Formalizing Dynamic Software Updating

Gavin Bierman[†] Michael Hicks[‡] Peter Sewell[†*] Gareth Stoyle[†]

[†]University of Cambridge
{First.Last}@cl.cam.ac.uk

[‡]University of Maryland, College Park
mwh@cs.umd.edu

Abstract

Dynamic software updating (DSU) is a process by which a running program can be updated with new code and data without interrupting its execution. DSU is critical for systems such as air-traffic control systems, financial transaction processors, and networks, which must provide constant service but nonetheless be updated to fix bugs and add new features.

While DSU is widely used in practice, and a number of language-based approaches have been implemented, there is little general understanding of how DSU is best used and/or achieved. A major open issue is how to, in a general sense, write programs that can be updated *safely* in nearly arbitrary ways. While some informal understanding has been achieved, we believe a formal, mathematical approach should be developed to set a firm foundation for both users and implementors of DSU technology.

In this paper, provide a preliminary step in this direction by presenting the BLAH calculus. This calculus provides two basic mechanisms for updating programs: an **update** primitive that allows bindings in the program to be changed safely, and way for the programmer to control which parts of the program will notice this change. This calculus was inspired by, and generalizes to an extent, the DSU technology present in the language Erlang. In addition to presenting the calculus, we present ways in which we it can be used to model existing updating implementations, and postulate how it can be extended prove safety properties of interest.

1. Introduction

Rough intro:

Many systems provide some form of run-time modification of system functionality. Little understanding of questions of usage: how to build updateable systems most effectively? How to know when it's safe to perform an update, so that the system smoothly transitions from its old version to the new one? Also, how should updating itself be implemented? Is there a common mechanism that underlies updating in different languages, whether functional, object-oriented, or imperative? What language features complicate or simplify updating?

We believe that the best way to study these questions is via a formal system that models updating at its essence. Unfortunately, there is little formal understanding of update at this time. There are quite a number of updating implementations (litany here, sure to include Erlang), and many ideas as to when updating should be considered safe, but most are not rigorously defined (Gupta, Bloom, Lee, Hicks, others?). There are few formalized programming languages for modeling updating or its aspects—Gupta, Dynamic ML and Duggan (others?). Of these, Gupta is quite high-level, and Dynamic ML and Duggan lack generality, so none provides sufficient answer our queries.

Therefore, we believe that a simple formal system should be established with the primary goals of understanding the underlying foundations of update, for the purpose of understanding how to best build reliable updateable programs.

2. Related Work

While there have been a variety of implementations of DSU (XXX cite), comparatively little work has been done in the two areas we are interested in: analyses as to how to update programs safely; and formal, language-based models of DSU. Here we briefly overview past work in these areas.

2.1 Updating Programs Safely

An important question for any dynamic update is whether that update is *valid*. Intuitively, we are interested in the question of whether a change to a system's code, realized dynamically, will properly transform the system to reflect the new code base. Gupta *et. al* developed a formal framework proved that the problem of update validity is, in general, undecidable. [6, 7]. In their model, a running program P is a pair (Π, s), where Π is the program code and s is the program state, encapsulating the notion of the stack, heap, and machine registers. An *update* to P is a pair (Π', S), where Π' is the new program code, and S is a state transformer function that maps the old state to a new state. Applying the update yields a new program (Π', s') where $s' = S(s)$. An update is *valid* if and only if the new program's state s' eventually becomes *reachable*. Reachability is defined as follows. A state s, relative to code Π, is *reachable* if and only if a program (Π, s_{Π_0}), where s_{Π_0} is a legal initial state,

*Corresponding Author. Computer Laboratory, University of Cambridge, JJ Thomson Avenue, Cambridge, CB3 0FD, UK. Fax: +44 1223 334678.

can evaluate to (Π, s) at some time for some inputs. Gupta *et al.* show that, in general, determining that a change is valid is undecidable by relating to the halting problem.

This means that any analysis proving validity must be conservative. Gupta *et. al* developed an analysis that compares the old and new versions of C code (but not including functions, stack allocation, or heap allocation) and identifies, based on a syntactic analysis, program points that would preserve update validity. This analysis is quite conservative, and can only handle restructurings of the same algorithm, not changes to program functionality.

