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

April 14, 2003

Abstract

Dynamic software updating (DSU) enables running programs to be updated with new code and data without interrupting their execution. A number of DSU systems have been designed, but there is still little rigorous understanding of how to use DSU technology so that updates are safe. As a first step in this direction, we introduce a small *update calculus* with a precise mathematical semantics. The calculus is formulated as an extension of a typed lambda calculus, and supports updating technology similar to that of the programming language Erlang [2]. Our goal is to provide a simple yet expressive foundation for reasoning about dynamically updateable software. In this paper, we present the details of the calculus, give some examples of its expressive power, and discuss how it might be used or extended to guarantee safety properties.

1 Introduction

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 continuous 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 at present little general understanding of how DSU is best provided and used. There are many open questions: How to know when it is safe to perform an update, so that the system smoothly transitions from its old version to the new one? Or conversely, how can we build programs that are update-safe by construction? How should updating itself be implemented? What language features complicate or simplify updating? To what extent can one use common mechanisms for updating in different languages, whether functional, object-oriented, or imperative?

There are quite a number of DSU implementations (e.g. [2, 13, 16, 9, 19, 15, 6, 14] among others), and many informal or incomplete ideas as to when a dynamic update should be considered safe [12, 3, 4, 16, 13]. However, we believe a formal, mathematical approach, with clear operational semantics, should be developed to set a firm foundation for both users and implementors of DSU technology. Unfortunately, little semantic work has been carried out to date. The most

substantial work is by Gupta [11, 12], who showed that we cannot prove, for an arbitrary program and an arbitrary update to it, that the update is *valid* in a particular sense – i.e. that it will eventually result in a reachable program state of the new code (we discuss validity and other desirable correctness properties of updates in Section 4).

Any analysis must therefore consider only particular programs or program structures, suggesting that a language-based formalism is needed. However, there are few formalized programming languages for modeling DSU or its aspects [20, 7], and these lack both simplicity and generality. Therefore, we believe that a simple formal system should be established with the goal of understanding the underlying foundations of update, for the purpose of understanding how to best build reliable updateable programs.

In this paper, we present a starting point for such a formal system with our *update calculus*. This is a model programming language with the following characteristics:

1. *Simplicity.* It straightforwardly extends the first-order simply-typed lambda-calculus with mutually-recursive modules and a primitive for updating them.

2. *Flexibility.* We allow any module in the system to be updated, including changes to the types of its definitions, as long as the resulting program is type-correct. Furthermore, the timing of an update can be controlled by the programmer, based on the insertion of an `update` primitive. Finally, the effects of an update can be controlled by using appropriate variable syntax. In combination, these features allow us to model a range of systems, from those that allow updating at any time in arbitrary (but type-correct) ways, to those that have timing and/or update-content restrictions (e.g. [10]).

3. *Practicality.* Our calculus is informed by our own implementation experience [13], and that of other DSU implementations, notably the one in Erlang [2]. We show how our calculus can be used to model a number of realistic situations and updating strategies.

In the next section we present the calculus. In the remainder of the paper, we show how it can be used to express various styles of update to a a server application (§3), discuss how it can be extended to prove safety properties of interest (§4), relate to prior work (§5), and conclude.

2 The update calculus

In this section we introduce the update calculus as a simple formal model of dynamic update. We describe the syntax and semantics of the language, focusing on the key mechanisms that enable DSU. We defer presentation of detailed examples to the next section.

2.1 Syntax

Figure 1 shows the syntax of the language, which is basically a first-order, simply-typed, call-by-value lambda calculus with two extensions: (1) a simple

Natural numbers	n		
Identifiers	x, y		
Module names	M		
Module component identifiers	z, f		
Versioned module names	M^n		
Simple types	S	::=	$\mathsf{int} \mid \mathsf{unit} \mid S * S$
Function types	F	::=	$S \to S$
All types	T	::=	$S \mid F$
Module interfaces	σ	::=	$\{z_1{:}T_n, ..., z_n{:}T_n\}$
Expressions	e	::=	$n \mid () \mid (e, e') \mid \pi_r e \quad (r = 1, 2) \mid \lambda x{:}S.e \mid e\ e$
		$\mid$	$\texttt{let}\ x{:}T = e\ \texttt{in}\ e' \mid x \mid M.z \mid M^n.z \mid \texttt{update}$
Values	v	::=	$n \mid () \mid (v, v') \mid \lambda x{:}S.e$
Module bodies	m	::=	$\{z_1{:}T_1 = v_1\ ...\ z_n{:}T_n = v_n\}$
Module sets	ms	::=	$\{\texttt{module}\ M_1^{n_1} = m_1, .., \texttt{module}\ M_k^{n_k} = m_k\}$
Programs	P	::=	$\texttt{modules}\ ms\ \texttt{in}\ e$
Atomic expr. contexts	A_1	::=	$(\_, e) \mid (v, \_) \mid \pi_r \_ \mid \_ e \mid (\lambda x{:}S.e)\_$
		$\mid$	$\texttt{let}\ x{:}T = \_\ \texttt{in}\ e$
Expression contexts	E_1	::=	$\_ \mid A_1 \cdot E_1$
Module contexts	E_2	::=	$\texttt{modules}\ ms\ \texttt{in}\ \_$

