# *Mutatis Mutandis*: Safe and Predictable Dynamic Software Updating

GARETH STOYLE
University of Cambridge
MICHAEL HICKS
University of Maryland, College Park
GAVIN BIERMAN
Microsoft Research, Cambridge
PETER SEWELL
University of Cambridge
and
IULIAN NEAMTIU
University of Maryland, College Park

This paper presents Proteus, a core calculus that models dynamic software updating, a service for fixing bugs and adding features to a running program. Proteus permits a program's type structure to change dynamically but guarantees the updated program remains type-correct by ensuring a property we call "con-freeness." We show how con-freeness can be enforced dynamically, and how it can be approximated via a novel static analysis. This analysis can be used to assess the implications of a program's structure on future updates, to make update success more predictable. We have implemented Proteus for C, and briefly discuss our implementation, which we have tested 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
Additional Key Words and Phrases: dynamic software updating, updateability analysis, type inference, capability, Proteus

## 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, all of which must provide continuous service but nonetheless must 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. In a large enterprise, such reboots can be costly [Zorn 2005; Oppenheimer et al. 2002].

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 calculus for modeling dynamic updates in imperative programs. PROTEUS carefully balances the concerns of safety and flexiblity, and adds assurances of predictability. PROTEUS programs consist of function and data definitions, together with 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 $\tau$ interchangeably but, as we shall see, the distinction lets us control updates. Dynamic updates can add new types and new definitions, and can also provide replacements for existing ones, where the type of a replacement may be different from the original. 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 [Armstrong and Virding 1991; Hicks 2001; Buck and Hollingsworth 2000]. We also support updating function pointers.

When updating a named type t from its old representation $\tau$ to a new one $\tau'$, the user provides a *type transformer function* c with type $\tau \to \tau'$. 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 [Gupta 1994; Hicks 2001], or else provide a *backward type transformer* to convert new values back to an older representation should they reappear in old code [Duggan 2001]. While these approaches are type safe, the first permits old and new code to manipulate different copies of the same logical data, which is likely to be disastrous in a language with side-effects; the second may result in information loss when converting a value backwards and then forwards again, since most type changes add information (§2).

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 notion of con-freeness is easily extended to functions and global variables—where a concrete usage is a function call or dereference, respectively—to ensure that updates with types different than the original pose no threat to type safety. The fact that we are only concerned about subsequent *concrete* uses is important: if old 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 data can be used concretely or abstractly at any program point and, moreover, neither use is denoted by syntax. 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.

As a simple example of the need for con-freeness, imagine that the running program is evaluating some function f, and that function is just about to access the second field of some record type t. At that moment, a dynamic update occurs which changes t to have only a single field, and correspondingly changes f to use only that field. While the update is type-correct in itself (the new f is compatible with the

new t), it is not con-free at the current program point, since the updated type t is about to be manipulated concretely by code that expects the old representation: the old f will attempt to access the second field of the t value after the update (the new f will only execute the next time f is called). But this t value will have been transformed by the user-provided type transformer function $\mathtt{c}_\mathtt{t}$ to have but a single field, leading to a type error. The flexibility of updates is partially to blame for this situation: if we were not allowed to change the definitions of types, or we were not allowed to change actively running code, we would not have this problem. However, as we argue in §2, these features are crucial to updating programs in practice, so we must deal with them.

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 concrete type $\tau$), and $\mathbf{con}_\mathtt{t}\ e$ does the reverse at points where t is used concretely. Thus, when some type t is updated, we could 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. Once an update is accepted, the occurrences of expressions $\mathbf{abs}_\mathtt{t}$ indicate where values of type t exist in the program, so that the proper type transformer can be applied.

While illuminating theoretically, the con-free check as defined above is problematic for a practical implementation, for two reasons. First, it is non-trivial to implement, since it must examine the entire run-time execution context of the program—the code, stack, and heap. Second, and more importantly, the results of the check are unpredictable. It may be hard to tell whether an update failure is transient—the update is not valid for this program state—or permanent—the update is invalid for all program states. This is because the dynamic check is with respect to a particular program state. Rather, one would prefer to reason about update behavior statically, covering all possible program states, to (among other things) assess whether there are sufficient 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. For example, we can automatically infer those points at which the program is con-free for all types t, precluding dynamic failure for *any* well-formed update at those points.

We have built a compiler and run-time system to support dynamic updating of C programs based on PROTEUS, and we have used it to dynamically update three open source programs: OpenSSH's `sshd` daemon, the "Very Secure" FTP daemon, `vsftpd`, and the GNU `zebra` routing daemon. PROTEUS's updating model is powerful enough that we were able to construct and apply dynamic updates for more than three years of releases for each program. Indeed, without the flexibility to update active code—needed to update the main event loops—and to change the program's type structure, we would not have been able to update these programs over such a long stretch. Based on a brief study of the evolution of these and other software programs, we believe these needs are not atypical. As such, the notion of con-freeness we present in this paper is crucial to ensuring that dynamic updates are both type-safe and representation consistent. The updateability analysis ensures

that updates can be applied predictably.

This work focuses on updating a single-threaded process, as opposed to multi-threaded or distributed programs. To support these systems requires some level of coordination. For example, to update a distributed system to use a new (non-backwards-compatible) communication protocol would require coordination to prevent an updated program from sending a confusing message to a not-as-yet-updated one. As a step in this direction, we have sketched how we could adapt our current work to support multi-threaded programs by treating **update** as a synchronization point between threads when an update is available (§6). This makes our analysis sound, but could allow an update to be unduly delayed, or worse, could cause the system to deadlock. We are exploring these issues in our current research.

In summary, this paper makes the following contributions:

—We present PROTEUS, a simple and flexible calculus for reasoning about type-safe, representation-consistent dynamic software updating in single-threaded, imperative languages (§3 and §4), with various extensions (§6). 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). Crucially, PROTEUS permits updating active code, and changing the type structure of programs.

—We formally define the notion of con-freeness, and prove that it is sufficient to establish type safety in updated programs (§4.4). We believe this notion is useful beyond DSU. For example, we have applied it to the problem of ensuring that a dynamic update of a security policy does not impact the security properties of a running program that uses it [Hicks et al. 2005].

—We present a novel updateability analysis that statically infers the types for which a given **update** point is not con-free (§5). While space constraints preclude a full description, we present some preliminary experience with our dynamic updating implementation that applies our analysis to C programs (§5.4).

In §8 and §9 we discuss related work and conclude. This is a revised and extended version of a paper presented at the 2005 ACM SIGPLAN Symposium on Principles of Programming Languages [Stoyle et al. 2005].

## 2. SOFTWARE EVOLUTION IN PRACTICE

The goal of a dynamic software updating system is to be able to modify a program to fix bugs and add new features without shutting it down. For each dynamic update, modifications are collected into a *dynamic patch* that is applied to the running system. There are a number of ways that dynamic patches could be constructed. For example, if a bug is discovered, a special-purpose dynamic patch could be constructed that fixes just that bug by replacing the offending function or functions with new versions. More generally, a dynamic patch could consist of all of the changes that occurred between two different releases of a software system. In this case, a new version of the software could ship with a dynamic patch to the prior version, for those users who wish to dynamically upgrade their running systems. This would permit *on-line software evolution.* We would not expect users to dynamically upgrade their programs in perpetuity; rather, DSU would allow programs to continue to run until some other, non-dynamically updateable part of the system, like the hardware, needed to be changed, thus extending the availability of the system.

To enable on-line evolution we must support the kinds of software changes that typically occur between releases. To characterize these changes, we studied how

| Total functions: | Vsftpd (14 releases) | OpenSSH (27 releases) | Zebra (6 releases) |
|---|---|---|---|
| Initial | 384 | 56 | 767 |
| Added | 107 | 955 | 134 |
| Deleted | 16 | 162 | 44 |
| Changed Body | 332 | 2673 | 321 |
| Changed Type | 11 | 195 | 13 |

Fig. 1. Function evolution in Vsftpd, OpenSSH, and Zebra

the source code has changed over time in some long-running C programs. We built a tool that parses two versions of a program, compares their abstract syntax trees, and reports the differences [Neamtiu et al. 2005]. In particular, it reports the definitions that are different between versions; that is, those function definitions, type definitions, or global variable definitions that have been added, deleted, or changed. For definitions that have changed, the tool also reports changes in type structure. For example, it might report that a function has an additional argument, or that a `struct` definition has two new fields.

We used the tool to compare 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); Vsftpd version 1.0.1 (Nov. 2001) to 2.0.3 (March 2005); OpenSSH, version 1.2 (Oct. 1999) to 4.2 (Sept. 2005), and GNU Zebra, version 0.92a (Aug. 2001) to 0.95a (Sept. 2005). These programs are in wide use. Linux is now quite common; Apache, BIND and Vsftpd are the de-facto web server, name server and FTP server, respectively, in major Unix distributions; the OpenSSH suite is the standard open-source release of the widely-used secure shell protocols; and many Linux/BSD-based dedicated BGP routers are built using Zebra or its spin-off, Quagga.

In what follows, we focus on Vsftpd, OpenSSH, and Zebra, as we have used them as test applications for our implementation (§7). We chose them because each maintains state that should be preserved by an update to maintain service (the same is true for BIND and Linux). Stopping and restarting the FTP daemon would abruptly end all the active file transfer sessions, rendering the clients with incomplete transfers. Taking a machine's SSH server down for update would close all the SSH connections, hence terminating all the remote shell sessions clients had on that particular machine. Terminating the zebra daemon on a machine used as a router would disable the routing functionality, and, upon restart, the daemon would need to re-learn all the routes it has amassed prior to termination. For these three programs, we analyzed versions as far back as possible; older versions need older compilers and headers. The changes followed a few key trends.

*Functions.* By far the most common form of version changes consist of added functions, or changes to existing functions which do not involve a changed type signature. Function deletions and body changes which involved a changed type signature occur significantly less often, but regularly. These trends are illustrated in Figure 1, which shows the evolution of functions in Vsftpd, OpenSSH, and Zebra for the span we measured. The parenthesized numbers in the heading indicate the number of individual releases measured. These trends are similar for the shorter spans we measured for BIND and Linux as well.

*Global Variables.* The number of global variables tends to be fairly static, adding a few and deleting a few with each change, but growing over time. For example,

the numbers of global variables for Vsftpd, OpenSSH, and Zebra grew, respectively, from 141 to 226, from 51 to 264, and 235 to 280. In Vsftpd, deletions were extremely rare, with 8 in total across 14 releases, while at the other extreme, a total of 82 variables were deleted for OpenSSH over the measured span of 27 releases. Global variables change their static initializers fairly often; for the entire period we analyzed, there were 18 static initializer changes for Vsftpd, 63 for OpenSSH, and 11 for Zebra. However, global variable types change less frequently: 1 case for Vsftpd, 29 for OpenSSH and 18 for Zebra.

*Type Definitions.* Data representations, which is to say *type definitions*, do change between versions, though rarely. In C, types are defined with `struct` and `union` declarations (aggregates), `typedefs`, and `enum`s. Very often, the changes are to aggregates and involve adding or removing a field. For example, moving from Linux 2.4.20 to 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. This ratio was similar to that of the other programs we measured. Typedefs rarely changed.

Given this information, a DSU system must, at the least, support the addition of new definitions, and the replacement of existing definitions at the same type, since these changes comprise the majority of version changes. Implementing a system that supports these changes is fairly straightforward. Indeed, Altekar et al. [Altekar et al. 2005] found that many security patches satisfy these criteria, and thus can be applied through a simple DSU system. However, to support on-line software evolution, we must be able to change the types of definitions, and to delete definitions. Supporting these changes while preserving program type safety is more challenging.

Our initial presentation of PROTEUS, in the next section, permits changes to type definitions, and we sketch extensions to this system to support changes to function and global variable types, and to support deleting definitions, in §6. We have developed a prototype implementation that supports all of these changes.

## 3. PROTEUS

In what follows we define two core calculi—PROTEUS$^{\mathbf{src}}$ and PROTEUS$^{\mathbf{con}}$—that formalize our approach to dynamic software updating. In this section we present PROTEUS$^{\mathbf{src}}$, the language used by programmers for writing updateable programs. We define how dynamic updates are specified and show how the timing of an update could violate type safety. Section 4 presents PROTEUS$^{\mathbf{con}}$, an extension of PROTEUS$^{\mathbf{src}}$ that makes the usage of named types manifest in program by introducing *type coercions*; these are used to support the operational semantics of dynamic updating and to ensure that the process is type-safe by ensuring con-freeness. Section 5 presents PROTEUS$^{\Delta}$, an extension to PROTEUS$^{\mathbf{con}}$, for which con-freeness can be determined statically.

### 3.1 PROTEUS$^{\mathbf{src}}$ Syntax

PROTEUS$^{\mathbf{src}}$ models a type safe, imperative language augmented with dynamic updating; its syntax is given in Figure 2. Programs $P$ are a series of top-level definitions followed by an expression **return** $e$. A **fun** $\mathsf{z}\ldots$ defines a top-level recursive function, and **var** $\mathsf{z} : \tau \ldots$ defines a top-level *mutable* variable (i.e., it has type $\tau$ **ref**). We allow the extension of function definitions to a mutually recursive block of function definitions using the keyword **and**. A **type** $\mathsf{t} = \tau \ldots$ defines the

| | | | |
|---|---|---|---|
| Integers | $n \in \text{Int}$ | | |
| Local variables | $x, y, z \in \text{IVar}$ | | |
| Top-level identifiers | $\mathsf{x}, \mathsf{y}, \mathsf{z} \in \text{XVar}$ | | |
| Record labels | $\mathrm{l} \in \text{Lab}$ | | |
| References | $r \in \text{Ref}$ | | |
| Type names | $\mathsf{t}, \mathsf{s} \in \text{NT}$ | | |
| Variables | $\text{Var} = \text{IVar} \uplus \text{XVar} \uplus \text{NT}$ | | |
| Types | $\tau \in \text{Typ}$ | $::= \mathsf{t} \mid \mathsf{int} \mid \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\} \mid \tau\ \mathbf{ref}$ | |
| | | $\mid\ \tau_1 \to \tau_2$ | |
| Expressions | $e \in \text{Exp}$ | $::= n$ | integers |
| | | $\mid\ x$ | variables |
| | | $\mid\ \mathsf{z}$ | top-level identifiers |
| | | $\mid\ r$ | heap reference |
| | | $\mid\ \{\mathrm{l}_1 = e_1, \ldots, \mathrm{l}_n = e_n\}$ | records |
| | | $\mid\ e.\mathrm{l}$ | projection |
| | | $\mid\ e_1\ e_2$ | application |
| | | $\mid\ \mathbf{let}\ z : \tau = e_1\ \mathbf{in}\ e_2$ | let bindings |
| | | $\mid\ \mathbf{ref}\ e$ | ref. creation |
| | | $\mid\ e_1 := e_2$ | assignment |
| | | $\mid\ !e$ | dereference |
| | | $\mid\ \mathbf{if}\ e_1 = e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4$ | conditional |
| | | $\mid\ \mathbf{update}$ | dynamic update |
| Values | $v \in \text{Val}$ | $::= \mathsf{z} \mid n \mid \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_n = v_n\} \mid r \mid \mathbf{abs}_{\mathsf{t}}\ v$ | |
| Programs | $P \in \text{Prog}$ | $::= \mathbf{var}\ \mathsf{z} : \tau = v\ \mathbf{in}\ P$ | |
| | | $\mid\ \mathbf{fun}\ \mathsf{z}_1(x : \tau_1) : \tau_1' = e_1$ | |
| | | $\ldots$ | |
| | | $\mathbf{and}\ \mathsf{z}_n(x : \tau_n) : \tau_n' = e_n\ \mathbf{in}\ P$ | |
| | | $\mid\ \mathbf{type}\ \mathsf{t} = \tau\ \mathbf{in}\ P$ | |
| | | $\mid\ \mathbf{return}\ e$ | |

Fig. 2. Syntax for PROTEUS$^{\mathbf{src}}$

type $\mathsf{t}$. Top-level identifiers $\mathsf{z}$ must be unique within $P$, and are not subject to $\alpha$-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 $\{\mathrm{x}_1 : \tau_1, \ldots, \mathrm{x}_n : \tau_n\} \to \tau$. We similarly sugar calls to such functions.

PROTEUS$^{\mathbf{src}}$ also includes an **update** expression which permits a dynamic update to take place, if one is available. That is, at run-time a user signals that an update is ready, and the next time **update** is reached in the program, that update is applied. The **update** expression is integer-valued; it returns 0 when an update is available and successfully applied, and returns 1 if no update is available, or if an update is available but cannot be applied successfully. Providing **update** in the language makes issues of timing more manifest. At one extreme, **update** could be inserted by the programmer in a few places, and at another the compiler could insert **update** at all program points, simulating the situation in which a dynamic update could occur at any moment. We discuss an algorithm for automatically inserting **update**s in §5.

The typing rules and operational semantics for the non-updating part

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

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

fun dispatch (s : req) : handResult =
  let t : fdtype = decode (s.fd) in
  let u1 : int = update in
  if (t = Socket) then
    let k : sock = getsock (s.fd) in
    let flags : sflags = decode_sockopargs (s.rest, s.op)  in
    let h : sockhandler = getsockhandler (s.fd, s.op)  in
    let u2 : int = update in
    let res : handResult = h (k, s.buf, flags)  in
    let u3 : int = update in res
  else if (t = File) then ...
  else −1 in

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

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

return (loop 0)
```

Fig. 3. A simple kernel for files and socket I/O

PROTEUS$^{\mathbf{src}}$ are essentially standard, and they are considered in more detail when considering compilation to PROTEUS$^{\mathbf{con}}$ in §4.

*Example.* Figure 3 shows a simple kernel, which one might want to dynamically update, for handling read and write requests on files or sockets. 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).[1] 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) and streaming connections (TCP). 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

[1] To aid readability we have defined fdtype using a union type—which is not in the core language we formalize—but it could have been defined as an integer.

```
UN : sockhandler  ↦
       ({sock : sock, buf : buf, sflags : sflags, cookie : cookie} → int,  sockh_coer)
     sock         ↦ ({daddr : int, ... }, sock_coer)
AN : sockhandler  ↦ (cookie, int)
UB : dispatch     ↦
       (req → handResult,
         λ(s).... h(k,  s.buf,  flags,  (security_info ()) ... )
AB : udp_read′    ↦
       ({sock : sock, buf : buf, sflags : sflags, cookie : cookie} → int, λ(x).... )
     udp_write′   ↦
       ({sock : sock, buf : buf, sflags : sflags, cookie : cookie} → int, λ(x).... )
     sockh_coer   ↦ (({sock : sock, buf : buf, sflags : sflags} → int) →
       ({sock : sock, buf : buf, sflags : sflags, cookie : cookie} → int),
       λ(f).if f = udp_read then udp_read′ else if f = udp_write then udp_write′ )
     sock_coer    ↦ ...
     security_info ↦ (int → cookie, λ(x).... )
```

Fig. 4. A sample update to the I/O kernel

files. After dispatch completes, its result is posted to the user, and the loop continues. The dispatch function defines three **update** points at which a dynamic update is permitted to occur, if one is available.

### 3.2 Specifying Dynamic Updates

A dynamic update is a specification to modify the running program with new and replacement definitions. Formally, a dynamic update, upd, consists of four partial maps, written as a record with the labels UN, UB, AN, and AB:

—UN (Updated Named types) is a map from type names to pairs of a type and an expression. Each entry, $\mathsf{t} \mapsto (\tau, \mathsf{c})$, specifies a named type to be replaced ($\mathsf{t}$), its new representation type ($\tau$), and a type transformer function $\mathsf{c}$ from the old representation type to the new.

—AN (Added Named types) is a map from type names $\mathsf{t}$ to type environments $\Omega$, which are lists of type definitions. This is used to define new named types. Each entry $\mathsf{t} \mapsto \Omega$ specifies a type $\mathsf{t}$ in the existing program, and the new definitions are inserted just above $\mathsf{t}$ in the updated program.

—UB (Updated Bindings) is a map from top-level identifiers to pairs of a type and a *binding value* $b_v$, which is either a function, written as $\lambda(x).e$ or a value $v$. These specify replacement **fun** and **var** definitions. Each entry $\mathsf{z} \mapsto (\tau, b_v)$ contains the binding to replace ($\mathsf{z}$), the type of the new binding as it appears in the source program ($\tau$) and the new binding ($b_v$). For now, we require updated bindings to have the same type of the definitions they replace. In §6.1 we show how this restriction can be relaxed by extending our technique for supporting changes to named types.

—AB (Added Bindings) is a map from top-level identifiers $\mathsf{z}$ to pairs of types and binding values. These are used to specify new **fun** and **var** definitions. All added functions are assumed to be mutually recursive, for simplicity.

We consider how to extend specifications to support deletion of bindings, and updating functions and data at new types in §6.

As an example, say we wish to modify socket handling in Figure 3 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. We then must provide transformer 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 sock_coer and sockh_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 sockh_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 [Frieder and Segal 1991; Hicks 2001], forwarding calls to the new version, and filling in the added argument. In Section 6 we explain how our approach can be easily extended to support updating bindings at new types with the same safety guarantees, making stubs unnecessary.

It is not necessary for programmers to write update specifications manually. Rather, it is straightforward for a tool to examine the old and new versions of a program to determine which bindings have changed, and which have been added [Hicks 2001]. Type transformers can be generated automatically [Hicks 2001] for simple changes, but generally require a programmer's attention.

### 3.3 Update Timing

