Principled Dynamic Code Improvement

John Peterson   Paul Hudak   Gary Shu Ling

Yale University
Department of Computer Science
New Haven, CT 06520
{peterson-john,hudak,ling}@cs.yale.edu

Research Report YALEU/DCS/RR-1135

July 15, 1997

Abstract

A method for dynamically updating running code is described. The method is "principled" in that old functionality is not destroyed. Rather, new functionality is added which can selectively and dynamically change the overall behavior of the system, yet all formal properties of the original system are guaranteed to still hold. Higher-order functions are the key to making this work, and an implementation in Hugs, a Haskell byte-code interpreter, is described.

1 Introduction

An increasingly common requirement of real-time software systems is the ability to update code “on the fly;” that is, while the system is still running. For whatever reason, such systems cannot afford to be shut down while the changes take place, and a redundant system design is typically impractical. The need for such *dynamic code improvements* arises not only because of conventional software bugs, but because of planned upgrades to the system for enhanced or altered functionality that could not be predicted when the system was originally built.

Most programming languages and environments do not support dynamic code improvement. When viewed from the level of object code, an obvious requirement is a flexible linking mechanism that allows modules to be swapped in dynamically. But additionally there needs to be some kind of synchronization mechanism to allow switching from the old functionality to the new, a mechanism to recover from linking errors, a means to reclaim unused code, and possibly a persistent store.

Although the low-level details of linking and memory management are important, they are for the most part straightforward, and several schemes have been successfully used. On the other hand, solutions to the synchronization problem are less clear. Indeed, a problem with existing mechanisms for dynamic code improvement is the lack of any principles by which one can reason about the *dynamics of change*. What does it mean for a program to change its behavior arbitrarily? What happens to formal properties that one might have proven about a system in the presence of such potentially radical changes? How can new properties be established, and how might they account for future changes?

In this paper we outline a simple strategy for *principled* dynamic code improvement that solves most of these problems. Our strategy takes full advantage of the functionality of HOT (“higher-order and typed”) programming languages, in that we rely critically on higher-order functions to capture the dynamics of code improvement, and on types to capture conventional compile-time safety. We have also implemented our ideas in the Hugs [6] implementation of Haskell [5].

2 Principled Dynamic Code Improvement

Our basic idea is rather simple: instead of implicitly switching execution from one code block to another, we will do so *explicitly* in the source language, using higher-order functions to capture the components as “first-class citizens.” To make this work all we need is a way to explicitly load and then invoke the new functionality. We will describe how this is done in our Hugs implementation of Haskell,[1] from which it should be clear how it might be done in some other HOT language.

Haskell has a (rather conventional) module system which permits the usual control over namespaces and facilitates separate compilation. Cross-module references need to be sorted out at compile-time in the usual way: assuming that the module references form a DAG, the

[1]Hugs development is being done in collaboration with the University of Nottingham.

modules may be loaded in any order that is a topological sort of the DAG. Hugs' "import chasing" feature in fact does this automatically for the user.

The thing to note is that, once a set of modules is loaded into a running system, the DAG structure may be extended at the leaves without disturbing any of the previous references. In other words, more modules could be loaded into the running system without the need for re-directing any existing references. Indeed, Hugs permits this option manually for the user. What is new is that we now make this option available to the programmer by adding to Haskell the following monadic action:

$$\mathit{loadAndGo} \;::\; \mathit{FName} \to \mathit{IO}\;()$$

where *FName* is the filename being loaded (with the normal Hugs convention that filename and module name match). Once loaded, the expression *main* is evaluated in the new module.

In essence, a call *loadAndGo* "M" transfers the execution context to *main* in the module M. Since M is at the bottom of the module hierarchy, evaluation of *main* can reference any value that has been imported from any pre-existing module. It can, in effect, transfer control again to anywhere the programmer desires.