Lee [9] describes a way to decompose a valid update into a set of smaller valid updates. A directed graph is constructed such that each node in the graph represents a function to be replaced, and an edge from f to g implies that g *should be updated before or with* f. The strongly connected components of the graph then represent functions that must be updated together. Lee does not formalize why one procedure should be updated before another; in some cases this is easy to determine (*e.g.* if the types of functions change), but in others it is not straightforward. Furthermore, a valid update must be known before it can be deconstructed, but no guidance is provided in finding such an update. Bloom *et al* develop a similar, but more complicated, model for Argus [2, 3]. XXX say more here

A number of researchers have observed that dynamic updates can become invalid if they are applied at an inopportune time. Assuming that an update can become available at any time, rather than perform the update at that moment, the update can be delayed until certain conditions are satisfied. For example, in Lee's DYMOS [9], the programmer specifies *when-conditions* along with the patches to update as in

```
update P, Q when P, M, S idle
```

This specifies that procedures `P` and `Q` should be updated only when procedures `P`, `M`, and `S` do not have activations in any thread stack. As a degenerate version of this idea, many systems simply impose the restriction that updates may only occur to inactive code (e.g. [5]). However, in none of these systems is there any well developed evidence as to what conditions are needed to guarantee validity.

An alternative to dynamically specified conditions are statically guaranteed ones. In particular, we have proposed in prior work [8] (XXX also cite the rebinding paper, maybe PLDI paper too) that programs should designate their own update points, perhaps indirectly by calling an updating mechanism directly, such as a class loader or dynamic linker. This trades flexibility for predictability, perhaps by formal analysis, and forms part of the BLAH calculus presented in Section 3.

XXX Dmetriev, and other USE02 papers ...

2.2 Language-based Formal Systems

XXX need to shorten this stuff a bunch

As mentioned, Gupta defines a language-inspecific model of computation that has an abstract notion of program code and state, and shows update validity is undecidable in general. For this reason, language-specific models are needed to show what updating properties are tractable. We briefly review related formalisms in this area.

Dynamic ML [5] is a proposed implementation of ML with a formalized abstract machine [10] that enables replacement of modules at runtime. User-defined types in replaced modules can be changed so long as they are *abstract*, meaning their representation is hidden from users. Thus a dynamic change of a module's type implementation need not require a dynamic change of that module's users. To ensure soundness, existing instances of the abstract type will be converted to the new representation during a garbage collection phase at update-time. Different versions of modules may not coexist, and a module must be inactive (e.g. not on the runtime stack) before it can be replaced. The abstract machine is used to define how replacement takes place via garbage collection, and to show that evaluation in the context of replacement is sound.[1]

Duggan [4] defines a formal language that relaxes two of Dynamic ML's restrictions with one change: module types may be converted lazily during program execution, rather than at once during garbage collection. As a result, different versions of a type/module may coexist during program execution, and must be convertible from the old to new version and *vice versa*. A novel type-system is presented and type soundness is proved.

While past work on formalization is instructive, prior formalizations are either too language- or implementation-specific to draw broad conclusions. Dynamic ML's abstract machine is intimately tied to SML, and it fixes the implementation strategy as a form of garbage collection for changed types. Duggan's framework is less tied to a particular language, but otherwise suffers the same limitations. Both calculi lack machinery for understanding when and how code can and should be updated, focusing only on the problem of converting instances of changed types following an update.

XXX more criticisms? Are these bogus? what are we really trying to say?

3. The BLAH calculus

XXX Introduction to calculus: not quite sure it says what I want it to say yet .... XXX In this section we introduce the BLAH calculus as a simple formal model of dynamic update. The calculus is roughly equivalent to Erlang [1] with respect to its updating power and the nature of the update mechanism. However, this is where the resemblance ends as our language is based on the simply typed lambda calculus and Erlang is dynamically typed. It might be argued that the static type system we employ makes our system too rigid in what is essentially a dynamic world – we are performing *dynamic* update. However dynamic update is often used in a mission critical environment, where the added safety offered by a static type system outweighs the convenience introduced by dynamic types. Add to this that the static type system presented here can easily be made more flexible by introducing parametric polymorphism and subtyping, and one has a strong argument for static typing in this dynamic context.

3.1 The Formalism

Figure 1 and 2 present the syntax and dynamics of the

[1]Interestingly, I could not locate the restriction in the formalization in Walton's thesis that module's that are active may not be updated. The end of Dynamic Replacement chapter indicates they don't consider how updating is initiated, which is fine, but it seems like this is important in order to show soundness. That is, "appropriate update points" affect their proof. Perhaps we can contact Chris about this?