```
let i : int = post (
  let u2 : int = update in
  let res : handResult = udp_read{sock = v_sock, buf = v_req.buf, sflags = v_sflags} in
  let u3 : int = update in res
) in loop i
```

Fig. 5. Example active expression

Because update specifications permit type definitions to change, to preserve type safety an update must only be applied at certain times during a program's execution. For example, Figure 5 shows the expression fragment of our example program after some evaluation steps (the outer **let** i = ... binding comes from loop and the argument to post is the partially-evaluated dispatch function; we are assuming a

substitution-style semantics for function application). The **let** u2 = **update**... is in redex position, and suppose that the update described in §3.2 is available, which updates sockhandler to have an additional cookie argument, amongst other things. If this update were applied then the user's type transformer sockh_coer would be inserted to convert udp_read, to be called next. Evaluating the transformer replaces udp_read with udp_read′ yielding the expression udp_read′($v_{\mathsf{sock}}, v_{\mathsf{req}}.\mathrm{buf}, v_{\mathsf{sflags}}$). But this would be type-incorrect! The function udp_read′ expects four arguments (more precisely, a record with four fields), 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 dispatch, which expects 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*. A value is used concretely when it is destructed, e.g., dereferencing a reference, calling a function, or extracting a field from a record, because these operations rely on the type of the value. If this type were to change, the operations would be type-incorrect.

## 4. PROTEUS$^{\mathbf{CON}}$

To prevent updates to named types from violating type safety, we define the language PROTEUS$^{\mathbf{con}}$ which extends PROTEUS$^{\mathbf{src}}$ with explicit *type coercions* to make manifest the concrete usage of named types. Ensuring proper update timing then reduces to ensuring that, roughly speaking, no type coercions that indicate a concrete usage for types to be updated appear in the active program. Type coercions also are handy for implementing dynamic updates operationally.

In this section, we present the syntax of PROTEUS$^{\mathbf{con}}$ and show how PROTEUS$^{\mathbf{src}}$ programs can be compiled to PROTEUS$^{\mathbf{con}}$ programs. Then we present the operational semantics of PROTEUS$^{\mathbf{con}}$, and define con-freeness as a predicate on the running program's state at the time of the update. We then prove that this predicate is sufficient to ensure that updates preserve type safety.

### 4.1 Syntax and Typing

Prior to evaluation, PROTEUS$^{\mathbf{src}}$ programs (as well as program fragments appearing in update specifications) are compiled to the language PROTEUS$^{\mathbf{con}}$, which extends PROTEUS$^{\mathbf{src}}$ with type coercions:

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathbf{abs}_{\mathsf{t}}\ e \mid \mathbf{con}_{\mathsf{t}}\ e \\ v & ::= & \ldots \mid \mathbf{abs}_{\mathsf{t}}\ v \end{array}$$

Given a type definition **type** t = $\tau$, the PROTEUS$^{\mathbf{src}}$ typing rules effectively allow values of type t and type $\tau$ to be used interchangeably, as is typical. (We present the PROTEUS$^{\mathbf{src}}$ typing rules when describing compilation to PROTEUS$^{\mathbf{con}}$ in §4.2.) For example, in Figure 3, the expression $h$ ($k$, $s$.buf, *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 PROTEUS$^{\mathbf{con}}$ all uses of a named type definition t = $\tau$ must be explicit, using type coercions: $\mathbf{abs}_{\mathsf{t}}\ e$ converts $e$ to type t (assuming $e$ has the proper type $\tau$), and $\mathbf{con}_{\mathsf{t}}\ e$ does the reverse. Figure 6 illustrates the dispatch function from Figure 3 with these coercions inserted. As examples, we can see that a handResult value is constructed in the last line from –1 via the coercion ($\mathbf{abs}_{\mathsf{handResult}}\ -1$). Conversely,

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

Fig. 6. dispatch with explicit type coercions

to invoke h (in the expression for res), it must be converted from type sockhandler via the coercion ($\mathbf{con}_{\mathsf{sockhandler}}$ h) (...).

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 (§4.4).

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 $\mathsf{c}_{\mathsf{t}}$ which converts from the old representation of t to the new, we can rewrite each instance at update-time to be $\mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}_{\mathsf{t}}\ e)$. This leads to a natural call-by-value evaluation strategy for transformers in conjunction with the rest of the program (§4.3).

The PROTEUS$^{\mathbf{con}}$ typing rules for coercions are simple:

$$\frac{\Gamma \vdash e : \mathsf{t} \qquad \Gamma(\mathsf{t}) = \tau}{\Gamma \vdash \mathbf{con}_{\mathsf{t}}\ e : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma(\mathsf{t}) = \tau}{\Gamma \vdash \mathbf{abs}_{\mathsf{t}}\ e : \mathsf{t}}$$

### 4.2 Compiling PROTEUS$^{\mathbf{src}}$ to PROTEUS$^{\mathbf{con}}$

Compiling a PROTEUS$^{\mathbf{src}}$ program to a PROTEUS$^{\mathbf{con}}$ program requires inserting type coercions to make explicit the implicit uses of the type equality in the source program. Our methodology is based on coercive subtyping [Breazu-Tannen et al. 1991]. As is usual for coercive subtyping systems, we define our translation inductively over source language typing derivations. In particular, we define a judgement $\Gamma \vdash P : \tau \rightsquigarrow P'$ by which a PROTEUS$^{\mathbf{src}}$ program $P$ is translated to PROTEUS$^{\mathbf{con}}$ program $P'$. (The typing environment $\Gamma$ is a finite mapping from variables to types and from type names to types. As is usual we will sometimes write a mapping using a list notation, e.g., $\Gamma \equiv x\colon \tau, \mathsf{t} = \tau'$.) Our primary aim is that the translation be *deterministic*, so that *where* coercions are inserted is intuitive to the programmer. Secondarily, we wish the resulting PROTEUS$^{\mathbf{con}}$ program to be *efficient*, with a minimal number of inserted coercions and other computational machinery.

The remainder of this subsection proceeds as follows. First, we show how abstraction and concretion of values having named type can be represented with subtyping. Second, we show how to derive an algorithmic subtyping relation. Finally, we show how to derive an algorithmic typing relation for expressions and use this to present

the full translation rules. The reader not interested in the details of the translation can safely skip this subsection and proceed to the operational semantics of PROTEUS$^{\mathbf{con}}$ in §4.3.

4.2.1 *Abstraction and Concretion as Subtyping.* To properly insert $\mathbf{con}_{\mathsf{t}}$ and $\mathbf{abs}_{\mathsf{t}}$ coercions in the source program $P$, we must identify where $P$ uses values of type $\mathsf{t}$ concretely and abstractly. We do this by defining a mostly-standard subtyping judgement for PROTEUS$^{\mathbf{src}}$, written $\Gamma \vdash \tau <: \tau'$, with two key rules. First, given a value of type $\tau$ it can be *abstracted* as having type $\mathsf{t}$ under the assumption $\mathsf{t} = \tau$:

$$\Gamma, \mathsf{t} = \tau \ \vdash \ \tau <: \mathsf{t}$$

Conversely a value of type $\mathsf{t}$ can be treated *concretely* as having type $\tau$:

$$\Gamma, \mathsf{t} = \tau \ \vdash \ \mathsf{t} <: \tau$$

These two rules, along with subtyping transitivity, allow a named type to be treated as equal to its definition.

The basic compilation strategy is to relate a subtyping derivation to a *coercion context* $\mathbb{C}$ using the judgement $\Gamma \ \vdash \ \tau <: \tau' \ \rightsquigarrow \ \mathbb{C}$. This context is applied to the relevant program fragment $e$ as part of the derivation $\Gamma \ \vdash \ e : \tau \ \rightsquigarrow \ e'$ using the expression subtyping rule:

$$\frac{\Gamma \vdash e : \tau \ \rightsquigarrow \ e' \qquad \Gamma \vdash \tau <: \tau' \ \rightsquigarrow \ \mathbb{C}}{\Gamma \vdash e : \tau' \ \rightsquigarrow \ \mathbb{C}[e']}$$

Here, a coercion context $\mathbb{C}$ is defined by the following grammar:

$$\mathbb{C} \ ::= \ \_ \mid \mathbf{abs}_{\mathsf{t}} \ \mathbb{C} \mid \mathbf{con}_{\mathsf{t}} \ \mathbb{C} \mid \mathbf{let} \ x : \tau = \mathbb{C} \ \mathbf{in} \ e \mid e \ \mathbb{C}$$

The syntax $\mathbb{C}[e]$ defines context application, where $e$ fills the "hole" (written $\_$) present in the context. For the abstraction and concretion subtyping rules, the translation rules are:

$$\Gamma, \mathsf{t} = \tau \ \vdash \ \tau <: \mathsf{t} \ \rightsquigarrow \ \mathbf{abs}_{\mathsf{t}} \ \_$$
$$\Gamma, \mathsf{t} = \tau \ \vdash \ \mathsf{t} <: \tau \ \rightsquigarrow \ \mathbf{con}_{\mathsf{t}} \ \_$$

To see how this works, here is an example translation derivation using the above rules, where $\Gamma \equiv \mathsf{t} = \mathsf{int} \to \mathsf{int}, \mathsf{f} : \mathsf{t}$:

$$\frac{\dfrac{\Gamma \vdash \mathsf{f} : \mathsf{t} \ \rightsquigarrow \ \mathsf{f} \qquad \Gamma \vdash \mathsf{t} <: \tau \ \rightsquigarrow \ \mathbf{con}_{\mathsf{t}} \ \_}{\Gamma \vdash \mathsf{f} : \mathsf{int} \to \mathsf{int} \ \rightsquigarrow \ \mathbf{con}_{\mathsf{t}} \ \mathsf{f}} \qquad \Gamma \vdash 1 : \mathsf{int} \ \rightsquigarrow \ 1}{\Gamma \vdash \mathsf{f} \ 1 : \mathsf{int} \ \rightsquigarrow \ (\mathbf{con}_{\mathsf{t}} \ \mathsf{f}) \ 1}$$

Notice how on the left-hand side of the derivation we apply coercion context $\mathbf{con}_{\mathsf{t}} \ \_$ to $\mathsf{f}$ to get $\mathbf{con}_{\mathsf{t}} \ \mathsf{f}$. Standard coercive subtyping relates subtyping judgements to *functions*, rather than contexts, so that this application would occur at run-time, rather than during compilation.

4.2.2 *Making Subtyping Algorithmic.* Unfortunately, the strategy described above is not directly suitable for implementation. The problem is that neither the typing relation for expressions nor the subtyping relation are syntax directed, meaning that many derivations are possible. This is not a merely theoretical concern: we can observe the difference between these derivations due to the effect of inserted $\mathbf{con}_{\mathsf{t}}$ coercions on run-time updates.

For example, assuming $\Gamma \equiv \mathsf{t} = \tau, \mathsf{x} : \tau$, we could translate the PROTEUS$^{\mathbf{src}}$ term $\mathsf{x}$ using the following derivation:

$$\Gamma \vdash \mathsf{x} : \tau \rightsquigarrow \mathsf{x}$$

We could also use the following derivation which uses subsumption twice:

$$\dfrac{\dfrac{\Gamma \vdash \mathsf{x} : \tau \rightsquigarrow \mathsf{x} \qquad \Gamma \vdash \tau <: \mathsf{t} \rightsquigarrow \mathbf{abs}_{\mathsf{t}}\ \_}{\Gamma \vdash \mathsf{x} : \tau \rightsquigarrow \mathbf{abs}_{\mathsf{t}}\ \mathsf{x}} \qquad \Gamma \vdash \mathsf{t} <: \tau \rightsquigarrow \mathbf{con}_{\mathsf{t}}\ \_}{\Gamma \vdash \mathsf{x} : \tau \rightsquigarrow \mathbf{con}_{\mathsf{t}}\ (\mathbf{abs}_{\mathsf{t}}\ \mathsf{x})}$$

Because $\mathbf{con}_{\mathsf{t}}$ coercions may impede a proposed dynamic update to the type $\mathsf{t}$, an update to the first program may succeed while the second fails. Moreover, because coercions perform computation at run time, the second program is less efficient than the first.

To remedy these problems, we make both the subtyping relation and the typing relation deterministic. Ignoring contexts $\mathbb{C}$ for the moment, here is our initial subtyping relation for PROTEUS$^{\mathbf{src}}$:

$$\frac{}{\Gamma \vdash \tau <: \tau}\ [\mathsf{REFL}]$$

$$\frac{\Gamma, \mathsf{t} = \tau \vdash \tau <: \tau'}{\Gamma, \mathsf{t} = \tau \vdash \mathsf{t} <: \tau'}\ [\mathsf{CON}] \qquad \frac{\Gamma, \mathsf{t} = \tau \vdash \tau' <: \tau}{\Gamma, \mathsf{t} = \tau \vdash \tau' <: \mathsf{t}}\ [\mathsf{ABS}]$$

$$\frac{\Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma \vdash \tau_2 <: \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}\ [\mathsf{FUN}]$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_1' \quad \cdots \quad \Gamma \vdash \tau_k <: \tau_k' \quad k \leq n}{\Gamma \vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n\} <: \{l_1 : \tau_1', \ldots, l_k : \tau_k\}}\ [\mathsf{REC}]$$

We have made the standard first step of removing the transitivity rule and embedding its action into the other rules (in this case, the abstraction and concretion rules). Two other things are worthy of note. First, the [REFL] rule imposes an *invariance* restriction on reference types (i.e., $\tau$ **ref** $<: \tau'$ **ref** if and only if $\tau$ and $\tau'$ are identical). While such an invariance is standard, it usually does not apply when considering named types as equal to their definition. For example, if we had $\Gamma \equiv \mathsf{t} = \mathsf{int}$, we might expect that $\Gamma \vdash \mathsf{t}$ **ref** $<: \mathsf{int}$ **ref**. However, when subtyping is used to add coercions, this approach will not work: there is no way to coerce a term having the former type to one having the latter. We have not found this to be a problem in practice.

Second, we can see that the rules [CON] and [ABS] are not syntax-directed. For example, consider the context $\Gamma \equiv \mathsf{t} = \mathsf{int}, \mathsf{s} = \mathsf{t}, \mathsf{u} = \mathsf{s}, \mathsf{v} = \mathsf{t}$. Here are two different derivations of the judgement $\Gamma \vdash \mathsf{u} <: \mathsf{v}$, with the preferred on the left:

$$\frac{\Gamma \vdash \mathsf{t} <: \mathsf{t}}{\Gamma \vdash \mathsf{t} <: \mathsf{v}} \qquad \frac{\Gamma \vdash \mathsf{t} <: \mathsf{t}}{\Gamma \vdash \mathsf{t} <: \mathsf{int}}$$

$$\frac{\Gamma \vdash \mathsf{t} <: \mathsf{v}}{\Gamma \vdash \mathsf{s} <: \mathsf{v}} \qquad \frac{\Gamma \vdash \mathsf{s} <: \mathsf{int}}{\Gamma \vdash \mathsf{s} <: \mathsf{t}}$$

$$\frac{\Gamma \vdash \mathsf{s} <: \mathsf{v}}{\Gamma \vdash \mathsf{u} <: \mathsf{v}} \qquad \frac{\Gamma \vdash \mathsf{s} <: \mathsf{t}}{\Gamma \vdash \mathsf{u} <: \mathsf{t}}$$

$$\frac{\Gamma \vdash \mathsf{u} <: \mathsf{t}}{\Gamma \vdash \mathsf{u} <: \mathsf{v}}$$

The problem with rightmost derivation is the pointless concretion/abstraction of type $\mathsf{t}$. As we explained above this will be compiled to a coercion that will possibly inhibit future updates and add unnecessary computation.

The [CON] and [ABS] rules capture the cases of $\tau <: \tau'$ where one or both of $\tau$ and $\tau'$ is a name. The essence of our solution to the above problem is to break down these cases into separate rules and also to avoid unnecessary concretions/abstractions. We replace the [CON] and [ABS] rules with the following four rules:

$$\frac{\mathit{NotNameType}(\tau') \qquad \Gamma, \mathsf{t} = \tau \vdash \tau <: \tau'}{\Gamma, \mathsf{t} = \tau \vdash \mathsf{t} <: \tau'}\ [\mathsf{CONc}]$$

$$\frac{\mathit{NotNameType}(\tau') \qquad \Gamma, \mathsf{t} = \tau \vdash \tau' <: \tau}{\Gamma, \mathsf{t} = \tau \vdash \tau' <: \mathsf{t}}\ [\mathsf{ABSc}]$$

$$\frac{\Gamma(\mathsf{t}) = \tau \quad \Gamma(\mathsf{s}) = \tau' \qquad \mathit{Height}(\mathsf{t}, \Gamma) \geq \mathit{Height}(\mathsf{s}, \Gamma) \qquad \Gamma \vdash \tau <: \mathsf{s}}{\Gamma \vdash \mathsf{t} <: \mathsf{s}}\ [\mathsf{CONn}]$$

$$\frac{\Gamma(\mathsf{t}) = \tau \quad \Gamma(\mathsf{s}) = \tau' \qquad \mathit{Height}(\mathsf{s}, \Gamma) > \mathit{Height}(\mathsf{t}, \Gamma) \qquad \Gamma \vdash \mathsf{t} <: \tau'}{\Gamma \vdash \mathsf{t} <: \mathsf{s}}\ [\mathsf{ABSn}]$$

The predicate $\mathit{NotNameType}(\tau)$ returns false if $\tau$ is a name and true otherwise. [CONc] and [ABSc] deal with the case $\tau <: \tau'$ where only one of $\tau$ and $\tau'$ is a name. However, we still need to handle the case when they are both names, e.g. $\Gamma, \mathsf{t} = \tau, \mathsf{t}' = \tau' \ \vdash \ \mathsf{t} <: \mathsf{t}'$. Should we unfold the name $\mathsf{t}$, or $\mathsf{t}'$? To break this symmetry we make use of a function *Height*, which is defined as follows:

$$\begin{array}{lcll} \mathit{Height}(\mathsf{t}, \Gamma) & = & 1 & \text{if } \Gamma(\mathsf{t}) = \tau \text{ and } \mathit{NotNameType}(\tau) \\ \mathit{Height}(\mathsf{t}, \Gamma) & = & 1 + h & \text{if } \Gamma(\mathsf{t}) = \mathsf{s} \text{ and } h = \mathit{Height}(\mathsf{s}, \Gamma) \end{array}$$

Given a typing context $\Gamma$ and name $\mathsf{t}$, $\mathit{Height}(\mathsf{t}, \Gamma)$ returns the number of "unfoldings" we need to make to the name until we get a type constructor. For example, given $\Gamma \equiv \mathsf{t} = \mathsf{int}, \mathsf{s} = \mathsf{t}, \mathsf{u} = \mathsf{s}, \mathsf{v} = \mathsf{t}$ then

$$\begin{array}{rcl} \mathit{Height}(\mathsf{u}, \Gamma) & = & 3, \\ \mathit{Height}(\mathsf{s}, \Gamma) & = & 2, \\ \mathit{Height}(\mathsf{t}, \Gamma) & = & 1, \text{ and} \\ \mathit{Height}(\mathsf{v}, \Gamma) & = & 2. \end{array}$$

We can use this algorithmic subtyping relation to generate coercion contexts as shown in Figure 7. As an example, we show how the derivation of $\mathsf{u} <: \mathsf{v}$ from earlier would be translated (assuming that $\Gamma \equiv \mathsf{t} = \mathsf{int}, \mathsf{s} = \mathsf{t}, \mathsf{u} = \mathsf{s}, \mathsf{v} = \mathsf{t}$).

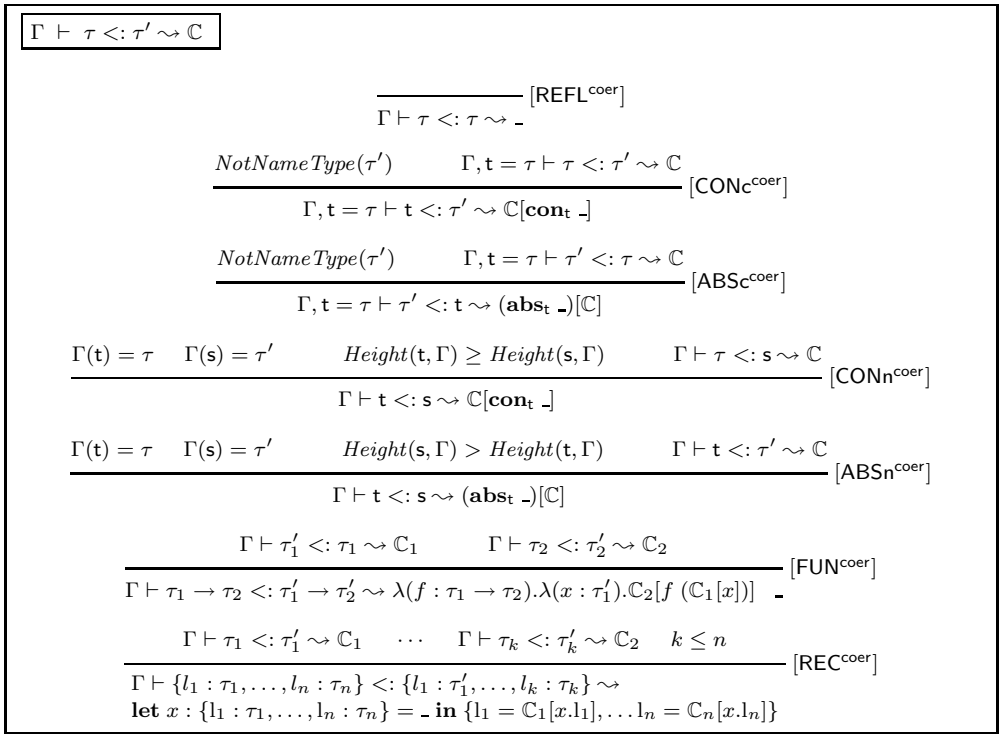


Fig. 7. Coercion generation via the subtyping relation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathsf{t} <: \mathsf{t} \rightsquigarrow \_} [\mathsf{REFL}^{\mathsf{coer}}] \qquad Height(\mathsf{v}, \Gamma) > Height(\mathsf{t}, \Gamma)}{\Gamma \vdash \mathsf{t} <: \mathsf{v} \rightsquigarrow \mathbf{abs}_{\mathsf{v}}\ \_} [\mathsf{ABSn}^{\mathsf{coer}}] \qquad Height(\mathsf{s}, \Gamma) \geq Height(\mathsf{v}, \Gamma)}{\Gamma \vdash \mathsf{s} <: \mathsf{v} \rightsquigarrow \mathbf{abs}_{\mathsf{v}}\ \mathbf{con}_{\mathsf{s}}\ \_} [\mathsf{CONn}^{\mathsf{coer}}] \qquad Height(\mathsf{u}, \Gamma) \geq Height(\mathsf{v}, \Gamma)}{\Gamma \vdash \mathsf{u} <: \mathsf{v} \rightsquigarrow \mathbf{abs}_{\mathsf{v}}\ \mathbf{con}_{\mathsf{s}}\ \mathbf{con}_{\mathsf{u}}\ \_} [\mathsf{CONn}^{\mathsf{coer}}]$$

It is relatively routine to show that whilst this system limits the number of derivations, it still encodes the same subtyping relation.

**Theorem 4.1.** *We refer to the subtyping relation with* [CON] *and* [ABS] *as* $\vdash^{nd}$ *and the relation that replaces these with* [CONn], [ABSn], [CONc], [ABSc] *as* $\vdash^{alg}$. *Then* $\Gamma \vdash^{nd} \tau <: \tau'$ *if and only if* $\Gamma \vdash^{alg} \tau <: \tau'$.

*Proof.* This is proved by relatively straightforward proof-theoretic techniques. □

4.2.3 *Algorithmic expression typing.* The final step toward a deterministic algorithm is to apply subtyping algorithmically within the typing relation. The standard approach is to remove the subsumption rule and incorporate it directly into the other rules, allowing subsumption only at the argument for application and for the right-most expression of an assignment. However, in the presence of named types, this approach is insufficient for maintaining a tight correspondence with the non-algorithmic relation. Consider an application $e_1\ e_2$. The problem occurs when $e_1$ has a named type, because subsumption is not available to expand the definition to a function type. While we cannot allow subsumption at both $e_1$ and $e_2$ as it would

not be algorithmic, we only require *unfolding*, a weaker form of subsumption, at $e_1$.[2] To this end, we introduce an unfolding judgement $\Gamma \vdash \mathsf{t} \lhd \tau$, that relates $\mathsf{t}$ and $\tau$ if the definition $\mathsf{t} = \tau$ holds directly or transitively, inserting an explicit concretion every time it unfolds a named type to its definition. The unfolding judgement is then used whenever a specific type is required in the typing judgement, i.e., in the application, dereference, assignment, and projection rules. This relation is defined as follows:

$$\frac{}{\Gamma \vdash \tau \lhd \tau \rightsquigarrow \_}$$

$$\frac{\Gamma(\mathsf{t}) = \tau' \quad \Gamma \vdash \tau' \lhd \tau \rightsquigarrow \mathbb{C}}{\Gamma \vdash \mathsf{t} \lhd \tau \rightsquigarrow \mathbb{C}[\mathbf{con}_{\mathsf{t}}\ \_]}$$

The complete rules for the translation are given in Figures 7, 8 and 9. (The normal typing rules for PROTEUS$^{\mathbf{src}}$ can be read from these figures by simply ignoring the $\rightsquigarrow\ \mathbb{C}$ parts.)

### 4.3 Operational Semantics

The operational semantics is defined using a single-step reduction relation between *configurations*, which are triples consisting of a type environment $\Omega$, a heap $H$ and an expression $e$, as shown in Figure 11. We use evaluation contexts to encode the (call-by-value) evaluation strategy.

The type environment $\Omega$ defines a configuration's named types. Each type in $\mathrm{dom}(\Omega)$ maps to a single representation $\tau$; some related approaches [Duggan 2001; Hicks 2001] would permit $\mathsf{t}$ to map to a set of representations 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 $\rho$ to pairs $(\omega, b)$, where $\omega$ is a *type tag* and $b$ is a binding. We use the heap to store both mutable references created with **ref** and top-level bindings created with **var** and **fun**; therefore $\rho$ ranges over locations $r$ and top-level identifiers $\mathsf{z}$. For locations, the type tag $\omega$ is simply $\cdot$, indicating the absence of a type, and for identifiers, e.g. $\mathsf{z}$, it is the type $\tau$ which appeared in the definition of $\mathsf{z}$ in the program. Type tags are used to type check new and replacement definitions provided by a dynamic update.

Normal configuration evaluation is defined by a relation between configurations, written $\Omega; H; e \longrightarrow \Omega; H'; e'$, and is given in Figure 12. This relation consists of a series of computations, the order of which is determined by evaluation contexts. All expressions $e$ can be uniquely decomposed into $\mathbb{E}[e']$ for some evaluation context $\mathbb{E}$ and expression $e'$, so the choice of computation rule is unambiguous. The computation rules are mostly standard; we will elaborate on the interesting rules below. We use the notation $e[v/x]$ to denote the capture-avoiding substitution of $v$ for $x$ within $e$.

[2] The alert reader may have noticed that the subtype relation is in fact sufficient to get algorithmic behaviour at application, provided we drop the notion of applying subsumption to the arguments of functions, and instead apply it to the *function type*. Although theoretically simpler, in practice we want to avoid coercing functions, which is expensive. Moreover, we need the unfolding at projection and dereference in any case, thus it seems a more general concept to apply unfolding at *destruct positions* — points where the top-level structure of a value is deconstructed [Bierman et al. 2003].

$\boxed{\Gamma \vdash e : \tau \rightsquigarrow e'}$

$$\Gamma \vdash n : \mathsf{int} \rightsquigarrow n$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow x}$$

$$\frac{\Gamma(\mathsf{z}) = \tau}{\Gamma \vdash \mathsf{z} : \tau \rightsquigarrow \mathsf{z}}$$

$$\frac{\Gamma \vdash e_i : \tau_i \rightsquigarrow e'_i \qquad (i \in 1..n)}{\Gamma \vdash \{\mathrm{l}_1 = e_1, \ldots, \mathrm{l}_n = e_n\} : \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\} \rightsquigarrow \{\mathrm{l}_1 = e'_1, \ldots, \mathrm{l}_n = e'_n\}}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e' \qquad \Gamma \vdash \tau \lhd \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\} \rightsquigarrow \mathbb{C}}{\Gamma \vdash e.\mathrm{l}_i : \tau_i \rightsquigarrow \mathbb{C}[e'].\mathrm{l}_i}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \rightsquigarrow e'_1 \qquad \Gamma \vdash e_2 : \tau'_1 \rightsquigarrow e'_2 \\ \Gamma \vdash \tau \lhd \tau_1 \to \tau_2 \rightsquigarrow \mathbb{C}_1 \qquad \Gamma \vdash \tau'_1 <: \tau_1 \rightsquigarrow \mathbb{C}_1\end{array}}{\Gamma \vdash e_1\ e_2 : \tau_2 \rightsquigarrow \mathbb{C}_1[e'_1]\ \mathbb{C}_2[e'_2]}$$

$$\frac{\Gamma \vdash e_1 : \tau'_1 \rightsquigarrow e'_1 \qquad \Gamma \vdash \tau'_1 <: \tau_1 \rightsquigarrow \mathbb{C} \qquad \Gamma, x\colon \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow e'_2}{\Gamma \vdash \mathbf{let}\ x : \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau_2 \rightsquigarrow \mathbf{let}\ x : \tau_1 = \mathbb{C}[e'_1]\ \mathbf{in}\ e'_2}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e'}{\Gamma \vdash \mathbf{ref}\ e : \tau\ \mathbf{ref} \rightsquigarrow \mathbf{ref}\ e'}$$

$$\frac{\Gamma \vdash e : \tau' \rightsquigarrow e' \qquad \Gamma \vdash \tau' \lhd \tau\ \mathbf{ref} \rightsquigarrow \mathbb{C}}{\Gamma \vdash\ !e : \tau \rightsquigarrow\ !\mathbb{C}[e']}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau' \rightsquigarrow e'_1 \qquad \Gamma \vdash e_2 : \tau'' \rightsquigarrow e'_2 \\ \Gamma \vdash \tau' \lhd \tau\ \mathbf{ref} \rightsquigarrow \mathbb{C}_1 \qquad \Gamma \vdash \tau'' <: \tau \rightsquigarrow \mathbb{C}_2\end{array}}{\Gamma \vdash e_1 := e_2 : \tau \rightsquigarrow \mathbb{C}_1[e'_1] := \mathbb{C}_2[e'_2]}$$