The only difficulty with this scheme is the absence of any kind of *persistence*. In particular, there will likely be values that we would like to save and then restore during the transition between old and new functionality. In a language with side effects, a global variable could be used for this purpose. Although standard Haskell has no global variables, most implementations, including Hugs, support *monadic references*. Using these primitives we are able to implement the following two functions:

```
saveState    :: SystemState → IO ()
restoreState :: IO SystemState
```

for any application-specific type *SystemState*.

2.1 A Simple Example

Suppose there is a module *Top* which implements a loop *loop* that depends on some state s (an arbitrary value). Further suppose that somewhere in this loop the function $f :: T1 \to T2$ is called whose functionality is intended to change dynamically when predicate p is true. The loop, then, needs to be parameterized not just with s, but with f too. Suppose now that somewhere in the loop the following monadic code is executed, shown here within the structure of the module *Top*:

```
module Top (loop) where

loop     :: (T1 → T2) → SystemState → IO ()
loop f s  =  ...
             if p then do saveState (f, s′)
                          loadAndGo "M"
                  else loop f s′
```

where M is the name of the new module. If the predicate is false, the loop is executed normally with some modified state s'. But if it is true, f and s' are saved, module M is loaded, and control is passed to the evaluation of *main*.

Module M might look something like this:

```
module M where
import Top

newf  ::  T1 → T2
newf  =   ...

main  =   do (oldf, olds) ← restoreState
             loop newf olds
```

Note that the module *Top* is imported, thus allowing access to the function *loop*. Evaluation of *main* results in restoration of the old state, after which *loop* is called with same old state but with the newly loaded function *newf*. In general, the newly loaded module can import any of the old ones (including ones loaded since compile-time), do a *restoreState*, and then call a re-entry point with arbitrary functionality.

Clearly in order to use our technique one must be able to anticipate in advance exactly which pieces of code might be updated. In fact, this is true of all existing methods that we know of, which is not surprising given the synchronization problem: the extent of the dynamics needs to be delimited somewhere. In the worst case, there might be a single function delimiting the functionality of the entire program, thus allowing a complete change in functionality.

2.2 Errors

It is important to realize that *loadAndGo* cannot redefine any existing functions. If this were attempted, the result would be the same as if the module were loaded at compile time: a name-clash error would occur. The only difference is that now the error happens at run-time.

Because of this possiblilty of run-time error, of course, care must be taken to provide a means to recover from it. This is easily done in Haskell using its monadic *catch* operator. As an example, here is a rewrite of the conditional in the loop given earlier, that restores the old functionality if a loading error occurs:

```
if p then do saveState (f, s')
             catch (loadAndGo "M") (\e → loop f s')
     else loop f s'
```

This version ignores the specifics of the error e and simply loops with the old functionality, but it is also possible to inspect e and react differently depending on the type of error that has occurred (the important cases being "file not found" and "compilation error").

2.3 Reasoning About Change

Since the new module cannot redefine any existing bindings, any formal properties of the original system must still hold in the new. For instance, in the example above, any properties involving f, *loop*, etc. must still hold. Indeed, it may be that the old functionality (for example, the function f) is still used in some other parts of the system, and we certainly don't want those parts to become suddenly inoperative (an example of this arises later when we consider concurrency).

On the other hand, if the operative aspect of the system is the combination of *loop* and its parameterized function f, then presumably there are properties about the combination of *loop* and f that must be reestablished for *loop* and *newf*. Nevertheless, this kind of reasoning is perfectly sound, and one can hardly expect to do less work than this.

To facilitate equational reasoning, the following program is equivalent to the one given earlier, assuming that a run-time error did not occur as a consequence of loading the new module:

```
module Top (loop) where
import qualified M (main)

loop     :: (T1 → T2) → SystemState → IO ()
loop f s  = ...
            if p then do saveState (f, s')
                         M.main
                 else loop f s'
```

In other words, the new system is equivalent to the system that would have resulted from loading the new module at the very beginning. Note how *main* from module M is explicitly imported and evaluated in the appropriate context.

