Mutatis Mutandis: Safe and Predictable Dynamic Software Updating[*]

Gareth Stoyle[†] Michael Hicks[⋆] Gavin Bierman[‡] Peter Sewell[†] Iulian Neamtiu[⋆]

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

[‡]Microsoft Research, Cambridge
gmb@microsoft.com

[⋆]University of Maryland, College Park
{mwh,neamtiu}@cs.umd.edu

ABSTRACT

Dynamic software updates can be used to fix bugs or add features to a running program without downtime. Essential for some applications and convenient for others, low-level dynamic updating has been used for many years. Perhaps surprisingly, there is little high-level understanding or language support to help programmers write dynamic updates effectively.

To bridge this gap, we present Proteus, a core calculus for dynamic software updating in C-like languages that is flexible, safe, and predictable. Proteus supports dynamic updates to functions (even active ones), to named types and to data, allowing on-line evolution to match source-code evolution as we have observed it in practice. We ensure updates are type-safe by checking for a property we call "con-freeness" for updated types `t` at the point of update. This means that non-updated code will not use `t` *concretely* beyond that point (concrete usages are via explicit coercions) and thus `t`'s representation can safely change. We show how con-freeness can be enforced dynamically for a particular program state. We additionally define a novel and efficient static *updateability analysis* to establish con-freeness statically, and can thus automatically infer program points at which all future (well-formed) updates will be type-safe. We have implemented our analysis for C and tested it on several well-known programs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.3.3 [**Programming Languages**]: Formal Definitions and Theory—*Semantics,Syntax*

General Terms

Design, Languages, Reliability, Theory, Verification

Keywords

dynamic software updating, updateability analysis, type inference, capability, Proteus

[*]This work supported by a Royal Society University Research Fellowship, a Marconi EPSRC CASE Studentship, EC FET-GC project IST-2001-33234 PEPITO, and NSF Contract #0346989.

1. INTRODUCTION

Dynamic software updating (DSU) is a technique by which a running program can be updated with new code and data without interrupting its execution. DSU is critical for non-stop systems such as air-traffic control systems, financial transaction processors, enterprise applications, and networks, which must provide continuous service but nonetheless be updated to fix bugs and add new features. DSU is also useful for avoiding the need to stop and start a non-critical system (e.g., reboot a personal operating system) every time it must be patched.

Providing general-purpose DSU is particularly challenging because of the competing concerns of *flexibility* and *safety*. On the one hand, the form of dynamic updates should be as unrestricted as possible, since the purpose of DSU is to fix bugs or add features not necessarily anticipated in the initial design. On the other hand, supporting completely arbitrary updates (e.g., binary patches to the existing program) makes reasoning about safety impossible, which is unacceptable for mission-critical software.

In this paper we present Proteus, a general-purpose DSU formalism for C-like languages that carefully balances these two concerns, and adds assurances of predictability. Proteus programs consist of function and data definitions, and definitions of *named types*. In the scope of a named type declaration $\mathtt{t} = \tau$ the programmer can use the name `t` and representation type τ interchangeably but the distinction lets us control updates. Dynamic updates can add new types and new definitions, and also provide replacements for existing ones, with replacement definitions possibly at changed types. Functions can be updated even while they are on the call-stack: the current version will continue (or be returned to), and the new version is activated on the next call. Permitting the update of active functions is important for making programs more available to dynamic updates [1, 12, 4]. We also support updating function pointers. Based on our experience [12] and a preliminary study on the evolution of C programs (§2), we believe Proteus is flexible enough to support a wide variety of dynamic updates.

When updating a named type `t` from its old representation τ to a new one τ', the user provides a *type transformer function* `c` with type $\tau \rightarrow \tau'$. This is used to convert existing `t` values in the program to the new representation. To ensure an intuitive semantics, we require that at no time can different parts of the program expect different representations of a type `t`; a concept we call *representation consistency*. The alternative would be to allow new and old definitions of a type `t` be valid simultaneously. Then, we could *copy* values when transforming them, where only new code sees the copies [10, 12]. While this approach would be type safe, old and new code could manipulate different copies of the same data, which is likely to be disastrous in a language with side-effects.

To ensure type safety and representation consistency, we must guarantee the following property: after a dynamic update to some type t, no updated values v' of type t will ever be manipulated *concretely* by code that relies on the old representation. We call this property "con-t-freeness" (or simply "con-freeness" when not referring to a particular type). The fact that we are only concerned about subsequent *concrete* uses is important: if code simply passes data around without relying on its representation, then updating that data poses no problem. Indeed, for our purposes the notion of con-freeness generalizes notions of encapsulation and type abstraction in object-oriented and functional languages. This is because concrete versus abstract uses of data are not confined to a single linguistic construct, like a module or object, but could occur at arbitrary points in the program. Moreover, con-freeness is a flow-sensitive property, since a function might manipulate a t value concretely at its outset, but not for the remainder of its execution.

To enforce con-freeness, Proteus programs are automatically annotated with explicit *type coercions*: $\mathbf{abs}_{\mathtt{t}}\ e$ converts e to type t (assuming e has the proper type τ), and $\mathbf{con}_{\mathtt{t}} e$ does the reverse at points where t is used concretely. Thus, when some type t is updated, we can dynamically analyze the active program to check for the presence of coercions $\mathbf{con}_{\mathtt{t}}$, taking into account that subsequent calls to updated functions will always be to their new versions. If any $\mathbf{con}_{\mathtt{t}}$ occurrences are discovered, then the update is rejected.

Unfortunately, the unpredictability of a dynamic con-free check could make it hard to tell whether an update failure is transient or permanent, since the dynamic check is for a particular program state. Rather, we would prefer to reason about update behavior statically, to (among other things) assess whether there are enough update points. Therefore, we have developed a novel *static updateability analysis*. We introduce an **update** expression to label program points at which updates could be applied. For each of these, we estimate those types t for which the program may not be con-t-free. We annotate the **update** with those types, and at run time ensure that any dynamic update at that point does not change them. This is simpler than the con-free dynamic check, and more predictable. In particular, we can automatically infer those points at which the program is con-free for all types t, precluding dynamic failure.

This paper makes the following contributions:

- We present Proteus, a simple and flexible calculus for reasoning about representation-consistent dynamic software updating in C-like languages (§3). We motivate our DSU support in Proteus with a brief study of the changes over time to some large C programs, taking these as indicative of dynamic updates that we have to support (§2).

- We formally define the notion of con-freeness, and prove that it is sufficient to establish type safety in updated programs (§3.4).

- We present a novel updateability analysis that statically infers the types for which a given **update** point is not con-free (§4). We present some preliminary experience with an implementation of our analysis for C programs (§4.4).

In §6 and §7 we discuss related work and conclude.

2. DYNAMIC SOFTWARE UPDATING

To enable on-line evolution we must support software changes unanticipated during the initial design. The kind of changes that must be support on-line are, we believe, similar to those that can be observed off-line in the program source tree. Therefore, to motivate our approach to DSU we describe the results of a small study we did on the source code evolution of some long-running services.

Using a custom tool, we compared increasing versions of a few large C programs. These include Linux, version 2.4.17 (Dec. 2001) to 2.4.21 (Jun. 2003); BIND, versions 9.2.1 (May 2002) to 9.2.3 (Oct. 2003); Apache, version 1.2.6 (Feb. 1998) to 1.3.29 (Oct. 2003); and OpenSSH, version 1.2 (Oct. 1999) to 3.8 (Feb. 2004). For these programs, the changes followed a few key trends:

- The overwhelming majority of a version change consists of added functions, or changes to existing functions which do not involve a changed type signature. Few, if any, functions are deleted. For example, BIND 9.2.2—9.2.3 resulted in 30 new functions added and 890 changed (starting from 3214 total functions). Of the changed functions, only 28 had a change of signature, of which about two-thirds were to add or remove arguments with the rest changing the type of an argument. As another example, Apache 1.3.0–1.3.6 resulted in 51 functions added, 10 deleted, and 290 changed, of which 4 changed their type signature (starting from 836 total functions).

- Global variables tend to be fairly static, adding a few and deleting a few, but growing over time. For example, OpenSSH grew from 106 to 251 total global variables from version 1.2.2 to 3.7, adding from 3 to 30 new variables per release, deleting up to 10. One change in Apache added 88 variables and deleted 37, but most added or deleted fewer than 10. It is extremely rare for a global variable to change type.

- Data representations, which is to say *type definitions*, change between versions, though rarely. In C, types are defined with `struct` and `union` declarations (aggregates), `typedefs`, and `enums`. Very often, the changes are to aggregates and involve adding or removing a field. For example, moving from Linux 2.4.20—2.4.21 resulted in 36 changes to `struct` definitions (out of 1214 total), of which 21 were the addition or removal of fields, while the remaining 15 were changes to the types of some fields. Type definitions are rarely deleted.

In short, to match these kinds of changes DSU must readily support the addition of new definitions (functions, data, or types), and the replacement of existing definitions (data or functions) at the same type. It must also allow changes to function types and data representations, but not necessarily to the types of global variables. Proteus supports these kinds of changes.

3. PROTEUS

This section defines a core calculus Proteus that formalizes our approach to dynamic software updating.

3.1 Syntax

Proteus models a type-safe, C-like language augmented with dynamic updating; its syntax shown in Figure 1. Programs P are a series of top-level definitions followed by an expression e. A **fun** z ... defines a top-level recursive function, and **var** z : τ ... defines a top-level *mutable* variable (i.e., it has type τ **ref**). A **type** t $= \tau$ ... defines the type t. Top-level variables z (a.k.a. *external names*) must be unique within P, and are not subject to α-conversion, so they can be unambiguously updated at run-time. Expressions e are largely standard. We abuse notation and write multi-argument functions

$$\mathbf{fun}\ \mathsf{f}(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e$$

which are really functions that take a record argument, thus having type $\{\mathrm{x}_1 : \tau_1, \ldots, \mathrm{x}_n : \tau_n\} \rightarrow \tau$. We similarly sugar calls to such functions.

$$
\begin{array}{rcl}
\text{Types } \tau & ::= & \mathsf{t} \mid \mathsf{int} \mid \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\} \\
& \mid & \tau\ \mathbf{ref} \mid \tau_1 \rightarrow \tau_2 \\
\text{Expressions } e & ::= & n \mid x \mid \mathsf{z} \mid r \\
& \mid & \{\mathrm{l}_1 = e_1, \ldots, \mathrm{l}_n = e_n\} \mid e.\mathrm{l} \\
& \mid & e_1\ e_2 \mid \mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2 \\
& \mid & \mathbf{ref}\ e \mid !e \mid e_1 := e_2 \\
& \mid & \mathbf{if}\ e_1 = e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 \\
& \mid & \boxed{\mathbf{update} \mid \mathbf{abs}_{\mathsf{t}}\ e \mid \mathbf{con}_{\mathsf{t}}\ e} \\
\text{Values } v & ::= & \mathsf{z} \mid n \mid \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_n = v_n\} \\
& \mid & r \mid \mathbf{abs}_{\mathsf{t}}\ v \\
\text{Programs } P & ::= & \mathbf{fun}\ \mathsf{z}(x : \tau) : \tau' = e\ \mathbf{in}\ P \\
& \mid & \mathbf{var}\ \mathsf{z} : \tau = v\ \mathbf{in}\ P \\
& \mid & \mathbf{type}\ \mathsf{t} = \tau\ \mathbf{in}\ P \\
& \mid & e
\end{array}
$$

Figure 1: Syntax for Proteus

The $\boxed{\text{boxed}}$ expressions support dynamic updates; we describe them further below. The typing rules define the judgment $\Gamma \vdash P : \tau$, where Γ is a map from type names t, external names z, and local variables x to types τ. Aside from our new constructs, type checking is standard.