$$\Gamma \vdash \mathbf{update} : \mathsf{int} \rightsquigarrow \mathbf{update}$$

Fig. 8. Con/abs insertion for compiling PROTEUS$^{\mathbf{src}}$ expressions

$\boxed{\Gamma \vdash P : \tau \rightsquigarrow P'}$

$$\frac{\Gamma \vdash \tau\ \mathrm{OK} \qquad \Gamma, \mathsf{t} = \tau \vdash P : \tau \rightsquigarrow P'}{\Gamma \vdash \mathbf{type}\ \mathsf{t} = \tau\ \mathbf{in}\ P : \tau \rightsquigarrow \mathbf{type}\ \mathsf{t} = \tau\ \mathbf{in}\ P'}$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma, \mathsf{z}\colon \tau_1 \to \tau_2 \qquad \Gamma', x\colon \tau_1 \vdash e : \tau' \rightsquigarrow e' \\ \Gamma' \vdash \tau' <: \tau_2 \rightsquigarrow \mathbb{C} \qquad \Gamma' \vdash P : \tau \rightsquigarrow P'\end{array}}{\Gamma \vdash \mathbf{fun}\ \mathsf{z}(x : \tau_1) : \tau_2 = e\ \mathbf{in}\ P : \tau \rightsquigarrow \mathbf{fun}\ \mathsf{z}(x : \tau_1) : \tau_2 = \mathbb{C}[e']\ \mathbf{in}\ P'}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : \tau'' \rightsquigarrow e' \qquad \Gamma \vdash \tau'' <: \tau' \rightsquigarrow \mathbb{C} \\ \Gamma, \mathsf{z} : \tau'\ \mathbf{ref} \vdash P : \tau \rightsquigarrow P'\end{array}}{\Gamma \vdash \mathbf{var}\ \mathsf{z} : \tau' = v\ \mathbf{in}\ P : \tau \rightsquigarrow \mathbf{var}\ \mathsf{z} : \tau' = \mathbb{C}[e']\ \mathbf{in}\ P'}$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e'}{\Gamma \vdash \mathbf{return}\ e : \tau \rightsquigarrow e'}$$

Fig. 9. Con/abs insertion for compiling PROTEUS$^{\mathbf{src}}$ programs

$\boxed{\Gamma \vdash \tau \text{ OK}}$

$$\Gamma \vdash \mathsf{int} \qquad \frac{\mathsf{t} \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \mathsf{t}} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \tau\ \mathbf{ref}}$$

$$\frac{\Gamma \vdash \tau_i \qquad i \in 1..n}{\Gamma \vdash \{\mathsf{l}_1 : \tau_1, ..., \mathsf{l}_n : \tau_n\}} \qquad \frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}$$

Fig. 10. Type well-formedness

**Syntax**

| | | | |
|---|---|---|---|
| Heap (binding) expressions | $b \in \mathrm{HExp}$ | $::=$ | $e \mid \lambda(x).e$ |
| Heap (binding) values | $b_v \in \mathrm{HVal}$ | $::=$ | $v \mid \lambda(x).e$ |
| Heaps | $H \in \mathrm{Heap}$ | $::=$ | $\emptyset \mid r \mapsto (\cdot, b), H \mid \mathsf{z} \mapsto (\tau, b), H$ |
| Type environment | $\Omega \in \mathrm{TEnv}$ | $::=$ | $\emptyset \mid \mathsf{t} \mapsto \tau, \Omega$ |
| Configurations | $\mathrm{cfg} \in \mathrm{CFG}$ | $::=$ | $\Omega; H; e$ |
| Evaluation context | $\mathbb{E}$ | $::=$ | $\_ \mid \{\mathsf{l}_1 = v_1, \ldots, \mathsf{l}_i = \mathbb{E}, \ldots, \mathsf{l}_n = e_n\}$ |
| | | $\mid$ | $\mathbb{E}.\mathsf{l} \mid \mathbb{E}\ e \mid v\ \mathbb{E} \mid \mathbf{let}\ z = \mathbb{E}\ \mathbf{in}\ e$ |
| | | $\mid$ | $\mathbf{ref}\ \mathbb{E} \mid !\mathbb{E} \mid \mathbb{E} := e \mid r := \mathbb{E}$ |
| | | $\mid$ | $\mathbf{con}_{\mathsf{t}}\ \mathbb{E} \mid \mathbf{abs}_{\mathsf{t}}\ \mathbb{E}$ |
| | | $\mid$ | $\mathbf{if}\ \mathbb{E} = e\ \mathbf{then}\ e\ \mathbf{else}\ e$ |
| | | $\mid$ | $\mathbf{if}\ v = \mathbb{E}\ \mathbf{then}\ e\ \mathbf{else}\ e$ |

**Updates**

| | | | |
|---|---|---|---|
| Updates | $U \in \mathrm{Upd}$ | $::=$ | $\{\mathrm{UN} = \mathrm{un}, \mathrm{AN} = \mathrm{an}, \mathrm{UB} = \mathrm{ub}, \mathrm{AB} = \mathrm{ab}\}$ |
| Updated Named Types | un | $\in$ | $\mathrm{NT} \rightharpoonup \mathrm{Typ} \times \mathrm{XVar}$ |
| Additional Named Types | an | $\in$ | $\mathrm{NT} \rightharpoonup \mathrm{TEnv}$ |
| Additional Bindings | ab | $\in$ | $\mathrm{XVar} \rightharpoonup \mathrm{Typ} \times \mathrm{HVal}$ |
| Updated Bindings | ub | $\in$ | $\mathrm{XVar} \rightharpoonup \mathrm{Typ} \times \mathrm{HVal}$ |

Fig. 11. Syntax for dynamic semantics of PROTEUS$^{\mathbf{con}}$

A program $P$ is compiled into a configuration $\Omega; H; e = \mathcal{C}(\emptyset; \emptyset; P)$, as shown in the bottom of Figure 12. Type defintions **type** $\mathsf{t} = \tau$ in the program are added to the configuration's type environment $\Omega$. Function definitions **fun** $\mathsf{f}(x : \tau_1) : \tau = e$ are stored in the heap as lambda terms $\lambda(x).e$ indexed by their identifier $\mathsf{f}$. The operational rules will never permit these lambda terms from appearing in the active expression. Finally, top-level identifier definitions **var** $\mathsf{z} : \tau = v$ are stored in the heap as we would expect.

Next, we consider how our semantics expresses the interesting operations of dynamic updating: (1) updating top-level identifiers $\mathsf{z}$ with new definitions, and (2) updating type definitions $\mathsf{t}$ to have a different representation.

*Replacing Top-level Identifiers.* A top-level identifier $\mathsf{z}$ from the source program is essentially a statically-allocated reference cell. As a result, at update-time we can change $\mathsf{z}$'s binding in the heap; afterwards any code that accesses (dereferences) $\mathsf{z}$ 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 $(\tau_1 \to \tau_2)$ **ref**, we simply give them type $\tau_1 \to \tau_2$, and perform the dereference as part of the (CALL) rule. This has the pleasant side effect of rendering top-level functions immutable during

**Computation:** $H; e \longrightarrow H'; e'$

$$H; \{\mathsf{l}_1 = v_1, \ldots, \mathsf{l}_n = v_n\}.\mathsf{l}_i \longrightarrow H; v_i \qquad \text{(PROJ)}$$

$$H; \mathbf{con}_{\mathsf{t}}\ (\mathbf{abs}_{\mathsf{t}}\ v) \longrightarrow H; v \qquad \text{(CONABS)}$$

$$(H, \mathsf{z} \mapsto (\tau, \lambda(x).e\ )); \mathsf{z}\ v \longrightarrow (H, \mathsf{z} \mapsto (\tau, \lambda(x).e\ )); e[v/x] \qquad \text{(CALL)}$$

$$H; \mathbf{let}\ x : \tau = v\ \mathbf{in}\ e \longrightarrow H; e[v/x] \qquad \text{(LET)}$$

$$H; \mathbf{ref}\ v \longrightarrow (H, r \mapsto (\cdot, v)); r \qquad \text{(REF)}$$

$$(H, \rho \mapsto (\omega, e));\ !\rho \longrightarrow (H, \rho \mapsto (\omega, e)); \rho := e \qquad \text{(DEREF)}$$

$$(H, \rho \mapsto (\omega, e)); \rho := v \longrightarrow (H, \rho \mapsto (\omega, v)); v \qquad \text{(ASSIGN)}$$

$$H; \mathbf{if}\ v1 = v2\ \mathbf{then}\ e1\ \mathbf{else}\ e2 \longrightarrow H; e1 \quad (\text{where } v1 = v2) \qquad \text{(IF-T)}$$

$$H; \mathbf{if}\ v1 = v2\ \mathbf{then}\ e1\ \mathbf{else}\ e2 \longrightarrow H; e2 \quad (\text{where } v1 \neq v2) \qquad \text{(IF-F)}$$

$$H; \mathbf{update} \longrightarrow H; 1 \qquad \text{(NO-UPDATE)}$$

**Configuration Evaluation:** $\Omega; H; e \longrightarrow \Omega'; H'; e'$

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

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

otherwise: $\Omega; H; \mathbb{E}[\mathbf{update}] \xrightarrow{\text{upd}} \mathbb{E}[1]$

**Compilation:** $\mathcal{C}(\Omega; H; P) = \Omega; H; e$

$$\mathcal{C}(\Omega; H; e) = \Omega; H; e$$

$$\mathcal{C}(\Omega; H; \mathbf{type}\ \mathsf{t} = \tau\ \mathbf{in}\ P) = \mathcal{C}(\Omega, \mathsf{t} = \tau; H; P)$$

$$\mathcal{C}\left(\Omega; H; \begin{pmatrix} \mathbf{fun}\ \mathsf{f}_1(x : \tau_1) : \tau_1' = e_1 \ldots \\ \mathbf{and}\ \mathsf{f}_n(x : \tau_n) : \tau_n' = e_n\ \mathbf{in}\ P \end{pmatrix}\right) =$$
$$\mathcal{C}(\Omega; H, \mathsf{f}_1 \mapsto (\tau \to \tau', \lambda(x).e_1\ ), \ldots, \mathsf{f}_n \mapsto (\tau \to \tau', \lambda(x).e_n\ ); P)$$

$$\mathcal{C}(\Omega; H; \mathbf{var}\ \mathsf{z} : \tau = v\ \mathbf{in}\ P) = \mathcal{C}(\Omega; H, \mathsf{z} \mapsto (\tau, v); P)$$

Fig. 12. Dynamic semantics for PROTEUS$^{\mathbf{con}}$

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 $\tau$ to be able to type check new and replacement bindings. Some formulations of dynamic linking define a *heap interface*, which maps variables $\mathsf{z}$ to types $\tau$ [Hicks et al. 2000], but we find it more convenient to merge this interface into the heap itself.

*Updating Data of Named Type.* As mentioned in §4.2, PROTEUS$^{\mathbf{con}}$ uses coercions to identify where data of a type $\mathsf{t}$ 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 $\mathsf{c}$ for an updated type $\mathsf{t}$, we rewrite each occurrence $\mathbf{abs}_{\mathsf{t}}\ e$ to be $\mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ e)$ using the $\mathcal{U}[-]^{\text{upd}}$ transformation, explained next. 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, and our

$\boxed{\mathcal{U}[H]^{\mathrm{upd}}}$

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

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

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

$\boxed{\mathcal{U}[b]^{\mathrm{upd}}}$

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

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

For remaining $b$ containing subterms $e_1, \ldots, e_n$: $\mathcal{U}[b]^{\mathrm{upd}} = b$ with $\mathcal{U}[e_1]^{\mathrm{upd}} \ldots \mathcal{U}[e_n]^{\mathrm{upd}}$

$\boxed{\mathcal{U}[\Omega]^{\mathrm{upd}}}$

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

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

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

$\boxed{\mathcal{U}[\Gamma]^{\mathrm{upd}}}$

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

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

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

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

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

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

Fig. 13. Dynamically updating a program: $\mathcal{U}[-]^{\mathrm{upd}}$

(DEREF) rule evaluates the contents of the reference and then *writes back* the result before proceeding. Whether the coercions are in the heap or the program, when they are executed is (for all intents and purposes) unpredictable. Indeed, in our implementation (§7), we take a lazy approach to updating typed values, delaying the evaluation of the coercion function until the value is actually used, in the style of Duggan [Duggan 2001]. As a result, coercions should be written to act locally and avoid side-effecting computation. One could imagine enforcing this, but we do

**Configuration typing:** $\vdash\ \Omega; H, e : \tau$

$$\frac{\vdash\ \Omega \qquad \Omega; \Phi\ \vdash\ H \qquad \Omega, \Phi \vdash_{\mathsf{U}} e : \tau}{\vdash\ \Omega; H; e : \tau}$$

**Heap typing:** $\Omega; \Phi\ \vdash\ H$

$$\frac{\begin{array}{l}\mathrm{dom}(\Phi) = \mathrm{dom}(H) \\ \forall \mathsf{z} \mapsto (\tau \to \tau', \lambda(x).e\ ) \in H.\quad \Omega, \Phi, x : \tau \vdash e : \tau'\ \wedge\ \Phi(\mathsf{z}) = \tau \to \tau' \\ \forall \mathsf{z} \mapsto (\tau, e) \in H.\quad \Omega, \Phi \vdash e : \tau\ \wedge\ \Phi(\mathsf{z}) = \tau\ \mathbf{ref} \\ \forall r \mapsto (\cdot, e) \in H.\quad \Omega, \Phi \vdash e : \tau\ \wedge\ \Phi(r) = \tau\ \mathbf{ref}\end{array}}{\Omega; \Phi\ \vdash\ H}$$

**Type Environment Well-formedness:** $\Omega\ \vdash\ \Omega$

$$\Omega\ \vdash\ \emptyset \qquad \frac{\Omega, \mathsf{t} = \tau\ \vdash\ \Omega' \qquad \Omega\ \vdash\ \tau\ \mathrm{OK}}{\Omega\ \vdash\ \mathsf{t} = \tau, \Omega'}$$

Fig. 14. PROTEUS$^{\mathbf{con}}$ Configuration typing

not do so here.

*Update Semantics.* When no dynamic update is available, an **update** expression simply evaluates normally to 1, by (NO-UPDATE). An available dynamic update upd is modeled with a *labeled transition*, where upd labels the arrow. Rule (UPDATE) specifies that if upd is well-formed (by updateOK(−), described in the next section), the **update** evaluates to 0, and the program is updated by transforming the current type environment $\Omega$, heap $H$, and expression $e$ according to $\mathcal{U}[-]^{\mathrm{upd}}$, defined in Figure 13; otherwise the **update** expression evaluates to 1. To ensure representation consistency, $\mathcal{U}[-]^{\mathrm{upd}}$ applies type transformation functions to all $\mathbf{abs}_{\mathsf{t}}\ e$ expressions of a named type $\mathsf{t}$ that is being updated. When transforming the heap, it replaces top-level identifier definitions with their new versions, and adds all of the new bindings. When transforming $\Omega$, it replaces type definitions with their new versions, and inserts new definitions into specified places in the list. Also shown in the figure is the definition for updating a typing context $\Gamma$; this is used in the definition of updateOK(−), described in the next two sections.

*Typing.* We define typing rules for the well-formedness of configurations in Figure 14; these will be used in the proof of type soundness presented in §4.5. The judgement $\vdash\ \Omega; H; e : \tau$ indicates that a configuration is well-formed. Configuration well-formedness is predicated on the existence of some $\Phi$, called the *heap interface*, that properly maps top-level identifiers $\mathsf{z}$ and references $r$ to types $\tau$. That is, a configuration is well-formed as long as there exists some $\Phi$ sufficient to type check the heap ($\Omega; \Phi\ \vdash\ H$) and to type check the active expression. Note that we write $\Omega, \Phi$ to denote the concatenation of the heap interface and the configuration type environment, which defines the $\Gamma$ used to type check the active expression $e$.

The heap typing judgement establishes two facts: (1) each of the types in $\Phi$ accurately represents the types of the bindings found in $H$; (2) each of the bindings in the heap type checks under $\Phi$ and the current type environment $\Omega$. Assuming the existence of a $\Phi$ permits cycles in the reference graph.

The type environment $\Omega$ must be consistent, which we establish with the judgement $\Omega \vdash \Omega$. This is particularly important when an update is applied as we must

ensure that the resulting type environment is valid. The rules in the figure ensure this by requiring all types mentioned in other types to be both defined and linearly orderable (non-recursive). However, it is not hard to treat types $\mathsf{t}$ as iso-recursive since $\mathbf{con}_{\mathsf{t}}\ e$ and $\mathbf{abs}_{\mathsf{t}}\ e$ correspond precisely to the mediating coercions $\mathbf{fold}_t\ e$ and $\mathbf{unfold}_t\ e$ of iso-recursive types [Gapeyev et al. 2000]. Therefore, all of our types could be considered implicitly recursive.

### 4.4 Update Safety

The conditions for ensuring that updates are type-safe are formally expressed in the definition of the updateOK(−) predicate that is a precondition to the (UPDATE) rule in Figure 12. In particular, if updateOK(−) holds, then applying the update will yield a type-correct program.

The definition of updateOK(−) has two components: (1) it ensures that the update is compatible with the program's definitions, and (2) that it is compatible with the current state of the program, i.e., the **return** $e$ part and the heap $H$. The former is a static property, in the sense that the information to perform it is available provided one has the original source and the updates previously applied. The interesting part is the latter component, since satisfying it depends on the timing of the update.

$$
\begin{array}{l}
\textbf{let}\ \mathrm{i} = \mathsf{post}\ ( \\
\quad \textbf{let}\ \mathrm{u2} = \textbf{update in} \\
\quad \textbf{let}\ \mathrm{res} = (\mathbf{con}_{\mathsf{sockhandler}}\ \mathbf{abs}_{\mathsf{sockhandler}}\ \mathsf{udp\_read}) \\
\qquad \{\mathrm{sock} = v_{\mathsf{sock}}, \mathrm{buf} = (\mathbf{con}_{\mathsf{req}}\ v_{\mathsf{req}}).\mathrm{buf}, \mathrm{sflags} = v_{\mathsf{sflags}}\}\ \textbf{in} \\
\quad \textbf{let}\ \mathrm{u3} = \textbf{update in}\ \mathrm{res} \\
)\ \textbf{in}\ \mathsf{loop}\ \mathrm{i}
\end{array}
$$

Fig. 15. Example active expression

To ensure that an update is well-timed, it must be applied at a program point that is *con-free*. To understand what this means, consider again the example from §3.3. Figure 15 illustrates the active expression in PROTEUS$^{\mathbf{con}}$ equivalent to the PROTEUS$^{\mathbf{src}}$ epxression shown in Figure 5 of that section. Notice that now the call to udp_read as a concrete usage of type sockhandler is made manifest by type coercions. We can clearly see that this point in the program is not "con-t-free" since it will manipulate $\mathsf{t}$ values concretely immediately following the update. In general, we say a configuration $\Omega; H; e$ is *con-free* for an update upd if for all named types $\mathsf{t}$ that the update will change, $\mathbf{con}_{\mathsf{t}}$ is not a subexpression of (1) the active expression $e$ or (2) 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 16. Part (2) is captured by the first rule in the figure: functions to be replaced (i.e., those in upd.UB) are not checked. The existence of a $\mathbf{con}_{\mathsf{t}}$ in some other part of the program $P$ will cause $\mathrm{conFree}[\,P\,]^{\mathrm{upd}}$ to fail as exemplified by the rule for $\mathrm{conFree}[\,\mathbf{con}_{\mathsf{t}}\ e\,]^{\mathrm{upd}}$.

The conditions for update well-formedness in updateOK(−) aim to ensure that type-safety is maintained following the addition or replacement of code and types. The types($H$) predicate extracts all of the type tags from $H$ and constructs a suitable $\Gamma$ for typechecking the new or replacement bindings. Since heap objects are stored with their declared type $\tau$, if they are not functions then in $\Gamma$ they are

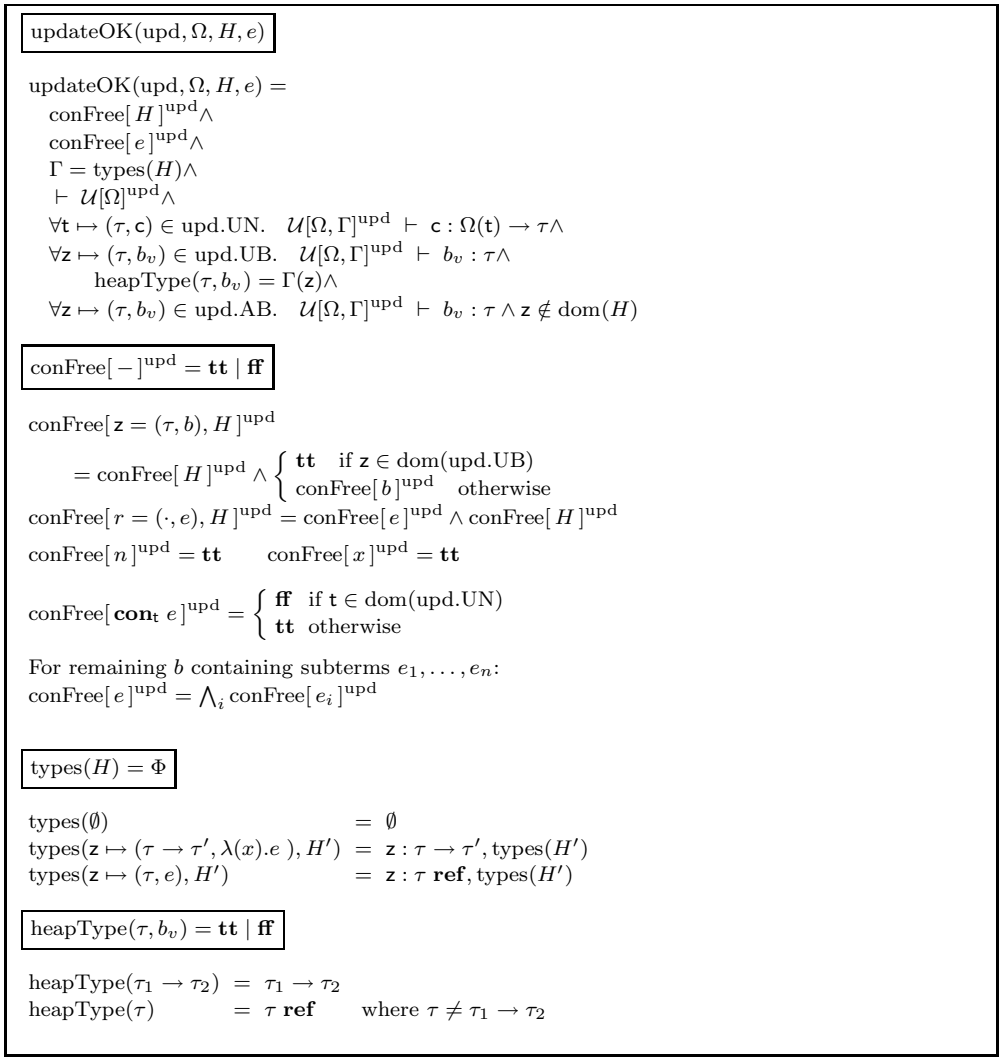


Fig. 16. updateOK(−) predicate for defining legal updates, and supporting definitions

given type $\tau$ **ref**. Next, the updated type environment $\mathcal{U}[\Omega]^{\mathrm{upd}}$ is checked for well-formedness. Then, using the updated $\Omega$ and $\Gamma$, we check that the type transformer functions, replacement bindings and new bindings are all well-typed. These type-checks apply only to expressions contained in the update—none of the existing code must be rechecked (though its types, as stored in the heap, are needed to check the new code).

A natural question is "how likely is it in practice that a given dynamic update will occur at an **update** that is con-free?" This depends on the placement of the **update** in the program text. If the **update** is at a program point for which there will be a fairly shallow call stack (corresponding to a small active expression $e$ in configuration $\Omega; H; e$), then there is less possibility that some $\mathbf{con}_{\mathsf{t}}$ exists that will prevent the update. The contents of the heap will not play a significant role in update timing, assuming that the new update indeed replaces functions that use a type whose definition has changed. For a program like our example that is

structured around a top-level event loop, placing **update** just prior to the recursive call of that loop yields the fewest restrictions: it is the top-level loop, so there is no call stack, and moreover the current invocation of the function that implements the loop has finished. An update at that point will be accepted even if the loop function has changed, because the next call to that function will be to the newest version, which is checked to be compatible with any changed type definitions at update-time. All single-threaded, long-running programs we have encountered have such update-friendly points in them [Hicks 2001].

### 4.5 Properties

Our main theorem is that PROTEUS$^{\mathbf{con}}$ is type safe.

**Theorem 4.2** (Type Soundness)**.** *If* $\vdash\ \Omega; H; e : \tau$ *then either*

*(i) there exists* $\Omega', H', e'$ *such that* $\Omega; H; e \to \Omega'; H'; e'$ *and* $\vdash \Omega'; H'; e' : \tau$ *or*
*(ii)* $e$ *is a value*

This theorem states that a well-typed program is either a value, or is able to reduce (and remain well-typed). The most interesting case in proving type preservation is the (UPDATE) rule, for which we must prove a lemma that well-formed and well-timed updates lead to well-typed programs:

**Lemma 4.3** ($\mathcal{U}[-]^{-}$ preserves type-safety)**.** *Given* $\vdash\ \Omega; H; e$ *and an update,* upd*, for which we have* updateOK$(\text{upd}, \Omega, H, e)$*, then* $\vdash\ \mathcal{U}[\,\Omega\,]^{\text{upd}}; \mathcal{U}[\,H\,]^{\text{upd}}, \mathcal{U}[\,e\,]^{\text{upd}} : \tau$*.*

Proof sketches appear in Appendix A.2.

## 5. STATIC UPDATE SAFETY

The updateOK$(-)$ predicate determines the legality of an update upd's form and its timing when applied to program $P$. Well-formedness is based only on the substance of upd and the types of the $P$'s *definitions*; that is, the bindings $\mathsf{z}$ in its heap $H$ and the type definitions $\mathsf{t}$ in its type environment $\Omega$. This type information is invariant with $P$'s execution, and so the well-formedness of upd can be known in advance, before it is applied, by checking it against the program text. In contrast, the well-timing of upd is determined by the conFree$[-]^{\text{upd}}$ predicate, which relies on the program's *current state*, which is the active expression $e$ and the bindings in the heap. Implementing this check directly has two drawbacks:

(1) It could be expensive. $P$'s active expression $e$ models the stack and program counter in an actual implementation, and so to implement conFree$[\,P\,]^{\text{upd}}$ would require traversing the stack and code to look for concrete uses of changed types.
(2) More importantly, it could fail if the update is applied at a bad time. While an easy remedy is to delay the update until an acceptable **update** location is reached, there is no assurance that such a location exists. For example, if all **update** points in the program $P$ immediately precede the expression $\mathbf{con}_{\mathsf{t}}\ e$, then *any* update that includes a change to $\mathsf{t}$ will always fail because conFree$[\,\mathbf{con}_{\mathsf{t}}\ e\,]^{\text{upd}}$ will fail for all upd. As this is due to the structure of $P$, the programmer would like to learn of this fact *before* $P$ is deployed, to ensure that future, unanticipated changes to $\mathsf{t}$ may safely take place.

In this section we present a way to *statically* infer, for each **update** point in the program, those named types $\mathsf{t}$ for which the program may not be con-$\mathsf{t}$-free at that

$$
\begin{array}{rrcl}
\text{Capabilities} & \Delta & ::= & \{\mathsf{t}_1, \ldots \mathsf{t}_n\} \mid \Delta \cap \Delta \\
\text{Updateability} & \mu & ::= & \mathsf{U} \mid \mathsf{N} \\
\text{Types} & \tau & ::= & \cdots \mid \tau \xrightarrow{\mu;\Delta} \tau \\
\text{Expressions} & e & ::= & \cdots \mid \mathbf{update}^{\Delta} \\
\text{Programs} & P & ::= & \cdots \mid \mathbf{fun}\ \mathsf{z}^{\mu;\Delta;\Delta'}(x:\tau_1):\tau_2 = e\ \mathbf{in}\ P \\
\text{Heap expressions} & b & ::= & e \mid \lambda^{\Delta}(x).e \\
\text{Heap values} & b_v & ::= & v \mid \lambda^{\Delta}(x).e
\end{array}
$$

Fig. 17. Extended syntax for PROTEUS$^{\Delta}$

point. With this information, we can discover which, if any, **update** points are con-free for *all* types, meaning that the **update** will accept any well-formed update, whatever it may be. This is the kind of guarantee enjoyed by standard dynamic linking, which only adds bindings to the program, but does not replace them. In addition to supporting this kind of reasoning, static inference also permits a simpler, more efficient implementation of updateOK$(-)$, without need of conFree$[-]$. We call our static inference an *updateability analysis*, and formulate it as a type and inference system. This section presents the type rules and shows the analysis to be sound.

### 5.1 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 **update** whether it will be acceptable to update some type $\mathsf{t}$. An update to $\mathsf{t}$ will be unacceptable if an occurrence of $\mathbf{con}_{\mathsf{t}}$ exists in any old code that could be evaluated in the continuation of the **update**. If we can discover all possible such occurrences of $\mathbf{con}_{\mathsf{t}}$, we could annotate **update** with a list of those types $\mathsf{t}$; call this annotation $\Delta$. Then, (along with its well-formedness checks) updateOK$(-)$ could ascertain con-freeness by merely checking that for all $\mathsf{t} \in$ upd.UN, $\mathsf{t} \notin \Delta$. This is substantially simpler than conFree$[-]$.

We call this annotation $\Delta$ a *capability*, since it serves as a bound on what types may be used concretely in the continuation of an **update**. That is, roughly speaking, any code following an **update** must type check using $\Gamma$ restricted to those types listed in the capability. Since an **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 **update** points with empty capabilities, we can be sure that *no* update will fail due to bad timing.

### 5.2 Typing

We define a type system that tracks the capability at each program point to ensure that **update**s are annotated soundly. To do this, we introduce a new target language, PROTEUS$^{\Delta}$, that differs from PROTEUS$^{\mathbf{con}}$ in two ways: (1) functions, function types, and **update** are annotated with capabilities; (2) each function and function type is annotated with an *updateability* $\mu$, which indicates whether a dynamic update may occur as a result of calling the function. The syntax changes are shown in Figure 17. We must also adjust compilation (Figure 12) in the case of functions to add the necessary annotation on the generated binding and type (the **and** case is obvious and not shown):