We work up to alpha-conversion of expressions throughout, with x binding in e in an expression $\lambda x{:}S.e$ and y binding in e' in an expression $\texttt{let}\ y{:}T = e\ \texttt{in}\ e'$. The M_k^n of a module set and the z_i of a module body do not bind, and so are not subject to alpha-conversion.

Figure 1: Update calculus syntax

module system with novel variable lookup rules, and (2) an `update` primitive that allows loading a new version of a module during program execution. We restrict the language to first-order functions only as updating higher-order functions (in particular closures) introduces many complexities.

A *program* P consists of a mutually-recursive set ms of module declarations and an expression e to evaluate. Module declarations are of the form $\texttt{module}\ M^n = m$, where M is a module name, n is a *version number*, and m is a module body. (Note that the version superscript n is part of the abstract syntax of programs, while a subscript k on a module name—or a variable or expression for that matter—as in $M_k^{n_k}$, is used only to notate enumerations.) Many different versions of the same module can coexist in a program, but each pair of a module name and a version number is unique. In turn, a module body m is a collection of bindings of values for module component identifiers, written $z{:}T = v$.

Expressions e are mostly standard, including pairs and pair projection π_r, function abstractions and applications, and `let` binders. To update a module with a new version, or insert a new module, we provide a primitive `update`, which will allow a module to be loaded into the program during execution. To allow staged transitions from old to new code, we allow flexible access to module