Example. Figure 2 shows a simple kernel for handling read and write requests on files or sockets, which one might want to dynamically update. Some functions and type definitions have been elided for simplicity. Reading from the bottom, the function loop is an infinite loop that repeatedly gets req objects (e.g., system calls) and then dispatches them to an appropriate handler using the dispatch function. This function first calls decode to determine whether a given file descriptor is a network channel or an open file (e.g., by looking in a table). If it is a network channel, dispatch calls getsock to get a sock object based on the given file descriptor (e.g., indexed in an array). Next, it decodes the remaining portion of the req to acquire the transmission flags. Finally, it finds an appropriate sockhandler object to handle the request and calls that handler. Handlers are needed to support different kinds of network channel, e.g., for datagram communications (UDP), for streaming connections (TCP), etc. Different handlers are implemented for each kind, and getsockhandler chooses the right one. A similar set of operations and types would be in place for files. After dispatch completes, its result is posted to the user, and the loop continues.

We now discuss dynamic updates to this kernel and explain our new constructs (the boxed syntax from Figure 1).

Update. The **update** expression permits a dynamic update to take place, if one is available. That is, at run-time a user provides an update through an out-of-band signaling mechanism, and the next time **update** is reached in the program, that update is applied. Informally, an update consists of the following:

- Replacement definitions for named types $\mathsf{t} = \tau$. Along with the new definition $\mathsf{t} = \tau'$, the user provides a *type transformer function* of type $\tau \rightarrow \tau'$, used by the runtime to convert existing values of type t in the program to the new representation.

- Replacement definitions for top-level identifiers z, having the same type as the original.

- New type, function, and variable definitions.

```
type handResult = int in
type sockhandler =
  {sock : sock, buf : buf, sflags : sflags} → handResult in

let udp_read(sock : sock, buf : buf, sflags : sflags)
  : handResult = ... in
let udp_write(sock : sock, buf : buf, sflags : sflags)
  : handResult = ... in

type req = {op : op, fd : int, buf : buf, rest : blob} in
type fdtype = File | Socket | Unknown  in

let dispatch (s : req) : handResult =
  let t = decode (s.fd) in
  if (t = Socket) then
    let k = getsock (s.fd) in
    let flags = decode_sockopargs (s.rest, s.op)  in
    let h = getsockhandler (s.fd, s.op)  in
    h (k, s.buf, flags)
  else if (t = File) then ...
  else  − 1 in

let post (r : handResult) : int = ... in

let loop (i : int) : int =
  let req = getreq (i) in
  let i = post (dispatch req) in
  loop i in

loop 0
```

Figure 2: A simple kernel for files and socket I/O

```
let dispatch(s : req) : handResult =
  let t = decode((con_req s).fd) in
  let u1 = update in
  if (con_fdtype t) = Socket then
    let k = getsock((con_req s).fd) in
    let flags =
      decode_sockopargs((con_req s).rest, (con_req s).op) in
    let h = getsockhandler((con_req s).fd, (con_req s).op) in
    let u2 = update in
    let res = (con_sockhandler h)(k, (con_req s).buf, flags) in
    let u3 = update in res
  else if (con_fdtype t) = File then ...
  else (abs_handResult −1)
```

Figure 3: dispatch **with explicit** update **and coercions**

We consider extending specifications to support binding deletion in §5. When writing an updateable program, the programmer can insert **update** manually, and/or the compiler can insert **update** automatically. Figure 3 shows dispatch from Figure 2 with some added **update** expressions (it also includes type coercions, discussed next). We consider a strategy for automatic insertion of **update** in §4.

Type Coercions. In Figure 2, within the scope of a type definition like **type** sockhandler = ... the type sockhandler is a synonym for its definition. For example, the expression $h\ (k,\ s.\mathrm{buf},\ \mathit{flags})$ in dispatch uses h, which has type sockhandler, as a function. In this case, we say that the named type sockhandler is being used *concretely*. However, there are also parts of the program that treat data of named type *abstractly*, i.e., they do not rely on its representation. For example, the getsockhandler function simply returns a sockhandler value; that the value is a function is incidental.

In the Proteus semantics (but not in the user source language) all uses of a named type definition $\mathsf{t} = \tau$ are made explicit with type

$$\begin{array}{llll}
\mathrm{UN}: & \mathsf{sockhandler} & \mapsto & (\{\mathrm{sock}:\mathsf{sock}, \mathrm{buf}:\mathsf{buf}, \mathrm{sflags}:\mathsf{sflags}, \mathrm{cookie}:\mathsf{cookie}\} \to \mathsf{int},\ \mathsf{sockh\_coer}) \\
 & \mathsf{sock} & \mapsto & (\{\mathrm{daddr}:\mathsf{int}, \ldots\}, \mathsf{sock\_coer}) \\
\mathrm{AN}: & \mathsf{sockhandler} & \mapsto & (\mathsf{cookie}, \mathsf{int}) \\
\mathrm{UB}: & \mathsf{dispatch} & \mapsto & (\mathsf{req} \to \mathsf{handResult}, \lambda(s).\ldots(\mathbf{con}_{\mathsf{sockhandler}}\ \mathrm{h})(\mathrm{k},\ (\mathbf{con}_{\mathsf{req}}\ \mathrm{s}).\mathrm{buf},\ \mathrm{flags},\ (\mathsf{security\_info}\ ())\ldots) \\
\mathrm{AB}: & \mathsf{udp\_read}' & \mapsto & (\{\mathrm{sock}:\mathsf{sock}, \mathrm{buf}:\mathsf{buf}, \mathrm{sflags}:\mathsf{sflags}, \mathrm{cookie}:\mathsf{cookie}\} \to \mathsf{int}, \lambda(x)....\ ) \\
 & \mathsf{udp\_write}' & \mapsto & (\{\mathrm{sock}:\mathsf{sock}, \mathrm{buf}:\mathsf{buf}, \mathrm{sflags}:\mathsf{sflags}, \mathrm{cookie}:\mathsf{cookie}\} \to \mathsf{int}, \lambda(x)....\ ) \\
 & \mathsf{sockh\_coer} & \mapsto & ((\{\mathrm{sock}:\mathsf{sock}, \mathrm{buf}:\mathsf{buf}, \mathrm{sflags}:\mathsf{sflags}\} \to \mathsf{int}) \to \\
 & & & (\{\mathrm{sock}:\mathsf{sock}, \mathrm{buf}:\mathsf{buf}, \mathrm{sflags}:\mathsf{sflags}, \mathrm{cookie}:\mathsf{cookie}\} \to \mathsf{int}), \\
 & & & \lambda(f).\mathbf{if}\ \mathrm{f} = \mathsf{udp\_read}\ \mathbf{then}\ \mathsf{udp\_read}'\ \mathbf{else\ if}\ \mathrm{f} = \mathsf{udp\_write}\ \mathbf{then}\ \mathsf{udp\_write}'\ ) \\
 & \mathsf{sock\_coer} & \mapsto & \ldots \\
 & \mathsf{security\_info} & \mapsto & (\mathsf{int} \to \mathsf{cookie}, \lambda(x)....\ )
\end{array}$$

Figure 4: A sample update to the I/O kernel

coercions: $\mathbf{abs}_{\mathsf{t}}\ e$ converts e to type t (assuming e has the proper type τ), and $\mathbf{con}_{\mathsf{t}}\ e$ does the reverse. For example, dispatch in Figure 3 constructs a handResult value from -1 via the coercion $\mathbf{abs}_{\mathsf{handResult}}\ -1$. Conversely, to invoke h, it must be converted from type sockhandler via the coercion $(\mathbf{con}_{\mathsf{sockhandler}}\ h)\ (\ldots)$.

Type coercions serve two purposes operationally. First, they are used to prevent updates to some type t from occurring at a time when existing code still relies on the old representation. In particular, the presence of $\mathbf{con}_{\mathsf{t}}$ clearly indicates where the concrete representation of t is relied upon, and therefore can be used as part of a static or dynamic analysis to avoid an invalid update (§3.4).

Second, coercions are used to "tag" abstract data so it can be converted to a new representation should its type be updated. In particular, all instances of type t occurring in the program will have the form $\mathbf{abs}_{\mathsf{t}}\ e$. Therefore, given a user-provided transformer function c_{t} which converts from the old representation of t to the new, we can rewrite each instance at update-time to be $\mathbf{abs}_{\mathsf{t}}\ (c_{\mathsf{t}}\ e)$. This leads to a natural CBV evaluation strategy for transformers in conjunction with the rest of the program (§3.3).

The typing rules for coercions are simple:

$$\frac{\Gamma, \mathsf{t} \mapsto \tau\ \vdash\ e : \mathsf{t}}{\Gamma, \mathsf{t} \mapsto \tau\ \vdash\ \mathbf{con}_{\mathsf{t}}\ e : \tau} \qquad \frac{\Gamma, \mathsf{t} \mapsto \tau\ \vdash\ e : \tau}{\Gamma, \mathsf{t} \mapsto \mathbf{abs}_{\mathsf{t}}\ e : \mathsf{t}}$$

Because variables are explicitly annotated with types, inserting $\mathbf{abs}$ and $\mathbf{con}$ coercions automatically is a relatively straightforward application of subtyping-as-coercions [3]; further details will appear in the extended version [19].

3.2 Specifying Dynamic Updates

Formally, a dynamic update upd consists of four elements, written as a record with the labels UN, UB, AN, and AB:

- UN (Updated Named types) is a map from a type name to a pair of a type and an expression. Each entry, $\mathsf{t} \mapsto (\tau, \mathsf{c})$, specifies a named type to replace (t), its new representation type (τ), and a type transformer function c from the old representation type to the new.

- AN (Added Named types) is a map from type names t to type environments Ω, which are lists of type definitions. This is used to define new named types. The domain specifies types t in the existing program, and the new definitions are inserted just above t for scoping purposes.