GPS:This is unsound at the moment

Integers	n		
Identifiers	x, y, z	(from a set of strings)	
Module names	M		
Simple Types	T	$::=$	$\mathsf{int} \mid \mathsf{unit} \mid A * A$
Function Types	F	$::=$	$A \rightarrow T$
All types	A	$::=$	$T \mid F$
Module types	σ	$::=$	$\{z_1{:}An, ..., zn{:}An\}$
Values	v	$::=$	$n \mid () \mid (v, v) \mid \lambda x{:}A.e$
Expressions	e	$::=$	$n \mid () \mid (e, e') \mid \pi_r e \mid ee \mid x \mid M.x \mid M^n.x \mid \textbf{let } x{:}A = e \textbf{ in } e' \mid \lambda x{:}A.e$
Module expressions	m	$::=$	$\textbf{letrec } \{z_1{:}A_1 = v_1 \text{and ...and } zn{:}An = vn\}$
Program	p	$::=$	$\textbf{module } M^n = m \textbf{ in } p \mid e$
Atomic expr. context	A_1	$::=$	$(\_, e) \mid (v, \_) \mid \_e \mid (\lambda x{:}A.e)\_ \mid \pi_r\_$
Atomic module context	A_2	$::=$	$\textbf{module } M = m \textbf{ in } \_$
Expression contexts	E_1	$::=$	$\_ \mid A_1.E_1$
Module contexts	E_2	$::=$	$\_ \mid A_2.E_2$
Prefix context	E_3	$::=$	$E_2.E_1$

Figure 1: BLAH calculus syntax

GPS:This is unsound at the moment

(ver) $\quad E_2.\textbf{module } M^n = m \textbf{ in } E_3.(M^n.z) \longrightarrow E_2.\textbf{module } M^n = m \textbf{ in } E_3.v$
$(z, v) \in \text{exports}(m)$

(unver) $\quad E_2.\textbf{module } M^n = m \textbf{ in } E_3.(M.z) \longrightarrow E_2.\textbf{module } M^n = m \textbf{ in } E_3.v$
$(z, v) \in \text{exports}(m) \text{ and } \neg\exists k > n.M_k \in \text{modules}(E_2)$

(update)
$$\frac{\text{env}(E_2) \vdash m' \qquad n = \max\{k \mid M_k \in \text{modules}(E_2, E_2')\} \qquad n' = succ(n)}{\begin{array}{l} E_2.\textbf{module } M^n = m \textbf{ in } E_2'.E_1.\textbf{update} \xrightarrow{M=m'} \\ E_2.\textbf{module } M^n = m \textbf{ in } E_2'.\textbf{module } Mn' = \{Mn'/M\}m' \textbf{ in } E_1.() \end{array}}$$

(let) $\quad E_3.\textbf{let } x = v \textbf{ in } e \longrightarrow E_3.\{v/x\}e$

(proj) $\quad E_3.\pi_r(v_1, v_2) \longrightarrow E_2.v_r$

(app) $\quad E_3.(\lambda x{:}A.e)v \longrightarrow E_3.\{v/x\}e$

(cong)
$$\frac{e \longrightarrow e'}{E_3.e \longrightarrow E_3.e'}$$

Figure 2: BLAH calculus reduction rules

GPS:This is unsound at the moment

$$\frac{\begin{array}{l}\Sigma;\Gamma\vdash p{:}T\\ \Sigma;\Gamma\vdash m{:}\sigma\\ M^n\notin \text{modules}(p)\end{array}}{\Sigma;\Gamma\vdash \textbf{module}\ = m\ \textbf{in}\ p{:}T}
\qquad
\frac{\begin{array}{l}\Sigma, M^n{:}\{z_1{:}A_1,...,zn{:}An\};\Gamma\vdash e_1{:}A_1\\ ...\\ \Sigma, M^n{:}\{z_1{:}A_1,...,zn{:}An\};\Gamma\vdash en{:}An\end{array}}{\Sigma;\Gamma\vdash \textbf{letrec}\ z_1=e_1\text{and ...and}\ zn=en{:}\{z_1{:}A_1,...,zn{:}An\}}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma,z{:}A\vdash e'{:}A'\\ \Sigma;\Gamma\vdash A{:}T\end{array}}{\Sigma;\Gamma\vdash \textbf{let}\ z=e\ \textbf{in}\ e'{:}A}
\qquad
\frac{\Sigma;\Gamma,x{:}T\vdash e{:}T'}{\Sigma;\Gamma\vdash \lambda x{:}T.e{:}T\rightarrow T'}$$