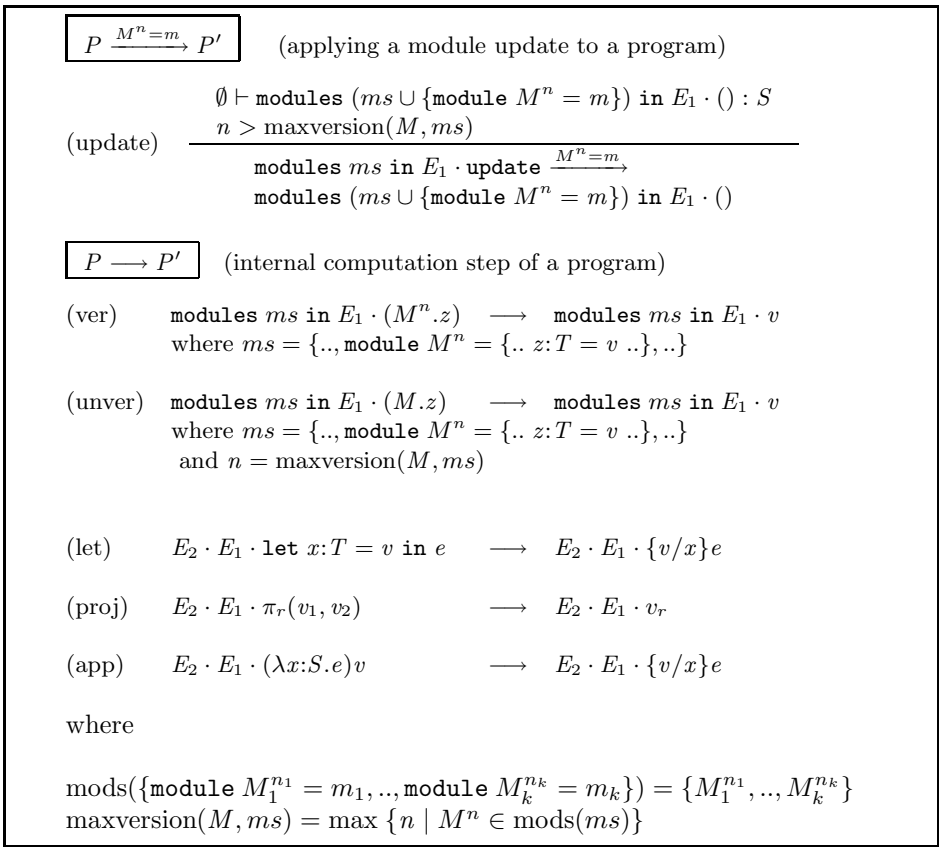

Figure 2: Update calculus reduction rules

components: to access the z component of a module named M, one can write either $M.z$, which will use the newest version of the module M, or $M^n.z$ (for some n), which will uses version n of the code. This semantics is analogous to the prefixing semantics of Erlang, but is slightly more general. In particular, Erlang requires *all* references to an external module to invoke the newest version of the code; control is only afforded when referencing bindings within the same module.

2.2 Semantics

Figure 2 presents the dynamics of the calculus. We define a small-step reduction relation $P \longrightarrow P'$, using evaluation contexts E_1 for expressions and E_2 for programs. Context composition is denoted by $\cdot$, as in $E_2 \cdot E_1 \cdot e$. The rules for (let), (app) and (proj) are standard, while the remaining three rules describe accessing module bindings and updating module definitions.

So that module updates work as we would expect, module component identifiers are not resolved by substitution, as is the case with local bindings (c.f. the (let) and (app) rules), but instead by 'lookup'. In particular, the (ver) rule will resolve the component identifier z from version n of module M when the expression $M^n.z$ appears in redex position. Similarly, the (unver) rule handles

Here Γ ranges over partial functions from identifiers x to types T, and Σ ranges over partial functions from module names M and versioned module names M^n to module types σ. Define

$$\text{modsig}(\{z_1{:}T_1 = v_1 \ ... \ z_n{:}T_n = v_n\}) \ = \ \{z_1{:}T_1, .., z_n{:}T_n\}$$

$$\text{modctx0}(\{\texttt{module}\ M_1^{n_1} = m_1, .., \texttt{module}\ M_k^{n_k} = m_k\}) \ = \ \{M_1^{n_1}{:}\text{modsig}(m_1), .., M_k^{n_k}{:}\text{modsig}(m_k)\}$$

$$\text{modctx}(ms) \ = \ \text{modctx0}(ms) \cup \{M{:}\sigma \mid \exists n.M^n{:}\sigma \in \text{modctx0}(ms) \wedge n = \text{maxversion}(M, ms)\}$$

$\boxed{\emptyset \vdash P{:}S}$ (type checking a program)

$$\frac{\Sigma = \text{modctx}(\{\texttt{module}\ M_1^{n_1} = m_1, .., \texttt{module}\ M_k^{n_k} = m_k\}) \quad \Sigma \vdash m_i{:}\text{modsig}(m_i) \quad (i = 1..k) \quad \Sigma; \emptyset \vdash e{:}S}{\emptyset \vdash \texttt{modules}\ \{\texttt{module}\ M_1^{n_1} = m_1, .., \texttt{module}\ M_k^{n_k} = m_k\}\ \texttt{in}\ e{:}S}$$

$\boxed{\Sigma \vdash m{:}\sigma}$ (type checking a module body)

$$\frac{\Sigma; \emptyset \vdash v_i{:}T_i \quad (i = 1..n)}{\Sigma \vdash \{z_1{:}T_1 = v_1 \ ... \ z_n{:}T_n = v_n\}{:}\{z_1{:}T_1, ..., z_n{:}T_n\}}$$

$\boxed{\Sigma; \Gamma \vdash e{:}T}$ (type checking an expression)

$$\frac{}{\Sigma; \Gamma \vdash n{:}\mathsf{int}} \qquad \frac{}{\Sigma; \Gamma \vdash ()\text{:}\mathsf{unit}} \qquad \frac{\Sigma; \Gamma \vdash e{:}S \qquad \Sigma; \Gamma \vdash e'{:}S'}{\Sigma, \Gamma \vdash (e, e'){:}S * S'}$$

$$\frac{\Sigma; \Gamma \vdash e{:}S_1 * S_2}{\Sigma; \Gamma \vdash \pi_r e{:}S_r} \qquad \frac{\Sigma; \Gamma \vdash e{:}S \to S' \qquad \Sigma; \Gamma \vdash e{:}S}{\Sigma; \Gamma \vdash e e'{:}S'}$$

