ABSTRACT

Title of dissertation:	PRACTICAL DYNAMIC SOFTWARE UPDATING
	Iulian Gheorghe Neamtiu Doctor of Philosophy, 2008
Dissertation directed by:	Professor Michael Hicks Department of Computer Science

This dissertation makes the case that programs can be updated while they run, with modest programmer effort, while providing certain update safety guarantees, and without imposing a significant performance overhead.

Few systems are designed with on-the-fly updating in mind. Those systems that permit it support only a very limited class of updates, and generally provide no guarantees that following the update, the system will behave as intended. We tackle the on-the-fly updating problem using a compiler-based approach called *dynamic software updating* (DSU), in which a program is patched with new code and data while it runs. The challenge is in making DSU *practical*: it should support changes to programs as they occur in practice, yet be safe, easy to use, and not impose a large overhead.

This dissertation makes both theoretical contributions—formalisms for reasoning about, and ensuring update safety—and practical contributions—Ginseng, a DSU implementation for C. Ginseng supports a broad range of changes to C programs, and performs a suite of safety analyses to ensure certain update safety properties. We performed a substantial study of using Ginseng to dynamically update six sizable C server programs, three single-threaded and three multi-threaded. The updates were derived from changes over long periods of time, ranging from 10 months to 4 years-worth of releases. Though the programs changed substantially, the updates were straightforward to generate, and performance measurements show that the overhead of Ginseng is detectable, but modest.

In summary, this dissertation shows that DSU can be practical for updating realistic applications as they are written now, and as they evolve in practice.

PRACTICAL DYNAMIC SOFTWARE UPDATING

by

Iulian Gheorghe Neamtiu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Părinţilor mei,

Elisabeta şi Gheorghe

Acknowledgments

I am deeply indebted to my advisor, Mike Hicks, for his relentless help and guidance, academic and otherwise, that have made this work possible. I am grateful to Mike for the late hours we spent working together prior to paper deadlines, for his patience in showing me how to write a type system and prove it correct, and for a million other things. But apart from teaching me the "skills of the trade" for doing research, Mike has taught me something equally important: that I could achieve what at first seemed impossible, if I was willing to dedicate myself to it.

I want to thank Jeff Foster for his help and advice, and for teaching an excellent class on Program Analysis and Understanding in Fall 2003 that opened my eyes to the rigor and elegance of Programming Languages research.

My first advisor, Liviu Iftode, got me started on conducting research very early in my Ph.D. program. Working with Liviu and his students at Rutgers University, Florin Sultan and Aniruddha Bohra, put me on the right track for my dynamic updating work.

I also thank the other members of my Ph.D committee, Jeff Hollingsworth, Bruce Jacob and Neil Spring, for taking time to go over my dissertation and providing feedback and suggestions.

I was fortunate to collaborate with Gareth Stoyle, Gavin Bierman, Peter Sewell, Manuel Oriol and Polyvios Pratikakis on both theoretical and practical aspects of this work. This dissertation has benefited greatly from our joint efforts.

The Programming Languages group at Maryland was a fun and productive environment to work in. Nikhil Swamy, Mike Furr, David Greenfieldboyce, Eric

Hardisty, Chris Hayden, Pavlos Papageorgiou, Nick Petroni, Khoo-Yit Phang, and Saurabh Srivastava were always willing to help whenever I needed suggestions, ideas, or feedback.

Finally, I thank my family for their unwavering support that allowed me to start a Ph.D. and carry it out to completion. My wife Monica brought much needed sunshine in my graduate student life: she put up with my moods when I was stressed, stood by my side, and provided comfort and encouragement. She did all these while being in a similarly stressful position herself: first a Ph.D. student, and, later on, an assistant professor. My parents Elisabeta and Gheorghe brought me up to respect and value education, and to strive for thoroughness. They have spared no effort in helping me succeed, and this dissertation is a small tribute to their endeavors.