$$\mathcal{C}(\Omega; H; \mathbf{fun}\ \mathsf{f}^{\mu;\Delta;\Delta'}(x:\tau):\tau' = e\ \mathbf{in}\ P) = \mathcal{C}(\Omega; H, \mathsf{f} \mapsto (\tau \xrightarrow{\mu;\Delta'} \tau', \lambda^{\Delta}(x).e\ ); P)$$

**Expression Typing:** $\Delta; \Gamma \vdash_\mu e : \tau; \Delta'$

$$\Delta; \Gamma \vdash_\mu n : \mathsf{int}; \Delta \qquad \text{(A.EXPR.INT)}$$

$$\frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash_\mu x : \tau; \Delta} \qquad \text{(A.EXPR.VAR)}$$

$$\frac{\Gamma(\mathsf{z}) = \tau}{\Delta; \Gamma \vdash_\mu \mathsf{z} : \tau; \Delta} \qquad \text{(A.EXPR.XVAR)}$$

$$\Delta; \Gamma, r : \tau\ \mathbf{ref} \vdash_\mu r : \tau\ \mathbf{ref}; \Delta \qquad \text{(A.EXPR.LOC)}$$

$$\frac{\Delta_i; \Gamma \vdash_\mu e_{i+1} : \tau_{i+1}; \Delta_{i+1} \qquad i \in 1..(n-1) \qquad n \geq 0}{\Delta_0; \Gamma \vdash_\mu \{\mathsf{l}_1 = e_1, \ldots, \mathsf{l}_n = e_n\} : \{\mathsf{l}_1 : \tau_1, \ldots, \mathsf{l}_n : \tau_n\}; \Delta_n} \qquad \text{(A.EXPR.RECORD)}$$

$$\frac{\Delta; \Gamma \vdash_\mu e : \{\mathsf{l}_1 : \tau_1, \ldots, \mathsf{l}_n : \tau_n\}; \Delta'}{\Delta; \Gamma \vdash_\mu e.\mathsf{l}_i : \tau_i; \Delta'} \qquad \text{(A.EXPR.PROJ)}$$

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash_\mu e_1 : \tau_1 \xrightarrow{\hat{\mu};\hat{\Delta}} \tau_2; \Delta' \\ \Delta'; \Gamma \vdash_\mu e_2 : \tau_1; \Delta'' \qquad \Delta''' \subseteq \Delta'' \\ (\hat{\mu} = \mathsf{U}) \Rightarrow (\mu = \mathsf{U} \wedge \Delta''' \subseteq \hat{\Delta})\end{array}}{\Delta; \Gamma \vdash_\mu e_1\ e_2 : \tau_2; \Delta'''} \qquad \text{(A.EXPR.APP)}$$

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash_\mu e : \tau; \Delta_1 \qquad \Delta_1; \Gamma \vdash_\mu e' : \tau; \Delta_2 \\ \Delta_2; \Gamma \vdash_\mu e_1 : \tau'; \Delta_3 \qquad \Delta_2; \Gamma \vdash_\mu e_2 : \tau'; \Delta_4\end{array}}{\Delta; \Gamma \vdash_\mu \mathbf{if}\ e = e'\ \mathbf{then}\ e_1 \mathbf{else}\ e_2 : \tau'; \Delta_3 \cap \Delta_4} \qquad \text{(A.EXPR.IF)}$$

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash_\mu e_1 : \tau_1'; \Delta' \\ \Delta'; \Gamma, x : \tau_1 \vdash_\mu e_2 : \tau_2; \Delta''\end{array}}{\Delta; \Gamma \vdash_\mu \mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2 : \tau_2; \Delta''} \qquad \text{(A.EXPR.LET)}$$

$$\frac{\Delta; \Gamma \vdash_\mu e : \tau; \Delta'}{\Delta; \Gamma \vdash_\mu \mathbf{ref}\ e : \tau\ \mathbf{ref}; \Delta'} \qquad \text{(A.EXPR.REF)}$$

$$\frac{\Delta; \Gamma \vdash_\mu e : \tau\ \mathbf{ref}; \Delta'}{\Delta; \Gamma \vdash_\mu !e : \tau; \Delta'} \qquad \text{(A.EXPR.DEREF)}$$

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash_\mu e_1 : \tau\ \mathbf{ref}; \Delta' \\ \Delta'; \Gamma \vdash_\mu e_2 : \tau; \Delta''\end{array}}{\Delta; \Gamma \vdash_\mu e_1 := e_2 : \tau; \Delta''} \qquad \text{(A.EXPR.ASSIGN)}$$

Fig. 18. Expression judgements for PROTEUS$^\Delta$ (part I)

Here, the $\Delta$ annotation on $\lambda^\Delta(x).e$ is used in the update-time safety check, as we show later. For the remainder of this section, we consider the type system for PROTEUS$^\Delta$, covering judgements for expressions, programs, and configurations.

5.2.1 *Expression Typing.* The rules for typing expressions are given in Figures 18 and 19, defining judgement $\Delta; \Gamma \vdash_\mu e : \tau; \Delta'$. Here, $\Delta$ is the capability before $e$ is evaluated, and $\Delta'$ is the capability afterward. Each rule is actually a family of rules parameterized by an *updateability* $\mu$: updateability $\mathsf{U}$ indicates a dynamic update may be performed while evaluating the given expression, while $\mathsf{N}$ indicates that no update is permitted. This is used to rule out dynamic updates in undesirable contexts, as we explain in the next subsection.

$$\frac{\Delta' \subseteq \Delta}{\Delta; \Gamma \vdash_{\mathsf{U}} \mathbf{update}^{\Delta'} : \mathsf{int}; \Delta'} \quad \text{(A.EXPR.UPDATE)}$$

$$\frac{\Delta' \subseteq \Delta \qquad \Delta'; \Gamma \vdash_{\mathsf{U}} e_1 : \tau'; \Delta_1 \qquad \Delta; \Gamma \vdash_{\mathsf{U}} e_2 : \tau'; \Delta_2}{\Delta; \Gamma \vdash_{\mathsf{U}} \mathbf{if}\ \mathbf{update}^{\Delta'} = 0\ \mathbf{then}\ e_1 \mathbf{else}\ e_2 : \tau'; \Delta_1 \cap \Delta_2} \quad \text{(A.EXPR.IFUPDATE)}$$

$$\frac{\Delta; \Gamma \vdash_{\mu} e : \mathsf{t}; \Delta' \qquad \Gamma(\mathsf{t}) = \tau \qquad \mathsf{t} \in \Delta'}{\Delta; \Gamma \vdash_{\mu} \mathbf{con}_{\mathsf{t}}\ e : \tau; \Delta'} \quad \text{(A.EXPR.CON)}$$

$$\frac{\Delta; \Gamma \vdash_{\mu} e : \tau; \Delta' \qquad \Gamma(\mathsf{t}) = \tau}{\Delta; \Gamma \vdash_{\mu} \mathbf{abs}_{\mathsf{t}}\ e : \mathsf{t}; \Delta'} \quad \text{(A.EXPR.ABS)}$$

$$\frac{\Delta; \Gamma \vdash_{\mu} e : \tau'; \Delta' \qquad \Gamma \vdash \tau' <: \tau \qquad \Delta'' \subseteq \Delta'}{\Delta; \Gamma \vdash_{\mu} e : \tau; \Delta''} \quad \text{(A.EXPR.SUB)}$$

Fig. 19. Expression judgements for PROTEUS$^{\Delta}$ (part II)

| | |
|---|---|
| **let** dispatch$^{\varepsilon;\varphi_1;\varphi_2}$(s : req) : handResult = | $\varphi_2 \subseteq \varphi_5$ |
| **let** t = decode((**con**$_{\mathsf{req}}$ s).fd) **in** | $\mathsf{req} \in \varphi_1$ |
| **let** u1 = **update**$^{\varphi_3}$ **in** | $\varphi_3 \subseteq \varphi_1, \varepsilon = \mathsf{U}$ |
| **if** (**con**$_{\mathsf{fdtype}}$ t) = Socket **then** | $\mathsf{fdtype} \in \varphi_3$ |
| **let** k = getsock((**con**$_{\mathsf{req}}$ s).fd) **in** | $\mathsf{req} \in \varphi_3$ |
| **let** flags = | |
| decode_sockopargs((**con**$_{\mathsf{req}}$ s).rest, (**con**$_{\mathsf{req}}$ s).op) **in** | $\mathsf{req} \in \varphi_3, \mathsf{req} \in \varphi_3$ |
| **let** h = getsockhandler((**con**$_{\mathsf{req}}$ s).fd, (**con**$_{\mathsf{req}}$ $s$).op) **in** | $\mathsf{req} \in \varphi_3, \mathsf{req} \in \varphi_3$ |
| **let** u2 = **update**$^{\varphi_4}$ **in** | $\varphi_4 \subseteq \varphi_3, \varepsilon = \mathsf{U}$ |
| **let** res = (**con**$_{\mathsf{sockhandler}}$ h)(k, (**con**$_{\mathsf{req}}$ s).buf, flags) **in** | $\mathsf{sockhandler} \in \varphi_4, \mathsf{req} \in \varphi_4$ |
| **let** u3 = **update**$^{\varphi_5}$ **in** res | $\varphi_5 \subseteq \varphi_4, \varepsilon = \mathsf{U}$ |
| **else if** (**con**$_{\mathsf{fdtype}}$ t) = File **then** ... | $\mathsf{fdtype} \in \varphi_3$ |
| **else** (**abs**$_{\mathsf{handResult}}$ $-1$) | |

where
$\varepsilon = \mathsf{U}$
$\varphi_1 = \{\mathsf{req}, \mathsf{fdtype}, \mathsf{sockhandler}\}$
$\varphi_2 = \emptyset$
$\varphi_3 = \{\mathsf{req}, \mathsf{fdtype}, \mathsf{sockhandler}\}$
$\varphi_4 = \{\mathsf{req}, \mathsf{sockhandler}\}$
$\varphi_5 = \emptyset$

Fig. 20. dispatch (from Figure 6) in PROTEUS$^{\Delta}$, with capability and updateability annotations

*Typing* **update** *and* **con**$_{\mathsf{t}}$ *e.* The capability $\Delta'$ on **update**$^{\Delta'}$ lists those types that *must not change* due to a dynamic update. Since any other type could change, the (A.EXPR.UPDATE) rule assumes that the capability can be at most $\Delta'$ following the update. The (A.EXPR.CON) rule states that to concretely access a value of type t, the type t must be defined in $\Gamma$, restricted to types listed in capability $\Delta'$.

Figure 20 shows the dispatch function from Figure 6 with capability and updateability annotations added. In the figure we put variables for these annotations with their solutions and constraints on their solutions to the right; we explain this more in Section 5.4. We can see that the **update** in **let** u1 = **update in** ... is annotated with a capability {fdtype, req, sockhandler}, since these types are used by **con** expressions that could be evaluated following that point within dispatch. By the same reasoning, the annotation on the u2 **update** is {req, sockhandler}, and the u3 **update** annotation can be empty. The (A.EXPR.UPDATE) rule requires updateability U; updates cannot be performed in a non-updateable (N) context.

(A.EXPR.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.EXPR.IFUPDATE) checks a special case of **if** with a guard $\mathbf{update}^{\Delta'} = 0$, which will be true only if an update successfully takes effect at run time. Therefore, the input capability of $e_1$ is $\Delta'$, while the input capability of $e_2$ is $\Delta$, unchanged.

*Function calls.* Function types have an annotation $\mu; \Delta'$, where $\Delta'$ is the output capability of the function. If calling a function could result in an update, the updateability $\mu$ must be $\mathsf{U}$. Thus, the dispatch function in Figure 20 has type $\mathsf{req} \xrightarrow{\mathsf{U};\emptyset} \mathsf{handResult}$.

In the (A.EXPR.UPDATE) rule, the output capability is bounded by the annotation on the **update**; in the (A.EXPR.APP) rule, the caller's output capability $\Delta'''$ 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 $\mu$ 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.EXPR.APP) is that a function f's output capability must mention those types used concretely by its callers following their calls to f. To illustrate, say we modify the type of post in Figure 3 to be int → int rather than handResult → int. As a result, loop would have to concretize the handResult returned by 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 dispatch to include handResult, which in turn would require that handResult appear in the capabilities of each of the **update** points in dispatch, preventing handResult from being updated.

Another unintuitive aspect of (A.EXPR.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 (and hence function types do not even mention the function's input capability). Instead, the type system assumes that all calls will be to a function's *most recent version*, which is guaranteed at update-time to be compatible with the program's type definitions (see §5.3). 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 unnecessary conservatism in the analysis.

*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 $\tau$ to $\tau'$ 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.

The type system permits subtyping via the (A.EXPR.SUB) rule, which also permits coarsening (making smaller) the output capability $\Delta$. Intuitively, this coarsening is always sound because it will put a stronger restriction on limits imposed by prior updates. Allowing subtyping adds flexibility to programs and to their updates. The interesting rule is (A.SUB.FUN) for function types. Output capabilities are

**Subtyping:** $\Gamma \vdash \tau_1 <: \tau_2$

$$\Gamma \vdash \mathsf{int} <: \mathsf{int} \qquad \text{(A.SUB.INT)}$$

$$\frac{\Gamma(\mathsf{t}) = \tau}{\Gamma \vdash \mathsf{t} <: \mathsf{t}} \qquad \text{(A.SUB.NT)}$$

$$\frac{\Gamma \vdash \tau_2 <: \tau_1 \qquad \Gamma \vdash \tau_1' <: \tau_2' \qquad \Delta_2 \subseteq \Delta_1 \qquad (\mu_2 = \mathsf{U}) \Rightarrow (\mu_1 = \mathsf{U})}{\Gamma \vdash \tau_1 \xrightarrow{\mu_1;\Delta_1} \tau_1' <: \tau_2 \xrightarrow{\mu_2;\Delta_2} \tau_2'} \qquad \text{(A.SUB.FUN)}$$

$$\frac{\Gamma \vdash \tau_1 <: \tau_1' \qquad i \in 1..n}{\Gamma \vdash \{\mathsf{l}_1 : \tau_1, \ldots, \mathsf{l}_n : \tau_n\} <: \{\mathsf{l}_1 : \tau_1', \ldots, \mathsf{l}_n : \tau_n'\}} \qquad \text{(A.SUB.RECORD)}$$

$$\frac{\Gamma \vdash \tau <: \tau' \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash \tau' \ \mathbf{ref} <: \tau \ \mathbf{ref}} \qquad \text{(A.SUB.REF)}$$

Fig. 21. Subtyping judgement for PROTEUS$^{\Delta}$

**Program Typing:** $\Gamma \vdash P : \tau$

$$\frac{\Gamma, \mathsf{t} = \tau' \vdash P : \tau \qquad \Gamma \vdash \tau' \text{ OK}}{\Gamma \vdash \mathbf{type}\ \mathsf{t} = \tau'\ \mathbf{in}\ P : \tau} \qquad \text{(A.PROG.TYPE)}$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma, \mathsf{z}_1 : \tau_1 \xrightarrow{\mu_1;\Delta_1'} \tau_1', \ldots, \mathsf{z}_n : \tau_1 \xrightarrow{\mu_n;\Delta_n'} \tau_n' \\ \Delta_i; \Gamma', x : \tau_i \vdash_{\mu} e_i : \tau_i; \Delta_i' \qquad i \in 1..n \qquad \Gamma' \vdash P : \tau\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{fun}\ \mathsf{z}_1^{\mu_1;\Delta_1;\Delta_1'}(x : \tau_1) : \tau_1' = e_1 \ldots \\ \qquad \mathbf{and}\ \mathsf{z}_n^{\mu_n;\Delta_n;\Delta_n'}(x : \tau_n) : \tau_n' = e_n\ \mathbf{in}\ P : \tau\end{array}} \qquad \text{(A.PROG.FUN)}$$

$$\frac{\emptyset; \Gamma \vdash_{\mathsf{N}} v : \tau'; \emptyset \qquad \Gamma, \mathsf{z} : \tau'\ \mathbf{ref} \vdash P : \tau}{\Gamma \vdash \mathbf{var}\ \mathsf{z} : \tau' = v\ \mathbf{in}\ P : \tau} \qquad \text{(A.PROG.VAR)}$$

$$\frac{\Delta; \Gamma \vdash_{\mathsf{U}} e : \tau; \Delta'}{\Gamma \vdash \mathbf{return}\ e : \tau} \qquad \text{(A.PROG.EXPR)}$$

Fig. 22. Program judgements for PROTEUS$^{\Delta}$

contravariant: if a caller expects a function's output capability to be $\Delta$, it will be a conservative approximation if the function's output capability is actually larger. A function that performs no updates can be a subtype of one that does, assuming they have compatible capabilities.

5.2.2 *Program Typing.* The rules for typing programs are given in Figure 22, defining judgement $\Gamma \vdash P : \tau$. The (A.PROG.TYPE) rule adds a new type definition to the global environment, and the (A.PROG.FUN) 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.PROG.VAR) rule checks it with an empty capability $\Delta$ and updateability $\mathsf{N}$. Finally, the (A.PROG.EXPR) rule type checks the body of the program using an arbitrary capability and updateability $\mathsf{U}$ to allow updates.

**Configuration typing:** $\vdash\ \Omega; H, e : \tau$

$$\frac{\vdash\ \Omega \qquad \Omega; \Phi\ \vdash\ H \qquad \Delta; \Omega, \Phi \vdash_{\mathsf{U}} e : \tau; \Delta'}{\vdash\ \Omega; H; e : \tau} \quad \text{(A.TYPE.CONFIG)}$$

**Heap typing:** $\Omega; \Phi\ \vdash\ H$

$$\frac{\begin{array}{l}\mathrm{dom}(\Phi) = \mathrm{dom}(H) \\ \forall \mathsf{z} \mapsto (\tau \xrightarrow{\mu;\Delta'} \tau', \lambda^{\Delta}(x).e\ ) \in H. \quad \Delta; \Omega, \Phi, x : \tau \vdash_{\mu} e : \tau'; \Delta'\ \wedge\ \Phi(\mathsf{z}) = \tau \xrightarrow{\mu;\Delta'} \tau' \\ \forall \mathsf{z} \mapsto (\tau, e) \in H. \quad \emptyset; \Omega, \Phi \vdash_{\mathsf{N}} e : \tau; \emptyset\ \wedge\ \Phi(\mathsf{z}) = \tau\ \mathbf{ref} \\ \forall r \mapsto (\cdot, e) \in H. \quad \emptyset; \Omega, \Phi \vdash_{\mathsf{N}} e : \tau; \emptyset\ \wedge\ \Phi(r) = \tau\ \mathbf{ref}\end{array}}{\Omega; \Phi\ \vdash\ H} \quad \text{(A.TYPE.HEAP)}$$

**Type Environment Well-formedness:** $\Omega\ \vdash\ \Omega$

$$\Omega\ \vdash\ \emptyset \quad \text{(A.TENV.EMPTY)} \qquad \frac{\Omega, \mathsf{t} = \tau\ \vdash\ \Omega' \qquad \Omega\ \vdash\ \tau\ \mathrm{OK}}{\Omega\ \vdash\ \mathsf{t} = \tau, \Omega'} \quad \text{(A.TENV.TYPE)}$$

Fig. 23. PROTEUS$^{\Delta}$ Configuration typing

5.2.3 *Configuration Typing.* Figure 23 shows the configuration typing rules for PROTEUS$^{\Delta}$. These have the same structure as those for PROTEUS$^{\mathbf{con}}$ (Figure 14), but they mention capabilities and updateabilities. For functions, we type check with the updateability indicated by the function's type, while for other bindings we assume $\mathsf{N}$.

## 5.3 Safety

Since we are approximating the conFree$[-]$ check using static capabilities, we can take advantage of this by refining the updateOK$(-)$ predicate for (UPDATE); this is shown in Figure 24 (contrast with the original definition in Figure 16). The two timing-related changes are highlighted by the boxes labeled (a) and (b). First, $\Delta$, taken from $\mathbf{update}^{\Delta}$, replaces $e$ as the last argument. This is used in part (a) to syntactically check that no types mentioned in $\Delta$ are changed by the update. Part (a) also refers to bindOK$[\Gamma]^{\mathrm{upd}}$ to ensure that all top-level bindings in the heap that use types in upd.UN concretely, as indicated by their input capability, are also replaced. 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 $\mathsf{t}$ is updated and has transformer $\mathsf{c}$, 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 $\Delta$ to type check, condition (b) conservatively mandates that transformers $\mathsf{c}$ must check in an empty capability, and may not perform updates ($\mathsf{c}$'s type must have updateability $\mathsf{N}$). These conditions

[3]Note that the con-free check is compatible with the analysis and therefore could simply be tried as an alternative if the static check fails.

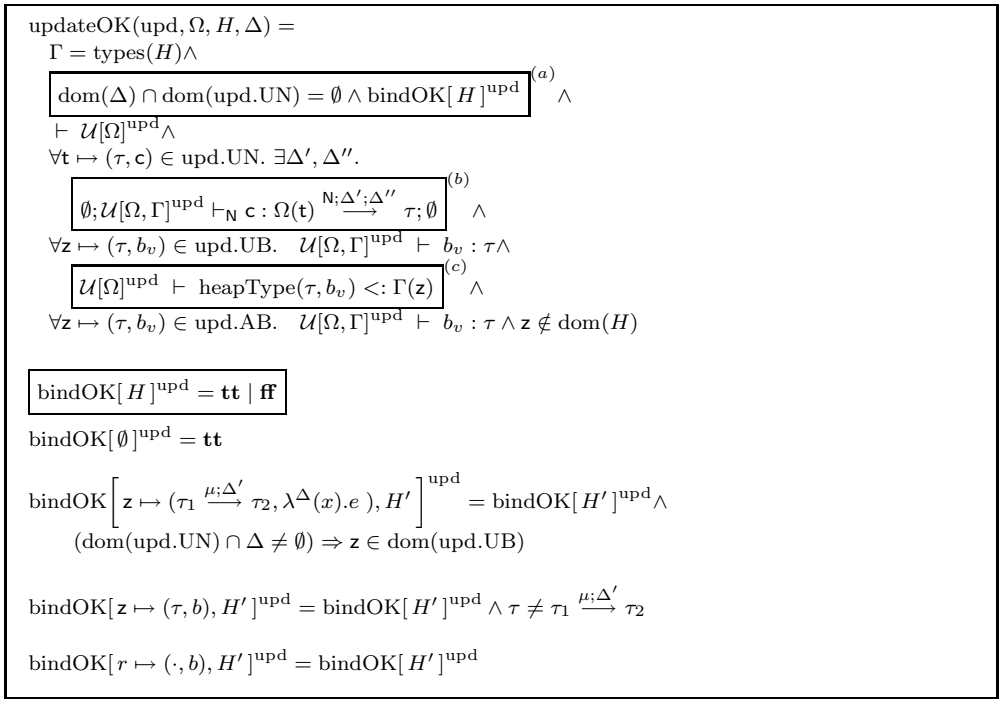


Fig. 24. Precondition for $\mathbf{update}^{\Delta}$ operational rule

are sufficient to ensure type correctness. Otherwise, a transformer function $\mathsf{c}$ is like any other function. For example, if it uses some type $\mathsf{t}$ concretely, it will have to be updated if $\mathsf{t}$ is updated. The ramifications of this fact are explored in §6.

Finally, we allow bindings to be 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 are called by) or what types they manipulate. Fortunately, we can always update an existing function with a function that causes no updates. In particular, say function $\mathsf{f}$ has type $\mathsf{t} \xrightarrow{\mathsf{U};\{\mathsf{t},\mathsf{t}'\}} \mathsf{t}'$, where $\mathsf{t} = \mathsf{int}$ and $\mathsf{t}' = \mathsf{int}$. Say we add a new type $\mathsf{t}'' = \mathsf{int}$ and want to change $\mathsf{f}$ to be the following:

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

The expected type of this function would be $\mathsf{t} \xrightarrow{\mathsf{N};\{\mathsf{t},\mathsf{t}''\}} \mathsf{t}'$, but it could just as well be given type $\mathsf{t} \xrightarrow{\mathsf{U};\{\mathsf{t},\mathsf{t}',\mathsf{t}''\}} \mathsf{t}'$, which is a subtype of the original, and thus an acceptable replacement. Replacements that contain **update** or that call functions that contain **update** are more rigid in their capabilities.

### 5.4 Inference

The type rules were designed so that type inference is straightforward, using constraints. In particular, we simply extend the definition of capability $\Delta$ to include variables $\varphi$ and updateability $\mu$ to include variables $\varepsilon$. Then we take a normal PROTEUS$^{\mathbf{con}}$ program and decorate it with fresh variables on each function def-

inition, function type, and **update** expression in the program. We also adjust the rules to use an algorithmic treatment of subtyping, eliminating the separate (A.EXPR.SUB) rule and adding subtyping preconditions to the (A.EXPR.APP) and (A.EXPR.ASSIGN) rules as is standard. This allows the judgement 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 [Heintze 1992]. Call the set of generated constraints $S$. A solution to $S$ consists of a substitution $\sigma$, which is a map from variables $\varphi$ to capabilities $\{\mathsf{t}_1, \ldots, \mathsf{t}_n\}$, and from variables $\varepsilon$ to updateabilities either $\mathsf{U}$ or $\mathsf{N}$. The constraints can be solved efficiently with standard techniques in time $O(n^3)$ in the worst case, where $n$ is the number of variables $\varphi$ 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.EXPR.SUB) |
| (2) | $\varepsilon = \mathsf{U}$ | (A.EXPR.UPDATE) |
| (3) | $(\hat{\mu} = \mathsf{U}) \Rightarrow C$ | (A.EXPR.APP), (A.EXPR.SUB) |
| (4) | $\varphi \subseteq \Delta$ | (A.EXPR.UPDATE), (A.EXPR.APP) |
| (5) | $\mathsf{t} \in \Delta$ | (A.EXPR.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 $\varphi$, since it will permit more dynamic updates. For $\mathbf{update}^{\varphi}$, a minimal $\varphi$ imposes fewer restrictions on the types that may be updated at that point. For functions $\tau \xrightarrow{\varepsilon;\varphi'} \tau'$, the smaller $\varphi'$ imposes fewer constraints on subtypes, which in turn permits more possible function replacements.

Here is one way to solve the constraints (which closely follows our implementation). First, we simplify the subtyping constraints (1) following the subtyping rules. For example, say we have the constraint

$$\Gamma \vdash \tau_1 \xrightarrow{\varepsilon_1;\varphi_1'} \tau_1' <: \tau_2 \xrightarrow{\varepsilon_2;\varphi_2'} \tau_2'$$

We remove this constraint from $S$ and replace it with constraints, following (A.SUB.FUN):

$$\begin{array}{l} \Gamma \vdash \tau_2 <: \tau_1 \\ \Gamma \vdash \tau_1' <: \tau_2' \\ \varphi_2' \subseteq \varphi_1' \\ (\varepsilon_2 = \mathsf{U}) \Rightarrow (\varepsilon_1 = \mathsf{U}) \end{array}$$

We continue until only simple subtyping constraints remain, e.g., $\Gamma \vdash \mathsf{int} <: \mathsf{int}$, and then remove these from $S$ (such constraints will always be satisfied by a program that type checks). Next, we find the greatest solution for updateability variables $\varepsilon$ appearing in constraints (2) and (3). That is, we make as many of the variables have updateability $\mathsf{U}$ as possible to allow for greater flexibility in future updates. As we discharge the implication constraints (2), additional constraints $C$ are added to $S$. Finally, the constraints that remain in $S$ are only subset constraints concerning capabilities (i.e., forms (4) and (5)), which are easily solved.

The right side of Figure 20 shows the constraints of forms (2), (4), and (5) that are generated from the dispatch function (we elide the subtyping and function call

constraints for simplicity). Following the algorithm above, we arrive at the solution at the bottom of the figure.

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* **update** *points.* Using the inference system, we can take a program that is devoid of **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 $\varphi$ 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 $\varphi$ on update points are unaffected by those on other update points; rather they are only impacted by occurrences of **con** in their continuations.

### 5.5 Properties

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

**Lemma 5.1** (Preservation)**.** *If* $\vdash \Omega; H; e : \tau$ *then*

*(i) if* $\Omega; H; e \to \Omega; H'; e'$ *then* $\vdash \Omega; H'; e' : \tau$

*(ii) if* $\Omega; H; e \xrightarrow{\text{upd}} \Omega'; H'; e'$ *then* $\vdash \Omega'; H'; e' : \tau$

The proof of part (1) of Preservation 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 5.2** (Update Program Safety)**.** *If* $\vdash \quad \Omega; H; \mathbb{E}[\mathbf{update}^{\Delta}] : \tau$ *and* $\text{updateOK}(\text{upd}, \Omega, H, \Delta)$ *then* $\vdash \mathcal{U}[\Omega]^{\text{upd}}; \mathcal{U}[H]^{upd}; \mathcal{U}[\mathbb{E}[0]]^{\text{upd}} : \tau$.

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 5.3** (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 $\Delta''$. In turn, this implies that the existing program could only use the types listed in $\Delta''$ concretely, and therefore it should be safe to update the other types in the program.

Another important element of the Update Program 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. Since type transformer functions

may not perform updates and are required to have an empty capability, this lemma allows them to be inserted at arbitrary program points:

**Lemma 5.4** (Capability strengthening)**.** *If* $\Delta; \Gamma \vdash_{\mathsf{N}} e : \tau; \Delta'$ *then for all* $\Delta''$ *we have* $\Delta \cup \Delta''; \Gamma \vdash_{\mathsf{N}} e : \tau; \Delta' \cup \Delta''$*.*

Proofs of these properties can be found in Appendix A.1.

## 6. EXTENSIONS

This section presents how PROTEUS$^{\Delta}$ can be extended with three features to improve its expressiveness: (1) permitting replacement bindings at changed types; (2) supporting binding deletion; and (3) extending PROTEUS to support multiple threads.

### 6.1 Replacing Bindings at New Types

Both PROTEUS$^{\mathbf{con}}$ and PROTEUS$^{\Delta}$ only permit replacement bindings having types compatible with the original; e.g., condition (c) of updateOK(−) in Figure 24 at best allows replacements to be subtypes of the original. However, as described in §2, it is not uncommon for bindings to change type; e.g., for a later version of a function to have an additional parameter. Given the type compatibility restriction, a simple way to effect type changes to functions is to add the new version at a new name, and then define a *stub* function to replace the old version [Frieder and Segal 1991; Hicks 2001]. The stub function has the same type as the original, but calls the new version at the new type, e.g., by generating default values for added parameters. While expedient, the stub approach has two drawbacks. First, a transformation from the old type to the new may be inefficient or impossible; transformations will occur with each call to the function, but only when the old call contains sufficient information to generate the new one. Second, there is no way to generate a stub for a non-function that has changed type.