$$\frac{\Sigma; \Gamma, x{:}S \vdash e{:}S'}{\Sigma; \Gamma \vdash \lambda x{:}S.e{:}S \to S'} \qquad \frac{\Sigma; \Gamma, x{:}T \vdash e'{:}T' \quad \Sigma; \Gamma \vdash e{:}T}{\Sigma; \Gamma \vdash \texttt{let}\ x{:}T = e\ \texttt{in}\ e'{:}T'}$$

$$\frac{}{\Sigma; \Gamma, x{:}T \vdash x{:}T} \qquad \frac{}{\Sigma, M^n{:}\{..., z{:}T, ...\}; \Gamma \vdash M^n.z{:}T}$$

$$\frac{}{\Sigma, M{:}\{..., z{:}T, ...\}; \Gamma \vdash M.z{:}T} \qquad \frac{}{\Sigma; \Gamma \vdash \texttt{update}\text{:}\mathsf{unit}}$$

Figure 3: Update calculus typing rules

the $M.z$ case, with the difference being that the most recent version of module M is used. This semantics is crucial for properly implementing updating.

The (update) rule defines the semantics of the `update` primitive, with labelled transitions $P \xrightarrow{M^n=m} P'$. The idea is that when this primitive is evaluated, the system will apply any waiting update to the running system. We express this idea by having the rule accept a module name M, a version number n, and a module body m. If the new module does not invalidate the type safety of the program, and if n is greater than any existing version of M, the new module is added (if type safety would be compromised, the update cannot take effect – see §4 for further discussion). Any unversioned existing references to M in the code will now refer to the newly loaded module.

We can now look at an example update. In the following take

$$\begin{array}{ll} ms \equiv & \{\texttt{module}\ M^0 = \{ \\ & \quad f = \lambda x{:}\mathsf{unit}.\texttt{let}\ y{:}\mathsf{unit} = \texttt{update in}\ M.z \\ & \quad z = 3\}\} \end{array}$$

to be the initial set of modules, an initial expression $M.f\ ()$, and $m \equiv \{z = (5,5)\}$ be a module body to be loaded. We have:

$$\begin{array}{ll} & \texttt{modules}\ ms\ \texttt{in}\ M.f\ () \\ \longrightarrow & \texttt{modules}\ ms\ \texttt{in}\ (\lambda x{:}\mathsf{unit}.\texttt{let}\ y{:}\mathsf{unit} = \texttt{update in}\ M.z)\ () \\ \longrightarrow & \texttt{modules}\ ms\ \texttt{in let}\ y{:}\mathsf{unit} = \texttt{update in}\ M.z \\ \xrightarrow{M^1=m} & \texttt{modules}\ ms'\ \texttt{in let}\ y{:}\mathsf{unit} = ()\ \texttt{in}\ M.z \\ \longrightarrow & \texttt{modules}\ ms'\ \texttt{in}\ M.z \\ \longrightarrow & \texttt{modules}\ ms'\ \texttt{in}\ (5,5) \end{array}$$

where $ms' = ms \cup \{\texttt{module}\ M^1 = \{z = (5,5)\}\}$.

At the point where the $M.f$ is resolved, in the first reduction step, the greatest extant version of M is M^0 – so $M.f$ is replaced by its $M^0.f$ definition. When the $M.z$ is resolved in the last reduction step, however, the greatest version of M is the M^1 supplied by the update – and so $M.z$ resolves to $(5,5)$ instead of 3.

The type system provides the necessary checks to ensure that loading a module does not result in a program that will reduce to a stuck state (one in which the expression is not a value and yet no reduction rule applies). Figure 3 shows the type system for our calculus. The rules for the judgment $\Sigma;\Gamma \vdash e{:}T$ are the standard ones for the simply typed lambda calculus, extended in the obvious way to deal with the typing of module components. The `update` command is statically uninteresting and types as `unit`, as this is the type of the () value it becomes after (update). The other two judgments are more interesting. $\Sigma \vdash P{:}S$ types whole programs and handles most of the complexity in typing modules. We use two auxiliary functions modsig and modctx: modsig determines the interface of a module given its body, and, given a set of modules, modctx determines the partial function that maps versioned module names M^n to their signatures and also maps the unversioned module names M to the signature of the highest versioned module with the same name. modctx can thus be used to determine the module context in which the program (including the module bodies themselves) should be typed. The single rule defining the judgement ensures that the expression and every module body can be typed in

this context; this means that the modules are allowed to be mutually recursive, as every module name is available in the typing of each module.

Typing of module bodies is expressed by the judgment $\Sigma \vdash m{:}\sigma$, i.e. that module body m has interface σ in the context of module declarations Σ; it simply requires that each component of the module has the appropriate type.