- UB (Updated Bindings) is a map from top-level identifiers to a pair of a type and a *binding value* b_v, which is either a function $\lambda(x).e$ or a value v. These specify replacement **fun** and **var** definitions. Each entry $\mathsf{z} \mapsto (\tau, b_v)$ contains the binding to replace (z), the type of the new binding as it appears in the source program (τ), which must be equal to the current type, and the new binding (b_v).

- AB (Added Bindings) is a map from top-level identifiers z to pairs of types and binding values. These are used to specify new **fun** and **var** definitions. Because our runtime semantics permits top-level definitions to be mutually recursive, their ordering with the existing program is of no consequence (we could easily change the source language to support this).

As an example, say we wish to modify socket handling in Figure 2 to include a *cookie* argument for tracking security information (this was done at one point in Linux). This requires four changes: (1) we modify the definition of sockhandler to add the additional argument; (2) we modify the sock type to add new information (such as a destination address for which the cookie is relevant); (3) we modify existing handlers, like udp_read, to add the new functionality, and (4) we modify the dispatch routine to call the handler with the new argument. The user must provide functions to convert existing sock and sockhandler objects.

The update is shown in Figure 4. The UN component specifies the new definitions of sock and sockhandler, along with type transformer functions sockh_coer and sock_coer, which are defined in AB. The AN component defines the new type cookie = int, and that it should be inserted above the definition of sockhandler (which refers to it). Next, UB specifies a replacement dispatch function that calls the socket handler with the extra security cookie, which is acquired by calling a new function security_info.

The AB component specifies the definitions to add. First, it specifies new handler functions udp_read′ and udp_write′ to be used in place of the existing udp_read and udp_write functions. The reason they are defined here, and not in UB, is that the new versions of these functions have a different type than the old versions (they take an additional argument). So that code will properly call the new versions from now on, the sock_coer maps between the old ones and the new ones. Thus, existing datastructures that contain handler objects (such as the table used by getsockhandler) will be updated to refer to the new versions. If any code in the program called udp_read or udp_write directly, we could replace them with *stub* functions [12, 7], forwarding calls to the new version, and filling in the added argument. Thus, Proteus indirectly supports updating functions to new types for those rare occasions when this is necessary.

3.3 Operational Semantics

The operational semantics is shown in Figure 5, defined as a judgment of the form $\Omega; H, e \ \longrightarrow\ \Omega; H', e'$ over configurations consisting of a type environment Ω, a heap H and an expression e. To evaluate a program P, we compile it into a configuration $\Omega; H, e = \mathcal{C}(\emptyset; \emptyset; P)$, as shown in the Figure.

The type environment Ω defines a configuration's named types. Each type in $\mathrm{dom}(\Omega)$ maps to a single representation τ; some related approaches [6, 12] would permit t to map to a set of repre-

$\boxed{H, P \;\longrightarrow\; H', P'}$

(PROJ)	$H, \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_n = v_n\}.\mathrm{l}_i \;\longrightarrow\; H, v_i$	(REF)	$H, \mathbf{ref}\ v \;\longrightarrow\; (H, r \mapsto (\cdot, v)), r$
(LET)	$H, \mathbf{let}\ x : \tau = v\ \mathbf{in}\ e \;\longrightarrow\; H, e[x := v]$	(ASSIGN)	$(H, \rho \mapsto (\omega, e)), \rho := v \;\longrightarrow\; (H, \rho \mapsto (\omega, v)), v$
(DEREF)	$(H, \rho \mapsto (\omega, e)),\ !\rho \;\longrightarrow\; (H, \rho \mapsto (\omega, e)), \rho := e$	(CALL)	$(H, \mathsf{z} \mapsto (\tau, \lambda(x).e\ )), \mathsf{z}\ v \;\longrightarrow\; (H, \mathsf{z} \mapsto (\tau, \lambda(x).e\ ), e[v/x]$
(IF-T)	$H, \mathbf{if}\ v1 = v2\ \mathbf{then}\ e1\ \mathbf{else}\ e2 \;\longrightarrow$ $H, e1 \quad$ (where $v1 = v2$)	(IF-F)	$H, \mathbf{if}\ v1 = v2\ \mathbf{then}\ e1\ \mathbf{else}\ e2 \;\longrightarrow$ $H, e2 \quad$ (where $v1 \neq v2$)
(CONABS)	$H, \mathbf{con}_{\mathsf{t}}\ (\mathbf{abs}_{\mathsf{t}}\ v) \;\longrightarrow\; H, v$	(NO-UPDATE)	$H, \mathbf{update} \;\longrightarrow\; H, 1$

$\boxed{\Omega; H, P \;\longrightarrow\; \Omega; H', P'}$

(CONG) $$\frac{H, e \;\longrightarrow\; H', e'}{\Omega; H, \mathbb{E}[e] \;\longrightarrow\; \Omega; H, \mathbb{E}[e']}$$

(UPDATE) $$\frac{\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \mathbb{E}[1])}{\Omega; H, \mathbb{E}[\mathbf{update}] \xrightarrow{\mathrm{upd}} \mathcal{U}[\,\Omega\,]^{\mathrm{upd}}; \mathcal{U}[\,H\,]^{\mathrm{upd}}, \mathcal{U}[\,\mathbb{E}[0]\,]^{\mathrm{upd}}}$$

otherwise: $\Omega; H, \mathbb{E}[\mathbf{update}] \xrightarrow{\mathrm{upd}}$ **UpdEx**

$$\begin{array}{llll}
\text{Eval Contexts} & \mathbb{E} & ::= & \_ \mid \mathbf{let}\ x = \mathbb{E}\ \mathbf{in}\ e \mid \mathbb{E}\ e \mid v\ \mathbb{E} \mid \mathbb{E}.\mathrm{l} \mid \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_i = \mathbb{E}, \ldots, \mathrm{l}_n = e_n\} \mid \mathbf{ref}\ \mathbb{E} \mid\ !\mathbb{E} \mid \mathbb{E} := e \mid v := \mathbb{E} \\
 & & \mid & \mathbf{con}_{\mathsf{t}}\ \mathbb{E} \mid \mathbf{abs}_{\mathsf{t}}\ \mathbb{E} \mid \mathbf{if}\ \mathbb{E} = e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \mathbf{if}\ v = \mathbb{E}\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2
\end{array}$$

$$\begin{array}{llllllll}
\text{Heap addresses} & \rho & ::= & \mathsf{z} \mid r & \text{Heap type tags} & \omega & ::= & \tau \mid \cdot \\
\text{Heap bindings} & b & ::= & \lambda(x).e\ \mid e & \text{Heap values} & b_v & ::= & \lambda(x).e\ \mid v
\end{array}$$

$\boxed{\mathcal{U}[b]^{\mathrm{upd}}}$ (updating bindings)

$$\mathcal{U}[n]^{\mathrm{upd}} = n \qquad \mathcal{U}[x]^{\mathrm{upd}} = x \qquad \mathcal{U}[\mathsf{x}]^{\mathrm{upd}} = \mathsf{x}$$

$$\mathcal{U}[\mathbf{abs}_{\mathsf{t}}\ e]^{\mathrm{upd}} = \begin{cases} \mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ \mathcal{U}[e]^{\mathrm{upd}}) \\ \quad \text{if } \mathsf{t} \mapsto (\tau', \mathsf{c}) \in \mathrm{upd.UN} \\ \mathbf{abs}_{\mathsf{t}}\ \mathcal{U}[e]^{\mathrm{upd}} \\ \quad \text{otherwise} \end{cases}$$

For remaining b containing subterms $e_1, \ldots, e_n$, we inductively apply $\mathcal{U}[e_i]^{\mathrm{upd}}$

$\boxed{\mathcal{U}[H]^{\mathrm{upd}}}$ (updating the heap)

$$\mathcal{U}[\,\mathsf{z} = (\tau, b), H\,]^{\mathrm{upd}}$$
$$= \begin{cases} \mathsf{z} = (\tau', b'), \mathcal{U}[\,H\,]^{\mathrm{upd}} \\ \quad \text{if upd.UB}(\mathsf{z}) = (\tau', b') \\ \mathsf{z} = (\tau, \mathcal{U}[\,b\,]^{\mathrm{upd}}), \mathcal{U}[\,H\,]^{\mathrm{upd}} \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{U}[r = (\cdot, b), H]^{\mathrm{upd}} = (r = (\cdot, \mathcal{U}[b]^{\mathrm{upd}})), \mathcal{U}[H]^{\mathrm{upd}}$$

$$\mathcal{U}[\emptyset]^{\mathrm{upd}} = \mathrm{upd.AB}$$

$\boxed{\mathcal{U}[\Gamma]^{\mathrm{upd}}}$ (updating environments)

$$\mathcal{U}[\emptyset]^{\mathrm{upd}} = \mathrm{types}(\mathrm{upd.AB})$$

$$\mathcal{U}[x : \tau, \Gamma]^{\mathrm{upd}} = x : \tau, \mathcal{U}[\Gamma]^{\mathrm{upd}}$$

$$\mathcal{U}[r : \tau, \Gamma]^{\mathrm{upd}} = r : \tau, \mathcal{U}[\Gamma]^{\mathrm{upd}}$$

$$\mathcal{U}[\mathsf{z} : \tau, \Gamma]^{\mathrm{upd}} = \begin{cases} \mathsf{z} : \mathrm{heapType}(\tau'), \mathcal{U}[\Gamma]^{\mathrm{upd}} & \text{if upd.UB}(\mathsf{z}) = (\tau', \_) \\ \mathsf{z} : \tau, \mathcal{U}[\Gamma]^{\mathrm{upd}} & \text{otherwise} \end{cases}$$

$$\mathcal{U}[\mathsf{t} = \tau, \Gamma]^{\mathrm{upd}} = \begin{cases} \mathsf{t} = \mathrm{upd.AN}(\mathsf{t}), \Gamma' & \text{if } \mathsf{t} \in \mathrm{dom}(\mathrm{upd.AN}) \\ \Gamma' & \text{otherwise} \end{cases}$$

$$\text{where } \Gamma' = \begin{cases} \mathsf{t} = \tau', \mathcal{U}[\Gamma]^{\mathrm{upd}} \\ \quad \text{if upd.UN}(\mathsf{t}) = (\tau', \_) \\ \mathsf{t} = \tau, \mathcal{U}[\Gamma]^{\mathrm{upd}} \\ \quad \text{otherwise} \end{cases}$$

Auxiliary function to convert update types to heap types:

$$\begin{array}{lcll}
\mathrm{heapType}(\tau_1 \to \tau_2) & = & \tau_1 \to \tau_2 & \\
\mathrm{heapType}(\tau) & = & t\ \mathbf{ref} & \text{where } \tau \neq \tau_1 \to \tau_2
\end{array}$$

$\boxed{\mathcal{C}(\Omega; H; P) = \Omega; H; e}$ (compilation from programs to configurations)

$$\begin{array}{ll}
\mathcal{C}(\Omega; H; e) & = \Omega; H; e \\
\mathcal{C}(\Omega; H; \mathbf{type}\ \mathsf{t} = \tau\ \mathbf{in}\ P) & = \mathcal{C}(\Omega, \mathsf{t} = \tau; H; P) \\
\mathcal{C}(\Omega; H; \mathbf{fun}\ \mathsf{f}(x : \tau) : \tau' = e\ \mathbf{in}\ P) & = \mathcal{C}(\Omega; H, \mathsf{f} \mapsto (\tau \to \tau', \lambda(x).e\ ); P) \\
\mathcal{C}(\Omega; H; \mathbf{var}\ \mathsf{z} : \tau = v\ \mathbf{in}\ P) & = \mathcal{C}(\Omega; H, \mathsf{z} \mapsto (\tau, v); P)
\end{array}$$

Figure 5: Proteus Operational Semantics

sentations indexed by a version. We refer to our non-versioned approach as being *representation consistent* since a running program has but one definition of a type at any given time.