Table of Contents

List of Figures

List of Abbreviations

AST	Abstract Syntax Tree
AVA	Abstraction-Violating Alias
DSU	Dynamic Software Updating
FTP	File Transfer Protocol
LOC	Lines Of Code
OS	Operating System
SSH	Secure SHell
TVC	Transactional Version Consistency
VC	Version Consistency
VM	Virtual Machine

Chapter 1

Overview

1.1 Motivation

Continuous operation is a requirement of many of today's computer systems. Examples range from pacemakers to cell phone base stations to nuclear power plant monitors. For ISPs, credit card providers, brokerages, and on-line stores, being available 24/7 is synonymous with staying in business: an hour of downtime can cost hundreds of thousands, or even millions of dollars [84, 90, 34], and longer downtimes put companies at increasingly higher risk. Despite the requirement that these systems "run forever," they must be updated to fix bugs and add new features.

The most common update method today, from data centers to desktops to sensor networks, is to stop the system, install the update and restart at the new version. For example, one study [65] found that 75% of nearly 6,000 outages of high-availability applications were planned-for hardware and software maintenance. Another example is critical updates to Windows Vista, where the update is considered so important that the operating system decides to apply the updates and reboot without giving the user the option to postpone installing the updates [13].

Internet access is quickly becoming ubiquitous, so more software vendors release their operating system or application patches online. Unfortunately, this leads to more frequent application restarts, or, when updating the OS, to more frequent reboots. To make a bad situation worse, experts suggest even higher patch release frequencies are needed, to reduce application and OS vulnerability [45].

As expected, this increase in patch release frequency, and hence restarts, is problematic. In a large enterprise, reboots can have a large administrative cost [116]. For embedded systems involved in mission-critical or medical applications, reboots are intolerable. For system administrators and end users, patches and reboots are burdensome: both categories are slow in applying patches because patches are disruptive and might introduce new bugs [97].

To fix these problems, we need to support *on-line updates*, i.e., applying software updates without having to restart or reboot. In prior work, many researchers have proposed variations of an approach to supporting on-line updates called *Dynamic Software Updating* (DSU). In this approach, a running program is patched with new code and data on-the-fly, while it runs.

This dissertation tackles the on-line updating problem using a fine-grained, compiler-based DSU approach. We compile programs specially so that they can be dynamically patched, generate most of a dynamic patch automatically, and finally, load the dynamic patch into the running program.

DSU is appealing because of its generality: in principle any program can be updated in a fine-grained way, without the need for redundant hardware or special-purpose software architectures. The challenge is in making DSU *practical*: it should be flexible and yet safe, efficient, and easy to use. We now describe each of these properties in detail.

1. *Flexibility.* DSU should be applicable to a broad range of applications (irrespective of abstraction level, software architecture, level of concurrency), and permit arbitrary updates to applications (since the form of future updates cannot be predicted).

2. *Efficiency.* Applications should require few modifications to support DSU, patches should be easy to write, and updateability should not degrade application performance.

3. *Safety.* DSU should provide safety guarantees that give application developers (and patch developers) assurances that following the update, the program will behave as intended.

Unfortunately, DSU systems presented in prior work fail to address one or more of these requirements. Many systems do not support all of the software changes as they appear in practice, i.e., are not flexible. Other systems are flexible, but provide no update safety guarantees.

To address these problems, we have built Ginseng, a new DSU system for C programs that aims to support most changes that appear in practice, and to satisfy the three practicality criteria laid out above. We have chosen C because it is a very popular language in the construction of long-running software. The kernels of Linux and the BSD OS family are written in C. A survey on safety-critical software [98] used in aerospace, transportation, medical and energy systems finds Ada, followed by assembler, C and C++ to be the predominant programming languages used to construct such systems. Popular long-running Internet servers such as BIND, Apache, Sendmail, and OpenSSH are also written in C, motivating our decision to pursue C as the target language for our DSU system.

A DSU system must support the kinds of software changes that typically occur between releases. To find out how programs typically change, we studied the source code evolution of some long-running C programs. We built a tool named ASTdiff that parses two versions of a program, compares their abstract syntax trees, and reports the differences. We used ASTdiff to compare versions of several large C programs (BIND, OpenSSH, Apache, Vsftpd, GNU Zebra and the Linux kernel) spanning several months to several years. We describe ASTdiff and our findings in Chapter 2. The results of the study show that, to enable long-term evolution, a DSU system must support *addition* of new definitions (functions, data, or types), the *replacement* of existing definitions (data or functions), and *changes to types* (data representations, function signatures, and types of global variables). Similar studies on the Linux kernel [89] and several substantial Java applications [28] show that changes to function signatures and class interfaces are part of software evolution for all programs analyzed.