2.3 Discussion

Many design decisions reflect our aim to keep the update calculus simple, but nonetheless practical and able to express different updating strategies for programs. We further consider some of those design decisions here.

The calculus addresses the run-time mechanisms involved in implementing updating (that is, loading new modules and allowing existing code or parts thereof to refer to them), but does not cover all the important software development issues of managing updateable code. In practice, we would expect compiler support for aiding the development process [13]. For example, user programs could refer to the 'current' and 'previous' versions of a module, and the compiler would fill in the absolute version number.

As many past researchers have observed, the timing of an update is critical to assuring its validity [11, 16, 9, 13]. We chose to support *synchronous* updates by using the `update` primitive to dictate when an update can occur. This makes it easier to understand the state(s) of the program which an update is applied to than the alternative *asynchronous* approach, and so makes it easier to write correct updates.

Of equal importance is the need to control an update's effect. Which modules will 'notice' the new version? Can an old version and a new version coexist? Different systems answer these questions differently. Many systems allow multiple versions to coexist [2, 9, 13, 7, 1, 18], while others prefer one version at a time [10, 11]. Our use of module versions allows multiple generations of a module to exist simultaneously, and provides explicit control over which version of a module we are referring to, allowing us to delimit the effect of an update. As such, we can model a variety of updating situations.

Finally, we assert that updateable programs must be reliable: if the program crashes frequently, it is probably not 'mission-critical', and thus it has little need to be updated dynamically! For this reason, we imposed a static (link-time) type system to ensure that if an update is accepted by the system, then the resulting program will be type-correct. In addition to improved reliability, we also believe that type-correct programs are easier to reason about.

3 Example: Updating a Server Application

To illustrate the expressive power of our calculus, we present some fairly realistic examples of updating a long-lived server application. There are many real-world examples of this class of system that employ or could benefit from DSU; e.g. financial transaction processors, web and database servers, network routers, intrusion detection sensors, and more. Because our calculus lacks concurrency (a non-trivial extension), we focus on a single-threaded, event-based architecture, which is not uncommon in server applications [17, 5, 21].

```
modules {
  module Handlers[1] = {
    handleGet = λ(q,e). ...
    handlePost = λ(q,e). ...
    handleUpdate = λ(q,e). update ; q
    ...
    }
  module Server[1] = {
    getevent = λq. ...
    handle = λq.
      let (q,e) = Queue.dequeue q in
      ...demultiplex...
      ... Handlers[1].handleGet (q,e) ...
      ... Handlers[1].handlePost (q,e) ...
      ... Handlers[1].handleUpdate (q,e) ...
    loop = λq.
      let q = Server.getevent q in
      let q = Server.handle q in
      Server.loop q
    }
  } in
Server.loop Queue.empty
```

Figure 4: An updateable (web) server

To make the code examples easier to read, we have taken some liberties with our syntax. In particular, we allow tuples rather than pairs, with pattern-matching; we have elided all typing annotations; we suppose the existence of booleans and conditionals; we suppose the existence of a type of queues, which could be implemented using lists; and we allow simultaneous updates of multiple modules. All these should be clear in context, and would be routine to add to the calculus definition.

3.1 Initial System

The initial program for our updateable server is shown in Figure 4. The `Server` module implements the basic event loop. The `loop` function has a queue of events, e.g. HTTP requests from clients or responses sent by handlers. New events are created by `getevent`, which queues any new events (such as client requests) and returns, or blocks if both the queue is empty and no new events have occurred. Once an event is extracted it is demultiplexed by the `handle` function, which calls the handlers implemented in the `Handlers` module. One possible event is a request to update the server. This is processed by the `Handlers.handleUpdate` function, which invokes `update`. Notice that because of the placement of the `update` primitive, updates will always occur just before the recursive call to `loop`, meaning that no computations will be incomplete when the update is accepted. This intuitively allows us to believe that an update will not result in an inconsistent state; we consider this point further in Section 4.1.

Also notice that calls to `Handlers` functions use an explicit version; the reason for this will be evident shortly. We assume that the program is using a module `Queue` to implement its event queue, which is not shown. The programs starts with a call to `loop` with an empty queue as its argument.