The heap H is a map from heap addresses ρ to pairs (ω, b), where ω is a type tag and b is a binding. We use the heap to store both mutable references created with $\mathbf{ref}$ and top-level bindings created with $\mathbf{var}$ and $\mathbf{fun}$; therefore ρ ranges over locations r and external names z. For normal references, the type tag ω is simply $\cdot$, indicating the absence of a type, and for identifiers, z, it is the type τ which appeared in the definition of z in the program. Type tags are used to type check new and replacement definitions provided by a dynamic update.

The operational semantics is given using evaluation contexts. All expressions e can be uniquely decomposed into $\mathbb{E}[e']$ for some evaluation context $\mathbb{E}$ and e', so the choice of rule is unambiguous. Next, we consider how our semantics expresses the interesting operations of dynamic updating: (1) updating top-level identifiers z with new definitions, and (2) updating type definitions t to have a different representation.

Replacing Top-level Identifiers. All top-level identifiers z from the source program are essentially statically-allocated reference cells. As a result, at update-time we can change z's binding in the heap, and afterward any code that accesses (dereferences) z will see the new version. However, our treatment of references differs somewhat from the standard one to facilitate dynamic updates.

First, since all functions are defined at the top-level, they are all references. However, rather than give top-level functions the type $(\tau_1 \rightarrow \tau_2)$ $\mathbf{ref}$, we simply give them type $\tau_1 \rightarrow \tau_2$, and perform the dereference as part of the (CALL) rule. This has the pleasant side effect of rendering top-level functions immutable during normal execution, as is typical, while still allowing them to be dynamically updated.

Second, as we have explained already, top-level bindings stored in the heap are paired with their type τ to be able to type check new and replacement bindings. Some formulations of dynamic linking define a *heap interface*, which maps variables z to types τ, but we find it more convenient to merge this interface into the heap itself.

Updating Data of Named Type. As mentioned in §3.1, Proteus uses coercions to identify where data of a type t is being used abstractly and concretely. The (CONABS) rule allows an abstract value $\mathbf{abs}_\mathsf{t}\ v$ to be used concretely when it is provided to $\mathbf{con}_\mathsf{t}$; this annihilates both coercions so that v can be used directly.

At update time, given a type transformation function c for an updated type t, we rewrite each occurrence $\mathbf{abs}_\mathsf{t}\ e$ to be $\mathbf{abs}_\mathsf{t}\ (\mathsf{c}\ e)$. Although only values can be stored in the heap initially, heap values of the form $\mathbf{abs}_\mathsf{t}\ v$, will be rewritten to be $\mathbf{abs}_\mathsf{t}\ (\mathsf{c}\ v)$, which is no longer a value. Therefore, $!r$ can potentially dereference an expression from the heap. While this is not a problem in itself, the transformation should be performed only once since it conceptually modifies the data in place. Therefore, the (DEREF) rule evaluates the contents of the reference and then *writes back* the result before proceeding.

Update Semantics. A dynamic update upd is modeled with a *labeled transition*, where upd labels the arrow. When no update is available, an $\mathbf{update}$ expression simply evaluates to 1, by (NO-UPDATE). Otherwise, (UPDATE) specifies that if upd is well-formed (by $\mathrm{updateOK}(-)$), the $\mathbf{update}$ evaluates to 0, and the program is updated by transforming the current type environment, heap, and expression according to $\mathcal{U}[-]^{\mathrm{upd}}$. When transforming expressions, $\mathcal{U}[-]^{\mathrm{upd}}$ applies type transformation functions to all $\mathbf{abs}_\mathsf{t}\ e$ expressions of a named type t that has been updated. When transforming the heap, it replaces top-level identifier definitions with their new versions, and adds all of the new bindings. When transforming Ω,[1] it replaces type definitions with their new versions, and inserts new definitions into specified slots in the list.

3.4 Update Safety

The conditions placed upon an update to guarantee type-safety are formally expressed by the precondition to the (UPDATE) rule given in Figure 7. The $\mathrm{updateOK}(-)$ predicate must determine that at the current point it is valid to apply this update—a dynamic property—and that the update is compatible with the program. The latter is a static property, in the sense that the information to perform it is available without recourse to the current state of the program, provided one has the original source and the updates previously applied.

```
let i = post (
  let u2 = update in
  let res = (con_sockhandler abs_sockhandler udp_read)
    {sock = $v_{\mathsf{sock}}$, buf = (con_req $v_{\mathsf{req}}$).buf,
     sflags = $v_{\mathsf{sflags}}$} in
  let u3 = update in res
) in loop i
```

Figure 6: Example active expression

Update Timing. To motivate the importance of timing, Figure 6 shows the expression fragment of our example program after some evaluation steps (the outer $\mathbf{let}\ \ \mathrm{i}\ =\ \ldots$ binding comes from loop and the argument to post is the partially-evaluated $\mathsf{dispatch}$ function). The $\mathbf{let}\ \ \mathrm{u2}\ =\ \mathbf{update}\ldots$ is in redex position, and suppose that the update described in §3.2 is available, which updates $\mathsf{sockhandler}$ to have an additional cookie argument, amongst other things. After applying the update, the user's type transformer $\mathsf{sockh\_coer}$ is inserted to convert $\mathsf{udp\_read}$, to be called next. Evaluating the transformer replaces $\mathsf{udp\_read}$ with $\mathsf{udp\_read}'$, and applying (CONABS) yields the expression $\mathsf{udp\_read}'(v_{\mathsf{sock}}, (\mathbf{con}_{\mathsf{req}}\ v_{\mathsf{req}}).\mathrm{buf}, v_{\mathsf{sflags}})$. But this is type-incorrect! The new version $\mathsf{udp\_read}'$ expects a fourth argument, but the existing call only passes three arguments.

The problem is that at the time of the update the program is evaluating the old version of $\mathsf{dispatch}$, which expects $\mathsf{sockhandler}$ values to take only three arguments. That is, this point in the program is not "con-t-free" since it will manipulate t values concretely. This fact is made manifest by the usage of $\mathbf{con}_{\mathsf{sockhandler}}$ in the active expression. In general, we say a configuration $\Omega; H, e$ is *con-free* for an update upd if for all named types t that the update will change, $\mathbf{con}_\mathsf{t}$ is not a subexpression of the active expression e or any of the bindings in the heap that are not replaced by the update. We write this as $\mathrm{conFree}[\,-\,]^{\mathrm{upd}}$; the definition is given in Figure 7.

Two other points are worth noting. First, the active expression only uses instances of $\mathsf{handResult}$ abstractly after the update (passing them to post), and so should we wish, $\mathsf{handResult}$ could be modified (assuming that post is modified as well). Second, the given update is only unsafe at the first $\mathbf{update}$ point; it could be safely applied at $\mathbf{let}\ \mathrm{u3}\ =\ \mathbf{update}\ldots$, since at that point there are no further concrete uses of any of the changed types.

[1] Ω is just a simpler form of Γ, so $\mathcal{U}[\Omega]^{\mathrm{upd}}$ is defined by $\mathcal{U}[\Gamma]^{\mathrm{upd}}$ shown in the Figure, with the exception that $\mathcal{U}[\emptyset]^{\mathrm{upd}} = \emptyset$ (i.e. the types of the new bindings are not added).

$$\begin{array}{l}
\mathrm{updateOK}(\mathrm{upd}, \Omega, H, e) = \\
\quad \mathrm{conFree}[\,H\,]^{\mathrm{upd}} \wedge \\
\quad \mathrm{conFree}[\,e\,]^{\mathrm{upd}} \wedge \\
\quad \Gamma = \mathrm{types}(H) \wedge \\
\quad \vdash\ \mathcal{U}[\Omega]^{\mathrm{upd}} \wedge \\
\quad \forall \mathsf{t} \mapsto (\tau, \mathsf{c}) \in \mathrm{upd.UN}.\quad \mathcal{U}[\Omega, \Gamma]^{\mathrm{upd}}\ \vdash\ \mathsf{c} : \Omega(\mathsf{t}) \to \tau \wedge \\
\quad \forall \mathsf{z} \mapsto (\tau, b_v) \in \mathrm{upd.UB}.\quad \mathcal{U}[\Omega, \Gamma]^{\mathrm{upd}}\ \vdash\ b_v : \tau \wedge \\
\qquad b_v = e \Rightarrow \exists \tau'.\ \Gamma(\mathsf{z}) = \tau'\ \mathbf{ref} \wedge \\
\qquad \mathrm{heapType}(\tau) = \Gamma(\mathsf{z}) \wedge \\
\quad \forall \mathsf{z} \mapsto (\tau, b_v) \in \mathrm{upd.AB}.\quad \mathcal{U}[\Omega, \Gamma]^{\mathrm{upd}}\ \vdash\ b_v : \tau
\end{array}$$

$$\begin{array}{l}
\mathrm{conFree}[\,\mathsf{z} = (\tau, b), H\,]^{\mathrm{upd}} \\
\qquad = \mathrm{conFree}[\,H\,]^{\mathrm{upd}} \wedge \begin{cases} \mathbf{tt} & \text{if } \mathsf{z} \in \mathrm{dom}(\mathrm{upd.UB}) \\ \mathrm{conFree}[\,b\,]^{\mathrm{upd}} & \text{otherwise} \end{cases} \\
\mathrm{conFree}[\,r = (\cdot, e), H\,]^{\mathrm{upd}} = \mathrm{conFree}[\,e\,]^{\mathrm{upd}} \wedge \mathrm{conFree}[\,H\,]^{\mathrm{upd}} \\
\mathrm{conFree}[\,n\,]^{\mathrm{upd}} = \mathbf{tt} \qquad \mathrm{conFree}[\,x\,]^{\mathrm{upd}} = \mathbf{tt} \\
\mathrm{conFree}[\,\mathbf{con}_{\mathsf{t}}\ e\,]^{\mathrm{upd}} = \begin{cases} \mathbf{ff} & \text{if } \mathsf{t} \in \mathrm{dom}(\mathrm{upd.UN}) \\ \mathbf{tt} & \text{otherwise} \end{cases}
\end{array}$$