A large number of compiler- or library-based DSU systems have been developed for C [42, 47, 20, 6], C++ [52, 60], Java [17, 86, 31, 70], and functional languages like ML [32, 43] and Erlang [8]. Many do not support all of the changes needed to make dynamic updates in practice. For example, updates cannot change type definitions or function prototypes [86, 31, 52, 60, 6], or else only permit such changes for abstract types or encapsulated objects [60, 43]. In many cases, updates to active code (e.g., long-running loops) are disallowed [43, 70, 42, 47, 60], and data stored

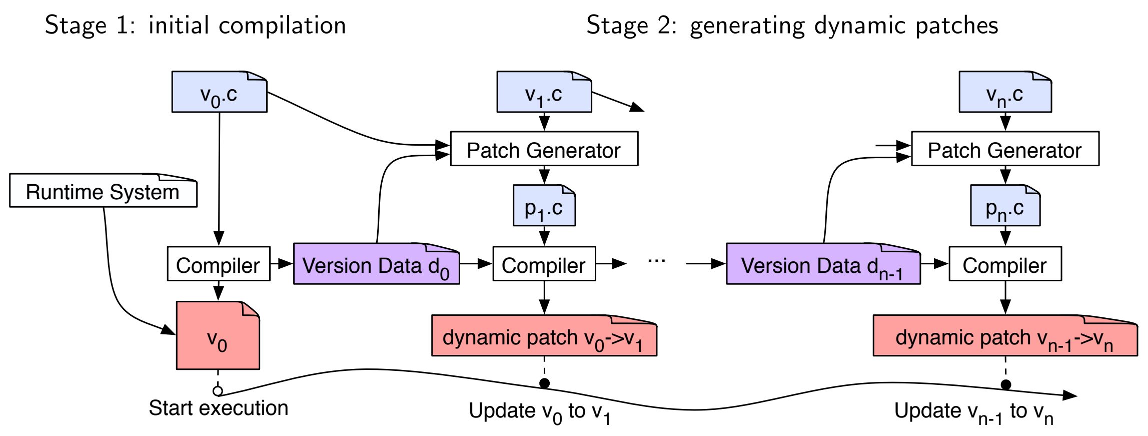

Figure 1.1: Building and dynamically updating software with Ginseng. In Stage 1, Ginseng compiles a C program into an updateable application. In Stage 2 and later, dynamic patches are generated and loaded into the application.

in local variables may not be transformed [50, 47, 42, 52]. Recent systems are more flexible, and support such changes [23, 24, 68], but provide no safety guarantees.

1.2 Ginseng

Ginseng is a compiler and tool suite for constructing updateable applications from C programs. Using Ginseng, we compile programs specially so that they can be dynamically patched, and generate most of a dynamic patch automatically. Ginseng performs a series of analyses that when combined with runtime support ensure that an update will not violate certain safety properties, while guaranteeing that data is kept up-to-date. We now proceed to presenting a high-level overview of our approach.

Ginseng consists of a *compiler*, a *patch generator* and a *runtime system* for building updateable software. The compiler and patch generator are written in Objective Caml using the CIL framework [80]. The runtime system is a library written in C.

Basic usage is illustrated in Figure 1.1, with Ginseng components in white boxes. There are two stages. First, for the initial version of a program, $\mathtt{v_0.c}$, the compiler generates an updateable executable $\mathtt{v_0}$, along with some type and analysis information (`Version Data` $\mathtt{d_0}$). The executable is then deployed. Second, when the program has changed to a new version ($\mathtt{v_1.c}$), the developer provides the new and old code to the patch generator to generate a patch $\mathtt{p_1.c}$ representing the differences. This is passed to the compiler along with the current version information, and turned into a `dynamic patch` $\mathtt{v_0} \rightarrow \mathtt{v_1}$. The runtime system links the dynamic patch into the running program, completing the on-line update. This process continues for each subsequent program version.

