynamic value. A simple type system for alternative (1) would only distinguish between two types, namely whether a value is of type `Dynamic` or of any other type. This information is sufficient to prevent an unsafe coercion of values that are not of type `Dynamic`. The checksum of alternative (2) can be used in a similar way by recomputing the checksum before inspecting the value itself. This approach would make the use of dynamic types as safe as the checksum that is used, which is an acceptable amount of safety for many applications.

The use of *existential types* [20] is supported by many existing Haskell implementations, including GHC, hugs, and nhc98. Existential types are commonly associated with dynamic types [7, 8], and have been previously used to provide *dynamic dispatch* in Haskell [19]. Indeed, since plugins are often used to dynamically dispatch functions, more research is needed on possible interactions between existential types and plugins.

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 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. Eventually, we hope to produce 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.

One caveat of using `pdynload` to type check object-code plugins (discussed in Section 5.3), is that the plugin's interface (`.hi`) 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. To this end, we plan to exploit GHC's recent support for making components of the compiler into a library.

10 Conclusion

This paper introduced a general framework for plugins in Haskell. We discussed how such a framework enables Haskell to be used as a strongly, statically typed *extension language*, which can be used to modify or add to an application's functionality at runtime without having to modify the application's source code.

We further discussed the relationship between EDSLs and extension languages, and we advocate the use of Haskell EDSLs as dynamic scripting languages, which is possible due to our plugin infrastructure. Languages with FFI bindings to Haskell immediately inherit these benefits, enabling them to use Haskell as an extension language, rather than designing their own ad-hoc language.

The plugin infrastructure library is currently used to extend PanTHeon, an image manipulation framework, which uses the Pan EDSL as its extension language. We described an interactive Haskell environment that evaluates Haskell code as strings, by compiling and loading such strings as plugins. We then showed how runtime compilation can be used via the FFI, to enable the use of dynamically generated Haskell plugins from within C and Objective-C programs.

The source code for the examples in the paper as well as the plugin library (`hs-plugins`) is available to view and download at:

`http://www.cse.unsw.edu.au/~dons/hs-plugins/paper`

Acknowledgments. We are grateful to Kai Engelhardt, Mark Wotton, Simon Winwood and the rest of the IRC channel `slashnet.org/#maya` for feedback on drafts. Moreover, we thank the anonymous reviewers for their constructive criticism.

11 References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):237–268, 1991.

[2] M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, 1995.

[3] M. M. T. Chakravarty et al. *A Primitive Foreign Function Interface*, 2004. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`.

[4] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 90–104. ACM Press, 2002.

[5] C. Colby, K. Crary, R. Harper, P. Lee, and F. Pfenning. Automated techniques for provably safe mobile code. *Theor. Comput. Sci.*, 290(2):1175–1199, 2003.

[6] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, Aug. 2003. ACM Press.

[7] D. Duggan. Dynamic types have existential type. Technical report, University of Waterloo, December 1994.

[8] D. Duggan. Dynamic typing for distributed programming in polymorphic languages. *Transactions on Programming Languages and Systems*, Jan. 1999.

[9] D. Duggan. Type-safe linking with recursive DLLs and shared libraries. *Transactions on Programming Languages and Systems*, Nov. 2002.

[10] C. Elliott. Functional Image Synthesis. In *Proceedings Bridges 2001, Mathematical Connections in Art, Music, and Science*, 2001.

[11] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, May 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical report, The University of Edinburgh, 1997.

[14] H. Hegedüs and Z. Horváth. Distributed computing based on clean dynamics. In *Proceedings of the 6th International Conference on Applied Informatics*, January 2004.

[15] M. Hicks, S. Weirich, and K. Crary. Safe and flexible dynamic linking of native code. In *Proceedings of the 3rd Workshop on Types in Compilation*, pages 147–176, 2000.

[16] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[17] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[18] L. 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.

[19] K. Läufer. Combining type classes and existential types. In *Proceedings of the Latin American Informatic Conference (PANEL)*, Sept. 1994.

[20] K. Läufer. Type classes with existential types. *Journal of Functional Programming*, May 1996.