For remaining b containing subterms $e_1, \ldots, e_n$:
$\mathrm{conFree}[\,e\,]^{\mathrm{upd}} = \bigwedge_i e_i$

Figure 7: Proteus $\mathrm{updateOK}(-)-$ **check and auxiliary definitions**

Update well-formedness. The conditions for update well-formedness are part of the $\mathrm{updateOK}(-)$ predicate in Figure 7. In addition to checking proper timing with the $\mathrm{conFree}[\,-\,]$ checks, this predicate ensures that type-safety is maintained following the addition or replacement of code and types. The $\mathrm{types}(H)$ predicate extracts all of the type tags from H and constructs a suitable Γ for typechecking the new or replacement bindings. Since heap objects are stored with their declared type τ, if they are non-functions, in Γ they are given type τ $\mathbf{ref}$. Next, the updated type environment $\mathcal{U}[\Omega]^{\mathrm{upd}}$ is checked for well-formedness. Then, using the updated Ω and Γ, we check that the type transformer functions, replacement bindings and new bindings are all well-typed. The important fact to notice about these type-checks is that they only apply to expressions contained in the update. Only the types of existing code, not the code itself, are needed. The extra checks in the replacement bindings clause ensure that a heap cell doesn't change between a function an a reference cell and that the type of the cell is preserved.

3.5 Properties

Proteus enjoys an essentially standard type safety result.

Theorem 3.1 (Type safety). *If* $\vdash\ \Omega; H, e : \tau$ *then either*

1. $\Omega; H, e \longrightarrow \Omega'; H', e'$ *and* $\vdash\ \Omega'; H', e' : \tau$ *for some* Ω', H' *and* e'*;*
2. $\Omega; H, e \longrightarrow$ ***UpdEx****; or*
3. e *is a value*

This theorem states that a well-typed program is either a value, or is able to reduce (and remain well-typed), or terminates abruptly due to a failed dynamic update. The most interesting case in proving type preservation is the $\mathbf{update}$ rule, for which we must prove a lemma that well-formed and well-timed updates lead to well-typed programs:

Lemma 3.2 ($\mathcal{U}[\,-\,]^{-}$ **preserves types of programs).** *Given* $\vdash\ \Omega; H, e$ *and an update,* upd*, for which we have* $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, e)$*, then* $\vdash\ \mathcal{U}[\,\Omega\,]^{\mathrm{upd}}; \mathcal{U}[\,H\,]^{\mathrm{upd}}, \mathcal{U}[\,e\,]^{\mathrm{upd}} : \tau$*.*

Proofs will appear in the extended version [19].

4. ASSURING PROPER UPDATE TIMING

Type safety in the system we have described so far is predicated on a dynamic con-free check. Unfortunately, this check could be difficult to implement, and moreover could fail if the update is applied at a bad time. The main result of this section is that given an unannotated Proteus program (no $\mathbf{abs}_{\mathsf{t}}$ or $\mathbf{con}_{\mathsf{t}}$ or $\mathbf{update}$ expressions), we can statically infer all of the program points that are con-free with respect to all types in the program, and insert $\mathbf{update}$ there. Thus, we eliminate the need for $\mathrm{conFree}[\,-\,]$[2] and we make the update behavior of the program easier to reason about, since many acceptable update points are known statically. In this section, we present our *updateability analysis* as a type and inference system, establish its soundness, and present some preliminary performance measurements on large C programs which show that the analysis is quite efficient.

4.1 Capabilities

Our goal is to define and enforce a notion of con-freeness for a program, rather than a program state. In other words, we wish to determine for a particular $\mathbf{update}$ whether it will be acceptable to update some type t. An update to t will be unacceptable if an occurrence of $\mathbf{con}_{\mathsf{t}}$ exists in any old code evaluated in the continuation of the $\mathbf{update}$. Assuming we can discover all such occurrences of $\mathbf{con}_{\mathsf{t}}$, we could annotate $\mathbf{update}$ with those types t, indicating that they should not be updated. We call this annotation Δ a *capability*, since it serves as a bound on what types may be used concretely in the continuation of an $\mathbf{update}$. That is, any code following an $\mathbf{update}$ must type check using Γ restricted to those types listed in the capability. Since an $\mathbf{update}$ could change only types not in the capability, we are certain that existing code will remain type-safe. As a consequence, if we can type-check our program containing only $\mathbf{update}$ points with empty annotations, we can be sure that no update will fail due to bad timing.

4.2 Typing

We define a *capability* type system that tracks the capability at each program point to ensure that $\mathbf{update}$s are annotated soundly. First we change slightly the grammar for types:

$$\begin{array}{rrcl}
\text{Capabilities} & \Delta & ::= & \{\mathsf{t}_1, \ldots \mathsf{t}_n\} \mid \Delta \cap \Delta \\
\text{Updateability} & \mu & ::= & \mathsf{U} \mid \mathsf{N} \\
\text{Types} & \tau & ::= & \cdots \mid \tau \xrightarrow{\mu; \Delta; \Delta'} \tau
\end{array}$$

We assume that $\mathbf{update}$ occurrences are annotated with some Δ, and function definitions are annotated with some $\mu; \Delta; \Delta'$ (§4.4 explains how to infer such annotations).

[2]We may wish to combine it with the static analysis. See §4.3.

$\boxed{\Delta; \Gamma \vdash_\mu e : \tau; \Delta'}$

$$\Delta; \Gamma \vdash_\mu n : \mathsf{int}; \Delta \quad \text{(A.Int)} \qquad \Delta; \Gamma, x : \tau \vdash_\mu x : \tau; \Delta \quad \text{(A.Var)} \qquad \Delta; \Gamma, \mathsf{x} : \tau \vdash_\mu \mathsf{x} : \tau; \Delta \quad \text{(A.XVar)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_\mu e_1 : \tau_1'; \Delta' \\ \Delta'; \Gamma, x : \tau_1 \vdash_\mu e_2 : \tau_2; \Delta''\end{array}}{\Delta; \Gamma \vdash_\mu \mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2 : \tau_2; \Delta''} \ \text{(A.Let)} \qquad \frac{\begin{array}{l}\Delta; \Gamma \vdash_\mu e_1 : \tau_1 \overset{\hat{\mu};\hat{\Delta};\hat{\Delta}'}{\longrightarrow} \tau_2; \Delta' \\ \Delta'; \Gamma \vdash_\mu e_2 : \tau_1; \Delta'' \qquad \Delta''' \subseteq \Delta'' \\ (\hat{\mu} = \mathsf{U}) \Rightarrow (\mu = \mathsf{U} \wedge \Delta''' \subseteq \hat{\Delta}')\end{array}}{\Delta; \Gamma \vdash_\mu e_1\ e_2 : \tau_2; \Delta'''} \ \text{(A.App)}$$

$$\frac{\Delta_i; \Gamma \vdash_\mu e_{i+1} : \tau_{i+1}; \Delta_{i+1} \qquad i \in 1..(n-1) \qquad n \geq 0}{\Delta_0; \Gamma \vdash_\mu \{\mathrm{l}_1 = e_1, \ldots, 1_n = e_n\} : \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\}; \Delta_n} \ \text{(A.Rec)} \qquad \frac{\Delta; \Gamma \vdash_\mu e : \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\}; \Delta'}{\Delta; \Gamma \vdash_\mu e.\mathrm{l}_i : \tau_i; \Delta'} \ \text{(A.Proj)}$$

$$\frac{\Delta; \Gamma \vdash_\mu e : \tau; \Delta'}{\Delta; \Gamma \vdash_\mu \mathbf{ref}\ e : \tau\ \mathbf{ref}; \Delta'} \ \text{(A.Ref)} \qquad \frac{\Delta; \Gamma \vdash_\mu e : \tau\ \mathbf{ref}; \Delta'}{\Delta; \Gamma \vdash_\mu\ !e : \tau; \Delta'} \ \text{(A.Deref)} \qquad \frac{\begin{array}{c}\Delta; \Gamma \vdash_\mu e_1 : \tau\ \mathbf{ref}; \Delta' \\ \Delta'; \Gamma \vdash_\mu e_2 : \tau; \Delta''\end{array}}{\Delta; \Gamma \vdash_\mu e_1 := e_2 : \mathsf{unit}; \Delta''} \ \text{(A.Assign)}$$

$$\frac{\Delta; \Gamma \vdash_\mu e : \mathsf{t}; \Delta' \qquad \Gamma \upharpoonleft_{\Delta'} (\mathsf{t}) = \tau}{\Delta; \Gamma \vdash_\mu \mathbf{con}_{\mathsf{t}}\ e : \tau; \Delta'} \ \text{(A.Con)} \qquad \frac{\Delta; \Gamma \vdash_\mu e : \tau; \Delta' \qquad \Gamma(\mathsf{t}) = \tau}{\Delta; \Gamma \vdash_\mu \mathbf{abs}_{\mathsf{t}}\ e : \mathsf{t}; \Delta'} \ \text{(A.Abs)}$$

$$\frac{\Delta' \subseteq \Delta}{\Delta; \Gamma \vdash_{\mathsf{U}} \mathbf{update}^{\Delta'} : \mathsf{unit}; \Delta'} \ \text{(A.Update)} \qquad \frac{\Delta; \Gamma \vdash_\mu e : \tau'; \Delta'' \qquad \Gamma \vdash \tau' <: \tau \qquad \Delta' \subseteq \Delta''}{\Delta; \Gamma \vdash_\mu e : \tau; \Delta'} \ \text{(A.Sub)}$$

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash_\mu e : \tau; \Delta_1 \qquad \Delta_1; \Gamma \vdash_\mu e' : \tau; \Delta_2 \\ \Delta_2; \Gamma \vdash_\mu e_1 : \tau'; \Delta_3 \qquad \Delta_2; \Gamma \vdash_\mu e_2 : \tau'; \Delta_4\end{array}}{\Delta; \Gamma \vdash_\mu \mathbf{if}\ e = e'\ \mathbf{then}\ e_1 \mathbf{else}\ e_2 : \tau'; \Delta_3 \cup \Delta_4} \ \text{(A.If)} \qquad \frac{\begin{array}{l}\Delta' \subseteq \Delta \\ \Delta'; \Gamma \vdash_{\mathsf{U}} e_1 : \tau'; \Delta_1 \qquad \Delta; \Gamma \vdash_{\mathsf{U}} e_2 : \tau'; \Delta_2\end{array}}{\Delta; \Gamma \vdash_{\mathsf{U}} \mathbf{if}\ \mathbf{update}^{\Delta'} = 0\ \mathbf{then}\ e_1 \mathbf{else}\ e_2 : \tau'; \Delta_1 \cup \Delta_2} \ \text{(A.If.Update)}$$