3.2 First Update: Adding a Log

As a first example update, say we want to log all of the HTTP events that we process, so we want to add logging to the server. To do this, we need to change the `loop` and `handle` functions of `Server` to additionally take a log object as an argument. Upon handling each event, a record will be added to the log. To realize this change on-line, we note that the existing `Server.loop` function does not expect this extra parameter, and therefore we must introduce a 'transitional' function `loop` *at the old type* (that is, expecting only a queue as its argument), which then calls the new version of `loop'` at the new type (that is, expecting both the queue and log as arguments). This is shown in Figure 5. Transitional functions have been proposed in existing systems under various names and guises [13, 16, 9].

```
module Log[1] = {
  emptyLog = ...
  logevent = λ(l,e). ...
  }
module Server[2] = {
  getevent = λq. ...
  handle = λ(l,q).
    let (q,e) = Queue.dequeue q in
    let l = Log.logevent (l,e) in
    ...
    Handlers[1].handleGet (q,e)
    ...
  loop = λq.
    Server[2].loop' (Log[1].emptyLog, q)
  loop' = λ(l,q).
    let q     = Server.getevent q in
    let (l,q) = Server.handle (l,q) in
    Server.loop' (l,q)
  }
```

Figure 5: Adding a log to the server

When the original `Server`^{1}`.loop` function handles the update event, its recursive call to `Server.loop` will dispatch to the new `Server`^{2}`.loop` function by the (unver) reduction rule. This function creates an empty log and calls `Server`^{2}`.loop'`, which continues processing events. This function will call the new `handle` function, which expects the log as an argument, and will log the appropriate events.

It would seem unfortunate from a software engineering point of view that the name of the `loop` function changes to `loop'` in the new version. However,

this issue can be resolved with compiler/tool support, as we have motivated and implemented in other work [13]. Our goal is to focus on the run-time issues, since doing so keeps things simpler and will not impede our formal reasoning ability.

```
module Server³ = {
  getevent = λq. ...an event of the new type...
  handle = λ(l,q).
    let (q,e) = Queue.dequeue q in
    let l = Log.logevent (l,e) in
    ...
    Handlers².handleGet (q,e)
    ...
  loop' = λ(l,q).
    if (Queue.isempty q) then
      Server³.loop'' (l, Queue.empty)
    else
      let q = Server².getevent () in
      let q = Server².handle (l,q) in
      Server³.loop' (l,q)
  loop'' = λ(l,q).
    let q     = Server.getevent q in
    let (l,q) = Server.handle (l,q) in
    Server.loop'' (l,q)
  }
```

Figure 6: Complete existing events first

3.3 Second Update: Enriching Events

The first update added to the server's functionality; we might also wish to enrich or change existing functionality. For instance, we could extend the definition of events to include additional information, perhaps to refine existing event descriptions or to add new ones. Such a change will impact all of the code that manipulates events, including our event queue and handler functions.

Because we are making the change without shutting down the system, we have to consider the existing unprocessed events before switching to the new format. In particular, the server's existing event queue could be non-empty. Here are two possible choices: (1) convert all of the events in the existing queue to have the format expected by the new code, or (2) process the old events using the old code and then switch to using new code for the new events. The former strategy, which we shall dub *convert*, is taken by most proposed DSU systems (e.g. [7, 13, 10, 9, 16, 19]), while the latter, which we dub *complete*, is taken by fewer (e.g. [1, 18]). Each approach has advantages and disadvantages; a main goal for our calculus is to be able to express a range of design decisions, so as to evaluate these tradeoffs.

In both cases, we will need to create new versions of the `Handlers` and `Log`

```
module Server³ = {
  convertevent = λe. ...convert event e
  convert = λ(q,q').
    if (Queue.isempty q) then q'
    else
      let (q,e) = Queue.dequeue q in
      let e' = Server³.convertevent e in
      let q' = Queue.enqueue q' e' in
      Server³.convert (q,q') in
  getevent = λq. ...an event of the new type
  handle = λ(l,q).
    let (q,e) = Queue.dequeue q
    let l = Log.logevent (l,e)
    ...
    Handlers².handleGet (q,e)
    ...
  loop' = λ(l,q).
    let q' = Server.convert (q,Queue.empty) in
    Server³.loop'' (l,q')
  loop'' = λ(l,q).
    let q     = Server.getevent q in
    let (l,q) = Server.handle (l,q);
    Server.loop'' (l,q)
  }
```

Figure 7: Convert existing events to new format

modules that use the new form of events:

```
module Handlers² = ...handles new event type
module Log² = ...handles new event type
```

(we omit the details of these two).