To solve these problems, we can extend our notion of con-freeness to consider concrete uses of top-level term bindings, and not just concrete uses of named types. Top-level bindings are either global variables having type $\tau$ **ref**, or functions having type $\tau \to \tau'$, and their concrete uses are dereference and assignment for the former, and application for the latter. For example, if at the time of update a call to function $\mathsf{f}$ occurs in either the active expression $e$ or within a function not being replaced, then the program would not be considered con-free with respect to $\mathsf{f}$. This means that $\mathsf{f}$ should only be updated at a compatible type. On the other hand, if $\mathsf{f}$ is not called by the active expression (or in any function not being replaced), then it will be safe to update $\mathsf{f}$ at a new type as part of a well-formed update.

It is straightforward to extend PROTEUS$^{\Delta}$ to capture this extended notion of con-freeness statically. In particular, we augment types $\tau$ **ref** and $\tau \xrightarrow{\mu;\Delta} \tau'$ to include a set $L$ of identifiers $\mathsf{z}$; this set contains those top-level variables that may be given this type. In addition, we allow capabilities $\Delta$ to contain both named types and top-level identifiers. For simplicity, we will just consider functions in the following discussion, having augmented types $\tau \xrightarrow{\mu;\Delta;L} \tau'$. The constructor for such a type is a function definition:

$$\frac{\begin{array}{l}\Gamma' = \Gamma, \mathsf{z}_1 : \tau_1 \xrightarrow{\mu_1;\Delta_1';\{\mathsf{z}_1\}} \tau_1', \ldots, \mathsf{z}_n : \tau_n \xrightarrow{\mu_n;\Delta_n';\{\mathsf{z}_n\}} \tau_n' \\ \Delta_i; \Gamma', x : \tau_i \vdash_\mu e_i : \tau_i; \Delta_i' \qquad i \in 1..n \qquad \Gamma' \vdash P : \tau\end{array}}{\begin{array}{rl}\Gamma \vdash & \textbf{fun } \mathsf{z}_1^{\mu_1;\Delta_1;\Delta_1'}(x : \tau_1) : \tau_1' = e_1 \ldots \\ & \textbf{and } \mathsf{z}_n^{\mu_n;\Delta_n;\Delta_n'}(x : \tau_n) : \tau_n' = e_n \textbf{ in } P : \tau\end{array}} \text{ (A.PROG.FUN')}$$

This rule differs from the one in Figure 22 only in the structure of the arrow types in $\Gamma'$; the arrow type of each function $\mathsf{z}_i$ contains $\mathsf{z}_i$ in its $L$ set. The destructor rule is application:

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash_\mu e_1 : \tau_1 \xrightarrow{\hat{\mu};\hat{\Delta};L} \tau_2; \Delta' \\ \Delta'; \Gamma \vdash_\mu e_2 : \tau_1; \Delta'' \qquad \Delta''' \subseteq \Delta'' \qquad L \subseteq \Delta'' \\ (\hat{\mu} = \mathsf{U}) \Rightarrow (\mu = \mathsf{U} \wedge \Delta''' \subseteq \hat{\Delta})\end{array}}{\Delta; \Gamma \vdash_\mu e_1\ e_2 : \tau_2; \Delta'''} \text{ (A.EXPR.APPU')}$$

This rule differs from the one in Figure 18 in that the set $L$ must be included in the output capability $\Delta''$, indicating that any update prior to this call must not update those functions appearing in $L$. Finally, subtyping permits the set $L$ on an arrow to be smaller in the subtype:

$$\frac{\begin{array}{l}\Gamma \vdash \tau_2 <: \tau_1 \qquad \Gamma \vdash \tau_1' <: \tau_2' \\ \Delta_2 \subseteq \Delta_1 \qquad L_1 \subseteq L_2 \qquad (\mu_2 = \mathsf{U}) \Rightarrow (\mu_1 = \mathsf{U})\end{array}}{\Gamma \vdash \tau_1 \xrightarrow{\mu_1;\Delta_1;L_1} \tau_1' <: \tau_2 \xrightarrow{\mu_2;\Delta_2;L_2} \tau_2'} \text{ (A.SUB.FUN')}$$

As an example, consider the typing of the call to the socket handler as part of the dispatch function shown in Figure 6:

$$\begin{array}{l}\textbf{let } \mathrm{u2} : \mathsf{int} = \textbf{update}^{\varphi} \textbf{ in} \\ \textbf{let } \mathrm{res} : \mathsf{handResult} = (\textbf{con}_{\mathsf{sockhandler}}\ \mathrm{h})(\mathrm{k}, (\textbf{con}_{\mathsf{req}}\ \mathrm{s}).\mathrm{buf}, \mathrm{flags}) \textbf{ in } e\end{array}$$

We might type this fragment in the following context:

$$\begin{array}{rl}\Gamma = & \ldots, \\ & \mathsf{sockhandler} = \{\mathrm{sock} : \mathsf{sock}, \mathrm{buf} : \mathsf{buf}, \mathrm{sflags} : \mathsf{sflags}\} \xrightarrow{\mathsf{U};\emptyset;\{\mathsf{upd\_read},\mathsf{udp\_write},\ldots\}} \mathsf{handResult}, \\ & \mathsf{udp\_read} : \{\mathrm{sock} : \mathsf{sock}, \mathrm{buf} : \mathsf{buf}, \mathrm{sflags} : \mathsf{sflags}\} \xrightarrow{\mathsf{U};\emptyset;\{\mathsf{upd\_read}\}} \mathsf{handResult}, \\ & \mathsf{udp\_write} : \{\mathrm{sock} : \mathsf{sock}, \mathrm{buf} : \mathsf{buf}, \mathrm{sflags} : \mathsf{sflags}\} \xrightarrow{\mathsf{U};\emptyset;\{\mathsf{upd\_write}\}} \mathsf{handResult}, \\ & \mathrm{k} : \mathsf{sock}, \\ & \mathrm{flags} : \mathsf{sflags}, \\ & \mathrm{h} : \mathsf{sockhandler}, \ldots\end{array}$$

In such a context, the subexpression $\textbf{con}_{\mathsf{sockhandler}}\ h$ will have type $\{\mathrm{sock} : \mathsf{sock}, \mathrm{buf} : \mathsf{buf}, \mathrm{sflags} : \mathsf{sflags}\} \xrightarrow{\mathsf{U};\emptyset;\{\mathsf{upd\_read},\mathsf{udp\_write},\ldots\}} \mathsf{handResult}$ by the (A.EXPR.CON) rule so that applying (A.EXPR.APP') above requires that $\{\mathsf{upd\_read}, \mathsf{udp\_write}, \ldots\}$ be included in the current capability. This constraint will propagate backward so that applying (A.EXPR.UPDATE) requires $\{\mathsf{upd\_read}, \mathsf{udp\_write}, \ldots\} \subseteq \varphi$. As $\varphi$ is the annotation on the **update**, updates to upd_read and udp_write at that point will be prevented if they have different types than the originals.

Note that one way to effectively allow variables to change type without this extension would be to give each a distinct named type. Since named type definitions can change, this would effectively allow the types of variables to change. However, this approach does not quite work because one cannot directly ascribe to a function a named type. The best one can do is define a variable as having a named type,

and assign to it a function having that named type's definition. Of course, a compiler could automatically perform this transformation and generate the attendant transformer functions when variables are updated. The drawback in doing so is that it introduces extra implementation complexity, and extra run-time overhead, since each function call or variable would require a **con** coercion. Therefore, in our implementation we take the approach presented here.

### 6.2 Binding Deletion

While most changes we have observed in source programs are due to added or replaced definitions, occasionally definitions are deleted. Assuming that DSU is meant to prolong a program's running time for perhaps a few years, we could choose to ignore deleted functions, as in many cases the dead code will not take up much memory. However, an excessive number of 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.

To support deletion, we augment updates upd to contain a component DB—a list of top-level identifiers to delete. To preserve type safety, we can only apply such an update if the deleted bindings will never be used again by the updated program. Ensuring this condition is surprisingly similar to permitting a type change. In particular, the deletion of an identifier `z` is only permitted if the program is con-free with respect to `z`, which is implemented just as described above.

We have to be a bit careful how we implement this check. In particular, while the program could well be con-free with respect to some function `f` to delete, it might be executing `f` at the time of the update. Since the operational semantics models application by substitution, this is a non-issue, since the active expression is separate from the definition of `f` in the heap. This would not be the case in an actual implementation, however, so we would need to delay the deletion of `f` until it is no longer executing (which could be established, for example, by a kind of reference counting). Because the program was (and is) con-free with respect to `f`, it will never be called after this point, so the code can be safely deleted.

### 6.3 Threads

Many non-stop systems are written in a multi-threaded programming style, so we would like our approach to work for these systems as well. We define the syntax **fork** $e$ to mean that $e$ should be executed in a separate thread. We adjust the operational semantics so that configurations consist of a tuple of thread expressions $e_1, \ldots, e_n$, a heap $H$, and a type environment $\Omega$. The definition of (dynamic) con-freeness in this setting is the same as in the single-threaded case (Figure 16) but considers all expressions $e_i$. (Note that we would have to pause all threads at the time of the update to perform this check dynamically.)

Unfortunately, it is easy to see that our updateability analysis does not soundly approximate con-freeness in the multi-threaded case. In particular, capabilities mention the definitions with respect to which the *current* thread is not con-free, but say nothing about the restrictions due to the concurrent activities other threads.

A simple way to address this problem is to treat each $\mathbf{update}^{\Delta}$ as a synchronization barrier. When each thread $t$ has synchronized at some $\mathbf{update}^{\Delta_t}$, the update may proceed as long as the update has not changed the type of any definition in

$\bigcup_{\text{all } t} \Delta_t$. This ensures that the update is con-free with respect to all threads, rather than just one.

Implementing this semantics could be done as follows. First, we can give **fork** the following type rule:

$$\frac{\Delta; \Gamma \vdash e : \tau; \Delta'}{\Delta; \Gamma \vdash \textbf{fork}\ e : \mathsf{int}; \Delta}$$

Notice that the input capability of the child matches that of the parent, to preserve any restrictions imposed by past updates, while the output capability of the parent is indifferent to that of the child. This is because the child will perform updates asynchronously, so statically limiting the parent's capability is unnecessary.

Second, we modify the dynamic semantics of $\textbf{update}^{\Delta}$ to barrier synchronize. That is, for each thread that reaches $\textbf{update}^{\Delta}$, if updateOK$(\ldots, \Delta)$ holds then the thread pauses until all other threads have similarly paused. Then the update takes place and execution resumes on the modified program. As a degenerate case, if no types are updated (and no variables have changed type), there is no need to synchronize.

We believe this is a promising starting point for supporting multi-threaded programs, but there are still other problems to solve. The main problem is that using blocking synchronization to ensure safety may compromise the liveness of the system. In particular, depending on the program, it may take a while for each thread to reach a suitable update point, so some threads will not be doing useful work while they wait. At the extreme, pausing a thread at an **update** could induce deadlock if the thread holds mutual exclusion locks that prevent other threads from reaching suitable **update** points.

Nevertheless, we believe our current framework puts us in a good position to solve these problems, for two reasons. First, by using **update** to make explicit when a dynamic update might occur, we can statically reason about its potential to induce deadlock by adapting existing techniques [Foster et al. 2002; Xie and Aiken 2005; Ball and Rajamani 2002]. Second, our notion of con-freeness permits identifying safe update points at fairly fine grain, creating more opportunities for updating and thus reducing potential waiting time. We plan to consider these issues carefully in future work.

## 7. PRELIMINARY IMPLEMENTATION

Following the formal development of PROTEUS presented here, we have been implementing a compiler and run-time system for dynamically updating C programs. Though a full discussion of the implementation and its evaluation is beyond the scope of this paper, this section nonetheless presents an overview of our approach, and briefly describes our experience dynamically updating three of the open source programs mentioned in §2: the "Very Secure" FTP daemon, `vsftpd`, OpenSSH's `sshd` daemon and the `zebra` routing daemon from the GNU Zebra routing package. Our intent is to provide some evidence that PROTEUS does indeed represent the necessary core of a system by which real-world programs can be safely and flexibly updated on the fly. Full details of our implementation and experience, including performance measurements, are presented elsewhere [Neamtiu et al. 2006].

An overview of the update process is presented in Figure 25. Given a C program $\texttt{program}_1{}^{\text{src}}$, our *update compiler* (UC) transforms it into an updateable C program, $\texttt{program}_1{}^{\text{src}}$, which is then compiled into a dynamically updateable executable, $\texttt{program}_1{}^{\text{bin}}$. When a new version, $\texttt{program}_2{}^{\text{src}}$, becomes available, our

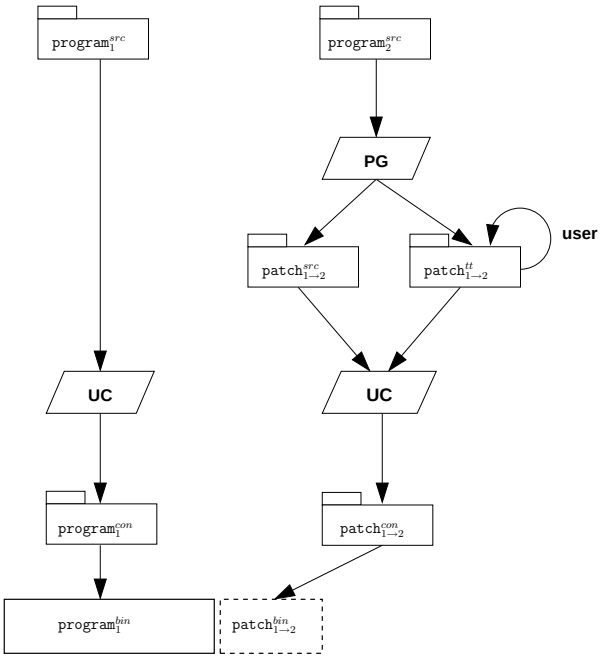


Fig. 25. Dynamic Update Methodology

*patch generator* (PG) generates a dynamic patch $\texttt{patch}_{1\to2}{}^{\text{src}}$, which is then passed to UC to generate an updateable patch $\texttt{patch}_{1\to2}{}^{\mathbf{con}}$, and finally compiled into a shared library $\texttt{patch}_{1\to2}{}^{\text{bin}}$. $\texttt{patch}_{1\to2}{}^{\text{bin}}$ can now be linked into $\texttt{program}_1{}^{\text{bin}}$'s address space so that run-time system can fix up the running program to start using the new and replacement code and data. Both the update compiler and the patch generator use CIL (*C Intermediate Language*) [Necula et al. 2002] for C code parsing and source-to-source transformation.

The update compiler makes a program updateable via a source-to-source translation of the program. Following prior work (e.g., Hicks [2001], Orso et al. [2002], Soules et al. [2003], and others), functions are made updateable by introducing a level of indirection at each direct call-site. At update-time, the target of each indirect call is modified to point to the new version of the function, thus ensuring that each call to the function is to the newest version, while existing versions may continue to run until they exit. Function pointers are treated a bit differently to ensure this invariant. Whenever the program uses a function $f$'s address as data, the compiler introduces a wrapper function, $f_{wrap}$ to be used instead. This wrapper simply contains an (indirected) call to $f$.

To implement updates to values of a named type $\texttt{t}$, uses of such values are compiled to use the explicit type coercions $\mathbf{con}_\texttt{t}$ and $\mathbf{abs}_\texttt{t}$, following the rules presented in §4.2. Values are compiled specially to be updateable (essentially using a level of indirection), and coercions are merely small functions that convert between this representation and the concrete representation expected by the program. Whenever a $\mathbf{con}_\texttt{t}$ is called, the necessary sequence of type transformers is invoked if the value is not up-to-date. Thus data transformation for named types is *lazy*, happening during program execution, rather than *eager*, happening at update-time. Duggan [Duggan 2001] also proposes lazy dynamic updates to types using type transformers, but our

work is the first work that explores the implementation of such primitives.

Once type coercions have been inserted, the update compiler performs the updateability analysis outlined in §5.4, including the extension to functions noted in §6.1. During this analysis the compiler can insert calls to **update**, which is merely a call into the runtime system, but so far we have just inserted one or two calls to **update** manually at *quiescent* program points, following Hicks [2001]. The analysis then infers those types that are not con-free at these **update** points, and the list is passed to the call to **update** to check well-timedness of an available update at run-time.

While this might be the end of the story for a type-safe, high-level language, C's weak type system and low level of abstraction create additional challenges. To ensure that compiling for updateability does not break otherwise correct programs, we must account for the `&` operator, unsafe casts, use of `void*` for polymorphism, physical subtyping, and other "low-level" language features. In some cases, we augmented the analysis to be more precise; for example we model some safe uses of `void*` and most uses of `&` as part of the updateability analysis. In other cases, low-level features render types or functions *non-updateable*; for example casting an integer to a pointer to a named type `t` would cause the analysis to deem `t` forever non-updateable.

A dynamic patch $\texttt{patch}_{1\to 2}{}^{\text{bin}}$ implements a dynamic update (§3.2) from $\texttt{program}_1{}^{\text{src}}$ to $\texttt{program}_2{}^{\text{src}}$. Starting from the two sources $\texttt{program}_1{}^{\text{src}}$ and $\texttt{program}_2{}^{\text{src}}$, the *patch generator* (PG) generates a C program $\texttt{patch}_{1\to 2}{}^{\text{src}}$ with the new and updated bindings and types (the UN, UB, AN and AB maps from §3.2). AN and AB are straightforward to generate, and they are written directly to $\texttt{patch}_{1\to 2}{}^{\text{src}}$. Constructing UN and UB is a bit more complicated, because writing converting functions for types (*type transformers*) and global variables (*state transformers*) cannot be fully automated. Instead we partially automate this process through a type-directed comparison of the old and new types, following Hicks [2001]. Since these state and type transformers might require user intervention, they are written into a file $\texttt{patch}_{1\to 2}{}^{\text{tt}}$ that the user can inspect and modify before being passed to later stages. $\texttt{patch}_{1\to 2}{}^{\text{src}}$ and $\texttt{patch}_{1\to 2}{}^{\text{tt}}$ are transformed by UC into an updateable program $\texttt{patch}_{1\to 2}{}^{\mathbf{con}}$ and finally compiled into a binary $\texttt{patch}_{1\to 2}{}^{\text{bin}}$ that will be linked into the running system $\texttt{program}_1{}^{\text{bin}}$.

We updated `vsftpd` from version 1.1.0 to 2.0.3, for a total of 13 releases, the OpenSSH daemon, from version 3.5 to 4.2, for a total of 11 releases, and `zebra` from version 0.92a to 0.95a, for a total of 6 releases. This represents the last three years in the lifespan of Vsftpd and OpenSSH, and the last four years of Zebra. Vsftpd grew from roughly 10K to 17K LOC, OpenSSH grew from 47K to 58K LOC, and Zebra grew from 41K to 45K LOC. We started with the oldest version of each program and generated a dynamic patch for each new release. With these patches, we can start running the oldest version of the program, and then successively patch it up to the most recent version. While testing the patches, we performed updates on the programs while they were actively serving requests. We also ran the relevant regression tests on the updated versions, to make sure they supported all of the added functionality.

The updateability analysis was helpful for ensuring that updates were performed at an appropriate time. For all the three programs, the quiescent **update** points that we picked manually were ratified by the analysis as having an empty capability, hence being con-free. The analysis also was very helpful for ensuring con-freeness

with respect to accessing global variables and calling functions, whose types did change over the lifetime of the two programs (see §2 for more detail). While in principle the static updateability analysis is more conservative than the dynamic con-free check, this was not a problem in practice; all of the bindings deemed not con-free by the analysis would have been deemed so by the dynamic check as well. The analysis itself was fairly efficient. On average, the entire compilation process was on the order of 10 seconds per patch for `vsftpd`, 70 seconds for OpenSSH daemon and 50 seconds for `zebra`. We intend to update some additional programs and more carefully evaluate our implementation as future work.

## 8. RELATED WORK

The main challenge in developing a dynamic software updating system is balancing the tension between safety and flexibility. Rather than survey the entirety of related work in this area (a reasonably thorough survey can be found in Hicks [2001]; updated in Hicks and Nettles [2005]), here we consider how systems that aim to provide type safety balance this goal with flexibility.

The simplest approach to ensuring type safety is to only permit replacement bindings at the same (or compatible) type as the original. This idea can be implemented directly by the compiler and run-time system, as in Orso et al. [2002], Drossopoulou and Eisenbach [2003], Hjálmtýsson and Gray [1998], and the K42 operating system [Soules et al. 2003; Baumann et al. 2005], among others; or it can be programmed directly using dynamic linking, as proposed by Appel [1994] and Peterson et al. [1997]. In earlier work, we formalized this approach using *dynamic binding*, considering different evaluation strategies [Bierman et al. 2003], and later added versioned identifiers for more precision [Bierman et al. 2003]. In essence, this definition of well-formedness trivially ensures that all updates are well-timed. However, as we showed in Section 2, over time software systems do tend to change their type structure, and so restricting updates to this form prevents natural, online evolution. It is nonetheless useful in some situations, such as simple security patches [Altekar et al. 2005].

Dynamic ML [Gilmore et al. 1997] supports updating modules defining *abstract* types `t`. The internal representation type of such a module is permitted to change, while the external interface must remain the same (or be a subtype). 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*). This approach thus extends the natural reusability benefits (due to representation independence [Mitchell 1986]) of abstract types to a dynamic setting. Our use of $\mathbf{abs}_t$ and $\mathbf{con}_t$ coercions generalizes this idea to non-abstract named types. This allows our $\mathrm{conFree}[-]^-$ check to be more precise, and thus more permissive in allowing safe updates, than Dynamic ML's "activeness" check. As examples, we could discover points *within* an abstract module at which it could be safely updated, and we can permit the module within which an infinite loop is executing to be updated; we have found the latter case particularly important in practice. Dynamic ML has no static notion of proper update timing, as we do with our updateability analysis. Moreover, as far as we can tell, there has never been formal proof that Dynamic ML's activeness check is sufficient to ensure type safety (though this seems plausible).

Dynamic ML's approach is a functional language analogue of object-oriented approaches to DSU, such as Hjálmtýsson and Gray [1998], and the K42 operating

system [Soules et al. 2003; Baumann et al. 2005], which require interface compatibility between versions of objects, while allowing internal representations to change. However, these systems do not provide the same strong encapsulation guarantees as abstract types in ML, since programmers may leak the internal state of an object. If the program contains pointers into the internal representation of objects at the time those objects are updated, the pointers may no longer be type correct. While the K42 designers cite the need for encapsulation, they provide no way to enforce it.

Duggan [2001] supports changing the representation of named types, which use constructs *fold* and *unfold* to create and destruct values of named type, similar to our $\mathbf{abs}_{\mathtt{t}}$ and $\mathbf{con}_{\mathtt{t}}$. However, rather than require representation consistency, in which programs have at most one definition of a type at a time, Duggan proposes allowing multiple versions of types to coexist. As such, 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. One 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). Finally, Duggan does not permit functions and global variables that are not defined as named types to be replaced with bindings having different types.

Hicks' earlier DSU approach [2001] permits the types of arbitrary bindings to change, but also at the cost of representation consistency. Type safety is preserved by copying data when transforming it to the new representation; in effect, for any bindings whose type has changed, old code uses the old value, while new code uses the new one. For functions, *stubs* (§3.2) can be used to direct calls having the old type to those having the new type, but there is no such facility for data. For named types, a representation change causes a new name to be generated by the compiler to be seen by the run-time system, and it is up to the programmer to manually copy and transform the data at update-time. This limits transformation to data that is reachable from global variables; stack-allocated data cannot be changed in general. In constrast, Proteus is representation-consistent, so there is no possibility of having two parts of the program operating on two copies of the same logical data, or calling two versions of the same function. Moreover, Proteus can update named types wherever they might be allocated, and is not limited to data reachable from global variables.