$\boxed{\Gamma \vdash_P P : \tau}$

$$\frac{\begin{array}{l}\Gamma, \mathsf{t} = \tau' \vdash_P P : \tau \\ \Gamma \vdash \tau'\ \mathrm{OK}\end{array}}{\Gamma \vdash_P \mathbf{type}\ \mathsf{t} = \tau'\ \mathbf{in}\ P : \tau} \ \text{(A.Type)} \qquad \frac{\begin{array}{l}\Gamma' = \Gamma, z : \tau_1 \overset{\mu;\Delta;\Delta'}{\longrightarrow} \tau_2 \\ \Delta; \Gamma', x : \tau_1 \vdash_\mu e : \tau_2; \Delta' \qquad \Gamma' \vdash_P P : \tau\end{array}}{\Gamma \vdash_P \mathbf{fun}\ \mathsf{z}^{\mu;\Delta;\Delta'}(x : \tau_1) : \tau_2 = e\ \mathbf{in}\ P : \tau} \ \text{(A.LetF)}$$

$$\frac{\emptyset; \Gamma \vdash_{\mathsf{N}} v : \tau'; \emptyset \qquad \Gamma, \mathsf{z} : \tau'\ \mathbf{ref} \vdash_P P : \tau}{\Gamma \vdash_P \mathbf{var}\ \mathsf{z} : \tau' = v\ \mathbf{in}\ P : \tau} \ \text{(A.LetV)} \qquad \frac{\Delta; \Gamma \vdash_{\mathsf{U}} e : \tau; \Delta'}{\Gamma \vdash_P e : \tau} \ \text{(A.Exp)}$$

$\boxed{\Gamma \vdash \tau <: \tau'}$

$$\Gamma \vdash \mathsf{int} <: \mathsf{int} \quad \text{(A.Sub.Int)} \qquad \Gamma, \mathsf{t} = \tau \vdash \mathsf{t} <: \mathsf{t} \quad \text{(A.Sub.Type)} \qquad \frac{\Gamma \vdash \tau <: \tau' \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash \tau'\ \mathbf{ref} <: \tau\ \mathbf{ref}} \ \text{(A.Sub.Ref)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \tau_2 <: \tau_1 \qquad \Gamma \vdash \tau_1' <: \tau_2' \\ \Delta_2' \subseteq \Delta_1' \qquad (\mu_2 = \mathsf{U}) \Rightarrow (\mu_1 = \mathsf{U})\end{array}}{\Gamma \vdash \tau_1 \overset{\mu_1;\Delta_1;\Delta_1'}{\longrightarrow} \tau_1' <: \tau_2 \overset{\mu_2;\Delta_2;\Delta_2'}{\longrightarrow} \tau_2'} \ \text{(A.Sub.Fun)} \qquad \frac{\tau_1 <: \tau_1' \qquad i \in 1..n}{\{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\} <: \{\mathrm{l}_1 : \tau_1', \ldots, \mathrm{l}_n : \tau_n'\}} \ \text{(A.Sub.Rec)}$$

$\boxed{\Gamma \vdash b : \tau}$

$$\frac{\Delta; \Gamma, x : \tau \vdash_\mu e : \tau'; \Delta'}{\Gamma \vdash \lambda(x).e : \tau \overset{\mu;\Delta;\Delta'}{\longrightarrow} \tau'} \ \text{(A.Bind.Fun)} \qquad \frac{\emptyset; \Gamma \vdash_{\mathsf{N}} e : \tau; \emptyset}{\Gamma \vdash e : \tau} \ \text{(A.Bind.Expr)}$$

$\boxed{\Gamma \vdash \Omega; H, e : \tau}$ $\qquad$ $\boxed{\Omega; \Phi \vdash H}$

$$\frac{\begin{array}{c}\vdash \Omega \qquad \Omega; \Phi \vdash H \\ \Delta; \Gamma, \Omega, \Phi \vdash_{\mathsf{U}} e : \tau; \Delta'\end{array}}{\Gamma \vdash \Omega; H, e : \tau} \qquad \frac{\begin{array}{l}\mathrm{dom}(\Phi) = \mathrm{dom}(H) \\ \forall \mathsf{z} \mapsto (\tau, e) \in H.\ \emptyset; \Phi \vdash_{\mathsf{N}} e : \tau; \emptyset \wedge \Phi(\mathsf{z}) = \tau\ \mathbf{ref} \\ \forall \mathsf{z} \mapsto (\tau, \lambda(x).e\ ) \in H.\ \Gamma \vdash \lambda(x).e : \tau \wedge \Phi(\mathsf{z}) = \tau \\ \forall r \mapsto (\cdot, e) \in H.\ \emptyset; \Phi \vdash_{\mathsf{N}} e : \tau; \emptyset \wedge \Phi(r) = \tau\ \mathbf{ref}\end{array}}{\Omega; \Phi \vdash H}$$

Type environment typing, $\vdash \Omega$, has a standard definition, essentially ensuring that there are no free type names.

Figure 8: Capability typing for Proteus programs

The type system is given in Figure 8, with judgments $\Delta; \Gamma \vdash_\mu e : \tau; \Delta'$ for expressions, and $\Gamma \vdash_P P : \tau$ for programs. For expression typings, Δ is the capability before e is evaluated, and Δ' is the capability afterward. Each rule is actually a family of rules parameterized by an *updateability* μ which indicates whether a dynamic update may be performed while evaluating the given expression. This is used to rule out dynamic updates in undesirable contexts, as we explain in the next subsection.

Typing **update** *and* $\mathbf{con}_\mathsf{t}\ e$. The capability Δ' on $\mathbf{update}^{\Delta'}$ lists those types that *must not change* due to a dynamic update. Since any other type could change, the (A.Update) rule assumes that the capability can be at most Δ' following the update. The (A.Con) rule states that to concretely access a value of type t, the type t must be defined in Γ, restricted to types listed in capability Δ'. Thus, to type check $\mathsf{dispatch}$ in Figure 3, we must annotate the **update** in **let** u1 = **update in** ... with a capability $\{\mathsf{fdtype}, \mathsf{req}, \mathsf{sockhandler}\}$, since these types are used by **con** expressions following that point within $\mathsf{dispatch}$. By the same reasoning, the annotation on the u2 update would be $\{\mathsf{req}, \mathsf{sockhandler}\}$, and the u3 update annotation can be empty. The (A.Update) rule requires updateability U; updates cannot be performed in a non-updateable (N) context.

(A.Update) assumes that any **update** could result in an update at run time. However, we can make our analysis more precise by incorporating the effects of a dynamic check. In particular, (A.If.Update) checks **if** when the guard is $\mathbf{update}^{\Delta'} = 0$, which will be true only if an update takes place at run time. Therefore, the input capability of e_1 is Δ', while the input capability of e_2 is Δ.

Function calls. Function types have an annotation $\mu; \Delta; \Delta'$, where Δ is the input capability and Δ' is the output capability. If calling a function could result in an update, the updateability μ must be U. Thus, in Figure 3, using the annotations on **update** mentioned above, the type for $\mathsf{dispatch}$ would be $\mathsf{req} \xrightarrow{\mathsf{U};\Delta;\emptyset} \mathsf{handResult}$, for some Δ satisfying the condition

$$\{\mathsf{req}, \mathsf{fdtype}, \mathsf{sockhandler}\} \subseteq \Delta$$

In the (A.Update) rule, the output capability is bounded by the annotation on the **update**; in the (A.App) rule, the caller's output capability Δ''' is bounded by the callee's output capability $\hat{\Delta}'$ for the same reason. This is expressed in the conditional constraint $(\hat{\mu} = \mathsf{U}) \Rightarrow (\mu = \mathsf{U} \ldots$, which also indicates the caller's updateability μ must allow the update. If the called function cannot perform an update, then the caller's capability and updateability need not be restricted. We will take advantage of this fact in how we define type transformer functions, described below.

A perhaps unintuitive effect of (A.App) is that a function f's output capability must mention those types used concretely by its callers following calls to f. To illustrate, say we modify the type of post in Figure 2 to be $\mathsf{int} \to \mathsf{int}$ rather than $\mathsf{handResult} \to \mathsf{int}$. As a result, loop would have to concretize the $\mathsf{handResult}$ returned by $\mathsf{dispatch}$ before passing it to post, resulting in the code

$$\mathbf{let}\ i = \mathsf{post}\ (\mathbf{con}_{\mathsf{handResult}}\ (\mathsf{dispatch}\ \mathit{req}))...$$

To type check the **con** would require the output capability of $\mathsf{dispatch}$ to include $\mathsf{handResult}$, which in turn would require that $\mathsf{handResult}$ appear in the capabilities of each of the **update** points in $\mathsf{dispatch}$, preventing $\mathsf{handResult}$ from being updated.

Another unintuitive aspect of (A.App) is that to call a function, we would expect that the caller's capability must be compatible with (i.e., must be a superset of) the function's input capability, but this condition is not necessary. Instead, the type system assumes that all calls will be to a function's most recent version, which will be guaranteed at update-time to be compatible with the program's type definitions (see §4.3). In effect, the type system approximates, for a given update point, the concretions in code that an updating function could *return to*, but not code it will later call, which is guaranteed to be safe. This is critical to avoid restricting updates unnecessarily.

$$\begin{array}{l}
\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \Delta) = \\
\quad \Gamma = \mathrm{types}(H) \wedge \\
\quad \boxed{\mathrm{dom}(\Delta) \cap \mathrm{dom}(\mathrm{upd.UN}) = \emptyset \wedge \mathrm{bindOK}[\,\Gamma\,]^{\mathrm{upd}}}^{(a)} \wedge \\
\quad \vdash\ \mathcal{U}[\Omega]^{\mathrm{upd}} \wedge \\
\quad \forall \mathsf{t} \mapsto (\tau, \mathsf{c}) \in \mathrm{upd.UN}. \\
\qquad \boxed{\emptyset; \mathcal{U}[\Omega, \Gamma]^{\mathrm{upd}} \vdash_{\mathsf{N}} \mathsf{c} : \Omega(\mathsf{t}) \xrightarrow{\mathsf{N};\Delta';\Delta''} \tau; \emptyset}^{(b)} \wedge \\
\quad \forall \mathsf{z} \mapsto (\tau, b_v) \in \mathrm{upd.UB}.\quad \mathcal{U}[\Omega, \Gamma]^{\mathrm{upd}} \vdash b_v : \tau \wedge \\
\qquad b_v = e \Rightarrow \exists \tau'.\, \Gamma(\mathsf{z}) = \tau'\ \mathbf{ref} \\
\qquad \boxed{\mathcal{U}[\Omega]^{\mathrm{upd}} \vdash \mathrm{heapType}(\tau) <: \Gamma(\mathsf{z})}^{(c)} \wedge \\
\quad \forall \mathsf{z} \mapsto (\tau, b_v) \in \mathrm{upd.AB}.\quad \mathcal{U}[\Omega, \Gamma]^{\mathrm{upd}} \vdash b_v : \tau
\end{array}$$

Figure 9: Precondition for $\mathbf{update}^{\Delta}$ operational rule

Other Rules. Unlike $\mathbf{con}_\mathsf{t}\ e$ expressions, $\mathbf{abs}_\mathsf{t}\ e$ expressions place no constraint on the capability. This is because a dynamic update that changes the definition of t from τ to τ' requires a well-typed type transformer c to rewrite $\mathbf{abs}_\mathsf{t}\ e$ to $\mathbf{abs}_\mathsf{t}\ (\mathsf{c}(e))$, which will always be well-typed assuming suitable restrictions on c described in the next subsection.