$$\frac{\begin{array}{l}M^n\in\Sigma \qquad (x=A)\in\Sigma(M^n)\\ n=\max\{k \mid M_k\in\Sigma\}\end{array}}{\Sigma;\Gamma\vdash M.x{:}A}
\qquad
\frac{M^n\in\Sigma \qquad (x=A)\in\Sigma(M^n)}{\Sigma;\Gamma\vdash M^n.x{:}A}$$

$$\frac{(x=A)\in\Gamma}{\Sigma;\Gamma\vdash x{:}A}
\qquad
\frac{\Sigma;\Gamma\vdash e{:}A \qquad \Sigma;\Gamma\vdash e'{:}A'}{\Sigma,\Gamma\vdash (e,e'){:}A*A'}
\qquad
\frac{\Sigma;\Gamma,z{:}A\vdash e{:}A'}{\Sigma;\Gamma\vdash \lambda z{:}A.e{:}A\rightarrow A'}$$

$$\frac{\Sigma;\Gamma\vdash e{:}A_1*A_2}{\Sigma;\Gamma\vdash \pi_r e{:}Ar}
\qquad
\frac{\Sigma;\Gamma\vdash e{:}T\rightarrow T' \qquad \Sigma;\Gamma\vdash e{:}T}{\Sigma;\Gamma\vdash e\,e'{:}T'}$$

Figure 3: BLAH calculus typing rules

calculus. It can be seen that (let), (app) and (proj) are standard call-by-value (CBV) reduction rules, with the remaining three rules being the interesting ones for this system. They deal with accessing the module bindings and updating module definitions.

A program consists of a sequence of module declarations followed by an expresion, which is where execution begins. We say that bindings in the module declarations are global, whereas bindings introduced by lets and lambdas are local. The two classes of binding do not have the same semantics, differing most notably in how eagerly the symbols are resolved. It is interesting to note however, that the calculus is still call-by-value as the instantiation of a variable always results in a value.

We should first ask why the treatment of instantiation of module definitions needs to be different to other variables. The reason for this is that bound variables in the lambda calculus are discharged during reduction by substitution. The problem is that if we resolve module names in a similar way, then we also dismiss the possibility of updating the module in the future. Consider:

$$\textbf{module}\ M = \textbf{letrec}\ f = \lambda x{:}T.x\ \textbf{in}\ (M.f2, M.f3) \longrightarrow ((\lambda x{:}T.x)2, (\lambda x{:}T.x)3)$$

the definition of the module has been distributed throughout the program and its associated name has been irretrievably lost

3.1.1 Update

The unit of update is the module, which in this language is a sequence of bindings that bind values to identifiers, these binding groups being mutually recursive.

XXX Why is our calculus better than just coding it up in a STLC with refs XXX

3.2 Examples

processing old transactions with old code (only update toplevel variable); changing things directly (as in our other Update calc); Optimizing the first using parts of the second

4. Future Work

4.1 Proving a Consistency Property

Talk about assertions idea

4.2 Understanding the Effect of Language Features

abstract types; references; threads; closures; object-orientation

5. Conclusions

6. REFERENCES

[1] J. L. Armstrong and R. Virding. Erlang — An Experimental Telephony Switching Language. In *XIII International Switching Symposium*, Stockholm, Sweden, May 27 – June 1, 1991.

[2] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, Laboratory for Computer Science, The Massachussets Institute of Technology, March 1983.

[3] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, March 1993.

[4] D. Duggan. Type-based hot swapping of running modules. In *International Conference on Functional Programming*, pages 62–73, 2001.

[5] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, Laboratory for the Foundations of Computer Science, The University of Edinburgh, December 1997.

[6] D. Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[7] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *Transactions on Software Engineering*, 22(2):120–131, Feb. 1996.

[8] M. W. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, The University of Pennsylvania, August 2001.

[9] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Department of Computer Science, University of Wisconsin, Madison, April 1983.

[10] C. Walton, D. Kirli, and S. Gilmore. An abstract machine for module replacement. Technical report, Laboratory for the Foundations of Computer Science, The University of Edinburgh, June 1998.