[21] X. Leroy, D. Doligez, J. Garrigue, D. Rmy, and J. Vouillon. The Objective Caml system 3.08. `http://ropas.kaist.ac.kr/caml/ocaml/`, July 2004.

[22] X. Leroy and M. 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.

[23] S. Liang and G. 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*, pages 36–44. ACM Press, 1998.

[24] H. Nilsson. Functional automatic differentiation with dirac impulses. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 153–164, Uppsala, Sweden, Aug. 2003. ACM Press.

[25] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, March 1998.

[26] M. Pil. First class file I/O. In *Implementation of Functional Languages*, pages 233–246, 1996.

[27] M. Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages*, pages 169–185, 1998.

[28] R. Plasmeijer and M. van Eekelen. Concurrent clean language report version 2.1. November 2002.

[29] S. Seefried, M. M. T. Chakravarty, and G. 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. To appear.

[30] J. Seward, S. Marlow, A. Gill, S. Finne, and S. P. Jones. Architecture of the Haskell Execution Platform (HEP). `http://www.haskell.org/ghc/docs/papers/`, 1999.

[31] M. Shields, T. Sheard, and S. Peyton Jones. Dynamic typing as staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–302. ACM Press, 1998.

[32] R. 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.

[33] The GHC Team. The Glasgow Haskell Compiler (GHC). `http://haskell.org/ghc`.

[34] A. van Weelden and R. 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.

[35] M. Wallace. hi — hmake interactive — compiler or interpreter? In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 339–345, 2000.

[36] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.

[37] S. Weirich. Type-safe cast: (functional pearl). In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 58–67. ACM Press, 2000.

A The Plugin Library Interface

A.1 The Dynamic Loader

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

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

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

$$\mathit{unload} \quad :: \mathit{Module} \rightarrow \mathit{IO}\ ()$$

$$\mathit{reload} \quad :: \mathit{Module} \rightarrow \mathit{Symbol} \rightarrow \mathit{IO}\ a$$

`load` takes the path to an object file as its first argument, along with a list of paths to search for any dependent modules to load (such as the application's API). The third argument is an optional list of Haskell library package information, in the standard `package.conf` format, so that dependent libraries can be found. The fourth argument is the symbol to search for in the loaded object. `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 to the application.

`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 typeing 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, returning `Nothing` otherwise.

`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 string representation of the required type. This is used to generate a Haskell expression asserting that the plugin's value has this type. The expression is then passed to GHC's type checker. If the check succeeds, then the plugin is safe and we return the resulting value, otherwise `Nothing` is returned.

`unload` takes a module and unloads the corresponding object from the address space. `reload` takes a module and a symbol. It unloads the module, recompiles its source, and then reloads the resulting object, returning the new code associated with the symbol. It is similar to performing `unload` and `make`, followed by a `load`.

A.2 The compilation manager

$$\begin{aligned}
&\mathit{make} && :: \mathit{FilePath}\\
&&& \rightarrow [\mathit{Arg}]\\
&&& \rightarrow \mathit{IO}\ (\mathit{FilePath}, \mathit{WasBuilt}, \mathit{Errors})
\end{aligned}$$

$$\begin{aligned}
&\mathit{makeWith} && :: \mathit{FilePath}\\
&&& \rightarrow \mathit{FilePath}\\
&&& \rightarrow [\mathit{Arg}]\\
&&& \rightarrow \mathit{IO}\ (\mathit{FilePath}, \mathit{WasBuilt}, \mathit{Errors})
\end{aligned}$$

`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 (such as the application's API) to link against. `make` returns a triple containing the path to the newly created object file, a boolean flag indicating whether the object file had to be rebuilt, and a list of errors produced by the compiler.

`makeWith` takes the path to a source file, the path to a stub file, and a list of directories to search in as its arguments. It returns the same values as `make`. `makeWith` merges the abstract syntax of its two file arguments to create a third source file, which is then compiled using `make`.

A.3 Foreign Language Bindings

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

`hs_eval_*` are foreign language interfaces to `eval`, which is a wrapper around the evaluation mechanism used in the plugs system. It is available to foreign languages who include `"RunHaskell.h"`. The foreign bindings use dynamic types to check that the compiled value has the type expected by the caller, yielding a null pointer otherwise.