Boyapati et al. [2003] and the K42 operating system [Soules et al. 2003; Baumann et al. 2005] ensure well-timed updates to objects in a multi-threaded setting. Both systems rely on object encapsulation to guarantee that no active code depends on an object's representation when the object is updated (Boyapati et al. guarantee proper encapsulation exists while K42 assumes it). 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. Compared to K42, our approach to safety in multi-threaded programs is *preemptive*, rather than *reactive*: when an update is pending, **update** points act as a barrier, while in K42 threads are allowed to proceed unless they might interfere with the update. However, their approach may have difficulty scaling to multi-object updates or those in which types have changed, since there may be no easy way to recover if an interfering action is detected. We plan to explore multi-threaded, dynamically updateable programs more carefully in future work.

Our approach focuses on updating single processes. To support updating processes in a distributed system would require coordination when the communication protocol is changed in a non-backward-compatible manner. The most recent work on this problem is that of Ajmani et al [Ajmani 2004; Ajmani et al. 2006]. They model distributed systems as processes communicating via remote procedure calls; in this light, a change in protocol amounts to a change in a remote function's type or usage pattern. They define conditions under which older versions of a system might simulate newer versions, and vice versa, to relax synchronization constraints. This idea might apply to our proposed idea of coordinating updates to multi-threaded programs.

While our con-freeness property and its relation to type-safe dynamic updating is new, others have considered other properties of dynamic updates. Gupta [1994] developed a high-level formal framework in which he proved that the problem of update *validity* is, in general, undecidable. He defines validity to mean that following a valid update, the existing program will eventually transition to the legal states of the new program. Bloom [1983; Bloom and Day [1993] explores the limitations of dynamic updates in a somewhat information-theoretic sense. That is, some dynamic updates are not possible simply because the existing program state does not contain the information to construct data structures needed by the new code.

The general formulation of our updateability analysis using capabilities is similar to other capability type systems [Walker et al. 2000; Walker 2000; Grossman et al. 2002]. For example, capabilities in the Calculus of Capabilities [Walker et al. 2000] 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.

## 9. CONCLUSIONS

In this paper we have presented PROTEUS, a simple calculus for modeling type-safe dynamic updates in imperative 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 continue our implementation of 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 operating systems. 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.

We have also starting applying our updateability analysis to the related problem of ensuring that a dynamic update of a security policy does not impact the security properties of a running program that uses it [Hicks et al. 2005]. We are working to scale up this basic insight to provide soundness guarantees for distributed secure systems for which policies change over time.

*Acknowledgements.* We thank Nikhil Swamy, Manuel Oriol, Mike Furr, and the anonymous referees for helpful comments on drafts of this paper. This work was supported by a Royal Society University Research Fellowship (Sewell), a Marconi EPSRC CASE Studentship (Stoyle), EC FET-GC project IST-2001-33234 PEPITO, NSF Contract #0346989, and EPSRC grant GR/T11715.

## A. PROOFS

### A.1 Proof of Type Soundness for Proteus$^{\Delta}$

In this section we give a proof of type soundness for Proteus$^{\Delta}$. The proof is based on standard techniques for proving syntactic type soundness. Proving soundness for the system guarantees that our inferred update points are safe and consequently any update judged suitable by updateOK() at runtime will not invalidate the type of the program.

Before we begin the proof we discuss the rôle of update capability weakening (Lemma A.13), a key lemma. Update capability weakening shows that the $\Delta$ annotation on updates is faithful to our intended meaning, that is, given an update point $\mathbf{update}^{\hat{\Delta}}$ in redex position in some larger expression, the only types used concretely following the update are contained in $\hat{\Delta}$. More formally, if

$$\Delta; \Gamma \vdash \mathbb{E}[\mathbf{update}^{\hat{\Delta}}] : \tau; \Delta'$$

then

$$\hat{\Delta}; \Gamma \vdash \mathbb{E}[\mathbf{update}^{\hat{\Delta}}] : \tau; \Delta'$$

Most of the work to establish this fact is done in proving a generalised $\mathbb{E}$-inversion lemma (Lemma A.12), from which Update Capability Weakening follows easily.

**Definition A.1** (Typing contexts)**.** A context $\Gamma$ is a finite partial function with the following entries:

$$\begin{array}{rl} \mathsf{z} : \tau & \text{types of external identifiers} \\ \mathsf{z} : \tau\ \mathbf{ref} & \text{types of references} \\ x : \tau & \text{types of local identifers} \\ \mathsf{t} = \tau & \text{named type definitions} \end{array}$$

We write $\Phi$ for typing contexts containing only external identifiers and references, and $\Omega$ for those containing only named type definitions.

We first note some standard Inversion, Canonical Forms and Weakening lemmas.

**Lemma A.2** (Inversion – expressions)**.**

(1) *If* $\Delta_0;\Gamma \vdash_\mu \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\};\Delta_n$ *then* $\exists \tau'_1, \ldots, \tau'_n, \Delta'_1, \ldots, \Delta'_{n-1}, \Delta'_n \supseteq \Delta_n$ *such that* $\forall i \in 1..n$ *we have* $\Gamma \vdash \tau'_i <: \tau_i$ *and* $\Delta'_{i-1};\Gamma \vdash_\mu e_i : \tau'_i;\Delta'_i$

(2) *If* $\Delta_0;\Gamma \vdash_\mu e_1\ e_2 : \tau_2;\Delta$ *then* $\exists \tau'_2, \Delta_1, \Delta_2, \hat{\Delta}', \hat{\mu}$ *such that* $\Delta_0;\Gamma \vdash_\mu e_1 : \tau_1 \xrightarrow{\hat{\mu};\hat{\Delta}} \tau_2;\Delta_1$ *and* $\Delta_1;\Gamma \vdash_\mu e_2 : \tau_1;\Delta_2$ *and* $\Delta' \subseteq \Delta_2 \wedge (\hat{\mu} = \mathsf{U} \Rightarrow (\mu = \mathsf{U} \wedge \Delta' \subseteq \hat{\Delta}'))$

□

**Lemma A.3** (Inversion – subtyping)**.**

(1) *If* $\tau <: \tau_1 \xrightarrow{\mu_1;\Delta_1} \tau'_1$ *then* $\exists \mu_2, \Delta_2, \tau_2, \tau'_2$ *such that* $\tau = \tau_2 \xrightarrow{\mu_2;\Delta_2} \tau'_2$

(2) *If* $\tau <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ *then* $\exists \tau'_1, \ldots, \tau'_n$ *such that* $\tau = \{l_1 : \tau'_1, \ldots, l_n : \tau'_n\}$

□

**Lemma A.4** (Canonical Forms)**.**

(1) *If* $v$ *is a value of type* $\mathsf{int}$ *then for some* $n \in \mathbb{N}$, $v = n$

(2) *If* $v$ *is a value of type* $\mathsf{t}$ *then for some value* $v'$, $v = \mathbf{abs}_{\mathsf{t}}$

(3) *If* $v$ *is a value of type* $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ *then* $\exists v_1, \ldots, v_n$ *such that* $v = \{l_1 = v_1, ..., l_n = v_n\}$

(4) *If* $v$ *is a value of type* $\mathsf{t}$ **ref** *then* $v = \rho$, *where* $\rho$ *ranges over references and external identifiers*

(5) *If* $v$ *is a value of type* $\tau_1 \xrightarrow{\mu;\Delta} \tau_2$ *then* $\exists \mathsf{z}.\ v = \mathsf{z}$

□

**Lemma A.5** (Weakening)**.** *If* $\Delta_1;\Gamma \vdash_\mu e : \tau;\Delta_2$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Delta_1;\Gamma' \vdash_\mu e : \tau;\Delta_2$ □

We now establish some basic facts about capabilities.

—If an update is judged safe for one update point then it is also safe for any more restricted update point.

—If an expression does not perform an update, it does not consume any of its capability. By "consume" we mean that its pre-capability is equal to its post-capability.

—If a term type checks given one capability the it type checks in a larger capability.

—Values type with any pre and post-capability (as long as pre is as least as permissive as post)

**Lemma A.6** (UpdateOK Capability Weakening)**.** *If* updateOK$(\text{upd}, \Omega, H, \Delta)$ *and* $\Delta' \subseteq \Delta$ *then* updateOK$(\text{upd}, \Omega, H, \Delta')$

*Proof.* The only clause in updateOK() that depends on $\Delta$ is $\text{dom}(\Delta) \cap \text{dom}(\text{upd.UN}) = \emptyset$, and the validity is unaffected by the shrinking of $\Delta$. □

**Lemma A.7** (Capability Strengthening)**.**

*(i) If* $\Delta_1;\Gamma \vdash_{\mathsf{N}} e : \tau;\Delta_2$ *then* $\forall \Delta_3.\ \Delta_1 \cup \Delta_3;\Gamma \vdash_{\mathsf{N}} e : \tau;\Delta_2 \cup \Delta_3$

*(ii) If* $\Delta_1;\Gamma \vdash_\mu e : \tau;\Delta_2$ *then* $\forall \Delta_3.\ \Delta_1 \cup \Delta_3;\Gamma \vdash_\mu e : \tau;\Delta_2$

*Proof.* We first prove (i) by induction on the typing derivation of $e$. Note that the (A.EXPR.UPDATE) case cannot occur as the annotation on the turnstyle is $\mathsf{U}$ not $\mathsf{N}$. We give the application case:

**case** (A.EXPR.APP) **:**

Assume

$$\frac{\begin{array}{l}\Delta;\Gamma \vdash_{\mathsf{N}} e_1 : \tau_1 \xrightarrow{\hat{\mu};\hat{\Delta}} \tau_2;\Delta' \\ \Delta';\Gamma \vdash_{\mathsf{N}} e_2 : \tau_1;\Delta'' \qquad \Delta''' \subseteq \Delta'' \\ \hat{\mu} \leq \mu \qquad \hat{\mu} = \mathsf{U} \implies \Delta''' \subseteq \hat{\Delta}'\end{array}}{\Delta;\Gamma \vdash_{\mathsf{N}} e_1\ e_2 : \tau_2;\Delta'''}$$

prove

$$\frac{\begin{array}{l}\Delta \cup \Delta_3;\Gamma \vdash_{\mathsf{N}} e_1 : \tau_1 \xrightarrow{\hat{\mu};\hat{\Delta}} \tau_2;\Delta' \cup \Delta_3 \quad {}^{(a)} \\ \Delta' \cup \Delta_3;\Gamma \vdash_{\mathsf{N}} e_2 : \tau_1;\Delta'' \cup \Delta_3 \quad {}^{(b)} \qquad \Delta''' \cup \Delta_3 \subseteq \Delta'' \cup \Delta_3 \quad {}^{(c)} \\ \hat{\mu} \leq \mu \quad {}^{(d)} \qquad \hat{\mu} = \mathsf{U} \implies \Delta''' \cup \Delta_3 \subseteq \hat{\Delta}' \quad {}^{(e)}\end{array}}{\Delta \cup \Delta_3;\Gamma \vdash_{\mathsf{N}} e_1\ e_2 : \tau_2;\Delta''' \cup \Delta_3}$$

(a) and (b) hold by induction. (c) holds if $\Delta''' \subseteq \Delta''$, which holds by assumption. (d) holds by assumption. If $\hat{\mu} = \mathsf{N}$ then (e) holds trivially. It cannot be the case that $\hat{\mu} = \mathsf{U}$ because then the annotation on the turnstyle for the typing of $e_1\ e_2$ must be $\mathsf{U}$ contradicting our assumption.

**case** (A.EXPR.UPDATE) **:**

This is trivially true as type checking update requires $\mathsf{U}$ annotation on the turnstyle.

The rest of the cases are similar.

(ii) can be proved by a similar induction on the typing derivation of $e$. □

**Lemma A.8** (Value Typing)**.** *If* $\Delta_1;\Gamma \vdash_\mu v : \tau;\Delta_2$ *then* $\forall \Delta_1', \Delta_2', \mu'$ *such that* $\Delta_2' \subseteq \Delta_1'$ *it holds that* $\Delta_1';\Gamma \vdash_{\mu'} v : \tau;\Delta_2'$

*Proof.* Proceed by induction of the typing derivation of $v$. First note that none of the rules A.EXPR.VAR,REFCELL,PROJ, APPU,IF, LET, REF DEREF, ASSIGN, UPDATE, CON or IF-UPDATE can the expression be a value.

**case** (A.EXPR.INT) **:**

By (A.EXPR.INT) $\Delta_2';\Gamma \vdash_\mu n : \mathsf{int};\Delta_2'$. By Capability Strengthening Lemma $\Delta_1';\Gamma \vdash_\mu n : \mathsf{int};\Delta_2'$ as required.

**case** (A.EXPR.XVAR) **:**

Similar to (A.EXPR.INT) case.

**case** (A.EXPR.RECORD) **:**

By Lemma A.4 (Canonical Forms) each element of the record must be a value in order for the record to be a value. The result follows by induction on each of the elements of the record and use of the (A.EXPR.RECORD) rule.

**case** (A.EXPR.ABS) **:**

By Lemma A.4 (Canonical Forms) of values $\mathbf{abs}_t$ is a value only if $e$ is a value, thus suppose $e = v$ for some $v$, then the result follows by induction on the typing derivation of $v$ and use of the (A.EXPR.ABS) rule.

**case** (A.EXPR.SUB) **:**

By straight-forward application of the IH.

□

**Lemma A.9** (Substitution)**.** *If* $\Delta_1;\Gamma, x : \tau' \vdash_\mu e : \tau;\Delta_2$ *and* $\Delta_3;\Gamma \vdash_\mu v : \tau';\Delta_3$ *then* $\Delta_1;\Gamma \vdash_\mu e[v/x] : \tau;\Delta_2$

*Proof.* This property follows by induction on the typing derivation of $e$ using the Value Typing Lemma as appropriate. □

**Lemma A.10** (Derivations have Well-Formed Types)**.** *If* $\Delta_1; \Gamma, \Omega, \Phi \vdash_\mu e : \tau; \Delta_2$ *then* $\Omega \vdash \tau$

**Lemma A.11** (Typing Weakens Capability)**.** *If* $\Delta_1; \Gamma \vdash_\mu e : \tau; \Delta_2$ *then* $\Delta_2 \subseteq \Delta_1$

*Proof.* By an easy induction on typing derivations. Note that the axioms enforce equality of capabilities; UPDATE, IF and IF-UPDATE rules allow weakening of capabilities; and the remainder act inductively w.r.t. capabilities. □

The following $\mathbb{E}$-inversion lemma describes the constraints on the capabilities of an expression if it is to be placed in a given evaluation context (continuation). The lemma essentially tells us that, given an expression $\mathbb{E}[e]$ which is checkable in capability $\Delta$, we can substitute any term $e'$ for $e$, that is checkable in capability $\hat{\Delta}$, provided its post-capability is at least that of the post-capability of $e$. i.e. we have to ensure that the computation has "enough capability left" to execute the continuation.

The reader may be surprised that the pre-capability of $e'$ (which is also the pre-capability of $\mathbb{E}[e']$) is not constrained in any way. Intuitively this is justified by the fact that capabilities are *flow-sensitive*, and that the expression $\mathbb{E}[e']$ represents an expression $e'$ with *continuation* $\mathbb{E}$. Thus, the execution of, and therefore the calculation of capabilities for, $\mathbb{E}[e']$ proceeds first by considering $e'$, and then by considering $\mathbb{E}$. Provided that after $e'$ is considered, there is enough capability left over to satisfy $\mathbb{E}$, then the capability we started out with is irrelevant.

This lemma is a key component in proving Update Capability Weakening (Lemma A.13).

**Lemma A.12** ($\mathbb{E}$-inversion)**.** *If* $\Delta; \Gamma \vdash_\mu \mathbb{E}[e] : \tau; \Delta'$ *then there exists* $\hat{\Delta}' \supseteq \Delta'$ *and* $\tau'$ *such that*

*(i)* $\Delta; \Gamma \vdash_\mu e : \tau'; \hat{\Delta}'$.

*(ii) for all* $e', \hat{\Delta}, \hat{\Delta}'' \supseteq \hat{\Delta}'$ *and* $\Gamma' \supseteq \Gamma$*, if* $\hat{\Delta}; \Gamma' \vdash_\mu e' : \tau'; \hat{\Delta}''$ *then* $\hat{\Delta}; \Gamma' \vdash_\mu \mathbb{E}[e'] : \tau; \Delta'$.

*Proof.* Proceed by induction on the expression typing derivation of $\mathbb{E}[e]$. In each case $\mathbb{E}$ may be $\_$ or a compound context. In the case where $\mathbb{E}$ can be a compound context we don't consider the $\_$ case as this holds trivially.

**case** (A.EXPR.VAR) **:**

In this case $\mathbb{E} = \_$, $e = x$ and $\Delta' = \Delta$.

Assume $\Delta; \Gamma, x : \tau \vdash_\mu x : \tau; \Delta$ and choose the existentially quantified variable $\hat{\Delta} = \Delta$. (i) holds by assumption.

To prove (ii) assume $\hat{\Delta}'' \supseteq \Delta$ (*), $\Gamma' \supseteq \Gamma, x : \tau$ and that for some $\hat{\Delta}$, $\hat{\Delta}; \Gamma' \vdash_\mu e' : \tau; \hat{\Delta}''$ (**). To complete we are required to show $\hat{\Delta}; \Gamma' \vdash_\mu e' : \tau; \Delta$, which follows from (*) and (**) using (A.EXPR.SUB) type rule.

**case** (A.EXPR.INT—XVAR—UPDATE) **:**

These cases all follow in a similar way to the (A.EXPR.VAR) case.

**case** (A.EXPR.RECORD) **:**

$\mathbb{E} = \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_i = \mathbb{E}'[e], \ldots, 1_n = e_n\}$. By assumption (where $\Delta \equiv \Delta_0, \Delta' \equiv \Delta_n$):

$$\frac{\begin{array}{c}\Delta_0; \Gamma \vdash_\mu v_1 : \tau_1; \Delta_1 \ldots \\ \Delta_{i-1}; \Gamma \vdash_\mu \mathbb{E}'[e] : \tau_i; \Delta_i \\ \ldots \Delta_{n-1}; \Gamma \vdash_\mu e_n : \tau_n; \Delta_n\end{array}}{\Delta_0; \Gamma \vdash_\mu \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_i = \mathbb{E}'[e], \ldots, 1_n = e_n\} : \{\mathrm{l}_1 : \tau_1, \ \ldots, \ \mathrm{l}_n : \tau_n\}; \Delta_n}$$

Prove that for some $\hat{\Delta}' \supseteq \Delta_n$ (i) and (ii) hold.

(i) By induction on the typing derivation of $\mathbb{E}[e]$ we have that for some $\hat{\Delta}' \supseteq \Delta_i$ it holds that $\Delta_0; \Gamma \vdash_\mu e : \tau'; \hat{\Delta}'$. By Typing Weakens Capability lemma $\Delta_i \supseteq \Delta_n$, therefore $\hat{\Delta}' \supseteq \Delta_n$ as required.

(ii) Assume $\hat{\Delta}; \Gamma' \vdash_\mu e' : \tau'; \hat{\Delta}''$ holds for some $\hat{\Delta}, \hat{\Delta}'' \supseteq \hat{\Delta}'$ and $\Gamma' \supseteq \Gamma$. Note that by Value Typing lemma and Weakening, we have $\hat{\Delta}; \Gamma' \vdash_\mu v_j : \tau_j; \hat{\Delta}$ for $1 \leq j \leq i-1$. Prove

$$\frac{\begin{array}{c}\hat{\Delta}; \Gamma' \vdash_\mu v_1 : \tau_1; \hat{\Delta} \ldots \\ \hat{\Delta}; \Gamma' \vdash_\mu \mathbb{E}'[e'] : \tau_i; \Delta_i \\ \ldots \Delta_{n-1}; \Gamma' \vdash_\mu e_n : \tau_n; \Delta_n\end{array}}{\hat{\Delta}; \Gamma' \vdash_\mu \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_i = \mathbb{E}'[e'], \ldots, 1_n = e_n\} : \{\mathrm{l}_1 : \tau_1, \ \ldots, \ \mathrm{l}_n : \tau_n\}; \Delta_n}$$

By induction, we have $\hat{\Delta}; \Gamma \vdash_\mu \mathbb{E}'[e'] : \tau; \Delta_i$. The reset of the premises follow from the assumptions using Weakening.

**case** (A.EXPR.APP) **:**

There are two possibilities for the form of $\mathbb{E}$: $v\ \mathbb{E}'$ and $\mathbb{E}'\ e$. We just consider the first as the second is similar.

Assume $\mathbb{E} = v\ \mathbb{E}'$ and

$$\frac{\begin{array}{l}\Delta; \Gamma \vdash_\mu v : \tau_1 \xrightarrow{\hat{\mu}; \Delta_f} \tau_2; \Delta' \\ \Delta'; \Gamma \vdash_\mu \mathbb{E}'[e] : \tau_1; \Delta'' \qquad \Delta''' \subseteq \Delta'' \\ \hat{\mu} \leq \mu \qquad \hat{\mu} = \mathsf{U} \implies \Delta''' \subseteq \Delta_f\end{array}}{\Delta; \Gamma \vdash_\mu v\ \mathbb{E}'[e] : \tau_2; \Delta'''}$$

Show that (i) and (ii) hold for some $\tau'$ and $\hat{\Delta}' \supseteq \Delta'''$.

(i) By IH there exists a $\hat{\Delta}' \supseteq \Delta''$ such that $\Delta; \Gamma \vdash_\mu e : \tau'; \hat{\Delta}'$. From the assumptions we can deduce $\hat{\Delta}' \supseteq \Delta'''$ as required.

(ii) By the typing judgement for $v$, Value Typing lemma and Weakening, for some $\Gamma' \supseteq \Gamma$ we have

$$\hat{\Delta}; \Gamma' \vdash_\mu v : \tau_1 \xrightarrow{\hat{\mu}; \Delta_f} \tau_2; \hat{\Delta} \tag{1}$$

Thus, it suffices to prove

$$\frac{\begin{array}{l}\hat{\Delta}; \Gamma' \vdash_\mu v : \tau_1 \xrightarrow{\hat{\mu}; \Delta_f} \tau_2; \hat{\Delta} \\ \hat{\Delta}; \Gamma' \vdash_\mu \mathbb{E}'[e'] : \tau_1; \Delta'' \qquad \Delta''' \subseteq \Delta'' \\ \hat{\mu} \leq \mu \qquad \hat{\mu} = \mathsf{U} \implies \Delta''' \subseteq \Delta_f\end{array}}{\hat{\Delta}; \Gamma' \vdash_\mu v\ \mathbb{E}'[e] : \tau_2; \Delta'''}$$

The typing for $v$ holds by 1 and the judgement for $\mathbb{E}'[e']$ by IH. Finally, the subset constraints hold directly by assumptions.

**case** (A.EXPR.CON) **:**

$\mathbb{E} = \mathbf{con}_{\mathsf{t}}$ . Assume

$$\frac{\Delta; \Gamma \vdash_{\mu} \mathbb{E}'[e] : \mathsf{t}; \Delta' \qquad \Gamma(\mathsf{t}) = \tau \qquad \mathsf{t} \in \Delta'}{\Delta; \Gamma \vdash_{\mu} \mathbf{con}_{\mathsf{t}}\ \mathbb{E}'[e] : \tau; \Delta'}$$

(i) follows by IH. To prove (ii) assume that for some arbitrary $e', \hat{\Delta}, \hat{\Delta}'' \supseteq \hat{\Delta}'$ and $\Gamma' \supseteq \Gamma$

$$\hat{\Delta}; \Gamma' \vdash_{\mu} e' : \mathsf{t}; \hat{\Delta}''$$

and prove

$$\frac{\hat{\Delta}; \Gamma' \vdash_{\mu} \mathbb{E}'[e'] : \mathsf{t}; \Delta' \qquad \Gamma'(\mathsf{t}) = \tau \qquad \mathsf{t} \in \Delta'}{\hat{\Delta}; \Gamma' \vdash_{\mu} \mathbf{con}_{\mathsf{t}}\ \mathbb{E}'[e] : \tau; \Delta'}$$

The first premise of which follows by induction and the second and third directly from the assumptions.

**case** (A.EXPR.IF) **:**

There are two cases for the form of $\mathbb{E}$: **if** $v = \mathbb{E}'$ **then** $e_1$**else** $e_2$ and **if** $\mathbb{E}' = e$ **then** $e_1$**else** $e_2$. We consider only the first as the second is similar.
Assume $\mathbb{E} = \mathbf{if}\ \mathbb{E}' = e\ \mathbf{then}\ e_1 \mathbf{else}\ e_2$ and

$$\frac{\begin{array}{cc}\Delta; \Gamma \vdash_{\mu} v : \tau; \Delta_1 & \Delta_1; \Gamma \vdash_{\mu} \mathbb{E}'[e] : \tau; \Delta_2 \\ \Delta_2; \Gamma \vdash_{\mu} e_1 : \tau'; \Delta_3 & \Delta_2; \Gamma \vdash_{\mu} e_2 : \tau'; \Delta_4\end{array}}{\Delta; \Gamma \vdash_{\mu} \mathbf{if}\ v = \mathbb{E}'[e]\ \mathbf{then}\ e_1 \mathbf{else}\ e_2 : \tau'; \Delta_3 \cap \Delta_4}$$

where $\Delta_3 \cap \Delta_4 \equiv \Delta'$. Prove (i) and (ii) hold. (i) holds by induction on the typing derivation of $\mathbb{E}'[e]$ and use of Weakening, Capability Strengthening, and Typing Weakens Capability lemmas. To prove (ii) assume for arbitrary $e', \hat{\Delta}, \hat{\Delta}'' \supseteq \hat{\Delta}'$ and $\Gamma' \supseteq \Gamma$ that $\hat{\Delta}; \Gamma' \vdash_{\mu} e' : \tau'; \hat{\Delta}''$ holds and show

$$\frac{\begin{array}{cc}\hat{\Delta}; \Gamma' \vdash_{\mu} v : \tau; \hat{\Delta}^{(A)} & \hat{\Delta}; \Gamma' \vdash_{\mu} \mathbb{E}'[e'] : \tau; {\Delta_2}^{(B)} \\ \Delta_2; \Gamma' \vdash_{\mu} e_1 : \tau'; {\Delta_3}^{(C)} & \Delta_2; \Gamma' \vdash_{\mu} e_2 : \tau'; {\Delta_4}^{(D)}\end{array}}{\hat{\Delta}; \Gamma' \vdash_{\mu} \mathbf{if}\ v = \mathbb{E}'[e']\ \mathbf{then}\ e_1 \mathbf{else}\ e_2 : \tau'; \Delta_3 \cap \Delta_4}$$

(A) holds by assumptions, Value Typing lemma and Weakening. (B) holds by induction on the typing derivation of $\mathbb{E}[e]$, while (C) and (D) hold straight from the assumptions by use of Weakening.

**case** (A.EXPR.PROJ-ABS-REF-DEREF-ASSIGN-LET-IFUPDATE-SUB) **:**

These cases follow by simple inductive arguments, similar to those presented above, using Value Typing lemma and Weakening lemma.

□

The following Update Capability Weakening lemma is used in the proof of Update Program Safety. It states that given an expression where the next redex is an update, this expression is checkable with the capability annotated on the update. Put another way, the capability annotated on the update is a sufficient capability for the execution of the continuation. If this is true, then the only types concreted by old code in the continuation are those not updated at this update point.

**Lemma A.13** (Update Capability Weakening)**.** *If* $\Delta; \Gamma \vdash_{\mu} \mathbb{E}[\mathbf{update}^{\Delta''}] : \tau; \Delta'$ *then* $\Delta''; \Gamma \vdash_{\mu} \mathbb{E}[\mathbf{update}^{\Delta''}] : \tau; \Delta'$.

*Proof.* Assume $\Delta; \Gamma \vdash_\mu \mathbb{E}[\mathbf{update}^{\Delta''}] : \tau; \Delta'$.
By $\mathbb{E}$-inversion lemma, for some $\hat{\Delta}'$: $\Delta; \Gamma \vdash_\mu \mathbf{update}^{\Delta''} : \tau; \hat{\Delta}'$.
By update type rule $\hat{\Delta}' = \Delta''$ and $\Delta'' \subseteq \Delta$.
Thus: $\Delta''; \Gamma \vdash_\mu \mathbf{update}^{\Delta''} : \tau; \Delta''$.
By $\mathbb{E}$-inversion: $\Delta''; \Gamma \vdash_\mu \mathbb{E}[\mathbf{update}^{\Delta'}] : \tau; \Delta'$, as required. □

**Lemma A.14** (Heap Extension)**.** *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\emptyset; \Omega, \Phi \vdash_{\mathsf{N}} v : \tau; \emptyset$ *and* $l \notin \mathrm{dom}(H)$ *then* $\Omega; \Phi, l : \tau\ \mathbf{ref} \vdash H, l \mapsto (\cdot, v)$

*Proof.* By definition of heap typing. □

**Lemma A.15** (Update Expression Safety)**.** *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\Delta_1; \Omega, \Phi, \Gamma \vdash_\mu e : \tau; \Delta_2$ *and* $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \Delta_1)$ *then* $\Delta_1; \mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash_\mu \mathcal{U}[e]^{\mathrm{upd}} : \tau; \Delta_2$

*Proof.* Proceed by induction on the derivation of $\Delta; \Omega, \Phi, \Gamma \vdash_\mu e : \tau; \Delta'$:

**case** (A.EXPR.INT) **:**
By assumption $\Delta; \Omega, \Phi, \Gamma \vdash_\mu n : \mathsf{int}; \Delta'$. Since $\mathcal{U}[n]^{\mathrm{upd}} = n$, we have $\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash_\mu \mathcal{U}[n]^{\mathrm{upd}} : \mathsf{int}; \Delta'$ follows from (A.EXPR.INT).

**case** (A.EXPR.VAR) **:**
By assumption $\Delta; \Omega, \Phi, \Gamma, x : \tau \vdash_\mu x : \tau; \Delta'$. Since $\mathcal{U}[x]^{\mathrm{upd}} = x$ and $\mathcal{U}[\Omega, \Phi, \Gamma, x : \tau]^{\mathrm{upd}} = \mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}}, x : \tau$, the result follows from (A.EXPR.VAR).