The new `Server` module for the *complete* strategy is illustrated in Figure 6. Here, if the existing queue is nonempty the new `loop'` processes the old events by explicitly referring to `Server`^{2}`.getevent` and `Server`^{2}`.handle` functions, which call the handlers from the `Handlers`^{1} module. Once the queue is empty, `loop'` calls the new `Server`^{3}`.loop''` function with an empty queue (to hold the new events).

Note that the *complete* strategy would not have worked if we had not explicitly included the version number when calling `Handlers`^{1} handling functions in `Server`^{2}`.handlers`. If instead we had used the unversioned syntax (e.g. `Handlers.handleGet`), the new versions of the handlers would have been called following the update, and this would have been flagged as a type error. On the other hand, had we used the unversioned syntax, we could have supplied an intermediate update that inserted the versioned variable syntax, and then proceeded with our original update!

The new module required for the *convert* strategy are shown in Figure 7. Here we have defined two new functions `convertevent` and `convert`; the former

converts an event from the old to the new format, and the latter recursively creates a new queue containing the converted events of the old one. At the time of the update, the `loop'` function will call `convert` to create a new queue, and then call $\texttt{Server}^3\texttt{.loop''}$ to proceed with processing the converted (and new) events with the new code.

4 Ongoing Work

The calculus we have presented is a first step towards a formal system for reasoning about Dynamic Software Updating. In this section, we briefly sketch how we envision using the calculus: to discuss properties of updates, and as a foundation for realistic updatable programming languages.

4.1 Ensuring Correctness Properties of Updates

As argued above, updatable programs must be reliable, yet updating itself introduces further complexity. To prevent total confusion, techniques are required for ensuring that updates are in some sense *safe*. Broadly, there are two classes of safety properties: simple type-safety properties and richer semantic correctness results.

Type Safety In the absence of any updates, the type system of our calculus prevents all runtime errors statically, e.g. applications of pairs to arguments. For updates, there are two main options. We have chosen to allow updates to change types arbitrarily – the signatures of different versions of a module need not be related – but to maintain type safety we perform a typecheck of the whole running program at update time. The obvious alternative would be to require new versions to be related by a subtype order to previous versions. This would rule out the example updates of §3, but would require only a subtype check for the new module at update-time, instead of a whole-program typecheck. For the first option, it would be interesting to identify sufficient conditions for the whole-program typecheck that could be performed off-line.

Our calculus does not specify what happens if an update-time check fails. We envisage tool support to return information to the updater, and that the system continues to execute.

Semantic Correctness If one had precise specifications of a system before and after a proposed update, then in principle one could prove the updated system will meet the latter specification. In practice, though, creating such specifications is extremely challenging, as is large-scale program proof.

On the other hand, when designing an update we do engage in delicate informal reasoning. For example, recall that in our updateable server of Figure 4, we placed the `update` primitive so that updates would always occur just before the recursive call to the event loop. This means that no computations would be incomplete when an update is accepted – in contrast to the alternative placing of update at the beginning of the event loop function.

Pragmatically, we envision formalizing this intuition by extending the calculus with logical assertions, using a Hoare-like logic. The idea is that a long-running loop, such as the one used in the server application from §3, expects

that the system state will respect some invariant. For example, our server loop assumes that upon entry to the loop, no events are being processed midstream; that is, all events not in the queue have been completely processed. In a more sophisticated server, this might imply that global data structures, like caches and maps, respect their own invariants. In turn, the new version of the server loop would assert a slightly different invariant, based on its new functionality. We could then use the logic to prove that given the update and the assertion in the old loop, after the computations performed in the transition function, the new loop will begin processing with its invariant held. Partial automation of such formal reasoning is becoming feasible, as work on Extended Static Checking [8] demonstrates.

Other Properties Gupta *et al.* investigated another property in [11, 12], which we detail in §5. There, an update is *valid* if (loosely) the resulting state eventually becomes a state that is reachable from the initial state of the new program text. One can envisage many reasonable updates that do not have this property, however – for example an update to our logging server that adds timestamps to log entries, recorded in a field of option type, with a `convert` function that attaches a `None` to old entries. Such old entries will persist, but the new program text would never generate them.

4.2 Language Design for Updatability

To extend our calculus of §2 to a full programming language, one must consider how standard constructs behave in the presence of update, e.g. higher-order functions, abstract types, concurrency, mutable state, and objects. Here we make some preliminary observations.

It is clear that the structure of a program plays an important rôle in determining the updatability of the resulting program. Sometimes structuring mechanisms that simplify normal programming actually obfuscate data that may be needed at an update point. A good example of this is higher-order functions, which at run-time lead to complex intermediate data structures being wrapped up in closures that are tied to the control flow. Further, it is unlikely that one would want to update (say) all partial applications of a function `f` uniformly, as they would be constructed in different environments.