This program equivalance permits us to use conventional equational reasoning when proving properties about the new system. For example, no dynamic lookup of names in a symbol table is required. We should also point out that Haskell's type system ensures that the additional functionality is type-safe: that is, type errors cannot occur at run-time. This is an important property when dealing with dynamic change, and is notably better than, for example, the type casting that results when using dynamic types in a language such as Java.

2.4 Concurrency

Sometimes it is desirable to swap in new functionality *gradually* instead of all at once as we have described above. For example, a server in a client-server architecture might communicate with multiple clients whose requests overlap in time. When the server functionality is upgraded, it may be desirable for the old server to finish any requests it is currently processing, while at the same time the new server picks up any new requests.

Our approach to dynamic code improvement deals nicely with this situation. Since the new functionality does not destroy any of the old, both can exist simultaneously; all that is needed is some form of concurrency. Concurrent Haskell (supported by Hugs), for example, allows one to fork off concurrent threads of computation, each independently performing IO actions:

```
do fork p1
   p2
```

Here $p1$ is forked while the current thread continues with $p2$. Communication between process is accomplished in one of a number of ways, including CSP-like channels.

To solve the server-client scenario, the scheduler for the server might fork off an instance of a "server function" for each request that arrived. Each of these instances would run in parallel until it completes its task (which might take seconds, hours, or weeks). Initially the scheduler would use whatever server function was originally installed, but if a new server function were dynamically loaded, it would start using that one, and so forth. So at any one time, there may be instances of many different server functions running. But, as the olds ones die off, they will be automatically purged from the system by the garbage collector.

2.5 Code Reclamation

As implied above, data structures and function closures created from the old functionality—say a closure for the function f in the earlier example—would naturally be reclaimed by the garbage collector soon after they are not needed any more. However, as new functionality is added to the system, the amount of *static code* also increases, possibly eventually causing memory space problems. The solution to this is to extend the garbage collection mechanism to include the reclamation of unused code. Hugs does not currently do this, but there are no technical problems preventing it.

2.6 Capitalizing on Type Classes

Another common approach to dynamic code improvement is based on *object oriented programming*. In this style, new objects, derived from pre-defined classes, are dynamicly added to a running system. The pre-defined classes serve as an interface by which dynamicly loaded objects interact with existing code.

The Haskell type class system, together with the ideas introduced earlier, can be used to achieve the same effect. Consider a program similar to the previous example in which a loop is defined in terms of a value which is a member of a particular class, instead of having a fixed type:

```
loop        ::  C a ⇒ a → SystemState → IO ()
loop obj s  =   ...
```

The code in ... uses methods in the class C to interact with the value *obj*. Now a dynamicly loaded file can define a new data type as an instance of the class C and pass new values of this type into the loop, thus allowing the set of methods associated with the class to be redefined by the new object. For example:

```
module M where

data NewType  =  ...

instance C NewType where
  method1  =  ...
  method2  =  ...

v            ::  NewType
v            =   ...

main         =   do olds = restoreState
                    loop v olds
```

This, in effect, packages up a group of related functions into a single object which is much easier to pass through the program. In fact, the methods associated with Haskell classes are passed implicitly rather than explicitly, making the use of dynamicly loaded class instances less intrusive on the program than passing changable functions explicitly through the program.[2]

2.7 Doing Without Persistence

It is possible to do away with the need for persistence by calling *main* directly with the desired state. Although we have not implemented this idea, it presents only a modest increase in implementation complexity, and is an arguably cleaner design.