**case** (A.EXPR.XVAR) **:**
By assumption $\Delta; \Omega, \Phi, \Gamma \vdash_\mu \mathsf{z} : \tau; \Delta'$. Thus $\Phi(\mathsf{z}) = \tau$ and by (a) $\mathsf{z} \in \mathrm{dom}(H)$.
By definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on expressions we have $\mathcal{U}[\mathsf{z}]^{\mathrm{upd}} = \mathsf{z}$.
There are three ways in which $\mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}}(\mathsf{z}) = \tau'$ can arise:
**case** $\mathsf{z} \in \mathrm{dom}(\mathrm{upd.AB})$ **:**
As $\mathsf{z} \in \mathrm{dom}(H)$ and by updateOK the domain of the heap and upd.AB are disjoint, we can conclude $\mathsf{z} \notin$ upd.AB, therefore this case cannot occur.
**case** $\mathsf{z} \in \mathrm{dom}(\mathrm{upd.UB})$ **:**
Let $\mathrm{upd.UB}(\mathsf{z}) = (\tau', b_v)$.
By definition of $\mathcal{U}[-]^{\mathrm{upd}}$ we have $\mathcal{U}[\Phi]^{\mathrm{upd}}(\mathsf{z}) = \mathrm{heapType}(\tau', b_v)$.
By updateOK assumption $\mathcal{U}[\Omega, \mathrm{types}(H)]^{\mathrm{upd}} \vdash \mathrm{heapType}(\tau', b_v) <: \tau$.
By Weakening $\mathcal{U}[\Omega, \Phi]^{\mathrm{upd}} \vdash \mathrm{heapType}(\tau', b_v) <: \tau$.
The result follows by use of (A.EXPR.SUB) type rule.
**case** $\mathsf{z} \notin \mathrm{dom}(\mathrm{upd.UB})$ **:**
By definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on contexts $\mathcal{U}[\Phi]^{\mathrm{upd}}(\mathsf{z}) = \tau$, thus $\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash_\mu \mathsf{z} : \tau; \Delta'$, as required.

**case** (A.EXPR.REFERENCE) **:**
Similar to the var case.

**case** (A.EXPR.ABS) **:**
By assumption

$$\frac{\Delta; \Omega, \Phi, \Gamma \vdash_\mu e : \tau; \Delta' \qquad [\Omega, \Phi](\mathsf{t}) = \tau}{\Delta; \Omega, \Phi, \Gamma \vdash_\mu \mathbf{abs}_{\mathsf{t}}\ e : \mathsf{t}; \Delta'}$$

Consider the form of upd.UN:
**case** $\mathsf{t} \notin \mathrm{dom}(\mathrm{upd.UN})$ **:**
By definition we have $\mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}}(\mathsf{t}) = \tau$ and $\mathcal{U}[\mathbf{abs}_{\mathsf{t}}\ e]^{\mathrm{upd}} = \mathbf{abs}_{\mathsf{t}}\ \mathcal{U}[e]^{\mathrm{upd}}$.
The desired result follows by induction and (A.EXPR.ABS).

**case** upd.UN($\mathsf{t}$) = $(\tau'', \mathsf{c})$ **:**

Observe $\mathcal{U}[\mathbf{abs}_{\mathsf{t}}\ e]^{\text{upd}} = \mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ \mathcal{U}[e]^{\text{upd}})$. Using (A.EXPR.ABS) and (A.EXPR.APP) we are required to prove:

$$\frac{\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash_\mu \mathsf{c} : \tau' \xrightarrow{\mathsf{N}; \Delta_c} \tau''; \Delta^{(a)} \quad \Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{upd} \vdash_\mu \mathcal{U}[e]^{upd} : \tau'; \Delta'^{(b)}}{\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash_\mu \mathsf{c}\ \mathcal{U}[e]^{\text{upd}} : \tau''; \Delta'} \quad [\Omega, \Phi, \Gamma](\mathsf{t}) = \tau'^{(c)}}{\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash_\mu \mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ \mathcal{U}[e]^{\text{upd}}) : \mathsf{t}; \Delta'}$$

(b) holds by induction. To prove (a):

| | |
|---|---|
| $\emptyset; \mathcal{U}[\Omega, \text{types}(H)]^{\text{upd}} \vdash_{\mathsf{N}} \mathsf{c} : \tau -> \tau'; \emptyset$ | By updateOK() assumption |
| $\Delta; \mathcal{U}[\Omega, \text{types}(H), \Gamma]^{\text{upd}} \vdash_{\mathsf{N}} \mathsf{c} : \tau \xrightarrow{\mathsf{N}; \Delta_c} \tau'; \Delta$ | By Cap. Strengthening lemma |
| $\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash_{\mathsf{N}} \mathsf{c} : \tau \xrightarrow{\mathsf{N}; \Delta_c} \tau'; \Delta$ | By Ctx. Weakening lemma |

Where the last step is valid because $\Omega; \Phi \vdash H$ and so $\text{types}(H) \subseteq \Phi$.
By case split $\Omega = (\mathsf{t} = \tau, \Omega')$ for some $\Omega'$.
By definition of $\mathcal{U}[]$ $\mathcal{U}[\mathsf{t} = \tau, \Omega', \Phi, \Gamma]^{\text{upd}} = \mathsf{t} = \tau'', \mathcal{U}[\Omega', \Phi, \Gamma]^{\text{upd}}$, thus (c) holds.

**case** (A.EXPR.CON) **:**

Assume

$$\frac{\mathsf{t} \in \Delta' \qquad \Omega(\mathsf{t}) = \tau \qquad \Delta; \Omega, \Phi, \Gamma \vdash_\mu e : \mathsf{t}; \Delta'}{\Delta; \Omega, \Phi, \Gamma \vdash_\mu \mathbf{con}_{\mathsf{t}}\ e : \tau; \Delta'} \tag{2}$$

$$\text{updateOK}(\text{upd}, \Omega, H, \Delta) \tag{3}$$

Suffices to show that the leaves of this derivation hold:

$$\frac{\mathsf{t} \in \Delta'^{(a)} \qquad \mathcal{U}[\Omega]^{\text{upd}}(\mathsf{t}) = \tau^{(b)} \qquad \Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash_\mu \mathcal{U}[e]^{\text{upd}} : \mathsf{t}; \Delta'^{(c)}}{\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash_\mu \mathbf{con}_{\mathsf{t}}\ \mathcal{U}[e]^{\text{upd}} : \tau; \Delta'}$$

(a) holds by assumptions. (c) holds by induction. Now show (b). Note that by (3) $\Delta \cap \text{dom}(\text{upd.UN}) = \emptyset$, so by assumption $\mathsf{t} \notin$ upd.UN.

| | |
|---|---|
| $\Omega = \mathsf{t} = \tau, \Omega'$ | for some $\Omega'$, by 3 |
| $\text{dom}(\Delta') \cap \text{dom}(\text{upd.UN}) = \emptyset$ | by 3 |
| $\mathcal{U}[\mathsf{t} = \tau, \Omega']^{\text{upd}} = (\mathsf{t} = \tau, \mathcal{U}[\Omega']^{\text{upd}})$ | as $\mathsf{t} \notin$ upd.UN |

**case** (A.EXPR.RECORD) **:**

By assumption (where $\Delta \equiv \Delta_0, \Delta' \equiv \Delta_n$):

$$\frac{\Delta_i; \Omega, \Phi, \Gamma \vdash_\mu e_{i+1} : \tau_{i+1}; \Delta_{i+1} \qquad i \in 1..(n-1) \qquad n \geq 0}{\Delta_0; \Omega, \Phi, \Gamma \vdash_\mu \{\mathsf{l}_1 = e_1, \ldots, \mathsf{l}_n = e_n\} : \{\mathsf{l}_1 : \tau_1, \ldots, \mathsf{l}_n : \tau_n\}; \Delta_n}$$

By typing weakens capability $\Delta_{i-1} \subseteq \Delta_i$ for $i \in 1..n$. By the fact that updateOK($-$) is preserved under weakening of capability, and induction, we can prove:

$$\Delta_i; \Omega, \Phi, \Gamma \vdash_\mu \mathcal{U}[e_{i+1}]^{\text{upd}} : \tau_{i+1}; \Delta_{i+1}$$

For each $i \in 0..n-1$. Finally, the result follows using (A.EQ.RECORD).

**case** (A.EXPR.SUB) **:**

Assume

$$\frac{\Delta; \Omega, \Phi, \Gamma \vdash_\mu e : \tau'; \Delta'' \qquad \Gamma \vdash \tau' <: \tau \qquad \Delta' \subseteq \Delta''}{\Delta; \Omega, \Phi.\Gamma \vdash_\mu e : \tau; \Delta'}$$

Prove

$$\frac{\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash_\mu \mathcal{U}[e]^{\text{upd}} : \tau'; \Delta'' \qquad \mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash \tau' <: \tau \qquad \Delta' \subseteq \Delta''}{\Delta; \mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash_\mu \mathcal{U}[e]^{\text{upd}} : \tau; \Delta'}$$

The typing judgement on $e$ holds by induction and the subset constraint holds by assumption. We are left to show the subtype assertion. By assumption $\Omega, \Phi, \Gamma \vdash \tau' <: \tau$. Furthermore, this judgement's only constraint on $\Omega, \Phi, \Gamma$ is that the free type names in $\tau$ and $\tau'$ are in the domain of $\Omega, \Phi, \Gamma$. It is easily proven that the domain of $\mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}}$ is a conservative extension of the domain of $\Omega, \Phi, \Gamma$. Thus the subtype judgement holds.

**case** (A.EXPR.UPDATE) **:**
Update checks to be int in any environment.

**case** (A.EXPR.(APP—PROJ—LET—REF—DEREF—ASSIGN—IF—IF-UPDATE)) **:**
All follow by a simple inductive argument.

□

The next lemma, Heap Update Safety, tells us that given a typeable heap and a valid update, applying that update to the heap leaves the heap well typed in the updated environment.

**Lemma A.16** (Heap Update Safety)**.** *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\text{updateOK}(\text{upd}, \Omega, H, \hat{\Delta})$ *then* $\mathcal{U}[\Omega]^{\text{upd}}; \mathcal{U}[\Phi]^{\text{upd}} \vdash \mathcal{U}[H]^{\text{upd}}$

*Proof.* First note that from $\Omega; \Phi \vdash H$ we can deduce that for all $\rho \in \text{dom}(H), \tau, e$:

(a) $\text{dom}(\Phi) = \text{dom}(H)$

(b) if $\rho = \mathsf{z}$ and $H(\mathsf{z}) = (\tau, e)$ then $\Omega, \Phi \vdash e : \tau$ and $\Phi(\mathsf{z}) = \tau$ **ref**

(c) if $\rho = \mathsf{z}$ and $H(\mathsf{z}) = (\tau, \lambda(x).e\ )$ then $\Omega, \Phi \vdash \lambda(x).e\ : \tau$ and $\Phi(\mathsf{z}) = \tau$

(d) if $\rho = r$ and $H(r) = (\cdot, e)$ then there exists a $\tau$ such that $\Omega, \Phi \vdash e : \tau$ and $\Phi(r) = \tau$ **ref**

So assume (a)-(d) and also:

$$\vdash \Omega \tag{4}$$

$$\text{updateOK}(\text{upd}, \Omega, H, \hat{\Delta}) \tag{5}$$

Via the same expansion we are required to prove for all $\rho \in \text{dom}(\mathcal{U}[H]^{\text{upd}}), \tau, e$ that

(i) $\text{dom}(\mathcal{U}[\Phi]^{\text{upd}}) = \text{dom}(\mathcal{U}[H]^{\text{upd}})$

(ii) if $\rho = \mathsf{z}$ and $\mathcal{U}[H]^{\text{upd}}(\mathsf{z}) = (\tau, e)$ then $\mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash e : \tau$ and $\mathcal{U}[\Phi]^{\text{upd}}(\mathsf{z}) = \tau$ **ref**

(iii) if $\rho = \mathsf{z}$ and $\mathcal{U}[H]^{\text{upd}}(\mathsf{z}) = (\tau, \lambda(x).e\ )$ then $\mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash \lambda(x).e\ : \tau$ and $\mathcal{U}[\Phi]^{\text{upd}}(\mathsf{z}) = \tau$

(iv) if $\rho = r$ and $\mathcal{U}[H]^{\text{upd}}(r) = (\cdot, e)$ then there exists $\tau$ such that $\mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash e : \tau$ and $\mathcal{U}[\Phi]^{\text{upd}}(r) = \tau$ **ref**

hold. (a) implies (i) by inspection of the definition of $\mathcal{U}[]$ on contexts and heaps. We are left to show (ii)-(iv).

Observe that types$(H) \subseteq \Phi$ because of (b) and (c).

Now consider the form of an arbitrary entry in $\mathcal{U}[H]^{\text{upd}}$:

**case** $r \mapsto (\cdot, e)$ **:**

In this case (ii) and (iii) hold trivially because the domain is a reference. To prove (iv) for $\mathcal{U}[H]^{\text{upd}}(r) = (\cdot, e)$ we show that, for some $\tau$

$$\frac{\emptyset; \mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash_{\mathsf{N}} e : \tau; \emptyset}{\mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash e : \tau} \tag{6}$$

$$\mathcal{U}[\Phi]^{\text{upd}}(r) = \tau \ \mathbf{ref} \tag{7}$$

By the action of $\mathcal{U}[-]^{\text{upd}}$ on heaps there exists an $e'$ such that $r \mapsto (\cdot, e') \in H$ and $e = \mathcal{U}[e']^{\text{upd}}$.

By (d) there exists a $\tau'$ such that

$$\Omega, \Phi \vdash e' : \tau' \tag{8}$$

$$\Phi(r) = \tau' \ \mathbf{ref} \tag{9}$$

By Update Expression Safety lemma $\mathcal{U}[\Omega; \Phi]^{\text{upd}} \vdash e' : \tau'$.

Take $\tau = \tau'$ to show 8 and 9.

By definition of $\mathcal{U}[-]^{\text{upd}}$ on heaps $\mathcal{U}[\Phi]^{\text{upd}}(r) = \tau'$ **ref** holds, which proves 9.

By UpdateOK Capability Weakening lemma updateOK$(\text{upd}, \Omega, \Phi, \emptyset)$.

By Update Expression Safety lemma:

$$\emptyset; \mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash_{\mathsf{N}} \mathcal{U}[e']^{\text{upd}} : \tau'; \emptyset$$

We obtain 8 by application of (A.BIND.EXPR).

**case** $\mathsf{z} \mapsto (\tau, b)$ **:**

In this case (iv) holds trivially and we are left to show (ii) and (iii).

**case** (ii) **:**

Assume $\mathcal{U}[H]^{\text{upd}}(z) = (\tau, e)$ and prove

$$\frac{\emptyset; \mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash_{\mathsf{N}} e : \tau; \emptyset}{\mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash e : \tau} \tag{10}$$

$$\mathcal{U}[\Phi]^{\text{upd}}(\mathsf{z}) = \tau \ \mathbf{ref} \tag{11}$$

By definition of $\mathcal{U}[-]^{\text{upd}}$ on heaps, there are three ways to generate elements of $\mathcal{U}[H]^{\text{upd}}$.

**case** $\mathsf{z} \in \text{dom}(H)$ and upd.UB$(\mathsf{z} = (\tau, e)$ **:**

As (a) holds by assumption and $\mathsf{z} \in \text{dom}(H)$ by case split, we have $\mathsf{z} \in \text{dom}(\Phi)$. Thus for some $\tau'$, $\Phi(\mathsf{z}) = \tau'$.

By definition of $\mathcal{U}[-]^{\text{upd}}$ on contexts $\mathcal{U}[\Phi]^{\text{upd}}(\mathsf{z}) = \tau$ **ref**, which proves 11.

By 9 $\mathcal{U}[\Omega, \text{types}(H)]^{\text{upd}} \vdash e : \tau$.

By context weakening $\mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash e : \tau$, which proves 10.

**case** $\mathsf{z} \in \text{dom}(H)$ and $\mathsf{z} \notin \text{dom}(\text{upd.UB})$ **:**

By the definition of $\mathcal{U}[-]^{\text{upd}}$ on heaps there must exist $b'$ such that $\mathcal{U}[\mathsf{z} \mapsto (\tau, b'), H']^{\text{upd}} = \mathsf{z} \mapsto (\tau, \mathcal{U}[b']^{\text{upd}}), \mathcal{U}[H']^{\text{upd}}$.

As $b = \mathcal{U}[b']^{\text{upd}}$ and this is an expression by fact we are in (ii) case split, $b'$ must also be an expression. Thus by (b) $\Omega, \Phi \vdash b' : \tau$ and $\Phi(\mathsf{z}) = \tau\ \mathbf{ref}$. Then by inversion $\emptyset; \Omega, \Phi \vdash_{\mathsf{N}} b' : \tau; \emptyset$ holds.

By Update Expression Safety lemma

$$\emptyset; \mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash_{\mathsf{N}} \mathcal{U}[b']^{\text{upd}} : \tau; \emptyset$$

which proves 10

As $\mathsf{z} \notin \text{dom(upd.UB)}$, by the definition of $\mathcal{U}[-]^{\text{upd}}$ on contexts we have $\mathcal{U}[\Omega, \Phi', \mathsf{z} : \tau\ \mathbf{ref}]^{\text{upd}} = \mathsf{z} : \tau\ \mathbf{ref}, \mathcal{U}[\Omega, \Phi']^{\text{upd}}$, which proves 11.

**case** $\mathsf{z} \notin \text{dom}(H)$ **:**

In this case it must hold that $\mathsf{z} \in \text{dom(upd.AB)}$.

By assumption$\mathcal{U}[H]^{\text{upd}}(\mathsf{z}) = (\tau, e)$ (where $e$ is in fact a value) therefore $\text{upd.AB}(\mathsf{z}) = (\tau, e)$.

By 9

$$\mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash e : \tau$$

which proves 10.

By inspection of the action of $\mathcal{U}[-]^{\text{upd}}$ on contexts we see that

$$\mathcal{U}[\Omega, \Phi]^{\text{upd}}(z) = \text{types(upd.AB)}(z) = \tau\ \mathbf{ref}$$

which proves 11, as required.

**case** (iii) **:**

Assume

$$\mathcal{U}[H]^{\text{upd}}(\mathsf{z}) = (\tau_1 \xrightarrow{\mu;\Delta'} \tau_2, \lambda^{\Delta}(x).e\ )$$

i.e. that $b = \lambda^{\Delta}(x).e$ and $\tau = \tau_1 \xrightarrow{\mu;\Delta'} \tau_2$. Prove

$$\mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash \mathcal{U}\big[\lambda^{\Delta}(x).e\ \big]^{\text{upd}} : \tau_1 \xrightarrow{\mu;\Delta'} \tau_2 \tag{12}$$

$$\mathcal{U}[\Omega, \Phi]^{\text{upd}}(\mathsf{z}) = \tau_1 \xrightarrow{\mu;\Delta'} \tau_2 \tag{13}$$

By definition of $\mathcal{U}[-]^{\text{upd}}$ on heaps, there are three ways to generate elements of $\mathcal{U}[H]^{\text{upd}}$.

**case** $\mathsf{z} \in \text{dom}(H)$ and $\mathsf{z} \in \text{dom(upd.UB)}$ **:**

By case split there exists $x, e, \tau_1, \tau_2, \mu', \Delta_1, \Delta_2$ such that $\text{upd.UB}(\mathsf{z}) = (\tau_1 \xrightarrow{\mu';\Delta_2} \tau_2, \lambda^{\Delta_1}(x).e\ )$.

$$\mathcal{U}[\Omega, \text{types}(H)]^{\text{upd}} \vdash \lambda^{\Delta_1}(x).e\ : \tau_1 \xrightarrow{\mu';\Delta_2} \tau_2 \qquad \text{by 9}$$

$$\mathcal{U}[\Omega, \text{types}(()H)]^{\text{upd}} \vdash \lambda^{\Delta_1}(x).e\ : \tau_1 \xrightarrow{\mu';\Delta_2} \tau_2 \ \text{ by weakening}$$

which proves 12. Finally, by (a), $\mathsf{z} \in \text{dom}(\Phi)$, so by definition of $\mathcal{U}[-]^{\text{upd}}$ on contexts: $\mathcal{U}[\Phi]^{\text{upd}}(z) = \tau_1 \xrightarrow{\mu';\Delta_2} \tau_2$ which proves 13

**case** $\mathsf{z} \in \text{dom}(H)$ and $\mathsf{z} \notin \text{dom(upd.UB)}$ **:**

By case split and definition of $\mathcal{U}[-]^{\text{upd}}$ on heaps, there exists $b', H'$ such that $\mathcal{U}[\mathsf{z} \mapsto (\tau, b'), H']^{\text{upd}} = z \mapsto (\tau, \mathcal{U}[b']^{\text{upd}}), \mathcal{U}[H']^{\text{upd}}$ and $H = \mathsf{z} \mapsto (\tau, b'), H'$.

Because $b'$ is a function by case split, then by the definition of $\mathcal{U}[-]^{\text{upd}}$ on bindings, $\mathcal{U}[b']^{\text{upd}}$ is a function, say $b' = \lambda^{\Delta}(x).e'$

By (c) and typing rules

$$\frac{\Delta;\Omega,\Phi \vdash_\mu e' : \tau_2;\Delta'}{\Omega,\Phi \vdash \lambda^{\Delta}(x).e' \ : \tau_1 \xrightarrow{\mu;\Delta'} \tau_2}$$

where $\tau = \tau_1 \xrightarrow{\mu;\Delta'} \tau_2$.
Required to prove 12 and 13.
By 9 bindOK[ types($H$) ]. By case split $\mathsf{z} \notin$ dom(upd.UB). By last two facts dom(upd.UN) $\cap\ \Delta = \emptyset$. It follows by the previous fact 9 that updateOK(upd, $\Omega, H, \Delta$).
By Update Expression Safety lemma $\Delta;\mathcal{U}[\Omega,\Phi]^{\mathrm{upd}} \vdash_\mu \mathcal{U}[e]^{\mathrm{upd}} : \tau_2;\Delta'$. Therefore, by use of (A.BIND.FUN), 12 holds.
By the definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on contexts it follows that $\mathcal{U}[\Phi]^{\mathrm{upd}}(z) = \tau$ making 13 holds, as required.

**case** $\mathsf{z} \notin \mathrm{dom}(H)$ **:**
The result follows similarly to this subcase in case (ii).

□

**Lemma A.17** (Update Program Safety)**.** *If*

*(i)* $\emptyset \vdash_\mu \Omega; H; \mathbb{E}[\mathbf{update}^{\hat{\Delta}}] : \tau$ *and*

*(ii)* updateOK(upd, $\Omega, H, \hat{\Delta}$)

*then* $\mathcal{U}[\emptyset]^{\mathrm{upd}} \vdash \mathcal{U}[\Omega]^{\mathrm{upd}};\mathcal{U}[H]^{\mathrm{upd}};\mathcal{U}[\mathbb{E}[0]]^{\mathrm{upd}} : \tau$

*Proof.* Assume

$$\frac{\begin{array}{c}\vdash \Omega \qquad \Omega;\Phi \vdash H \\ \Delta;\Omega,\Phi \vdash_{\mathsf{U}} \mathbb{E}[\mathbf{update}^{\hat{\Delta}}] : \tau;\Delta'\end{array}}{\emptyset \vdash \Omega; H; \mathbb{E}[\mathbf{update}^{\hat{\Delta}}] : \tau} \tag{14}$$

$$\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \hat{\Delta}) \tag{15}$$

It suffices to prove the hypotheses for this deduction:

$$\frac{\begin{array}{c}\vdash \mathcal{U}[\Omega]^{\mathrm{upd}}{}^{(a)} \qquad \mathcal{U}[\Omega]^{\mathrm{upd}};\Phi' \vdash \mathcal{U}[H]^{\mathrm{upd}}{}^{(b)} \\ \hat{\Delta};\mathcal{U}[\Omega]^{\mathrm{upd}},\Phi' \vdash_{\mathsf{U}} \mathcal{U}\left[\mathbb{E}[\mathbf{update}^{\hat{\Delta}}]\right]^{\mathrm{upd}} : \tau;\Delta'^{(c)}\end{array}}{\mathcal{U}[\emptyset]^{\mathrm{upd}} \vdash \mathcal{U}[\Omega]^{\mathrm{upd}};\mathcal{U}[H]^{\mathrm{upd}};\mathcal{U}\left[\mathbb{E}[\mathbf{update}^{\hat{\Delta}}]\right]^{\mathrm{upd}} : \tau}$$

Choose $\Phi' = \mathcal{U}[\Phi]^{\mathrm{upd}}$; then (a) follows from the definition of updateOK. (b) follows from Update Heap Safety lemma.

By Update Capability Weakening lemma and 14

$$\hat{\Delta};\Gamma,\Omega,\Phi' \vdash_{\mathsf{U}} \mathbb{E}[\mathbf{update}^{\hat{\Delta}}] : \tau;\Delta'$$

therefore (c) follows from Update Expression Safety lemma. □

The Non-update Expression Safety Lemma establishes that the typing relation is closed under reduction. One thing to stress is that we only require closure; the capabilities do not become more restrictive, indeed they can grow at function calls, which explains $\Delta_1' \supseteq \Delta_1$ and $\Delta_2' \supseteq \Delta_2$ in the existentially quantified variables.

**Lemma A.18** (Non-Update Expression Safety)**.** *If* $\vdash\ \Omega$ *and* $\Omega;\Phi\ \vdash\ H$ *and* $\Delta_1;\Omega,\Phi \vdash_\mu e:\tau;\Delta_2$ *and* $H;e \to H';e'$ *then* $\exists \Phi' \supseteq \Phi, \Delta_1' \supseteq \Delta_1, \Delta_2' \supseteq \Delta_2$ *such that*

*(i)* $\Omega;\Phi' \vdash H'$ *and*
*(ii)* $\Delta_1';\Omega,\Phi' \vdash_\mu e':\tau;\Delta_2'$

*Proof.* Proceed by induction on the derivation of $\Delta_1;\Gamma,\Omega,\Phi \vdash_\mu e:\tau;\Delta_2$.

**case** (A.EXPR.VAR) **:**
Expressions are closed w.r.t local variables, so this cannot occur.

**case** (A.EXPR.XVAR-INT-REFERENCE) **:**
These are values, so can not reduce.

**case** (A.TYPE.APPU) **:**
Assume

$$\Omega;\Phi \vdash H \tag{16}$$

$$\frac{\begin{array}{l}\Delta_1;\Omega,\Phi \vdash_\mu e_1 : \tau_1 \xrightarrow{\hat{\mu};\Delta'} \tau_2;\Delta_2 \\ \Delta_2;\Omega,\Phi \vdash_\mu e_2 : \tau_1;\Delta_3 \qquad \Delta_4 \subseteq \Delta_3 \\ \hat{\mu} \le \mu \qquad \hat{\mu} = \mathsf{U} \implies \Delta_4 \subseteq \Delta'\end{array}}{\Delta_1;\Omega,\Phi \vdash_\mu e_1\ e_2 : \tau_2;\Delta_4} \tag{17}$$

$$H;e_1\ e_2 \to H';e' \tag{18}$$

Required to prove that there exists $\Phi' \supseteq \Phi, \Delta_1' \supseteq \Delta_1, \Delta_2' \supseteq \Delta_2$ such that (i) and (ii) hold. The only possible expression reduction of 18 is (CALL). In this case

$$(H, \mathsf{z} \mapsto (\tau, \lambda^{\Delta}(x).e\ )), \mathsf{z}\ v \to (H, \mathsf{z} \mapsto \lambda^{\Delta}(x).e\ ), e[v/x]$$

Take $H' = H$ and $\Omega' = \Omega$ then (i) holds by 16.
Now prove (ii) by showing

$$\Delta \cup \Delta_1;\Gamma \vdash_\mu e[v/x] : \tau_2;\Delta_4 \tag{19}$$

By 17 we have $\Delta_2;\Omega,\Phi \vdash_\mu v : \tau_1;\Delta_3$ and by applying Value Typing lemma $\emptyset;\Omega,\Phi \vdash_\mu v : \tau_1;\emptyset$. By 16 $\Delta;\Omega,\Phi,x:\tau_1 \vdash_{\hat{\mu}} e : \tau_2;\Delta'$. Then by Substitution lemma

$$\Delta;\Omega,\Phi \vdash_{\hat{\mu}} e[v/x] : \tau_2;\Delta' \tag{20}$$

From typing of the LHS of the application

$$\Delta;\Omega,\Phi \vdash_\mu \mathsf{z} : \tau_1 \xrightarrow{\hat{\mu};\Delta'} \tau_2;\Delta_2 \tag{21}$$

It is clear that 20 must be derived either directly from the axiom (A.TYPE.XVAR) or by (possibly repeated use of) subsumption terminated by (A.TYPE.XVAR). It is easy to check that $<:$ is transitive allowing us to conclude that $\Omega \vdash \Phi(\mathsf{z}) <: \tau_1 \xrightarrow{\hat{\mu};\Delta'} \tau_2$. By Subtype Inversion Lemma there exists $\tau_3, \tau_4, \hat{\mu}', \Delta_5$ such that

$$\Omega \vdash \tau_3 \xrightarrow{\hat{\mu}';\Delta_5} \tau_4 <: \tau_1 \xrightarrow{\hat{\mu};\Delta'} \tau_2 \tag{22}$$

and thus

$$\tau_1 <: \tau_3 \qquad \tau_4 <: \tau_1 \qquad \Delta' \subseteq \Delta_5 \qquad \hat{\mu}' \le \hat{\mu} \tag{23}$$

By use of subsumption on 20 using facts from 23

$$\Delta;\Omega,\Phi \vdash_{\hat{\mu}'} e[v/x] : \tau_2;\Delta' \tag{24}$$

To show 19 we case on the value of $\hat{\mu}'$.

**case** $\hat{\mu}' = \mathsf{U}$ **:**

By 23 $\hat{\mu} = \mathsf{U}$. Thus by precondition of app rule $\mu = \mathsf{U}$ and $\Delta_4 \subseteq \Delta'$. By these derived facts, 24, Capability Strengthening Lemma, and subsumption rule we have the result.

**case** $\hat{\mu} = \mathsf{N}$ **:**

By 23 $\hat{\mu}$ can be either $\mathsf{U}$ or $\mathsf{N}$. If it is $\mathsf{U}$ we proceed as we did in the previous case, so suppose $\hat{\mu} = \mathsf{N}$. In this case $\mu$ is unconstrained so case on its value. If $\mu = \mathsf{U}$ then proceed as in $\hat{\mu}' = \mathsf{U}$ case. If $\mu = \mathsf{N}$ the result follows by Capability Strengthening Lemma and use of subsumption.

**case** (A.EXPR.CON) **:**

Assume

$$\frac{\Delta_1; \Omega, \Phi \vdash_\mu e' : \mathsf{t}; \Delta_2 \qquad \mathsf{t} \in \Delta_2 \qquad [\Omega, \Phi](\mathsf{t}) = \tau}{\Delta_1; \Omega, \Phi \vdash_\mu \mathbf{con}_\mathsf{t}\ e' : \tau; \Delta_2} \tag{25}$$

$$\Omega; \Phi \vdash H \tag{26}$$

The only possible expression reduction of $\mathbf{con}_\mathsf{t}\ e'$ is (CONABS). In this case $e' = \mathbf{abs}_\mathsf{t}\ v$ for some value $v$, and the result of the reduction is v. By inversion $\Delta_1; \Omega, \Phi \vdash_\mu v : \tau; \Delta_2$ as required.

**case** (A.EXPR.PROJ) **:**

Assume

$$\frac{\Delta_0; \Omega, \Phi \vdash_\mu e : \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\}; \Delta_1}{\Delta_0; \Omega, \Phi \vdash_\mu e.\mathrm{l}_i : \tau_i; \Delta_1} \tag{27}$$

$$\Omega; \Phi \vdash H \tag{28}$$

The only possible expression reduction is (PROJ). Assume

$$\frac{\dfrac{\Delta_{i-1}; \Omega, \Phi \vdash v_i : \tau_i; \Delta_i \qquad i \in 0..n \qquad n \geq 0}{\Delta_0; \Omega, \Phi \vdash \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_n = v_n\} : \{\mathrm{l}_1 : \tau_1, \ldots, \mathrm{l}_n : \tau_n\}; \Delta_n}}{\Delta_0; \Omega, \Phi \vdash \{\mathrm{l}_1 = v_1, \ldots, \mathrm{l}_n = v_n\}.\mathrm{l}_i : \tau_i; \Delta_n} \tag{29}$$

The result of the reduction is $v_i$. Choose $\Phi' = \Phi$, then (i) holds by 28. By typing weakens capability lemma we have $\Delta_i \subseteq \Delta_{i+1}$ for $i \in 0..n$. Therefore:

$$\begin{array}{ll} \Delta_0; \Omega, \Phi \vdash v_i : \tau_i; \Delta_i & \text{by capability strengthening} \\ \Delta_0; \Omega, \Phi \vdash v_i : \tau_i; \Delta_n & \text{by (A.EXPR.SUB)} \end{array}$$

as required to show (ii).

**case** (A.EXPR.LET) **:**

Assume

$$\frac{\begin{array}{c}\Delta_1; \Omega, \Phi \vdash_\mu e_1 : \tau_1'; \Delta_2 \\ \Delta_2; \Omega, \Phi, x : \tau_1 \vdash_\mu e_2 : \tau_2; \Delta_3\end{array}}{\Delta_1; \Omega, \Phi \vdash_\mu \mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2 : \tau_2; \Delta_3} \tag{30}$$

The only possible expression reduction is (LET). In this case $e_1$ is equal to some value. (i) holds by assumption and we are left to show (ii) where $e' = e_2[e_1/x]$.

$$\begin{array}{ll} \emptyset; \Omega, \Phi \vdash_\mu e_1 : \tau_1'; \emptyset & \text{by Value Typing lemma} \\ \Delta_2; \Omega, \Phi \vdash e_2[e_1/x]\tau_2; \Delta_3 & \text{by Substitution lemma} \\ \Delta_1; \Omega, \Phi \vdash e_2[e_1/x]\tau_2; \Delta_3 & \text{by Capability Strengthening lemma} \end{array}$$