Abstract data types, on the other hand, can make it easier to design updates, as the code that can access values of those types is collected together.

Another problematic area is that of introducing concurrent execution in the form of threads. Threads introduce non-determinism, which makes understanding the state of the program at update time harder. If one thread performs an update how should the other threads behave with respect to updated code and data? Determining when to do an update in complex situations such as this is hard, and we think that formal models will be invaluable in devising correct update strategies.

Of course, one need not be constrained to existing languages. It would be interesting to consider new structuring mechanisms that ease the writing of updatable code, and in fact this may be necessary – update primitives are not orthogonal to the ambient language. We should also consider the granularity of the update. In this paper we have explored a coarse-grained update mechanism, where the unit of update is a whole module. One can imagine using a more fine-

grained definition of update, although this raises practical problems in compiling and replacing only parts of modules, and eliminates the immediate use of many common compiler optimisations.

5 Related Work

While there have been a variety of implementations of DSU (e.g. [2, 13, 16, 9, 19, 15, 6, 14] among others), comparatively little work has been done in the two areas we are interested in here: analyses as to how to update programs safely; and formal, language-based models of DSU. Here we briefly overview past work in these areas.

5.1 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 in which they proved that the problem of update validity is, in general, undecidable. [11, 12]. 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 notions 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, 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 a reduction 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 [16] 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 [3, 4].

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 [16], 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. [10]). However, in none of these systems is there any well developed evidence as to what conditions are needed to guarantee validity. This has led us to focus on a model of synchronous updates, as mentioned in §2.3, in which the program designates its own update points.

5.2 Language-based Formal Systems

As far as we know, only two past efforts have formalized languages with DSU in mind. Dynamic ML [10] is a proposed implementation of ML with a formalized abstract machine [20] that enables replacement of modules at runtime; changes can include the alteration of abstract types, and the addition (and possibly deletion) of module definitions. The authors define an abstract machine that implements replacement via garbage collection, in which old instances of a changed type are converted at update-time. They show that evaluation in the context of replacement preserves type-safety. Duggan [7] defines a formal language in which 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 these formalizations are instructive, both calculi focus only on the problem of converting instances of changed types following an update, adopting particular language mechanisms to do so. We believe the wider questions of how and when to ensure safe updates require attention; a prime goal of the update calculus has been to use the simplest mechanisms possible in order to highlight commonality among various DSU systems.

6 Conclusions

In this paper, we make two contributions:

- We have motivated the need for a simple, yet flexible and practical formal framework for understanding effects of dynamic software updating.
- We have defined the update calculus (§2) as a first step towards such a framework. The calculus is expressive enough to model real-world examples, as presented in §3, and model design decisions present in existing DSU systems. At the same time, it is simple enough to admit formal reasoning, and extension to include other language features, as presented in §4.

In ongoing work, we plan to use the update calculus to expand our understanding of how to implement languages with updating capabilities, and how to use those features to effectively build reliable, updateable software.

Acknowledgments We acknowledge support from a Royal Society University Research Fellowship (Sewell), a Marconi EPSRC CASE Studentship (Stoyle), AFRL-IFGA IAI grant AFOSR F49620-01-1-0312 (Hicks, while at Cornell University) and EC grant PEPITO.

References

[1] A. Appel. Hot-Sliding in ML, December 1994. Unpublished manuscript.

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

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

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

[5] Boa Webserver. `http://www.boa.org`.

[6] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution*, October 2001.

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

[8] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for java. In *Proc. PLDI, 2002.*, 2002.

[9] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *Journal of Systems and Software*, 14(2):111–128, September 1991.

[10] 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.

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

[12] 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.

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

[14] JDRUMS, Java distributed run-time updating management system. `http://www.ida.liu.se/~jengu/jdrums/`.

[15] G. Kniesel. Type-safe delegation for run-time component adaptation. *Lecture Notes in Computer Science*, 1628:351–366, 1999.

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

[17] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable webserver. In *Proceedings of the USENIX Annual Technical Conference*, pages 106–119, Monterey, June 1999.

[18] J. Peterson, P. Hudak, and G. S. Ling. Principled dynamic code improvement. Technical Report YALEU/DCS/RR-1135, Department of Computer Science, Yale University, July 1997.

[19] Squeak Smalltalk-80 programming system. `http://www.squeak.org`.

[20] 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.

[21] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, October 2001.