Turning to program typing, the (A.type) rule adds a new type definition to the global environment, and the (A.LetF) rule simply checks the function's body using the capabilities and updateability defined by its type. Since v is a value and cannot effect an update, the (A.LetV) rule checks it with an empty capability Δ and updateability N. Finally, the (A.Exp) rule type checks the body of the program using an arbitrary capability and updateability U to allow updates.

Allowing subtyping adds flexibility to programs and to their updates. The interesting rule is (A.Sub.Fun) for function types. Output capabilities are contravariant: if a caller expects a function's output capability to be Δ, it will be a conservative approximation if the function's output capability is actually larger. There is no restriction on the input capability for updateable functions, since we always assume them to be compatible with the current set of type definitions for the program. A function that performs no updates can be a subtype of one that does, assuming they have compatible capabilities. During type checking, subtyping is invoked by the (A.Sub) rule, which can simultaneously coarsen (makes smaller) the output capability Δ. Intuitively, this is always sound because it will put a stronger restriction on limits imposed by prior updates.

We need additional typing judgments to ensure the well-formedness of configurations and the consistency of heaps. These properties are expressed by the $\Gamma \vdash \Omega; H, e : \tau$ and $\Omega; \Phi \vdash H$ judgments respectively. An additional judgment $\Gamma \vdash b : \tau$ is used for typing heap bindings.

4.3 Operational Semantics

The dynamic semantics from Figure 5 remains unchanged with the exception of the $\mathrm{updateOK}(-)$ predicate for (UPDATE), shown in Figure 9. The two timing-related changes are highlighted by the boxes labeled (a) and (b). First, Δ, taken from $\mathbf{update}^{\Delta}$, replaces e as the last argument. This is used in (a) to syntactically check that no types mentioned in Δ are changed by the update. Change (a) also refers to $\mathrm{bindOK}[\,\Gamma\,]^{\mathrm{upd}}$ to ensure that all top-level bindings in the heap that use types in $\mathrm{upd.UN}$ concretely, as indicated by their input capability, are also replaced (the definition is straightforward and not shown). This allows the type system to assume that calling a function is always safe, and need not impact its capability. Together, these two checks are analogous to the con-free dynamic check to ensure proper timing.[3]

Type transformers provided for updated types must not, when inserted, violate assumptions made by the updateability analysis. In particular, each $\mathbf{abs}_{\mathsf{t}}\ e$ appearing in the program type checks with some capability prior to an update, i.e., $\Delta; \Gamma \vdash_\mu \mathbf{abs}_{\mathsf{t}}\ e : \tau; \Delta'$. If type t is updated with transformer c, we require $\Delta; \Gamma \vdash_\mu \mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ e) : \tau; \Delta'$. Since $\mathbf{abs}_{\mathsf{t}}\ e$ expressions could be anywhere at update time, and could require a different capability Δ to type check, condition (b) conservatively mandates that transformers c must check in an empty capability, and may not perform updates (c's type must have updateability N). These conditions are sufficient to ensure type correctness. Otherwise, a transformer function c is like any other function. For example, if it uses some type t concretely, it will have to be updated if t is updated. The ramifications of this fact are explored in §5.

Finally, we allow bindings to updated at subtypes, as indicated by condition (c). This is crucial for functions, because as they evolve over time, it is likely that their capabilities will change depending on what functions they call or what types they manipulate. Fortunately, we can always update an existing function with a function that causes no updates. In particular, say function f has type $\mathsf{t} \xrightarrow{\mathsf{U};\{\mathsf{t},\mathsf{t}'\};\{\mathsf{t},\mathsf{t}'\}} \mathsf{t}'$, where $\mathsf{t} = \mathsf{int}$ and $\mathsf{t}' = \mathsf{int}$. Say we add a new type $\mathsf{t}'' = \mathsf{int}$ and want to change f to be the following:

$$\begin{array}{l}\mathbf{fun}\ \mathsf{f}(x : \mathsf{t}) : \mathsf{t}' = \\ \quad \mathbf{let}\ y = \mathbf{con}_{\mathsf{t}''}\ \mathbf{abs}_{\mathsf{t}''}\ 1\ \mathbf{in} \\ \quad \mathbf{let}\ z = \mathbf{con}_{\mathsf{t}}\ x\ \mathbf{in}\ \mathbf{abs}_{\mathsf{t}'}\ z + y\end{array}$$

The expected type of this function would be $\mathsf{t} \xrightarrow{\mathsf{N};\{\mathsf{t},\mathsf{t}''\};\{\mathsf{t},\mathsf{t}''\}} \mathsf{t}'$, but it could just as well be given type $\mathsf{t} \xrightarrow{\mathsf{U};\{\mathsf{t},\mathsf{t}',\mathsf{t}''\};\{\mathsf{t},\mathsf{t}',\mathsf{t}''\}} \mathsf{t}'$, which is a subtype of the original, and thus an acceptable replacement. Replacements that contain $\mathbf{update}$ or call functions that contain $\mathbf{update}$ are more rigid in their capabilities. We expect that experimenting with an implementation of Proteus will help us understand how this fact affects the program's ability to update itself over time.

4.4 Inference

It is straightforward to construct a type inference algorithm for our capability type system. In particular, we simply extend the definition of capability Δ to include variables φ and updateability μ to include variables ε. Then we take a normal Proteus program and decorate it with fresh variables on each function definition, function type, and $\mathbf{update}$ expression in the program. We also adjust the rules to use an algorithmic treatment of subtyping, eliminating the separate (A.Sub) rule and adding subtyping preconditions to the (A.App) and (A.Assign) rules as is standard. This allows the judgment to be syntax-directed.

As a result of these changes, conditions imposed on capability variables by the typing and subtyping rules become simple set and term constraints [11]. A solution consists of a substitution σ, which is a map from variables φ to capabilities $\{\mathsf{t}_1, \ldots, \mathsf{t}_n\}$, and from variables ε to updateabilities either U or N. The constraints can be solved efficiently with standard techniques in time $O(n^3)$ in the worst case (but far better on average), where n is the number of variables φ or set constants $\{\cdot\}$ mentioned in the constraints. The constraints have the following forms (shown with the rules that induce them):

(1)	$\Gamma \vdash \tau_1 <: \tau_2$	(A.Sub)
(2)	$\varepsilon = \mathsf{U}$	(A.Update)
(3)	$(\hat{\mu} = \mathsf{U}) \Rightarrow C$	(A.App), (A.Sub)
(4)	$\varphi \subseteq \Delta$	(A.Update), (A.App)
(5)	$\mathsf{t} \in \Delta$	(A.Con)

For updateabilities, we want the *greatest* solution; that is, we want to allow as many functions as possible to perform updates (with an unannotated program, this will vacuously be the case). For the capabilities, we are interested in the *least* solution, in which we minimize the set to substitute for φ, since it will permit more dynamic updates. For $\mathbf{update}^\varphi$, a minimal φ imposes fewer restrictions on the types that may be updated at that point. For functions $\tau \xrightarrow{\varepsilon,\varphi;\varphi'} \tau'$, the smaller φ' imposes fewer constraints on subtypes, which in turn permits more possible function replacements. When using inference for later versions of a program, we must introduce subtyping constraints between an old definition's (solved) type and the new version's to-be-inferred one. This ensures that the new definition will be a suitable dynamic replacement for the old one.

Inferring $\mathbf{update}$ *points.* Using the inference system, we can take a program that is absent of $\mathbf{update}$ expressions, and infer places to insert them that are con-free for all types. Define a source-to-source rewriting function $\mathsf{rewrite} : P \to P'$ that inserts $\mathbf{update}^\varphi$ at various locations throughout the program. Then we perform inference, and remove all occurrences of $\mathbf{update}^\varphi$ for which φ is not $\emptyset$ (call these *universal* update points as they do not restrict the types that may be updated). In the simplest case, the rewriting function could insert $\mathbf{update}^\varphi$ just before a function is about to return. Adding more points implies greater availability, but longer analysis times and more runtime overhead. Intuitively, this approach will converge because the annotations φ on update points are unaffected by those on other update points; rather they are only impacted by occurrences of $\mathbf{con}$ in their continuations.

Preliminary Implementation. We are currently implementing Proteus for C programs. We use CIL (*C Intermediate Language*) [16] for C code parsing and source-to-source transformation, and BANSHEE [14] for constraint solving. We have implemented the updateability analysis to operate in three stages. First, it automatically inserts $\mathbf{abs}_{\mathsf{t}}$ and $\mathbf{con}_{\mathsf{t}}$ coercions for uses of `typedef`, product (`struct`), sum (`union`) and enumeration (`enum`) types. Second, it considers $\mathbf{update}$ checks just before each `return` statement. Third, it performs capability inference, removing $\mathbf{update}$ checks within functions with updateability N, and may remove $\mathbf{update}$s that are not universal.

C's weak type system and low level of abstraction create challenges not present in a higher-level language like ML or Java. For example, the use of unsafe casts and/or the address-of (`&`) operator can reveal a type's representation through an alias. Say we have type `struct S { int x; int y; }`, and variable `struct S *p`. If we permit taking the address of `p`'s first field, `&(p->x)` having type `int *`, then an update to `struct S` to change the type of `x` to `int *` would lead to `p->x` and the alias ascribing different types to the same storage. To prevent this, for the time being we conservatively restrict `struct S` from being updated at all. Similarly, unsafe casts over function pointers conservatively force them

[3]Note that we could combine this with the con-free dynamic check as follows: let UN′ = UN restricted to those types in Δ. If UN′ is non-empty, and con-free check using UN′ succeeds, then the update is safe.

to have updateability N, which can have a ripple effect throughout the program, reducing the number of possible **update** points.