**case** (A.EXPR.RECORD) **:**
No expression reductions apply.

**case** (A.EXPR.UPDATE) **:**
The case is trivially true as the expression cannot do a non-update reduction.

**case** (A.EXPR.SUB) **:**
Follows directly by application of IH.

**case** (A.EXPR.REF) **:**
Assume $\vdash \Omega$ and $\Omega; \Phi \vdash H$ and $H, \mathbf{ref}\ e'' \longrightarrow H', e'$ and

$$\frac{\Delta; \Omega, \Phi \vdash_\mu e : \tau; \Delta'}{\Delta; \Omega, \Phi \vdash_\mu \mathbf{ref}\ e : \tau\ \mathbf{ref}; \Delta'}$$

The only reduction rule applicable here is ref. This implies, for some value $v$ and location $r \notin \mathrm{dom}(H)$, that $e'' = v$ and $H' = H, r \mapsto v$.
Required to prove
(i) $\Omega; \Phi, r : \tau\ \mathbf{ref} \vdash H, r \mapsto (\cdot., v)$
(ii) $\Delta; \Omega, \Phi, r : \tau\ \mathbf{ref} \vdash_\mu r : \tau\ \mathbf{ref}; \Delta'$
(i) follows from Heap Extension lemma. (ii) is deducible from (A.EXPR.LOC) and (A.EXPR.SUB).

**case** (A.EXPR.(IF—DEREF—ASSIGN—ABS—IF-UPDATE)) **:**
These cases follow similarly.

□

**Lemma A.19** (Non-Update Program Safety)**.** *If* $\vdash \Omega$*,* $\Omega; \Phi \vdash H$*,* $\Delta_1; \Omega, \Phi \vdash \mathbb{E}[e] : \tau; \Delta_2$ *and* $\Omega; H; \mathbb{E}[e] \longrightarrow \Omega; H'; \mathbb{E}[e']$*, then there exists* $\Phi' \supseteq \Phi, \Delta_1' \supseteq \Delta_1$ *and* $\Delta_2' \supseteq \Delta_2$ *such that*

*(i)* $\Omega; \Phi' \vdash H$

*(ii)* $\Delta_1'; \Omega, \Phi \vdash \mathbb{E}[e'] : \tau; \Delta_2'$

*Proof.* By $\mathbb{E}$-inversion lemma we have, for some $\tau'$ and $\hat{\Delta}' \supseteq \Delta_2'$, that $\Delta_1; \Omega, \Phi \vdash e : \tau'; \hat{\Delta}'$.
By inversion of derivation of program reduction $\Omega; H; e \longrightarrow \Omega; H'; e'$.
By Non-Update Expression Safety lemma there exists $\Phi' \supseteq \Phi, \Delta_1' \supseteq \Delta_1$ and $\hat{\Delta}'' \supseteq \hat{\Delta}'$ such that $\Delta_1'; \Omega, \Phi \vdash e' : \tau'; \hat{\Delta}'$ and $\Omega; \Phi \vdash H$. The latter proves (i).
By weakening $\Delta_1; \Omega, \Phi' \vdash \mathbb{E}[e] : \tau'; \Delta_2$.
By $\mathbb{E}$-inversion lemma $\Delta_1'; \Omega, \Phi' \vdash \mathbb{E}[e] : \tau'; \Delta_2$, which proves (ii) as required.

□

**Lemma A.20** (Preservation)**.** *If* $\emptyset \vdash \Omega; H; e : \tau$ *then*

*(i) if* $\Omega; H; e \to \Omega; H'; e'$ *then* $\emptyset \vdash \Omega; H'; e' : \tau$

*(ii) if* $\Omega; H; e \xrightarrow{\mathrm{upd}} \Omega'; H'; e'$ *then* $\emptyset \vdash \Omega'; H'; e' : \tau$

*Proof.* Suppose $\emptyset \vdash \Omega; H; e : \tau$ and consider the form of the transition:

**case** $\Omega; H; e \to \Omega; H'; e'$ **:**
(i) holds by Non-Update Program Safety lemma. (ii) trivially holds.

**case** $\Omega; H; e \xrightarrow{\mathrm{upd}} \Omega'; H'; e'$ **:**
This transition must be by the update rule, therefore $e = \mathbb{E}[\mathbf{update}^\Delta]$ for some $\mathbb{E}$ and $\Delta$; and either
(a) $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \Delta)$ $\Omega' = \mathcal{U}[\Omega]^{\mathrm{upd}}$ $H' = \mathcal{U}[H]^{\mathrm{upd}}$ $e' = \mathcal{U}[\mathbb{E}[0]]^{\mathrm{upd}}$

(b) $e' = \mathbb{E}[1]$
In the former case $\emptyset \vdash \Omega'; H'; e' : \tau$ by Update Program Safety lemma. In the latter case $\mathbb{E}[1]$ can be typed by $\mathbb{E}$-inversion lemma. In either case (ii) is confirmed. (i) holds trivially.

□

**Lemma A.21** (Progress)**.** *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\Delta_1; \Omega, \Phi \vdash_\mu e : \tau; \Delta_2$ *then either*

*(i) there exists* $\Omega', H', e'$ *such that* $\Omega; H; e \to \Omega'; H'; e'$ *or*
*(ii)* $e$ *is a value*

*Proof.* Proceed by induction on the derivation of $\Delta_1; \Omega, \Phi \vdash_\mu e : \tau; \Delta_2$.

**case** (A.EXPR.INT—XVAR—LOC) **:**
All values.

**case** (A.EXPR.VAR) **:**
Cannot occur because the context is closed under local variables.

**case** (A.EXPR.APP) **:**
Assume

$$\vdash \Omega \tag{31}$$

$$\Omega; \Phi \vdash H \tag{32}$$

$$\frac{\begin{array}{l}\Delta_1; \Omega, \Phi \vdash_\mu e_1 : \tau_1 \xrightarrow{\hat{\mu}; \Delta'}{}^{(a)} \tau_2; \Delta_2 \\ \Delta_2; \Omega, \Phi \vdash_\mu e_2 : \tau_1; \Delta_3 \qquad \Delta_4 \subseteq \Delta_3 \\ \hat{\mu} \leq \mu \qquad (\hat{\mu} = \mathsf{U}) \implies \Delta_4 \subseteq \Delta'\end{array}}{\Delta_1; \Omega, \Phi \vdash_\mu e_1\ e_2 : \tau_2; \Delta_4} \tag{33}$$

By IH one of (i)-(iii) holds for $e_1$:
**case** (i) holds for $e_1$ **:**
(i) holds for $e_1\ e_2$ by cong rule.
**case** (iii) holds for $e_1$ **:**
By IH there are three cases to consider for $e_1$:
**case** (i) holds for $e_2$ **:**
(i) holds for $e_1\ e_2$ by cong rule
**case** (iii) holds for $e_2$ **:**

$e_1 = \mathsf{z}$ by Canonical Forms lemma
$\Delta_1; \Omega, \Phi \vdash_\mu \mathsf{z} : \tau_1 \xrightarrow{\hat{\mu}; \Delta'} \tau_2; \Delta_2$ By (a)

Thus $\Phi(\mathsf{z}) = \tau_1 \xrightarrow{\hat{\mu}; \Delta'} \tau_2$ and $H(\mathsf{z}) = \lambda^\Delta(x).e$ by 32. Therefore the (CALL) reduction rule matches and (a) holds for $e_1\ e_2$.

**case** (A.EXPR.SUB) **:**
Follows directly by induction on the sub derivation.

**case** (A.EXPR.ABS) **:**
Assume

$$\frac{\Delta; \Omega, \Phi \vdash_\mu e : \tau; \Delta' \qquad \Gamma(\mathsf{t}) = \tau}{\Delta; \Omega, \Phi \vdash_\mu \mathbf{abs}_\mathsf{t}\ e : \mathsf{t}; \Delta'}$$

By IH there are three cases to consider:

**case** (i) holds for $e$ **:**

By cong reduction rule (i) holds.

**case** (iii) holds for $e$ **:**

$e$ is a value by case split, thus $\mathbf{abs}_t$ is also a value (by inspection of values)

**case** (A.EXPR.UPDATE) **:**

$\mathbf{update}^{\Delta}$ is not a value, so (i) must hold. There are two possible reductions for update, but both result in an integer. By Typing Weakens Capability and the Value Typing Lemma an integer can be made to type check in the same updateability and capability environments as $\mathbf{update}^{\Delta}$.

The rest of the cases are similar.

□

**Theorem A.22** (Type Soundness)**.** *If* $\emptyset \vdash_{\mu} \Omega; H; e : \tau$ *then either*

*(i) there exists* $\Omega', H', e'$ *such that* $\Omega;; He \rightarrow \Omega'; H'; e'$ *and* $\emptyset \vdash_{\mu} \Omega'; H'; e' : \tau$ *or*

*(ii)* $e$ *is a value*

*Proof.* Suppose $\emptyset \vdash_{\mu} \Omega; H; e : \tau$ then by progress one of the following hold:

(a) there exists $\Omega', H', e'$ such that $\Omega; H; e \rightarrow \Omega'; H'; e'$ or

(b) $e$ is a value

Suppose (a) holds, then by preservation $\emptyset \vdash_{\mu} \Omega'; H'; e' : \tau$ and (i) holds. Suppose (c) holds, then (iii) holds. □

## A.2 Proof of Type Soundness for Proteus

The proof of soundness for Proteus follows an almost identical structure to the proof for Proteus$^{\Delta}$. In this section we give a sketch of the proof, paying particular attention when it differs from that of the corresponding Proteus$^{\Delta}$ proof.

We state the three main theorems whose proofs can be easily reconstructed by following the structure of those in Proteus$^{\Delta}$.

**Lemma A.23** (Progress)**.** *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\Omega, \Phi \vdash e : \tau$ *then either*

*(i) there exists* $\Omega', H', e'$ *such that* $\Omega; H; e \to \Omega'; H'; e'$ *or*

*(ii)* $e$ *is a value*

**Lemma A.24** (Preservation)**.** *If* $\emptyset \vdash \Omega; H; e : \tau$ *then*

*(i) if* $\Omega; H; e \to \Omega; H'; e'$ *then* $\emptyset \vdash \Omega; H'; e' : \tau$

*(ii) if* $\Omega; H; e \xrightarrow{\text{upd}} \Omega'; H'; e'$ *then* $\emptyset \vdash \Omega'; H'; e' : \tau$

**Theorem A.25** (Type Soundness)**.** *If* $\emptyset \vdash \Omega; H; e : \tau$ *then either*

*(i) there exists* $\Omega', H', e'$ *such that* $\Omega; ; He \to \Omega'; H'; e'$ *and* $\emptyset \vdash \Omega'; H'; e' : \tau$ *or*

*(ii)* $e$ *is a value*

Proteus' type system is a simplification of that of Proteus$^{\Delta}$. To obtain Proteus, the capabilities are removed, along with the subtype relation and the subsumption rule. Subtyping can be removed as its only function was to provide subtype-polymorphic behaviour for the capabilities on function arrows, which do not exist in the dynamic system.

We first give some properties of conFree[ ] and updateOK(), the proofs for which are simple inductions on the syntax of terms.

**Lemma A.26** (conFree$[-]$ Congruence)**.** *For any* upd *and* $e$*, if* conFree$[e]^{\text{upd}}$ *holds, then for any subterm* $e'$ *of* $e$ *we have* conFree$[e']^{\text{upd}}$*. We say that* conFree$[-]^{\text{upd}}$ *is congruent to the syntax of expressions.*

**Lemma A.27** (updateOK$(-)$ Congruence)**.** *For any* $\text{upd}, \Omega, H$ *and* $e$*, if* $\text{updateOK}(\text{upd}, \Omega, H, e)$ *holds then for any subterm* $e'$ *of* $e$ *we have* $\text{updateOK}(\text{upd}, \Omega, H, e')$*. We say that* $\text{updateOK}(-\text{upd}, \Omega, H, -)$ *is a congruent to the syntax of expressions.*

The only change to the dynamic semantics is in the definition of updateOK. We now give the cases of proofs that depend on updateOK.

**Lemma A.28** (Update Expression Safety)**.** *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\Omega, \Phi, \Gamma \vdash_{\mu} e : \tau$ *and* $\text{updateOK}(\text{upd}, \Omega, H, e)$ *then* $\mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}} \vdash_{\mu} \mathcal{U}[e]^{\text{upd}} : \tau$

*Proof.* The proof is by induction on the typing derivation of $e$ in a similar way to the Proteus$^{\Delta}$proof. We give the cases for concretization, abstraction and top-level variables variables:

**case** (EXPR.XVAR) **:**

By assumption $\Omega, \Phi, \Gamma \vdash \mathsf{z} : \tau$. Thus $\Phi(\mathsf{z}) = \tau$ and by (a) $\mathsf{z} \in \text{dom}(H)$.

By definition of $\mathcal{U}[-]^{\text{upd}}$ on expressions we have $\mathcal{U}[\mathsf{z}]^{\text{upd}} = \mathsf{z}$.

There are three ways in which $\mathcal{U}[\Omega, \Phi, \Gamma]^{\text{upd}}(\mathsf{z}) = \tau'$ can arise:

**case** $\mathsf{z} \in \mathrm{dom}(\mathrm{upd.AB})$ **:**

As $\mathsf{z} \in \mathrm{dom}(H)$ and by updateOK the domain of the heap and upd.AB are disjoint, we can conclude $\mathsf{z} \notin$ upd.AB, therefore this case cannot occur.

**case** $\mathsf{z} \in \mathrm{dom}(\mathrm{upd.UB})$ **:**

Let $\mathrm{upd.UB}(\mathsf{z}) = (\tau', b_v)$.

By definition of $\mathcal{U}[-]^{\mathrm{upd}}$ we have $\mathcal{U}[\Phi]^{\mathrm{upd}}(\mathsf{z}) = \mathrm{heapType}(\tau', b_v)$.

By updateOK assumption $\tau = \mathrm{heapType}(\tau', b_v)$.

By (A.EXPR.XVAR) $\mathcal{U}[\Omega, \Phi]^{\mathrm{upd}} \vdash \mathsf{z} : \tau$, as required.

**case** $\mathsf{z} \notin \mathrm{dom}(\mathrm{upd.UB})$ **:**

By definition of $\mathcal{U}[-]^{\mathrm{upd}}$ on contexts $\mathcal{U}[\Phi]^{\mathrm{upd}}(\mathsf{z}) = \tau$, thus $\mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash \mathsf{z} : \tau$, as required.

**case** (EXPR.ABS) **:**

By assumption

$$\frac{\Omega, \Phi, \Gamma \vdash e : \tau \qquad [\Omega, \Phi](\mathsf{t}) = \tau}{\Omega, \Phi, \Gamma \vdash \mathbf{abs}_{\mathsf{t}}\ e : \mathsf{t}}$$

Consider the form of upd.UN:

**case** $\mathsf{t} \notin \mathrm{dom}(\mathrm{upd.UN})$ **:**

By definition we have $\mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}}(\mathsf{t}) = \tau$ and $\mathcal{U}[\mathbf{abs}_{\mathsf{t}}\ e]^{\mathrm{upd}} = \mathbf{abs}_{\mathsf{t}}\ \mathcal{U}[e]^{\mathrm{upd}}$. The desired result follows by induction and (A.EXPR.ABS).

**case** $\mathrm{upd.UN}(\mathsf{t}) = (\tau'', \mathsf{c})$ **:**

Observe $\mathcal{U}[\mathbf{abs}_{\mathsf{t}}\ e]^{\mathrm{upd}} = \mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ \mathcal{U}[e]^{\mathrm{upd}})$. Using (A.EXPR.ABS) and (A.EXPR.APP) we are required to prove:

$$\frac{\begin{array}{c}\mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash \mathsf{c} : \tau' \to \tau''^{(a)} \\ \mathcal{U}[\Omega, \Phi, \Gamma]^{upd} \vdash \mathcal{U}[e]^{upd} : \tau'^{(b)}\end{array}}{\dfrac{\mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash \mathsf{c}\ \mathcal{U}[e]^{\mathrm{upd}} : \tau'' \qquad [\Omega, \Phi, \Gamma](\mathsf{t}) = \tau'^{(c)}}{\mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash \mathbf{abs}_{\mathsf{t}}\ (\mathsf{c}\ \mathcal{U}[e]^{\mathrm{upd}}) : \mathsf{t}}}$$

(b) holds by induction. To prove (a):

$$\begin{array}{ll}\mathcal{U}[\Omega, \mathrm{types}(H)]^{\mathrm{upd}} \vdash \mathsf{c} : \tau \to \tau' & \text{By updateOK() assumption} \\ \mathcal{U}[\Omega, \mathrm{types}(H), \Gamma]^{\mathrm{upd}} \vdash \mathsf{c} : \tau \to \tau' & \text{By Cap. Strengthening lemma} \\ \mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash \mathsf{c} : \tau \to \tau' & \text{By Ctx. Weakening lemma}\end{array}$$

Where the last step is valid because $\Omega; \Phi \vdash H$ and so $\mathrm{types}(H) \subseteq \Phi$.

By case split $\Omega = (\mathsf{t} = \tau, \Omega')$ for some $\Omega'$.

By definition of $\mathcal{U}[]$ $\mathcal{U}[\mathsf{t} = \tau, \Omega', \Phi, \Gamma]^{\mathrm{upd}} = \mathsf{t} = \tau'', \mathcal{U}[\Omega', \Phi, \Gamma]^{\mathrm{upd}}$, thus (c) holds.

**case** (EXPR.CON) **:**

Assume

$$\frac{\Omega(\mathsf{t}) = \tau \qquad \Omega, \Phi, \Gamma \vdash e : \mathsf{t}}{\Omega, \Phi, \Gamma \vdash \mathbf{con}_{\mathsf{t}}\ e : \tau} \tag{34}$$

$$\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \mathbf{con}_{\mathsf{t}}\ e) \tag{35}$$

Suffices to show that the leaves of this derivation hold:

$$\frac{\mathcal{U}[\Omega]^{\mathrm{upd}}(\mathsf{t}) = \tau^{(a)} \qquad \mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash \mathcal{U}[e]^{\mathrm{upd}} : \mathsf{t}^{(b)}}{\mathcal{U}[\Omega, \Phi, \Gamma]^{\mathrm{upd}} \vdash \mathbf{con}_{\mathsf{t}}\ \mathcal{U}[e]^{\mathrm{upd}} : \tau}$$

By updateOK assumption: conFree$[\![\, \mathbf{con}_{\mathsf{t}}\ e \,]\!]^{\mathrm{upd}}$, thus by definition of conFree, $\mathsf{t} \notin \mathrm{dom}(\mathrm{upd.UN})$. It follows by definition of $\mathcal{U}[\![-]\!]^{\mathrm{upd}}$ that (a) holds.
By Confree Congruence lemma conFree$[\![\, e \,]\!]^{\mathrm{upd}}$. It can now be shown by application of IH that (b) holds.

□

**Lemma A.29** (Heap Update Safety)**.** *If* $\vdash \Omega$ *and* $\Omega; \Phi \vdash H$ *and* $\mathrm{updateOK}(\mathrm{upd}, \Omega, H, e)$ *then* $\mathcal{U}[\![\Omega]\!]^{\mathrm{upd}}; \mathcal{U}[\![\Phi]\!]^{\mathrm{upd}} \vdash \mathcal{U}[\![H]\!]^{\mathrm{upd}}$

*Proof.* First note that from $\Omega; \Phi \vdash H$ we can deduce that for all $\rho \in \mathrm{dom}(H), \tau, e$

(a) $\mathrm{dom}(\Phi) = \mathrm{dom}(H)$
(b) if $\rho = \mathsf{z}$ and $H(\mathsf{z}) = (\tau, e)$ then $\Omega, \Phi \vdash e : \tau$ and $\Phi(\mathsf{z}) = \tau$ **ref**
(c) if $\rho = \mathsf{z}$ and $H(\mathsf{z}) = (\tau, \lambda(x).e\ )$ then $\Omega, \Phi \vdash \lambda(x).e\ : \tau$ and $\Phi(\mathsf{z}) = \tau$
(d) if $\rho = r$ and $H(r) = (\cdot, e)$ then there exists a $\tau$ such that $\Omega, \Phi \vdash e : \tau$ and $\Phi(r) = \tau$ **ref**

So assume (a)-(d) and also

$$\vdash \Omega \tag{36}$$

$$\mathrm{updateOK}(\mathrm{upd}, \Omega, H, \hat{\Delta}) \tag{37}$$

Via the same expansion we are required to prove for all $\rho \in \mathrm{dom}(\mathcal{U}[\![H]\!]^{\mathrm{upd}}), \tau, e$ that

(i) $\mathrm{dom}(\mathcal{U}[\![\Phi]\!]^{\mathrm{upd}}) = \mathrm{dom}(\mathcal{U}[\![H]\!]^{\mathrm{upd}})$
(ii) if $\rho = \mathsf{z}$ and $\mathcal{U}[\![H]\!]^{\mathrm{upd}}(\mathsf{z}) = (\tau, e)$ then $\mathcal{U}[\![\Omega, \Phi]\!]^{\mathrm{upd}} \vdash e : \tau$ and $\mathcal{U}[\![\Phi]\!]^{\mathrm{upd}}(\mathsf{z}) = \tau$ **ref**
(iii) if $\rho = \mathsf{z}$ and $\mathcal{U}[\![H]\!]^{\mathrm{upd}}(\mathsf{z}) = (\tau, \lambda(x).e\ )$ then $\mathcal{U}[\![\Omega, \Phi]\!]^{\mathrm{upd}} \vdash \lambda(x).e\ : \tau$ and $\mathcal{U}[\![\Phi]\!]^{\mathrm{upd}}(\mathsf{z}) = \tau$
(iv) if $\rho = r$ and $\mathcal{U}[\![H]\!]^{\mathrm{upd}}(r) = (\cdot, e)$ then there exists $\tau$ such that $\mathcal{U}[\![\Omega, \Phi]\!]^{\mathrm{upd}} \vdash e : \tau$ and $\mathcal{U}[\![\Phi]\!]^{\mathrm{upd}}(r) = \tau$ **ref**

hold. (a) implies (i) by inspection of the definition of $\mathcal{U}[\!]$ on contexts and heaps. We are left to show (ii)-(iv).

Observe that $\mathrm{types}(H) \subseteq \Phi$ because of (b) and (c).

Now consider the form of an arbitrary entry in $\mathcal{U}[\![H]\!]^{\mathrm{upd}}$:

**case** $r \mapsto (\cdot, e)$ **:**
This case is dealt with as in the Proteus$^{\Delta}$case.

**case** $\mathsf{z} \mapsto (\tau, b)$ **:**
In this case (iv) holds trivially and we are left to show (ii) and (iii).

**case** (ii) **:**
This case is dealt with as in the Proteus$^{\Delta}$case as it only relies on properties of updateOK common between the two definitions.

**case** (iii) **:**
Assume

$$\mathcal{U}[\![H]\!]^{\mathrm{upd}}(\mathsf{z}) = (\tau_1 \to \tau_2, \lambda(x).e\ )$$

i.e. that $b = \lambda(x).e$ and $\tau = \tau_1 \to \tau_2$. Prove

$$\mathcal{U}[\![\Omega, \Phi]\!]^{\mathrm{upd}} \vdash \mathcal{U}[\![\lambda(x).e\ ]\!]^{\mathrm{upd}} : \tau_1 \to \tau_2 \tag{38}$$

$$\mathcal{U}[\![\Omega, \Phi]\!]^{\mathrm{upd}}(\mathsf{z}) = \tau_1 \to \tau_2 \tag{39}$$

By definition of $\mathcal{U}[-]^{\text{upd}}$ on heaps, there are three ways to generate elements of $\mathcal{U}[H]^{\text{upd}}$.

**case** $\mathsf{z} \in \text{dom}(H)$ and $\mathsf{z} \in \text{dom}(\text{upd.UB})$ **:**

This case is dealt with as in the Proteus$^{\Delta}$case as it only relies on properties of updateOK common between the two definitions.

**case** $\mathsf{z} \in \text{dom}(H)$ and $\mathsf{z} \notin \text{dom}(\text{upd.UB})$ **:**

By case split and definition of $\mathcal{U}[-]^{\text{upd}}$ on heaps, there exists $b', H'$ such that $\mathcal{U}[\mathsf{z} \mapsto (\tau, b'), H']^{\text{upd}} = z \mapsto (\tau, \mathcal{U}[b']^{\text{upd}}), \mathcal{U}[H']^{\text{upd}}$ and $H = \mathsf{z} \mapsto (\tau, b'), H'$.

Because $b'$ is a function by case split, then by the definition of $\mathcal{U}[-]^{\text{upd}}$ on bindings, $\mathcal{U}[b']^{\text{upd}}$ is a function, say $b' = \lambda(x).e'$

By (c) and typing rules

$$\frac{\Omega, \Phi \vdash_{\mu} e' : \tau_2}{\Omega, \Phi \vdash \lambda(x).e' \ : \tau_1 \to \tau_2}$$

where $\tau = \tau_1 \to \tau_2$.

Required to prove 38 and 39.

By 37 $\text{conFree}[\,H\,]^{\text{upd}}$, therefore $\text{conFree}[\,e\,]^{\text{upd}}$. By UpdateOK Congruence lemma $\text{updateOK}(\text{upd}, \Omega, H, e)$.

By Update Expression Safety lemma $\mathcal{U}[\Omega, \Phi]^{\text{upd}} \vdash_{\mu} \mathcal{U}[e]^{\text{upd}} : \tau_2$. Therefore, by use of (A.BIND.FUN), 38 holds.

By the definition of $\mathcal{U}[-]^{\text{upd}}$ on contexts it follows that $\mathcal{U}[\Phi]^{\text{upd}}(z) = \tau$ making 39 holds, as required.

**case** $\mathsf{z} \notin \text{dom}(H)$ **:**

The result follows similarly to this subcase in case (ii).

□

**Lemma A.30** ($\mathcal{U}[\,-\,]^{-}$ preserves type-safety)**.** *Given* $\vdash\ \Omega; H; e$ *and an update,* upd*, for which we have* $\text{updateOK}(\text{upd}, \Omega, H, e)$*, then* $\vdash\ \mathcal{U}[\,\Omega\,]^{\text{upd}}; \mathcal{U}[\,H\,]^{\text{upd}}, \mathcal{U}[\,e\,]^{\text{upd}} : \tau$*.*

*Proof.* Follows (Heap Update Safety) and (Update Expression Safety) and an application of the configuration typing rule. □

## REFERENCES


Ajmani, S. 2004. Automatic software upgrades for distributed systems. Ph.D. thesis, Laboratory of Computer Science, the Massachussetts Institute of Technology.

Ajmani, S., Liskov, B., and Shrira, L. 2006. Modular software upgrades for distributed systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

Altekar, G., Bagrak, I., Burstein, P., and Schultz, A. 2005. OPUS: Online patches and updates for security. In *Proceedings of the Fourteenth USENIX Security Symposium*. Baltimore, Maryland, USA, 287–302.

Appel, A. 1994. Hot-Sliding in ML. Unpublished manuscript.

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

Ball, T. and Rajamani, S. K. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL)*. Portland, Oregon, 1–3.

Baumann, A., Heiser, G., Appavoo, J., Silva, D. D., Krieger, O., , Wisniewski, R. W., and Kerr, J. 2005. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference*.

Bierman, G., Hicks, M., Sewell, P., and Stoyle, G. 2003. Formalizing dynamic software updating. In *Proceedings of USE 2003: the Second International Workshop on Unanticipated Software Evolution (Warsaw)*.

Bierman, G., Hicks, M., Sewell, P., Stoyle, G., and Wansbrough, K. 2003. Dynamic rebinding for marshalling and update with destruct-time $\lambda$. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*.

Bloom, T. 1983. Dynamic Module Replacement in a Distributed Programming System. Ph.D. thesis, Laboratory for Computer Science, The Massachussets Institute of Technology.

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

Boyapati, C., Liskov, B., Shrira, L., Moh, C.-H., and Richman, S. 2003. Lazy modular upgrades in persistent object stores. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

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

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

Drossopoulou, S. and Eisenbach, S. 2003. Flexible, source level dynamic linking and re-linking. In *Proceedings of the ECOOP 2003 Workshop on Formal Techniques for Java Programs*.

Duggan, D. 2001. Type-based hot swapping of running modules. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*.

Foster, J. S., Terauchi, T., and Aiken, A. 2002. Flow-sensitive type qualifiers. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*. Berlin, Germany, 1–12.

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

Gapeyev, V., Levin, M., and Pierce, B. C. 2000. Recursive subtyping revealed. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*.

Gilmore, S., Kirli, D., and Walton, C. 1997. Dynamic ML without dynamic types. Tech. Rep. ECS-LFCS-97-378, LFCS, University of Edinburgh. December.

Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., and Cheney, J. 2002. Region-based memory management in Cyclone. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*.

Gupta, D. 1994. On-line software version change. Ph.D. thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur.

Heintze, N. 1992. Set-based program analysis. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University.

Hicks, M. and Nettles, S. M. 2005. Dynamic software updating. *Transactions on Programming Languages and Systems 27,* 6 (November).

Hicks, M., Tse, S., Hicks, B., and Zdancewic, S. 2005. Dynamic updating of information-flow policies. In *Proceedings of the International Workshop on Foundations of Computer Security (FCS)*.

Hicks, M., Weirich, S., and Crary, K. 2000. Safe and flexible dynamic linking of native code. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC)*, R. Harper, Ed. Lecture Notes in Computer Science, vol. 2071. Springer-Verlag.

Hicks, M. W. 2001. Dynamic software updating. Ph.D. thesis, Department of Computer and Information Science, The University of Pennsylvania.

Hjálmtýsson, G. and Gray, R. 1998. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proceedings of the USENIX*.

Mitchell, J. C. 1986. Representation independence and data abstraction. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 263–276.

Neamtiu, I., Foster, J. S., and Hicks, M. 2005. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*.

Neamtiu, I., Hicks, M., Stoyle, G., and Oriol, M. 2006. Practical dynamic software updating for C. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*. 72–83.

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

Oppenheimer, D., Brown, A., Beck, J., Hettena, D., Kuroda, J., Treuhaft, N., Patterson, D. A., and Yelick, K. 2002. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Comput. 51,* 2, 100–107.

Orso, A., Rao, A., and Harrold, M. 2002. A technique for dynamic updating of Java software. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*.

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

Soules, C., Appavoo, J., Hui, K., Wisniewski, R. W., Silva, D. D., Ganger, G. R., Krieger, O., Stumm, M., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. 2003. System support for online reconfiguration. In *Proceedings of the USENIX Annual Technical Conference*.

Stoyle, G., Hicks, M., Bierman, G., Sewell, P., and Neamtiu, I. 2005. *Mutatis Mutandis*: Safe and predictable dynamic software updating. In *Proceedings of POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Long Beach)*. 183–194.

Walker, D. 2000. A type system for expressive security policies. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 254–267.

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

Xie, Y. and Aiken, A. 2005. Scalable Error Detection using Boolean Satisfiability. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 351–363.

Zorn, B. 2005. Personal communication, based on experience with Microsoft Windows customers.