Recall that the type of *loadAndGo* is *FName* → *IO* (), and *main* in the newly added module must have type *IO* (). Instead, we could use the following type for *loadAndGo*:

```
loadAndGo  ::  FName → a → IO ()
```

The idea is that, instead of saving and restoring the state explicitly, we can design *loadAndGo* such that it passes its second argument to *main* directly. If the type of that argument is T, then *main* would be expected to have type T → *IO* (). The previous example could then be rewritten as:

```
module Top (loop) where

loop      ::  (T1 → T2) → SystemState → IO ()
loop f s  =   ...
              if p then loadAndGo "M" (f, s′)
                   else loop f s′
```

[2]In reality, this method for dynamic code improvement requires *existential types* to implement properly, which does not exist in standard Haskell but is supported by several Haskell implementations.

```
module M where
import Top

newf             ::  T1 → T2
newf             =   ...

main             ::  (T1 → T2, SystemState) → IO ()
main (oldf, olds) =  loop newf olds
```

Note the type of *main*: it takes as argument the state that was previously being saved and restored.

To implement this idea simply requires that Hugs be informed of the instantiated type of the second argument to *loadAndGo* at run-time, so that it can perform the proper type-check on *main*.

3 Related Work

One of the most widely-cited examples of dynamic code improvement is that done for telephone switching systems, where clearly any interruption of service cannot be tolerated. Unfortunately, most of this work has been done in commercial environments, and we do not have any documentation describing it.

There is one notable exception to this, however, based on the functional language *Erlang* [2, 7] developed and used extensively by Ericsson in its telephone switching systems. That Erlang is a functional language is notable in its own right, but its support for dynamic code improvement is less principled than ours: existing modules can actually be edited and recompiled while the system is running. Although obviously flexible, such an approach presents dangerous opportunities for error, and makes the task of program verification extremely difficult. The approach also seems to rely inherently on the tail-recursive nature of top-level code.

Of course, Lisp systems have supported a form of dynamic code improvement for many years: new files may be dynmically loaded under programmer control at any time. Unfortunately, the method is highly unprincipled. Not only can any function be arbitrarily redefined, for example, but not every implementation supports this in the same way, some behave differently when compiled vs. interpreted, and the Scheme standard [3] is silent on the issue. Problems arise especially when mutually recursive functions are redefined. In general this "imperative" style of reloading code, shared by many other systems such as the ones discussed below, results in an unprincipled solution to dynamic code improvement.

Appel [1] discusses support for dynamic code improvement in SML of New Jersey [4], citing many of the issues that we have reported here. SML/NJ allows any module in an acyclic module system to be replaced, and uses indirection to avoid costly relinking (much like a conventional Lisp system). It also performs garbage collection of unused code.

One might also consider OLE, COM, Corba, and other approaches to component-based interoperability as viable mechanisms for implementing dynamic code improvement. Again, however, the result is far from disciplined. One does not generally even know if the same object will be invoked every time that it is called. The fact is, these component-based technologies are really solving a different problem, and are thus not the ideal solution for dynamic code improvement.

4 Acknowledgement

Thanks to Alastair Reid for help in implementing these ideas in Hugs. We also wish to thank our sponsors, Darpa and NSF, who have supported this research through Rome Labs contract number F30602-96-2-0232 and NSF grant number CCR-9633390, respectively.

References

[1] Andrew W. Appel. Hot sliding in standard ml, December 1994. unpublished manuscript.

[2] J.L Armstrong and S.R. Virding. Erlang – an experimental telephony programming language, June 1990.

[3] Clinger, W. et al. The revised revised report on scheme, or an uncommon lisp. AI Memo 848, Massachusetts Institute of Technology, August 1985.

[4] Robert Harper, Robin Milner, and Mads Tofte. The definition of standard ml version 2. Technical Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, Department of Computer Science - University of Edinburgh, August 1988.

[5] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[6] Mark P. Jones and John C. Peterson. Hugs 1.4 User Manual. Research Report YALEU/DCS/RR-1123, Yale University, Department of Computer Science, 1997.

[7] M. Persson, K. Odling, and D. Eriksson. A switching software architecture prototype using real time declarative language, 1993.