We have run our preliminary analysis implementation on a number of open-source server programs: `vsftpd`, `apache`, `opensshd`, and `bind`. This demonstrates the feasibility of our analysis. For example, running it on the 232Kloc `bind` code takes 80 seconds.[4] For `vsftpd` more than half of the types (9/16) and potential update points (`returns`) were discovered to be updateable, with roughly half of the latter universal (325/700). For the other examples fewer update points were found, reflecting the conservatism of the current analysis. We are working on addressing C's lower-level features which will be critical to making our implementation practical. Also necessary is an investigation of what coding styles would increase updateability; the examples considered here being written without update in mind.

4.5 Properties

The two important properties of the updateability analysis are soundness and predictability. As with the dynamic system, soundness is proved via *preservation* and *progress* lemmas. The former is stated as follows:

Lemma 4.1 (Preservation). *If* $\vdash \Omega; H, e : \tau$ *then*

1. *If* $\Omega; H, e \rightarrow \Omega; H', e'$ *then* $\vdash \Omega; H', e' : \tau$.
2. *If* $\Omega; H, e \xrightarrow{\text{upd}} \Omega'; H', e'$ *then* $\vdash \Omega'; H', e' : \tau$ *or else* $e' = $ ***UpdEx***.

The proof of part (1) is mostly standard. However, the proof of part (2) is more challenging, and reduces to proving the following lemma, which states that valid updates preserve typing:

Lemma 4.2 (Program Update Safety). *If* $\vdash \Omega; H, e : \tau$ *and* $\text{updateOK}(\mathit{upd}, \Omega, H, \Delta)$ *then* $\vdash \mathcal{U}[H]^{\Omega}; \mathcal{U}[H]^{\mathit{upd}}, \mathcal{U}[e]^{\mathit{upd}} : \tau$.

A core element of this proof is that we must show that by changing the named types listed in upd.UN we will not invalidate code in the existing program. We do this by proving the following lemma:

Lemma 4.3 (Update Capability Weakening). *If* $\Delta; \Gamma \vdash_{\mu} \mathbb{E}[\mathbf{update}^{\Delta''}] : \tau; \Delta'$ *then* $\Delta''; \Gamma \vdash_{\mu} \mathbb{E}[\mathbf{update}^{\Delta''}] : \tau; \Delta'$.

This states that for any expression that has $\mathbf{update}^{\Delta''}$ as its redex, we can typecheck that whole expression using capability Δ''. In turn, this implies that the existing program could only use the types listed in Δ'' concretely, and therefore it should be safe to update the other types in the program.

Another important element of the Program Update Safety lemma is that the insertion of type transformers will preserve type-safety. This must take into account that an inserted transformer will not have an adverse effect on the capability. The following lemma states that as long as a given expression e will not perform an update, it is always safe to increase its capability, and thus to insert it at an arbitrary program point:

Lemma 4.4 (Capability strengthening). *If* $\Delta; \Gamma \vdash_{\mathsf{N}} e : \tau; \Delta'$ *then for all* Δ'' *we have* $\Delta \cup \Delta''; \Gamma \vdash_{\mathsf{N}} e : \tau; \Delta' \cup \Delta''$.

Proofs will appear in the extended version [19].

[4]We ran the analysis on a dual Xeon 2.8GHz with 2GB of RAM running Red Hat Enterprise Linux WS.

5. EXTENSION: BINDING DELETION

While most changes we have observed in source programs are due to added or replaced definitions, occasionally definitions are deleted as well. It is also desirable to support removing definitions dynamically, for two reasons:

1. Dead bindings will unnecessarily consume virtual memory, which could be problematic over time.
2. Dead functions could hamper dynamic updates, since update well-formedness dictates that if some type `t` is updated, any function `f` that concretely manipulates `t` must also be updated. Therefore, even if some function `f` has been removed from the program sources, a future update to `t` would necessitate updating `f`. But how does one update a function that is no longer of use? This issue also arises with old type transformer functions.

Removing dead code reduces to a garbage collection problem. The programmer can specify which bindings should be eligible for deletion at update-time, and then those bindings not reachable by the current program can be removed. Bindings that are unreachable but not specified as dead should be preserved, presumably because they still exist in the program source and might be used later. Formally, we would modify updates upd to include a set of external variable names DB to be deleted. The (UPDATE) operational rule could then be changed to include the precondition

$$\begin{array}{l} \text{upd.DB} \subseteq \text{deadVar}(H, \mathbb{E}[\mathbf{update}^{\Delta}]) \\ H' = \text{delete}(H, e, \text{DB}) \\ \text{updateOK}(\text{upd}, \Omega, H', \Delta) \end{array}$$

Here, $\text{deadVar}()$ traverses the current program to discover which bindings are unreachable, and if all those specified in DB are unreachable, they are removed before the update proceeds (using H'). We could also imagine "marking" bindings eligible for deletion, and removing them as they die.

6. RELATED WORK

Dynamic software updating has been used in industry for many years and is well-studied in academia. To our knowledge, approaches taken in industry are often application-specific, or rely on redundant hardware, limiting their applicability. Academic approaches range from being quite flexible but type-unsafe, to type-safe but quite inflexible.

The systems of which we are aware are either less safe or less flexible than our approach. Many systems are either not type safe at all [7, 13, 10, 4], or could admit dynamic type errors [1]. Some systems are type-safe but not representation consistent [12, 6]. For example, Hicks [12] ensures type-safety by copying and transforming values from their old representation to the new; existing code will continue to use the old, stale values unless the programmer manually ensures otherwise. Other systems are too restrictive. For example, updates may only be permitted to individual class instances whose type cannot change [17, 5, 13, 18], or else representation changes are only permitted for abstract types or encapsulated objects [2, 18, 8]. In many cases, updates to active code are disallowed [8, 15, 7, 10, 18], and data stored in local variables may not be transformed [12, 10, 7, 13].

A number of systems use techniques that bear some resemblance to our approach. Dynamic ML [8] supports updating modules defining *abstract* types `t`. Since by definition clients of such a module must use values of type `t` abstractly, the module can be updated if none of its functions are on the call-stack (i.e. it is *inactive*).

Our use of $\mathbf{abs}_t$ and $\mathbf{con}_t$ coercions generalizes this idea to non-abstract named types, and permits more fine-grained determination of safe update points. In particular, we could discover points *within* an abstract module at which it could be safely updated. This allows our $\mathrm{conFree}[-]^-$ check to be more precise than Dynamic ML's "activeness" check. Dynamic ML has no static notion of proper update timing, as we do with our updatability analysis.

Duggan [6] supports dynamic updates to named types, which use constructs *fold* and *unfold* to create and destruct values of named type (similar to our $\mathbf{abs}_t$ and $\mathbf{con}_t$). However, updated programs are not representation-consistent. Rather, programmers must provide transformer functions that go both ways: from the old to the new representation and from the new version back to the old. Occurrences of *unfold* will dynamically compare the expected version of the `t` value with its actual version and apply some composition of forward or backward transformers to convert the value. This approach ensures well-formed updates are always well-timed. However, programs are harder to reason about. We might wonder: will the program still behave properly when converting a `t` value forward for new code, backward for old code, and then forward again? Moreover, it may not always be possible to write backward transformers, since updated types often contain more information than their older versions (§2).

Boyapati et al. [2] and the K42 operating system [18] ensure well-timed updates to objects. Both systems rely on object encapsulation to guarantee that no active code depends on an object's representation when the object is updated. In Boyapati et al., proper timing is enforced by programmer-defined database-style transactions: if an update occurs at an inopportune time, they abort the current transaction, perform the update, and then restart the transaction. In K42, an object to be updated is made *quiescent* by blocking new threads from using it, and waiting until all current threads that could be using it have terminated. Our approach uses the more general notion of con-freeness, rather than encapsulation. Transactions are approximated by automatically- or programmer-inserted **update** points, but without the benefit of rollback. To mimic this approach in our setting, we could force **update** points to synchronize in different threads; an update could proceed only when all threads have reached safe update points. We intend to flesh out this idea in future work.

While our updateability analysis is new, its general formulation is similar to other capability type systems [21, 20, 9]. For example, capabilities in the Calculus of Capabilities [21] statically prevent a runtime dereference of a dangling pointer by approximating the runtime heap. Our capabilities prevent runtime access to a value whose representation might have changed by approximating the current set of legal types.

7. CONCLUSIONS

In this paper we have presented Proteus, a simple calculus for modeling type-safe dynamic updates in C-like languages. To ensure that updates are type-safe in the presence of changes to named types, Proteus exploits the idea of "con-t-freeness:" a given update point is con-t-free if the program will never use a value of type `t` concretely at its old representation from then on. We have shown that con-freeness can be checked dynamically, and automatically inferred statically using our novel *updateability analysis*.

In the short term, we plan to implement Proteus in the context of single-threaded C, to explore its feasibility for existing non-stop services. Our next step will be to consider the addition of threads, and ultimately move to distributed programs, such as server farms operating concurrently on a shared database whose schema must be evolved. We also plan to explore reasoning techniques for other useful properties, such as update availability. Currently we can discover functions for which an update is never possible; conversely, we wish to understand how often an update is possible for some function, which depends more on runtime behavior. In the longer term, we wish to adapt our techniques to functional and object-oriented languages. On the one hand, these languages will be easier to reason about due to their strong abstraction and encapsulation properties. On the other hand, advanced features such as closures and objects are more challenging to update.

8. 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] C. Boyapati, B. Liskov, L. Shrira, C-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Proc. OOPSLA*, 2003.

[3] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and computation*, 93(1):172–221, 1991.

[4] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[5] S. Drossopoulou and S. Eisenbach. Flexible, source level dynamic linking and re-linking. In *Proc. ECOOP 2003 Workshop on Formal Techniques for Java Programs*, 2003.

[6] D. Duggan. Type-based hot swapping of running modules. In *Proc. ICFP*, 2001.

[7] 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.

[8] S. Gilmore, D. Kirli, and C. Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, LFCS, University of Edinburgh, December 1997.

[9] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proc. PLDI*, 2002.

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

[11] N. Heintze. *Set-Based Program Analysis*. PhD thesis, Department of Computer Science, Carnegie Mellon University, October 1992.

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

[13] G. Hjálmtýsson and R. Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proc. USENIX*, June 1998.

[14] J. Kodumal. BANSHEE: A toolkit for building constraint-based analyses. `http://bane.cs.berkeley.edu/banshee`.

[15] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proc. ECOOP*, 2000.

[16] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002.

[17] A. Orso, A. Rao, and M.J. Harrold. A technique for dynamic updating of Java software. In *Proc. IEEE International Conference on Software Maintenance (ICSM)*, 2002.

[18] C. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System support for online reconfiguration. In *Proc. USENIX*, June 2003.

[19] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating (extended version). To appear.

[20] D. Walker. A type system for expressive security policies. In *Proc. POPL*, pages 254–267, January 2000.

[21] D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, 2000.