1.2.1 Ginseng Compiler

The Ginseng compiler has two responsibilities: 1) it compiles programs to be dynamically updateable, and 2) it applies static analyses to ensure updates are safe even when type definitions change. We describe each of these in turn.

Compilation Techniques. The Ginseng compiler transforms an input C program so that existing functions will call replacement functions present in a dynamic patch, and data is converted to the latest representation whenever data types change.

The technique for updating functions is called *function indirection*; it permits old code to call new function versions by introducing a level of indirection (via a global variable) between a caller and the called function. To update a function to its new version, the runtime system dynamically loads the new function version and sets the indirection variable to the new function, so new calls go to the new function version.

Ginseng also must permit transformations to the state of the program, so the state is compatible with the new code. For this, Ginseng uses a technique called *type wrappers*: each definition of a named type `T` is converted into a "wrapped" version `wT` whose size is larger and allows room for future growth. When an update changes the definition of `T` in the original program, existing values of type `wT` in the compiled program must be transformed to have the new type's representation, to be compatible with the new code. This is done via a function called *type transformer*. For example, if the old definition of `T` is `struct { int x;}` and the new definition is `struct { int x; int y;}`, the type transformer's job is to copy the old value of `x` and initialize `y` to a default value. Code is compiled to notice when a typed value is out of date, and if so, to apply the necessary type transformer.

Safety Analyses. Ginseng combines static analysis with runtime support to ensure that updates are always *type-safe*, even when changes are made to function prototypes or type definitions. While supporting the addition of new definitions, or the replacement of data and functions at the same type, is relatively straightforward, supporting changes to types is challenging: if the old and new programs assume different representations for a certain type, then old code accessing new data, or new code accessing old data, leads to a *representation inconsistency*, i.e., a violation of type safety. To illustrate this, consider the following simple program; the old version is on the left and the new program version is on the right. The update changes the signature of foo to accept two arguments instead of one.

```
1  void foo (int i) { ... }
2  void bar () {
3    int i;
4     ...
5    foo(i);
6     ...
7  }
```

```
1  void foo (int i, int j) { ... }
2  void bar () {
3    int i,j;
4     ...
5    foo(i,j);
6     ...
7  }
```

Suppose the update is applied when the old program's execution reaches line 4. The new version of foo is loaded, and the call on line 5 will invoke the new version, passing it one argument, i. But this is incorrect, since the new version of foo expects two arguments. The correct thing to do is to postpone the update until after the call to foo. Ginseng performs two safety analyses (updateability analysis and abstraction-violating alias analysis) to ensure an update will not lead to such type safety violations, while guaranteeing that data is kept up-to-date. The basic idea is to examine the program to discover assumptions made about the types of updateable entities (i.e., functions or data) in the continuation of each program point. These assumptions become constraints on the timing of updates (Section 3.3 discusses the implementation of theses analyses).

This is in contrast to previous approaches that focus on the updating mechanism, rather than update safety, and as a consequence, support only limited-scale updates, or provide no safety guarantees.

1.2.2 Patch Generator

Another key factor in enabling dynamic updates to realistic programs is the ability to construct a dynamic patch automatically. The Ginseng patch generator (Section 3.4) has two responsibilities. First, it identifies those definitions (global variables, functions, or types) that have changed between versions. Second, for each type definition that has changed, it generates a type transformer function used to convert values from a type's old representation to the new one. The compiler inserts code so that the program will make use of these functions following a dynamic patch. If the new code assumes an invariant about global state (e.g., certain files are open, certain threads are started, or a list is doubly-linked), this invariant has to hold after the update takes place. Users can write *state transformer* functions that are run at update time to convert state and run initialization code for new features, as necessary. Users also may adjust the generated type transformers as necessary.

1.2.3 Runtime System and Update Points

The dynamic update itself is carried out by the Ginseng runtime system (Section 3.4), which is linked into the updateable program. Once notified, the runtime system will cause a dynamic patch to be dynamically loaded and linked at the next safe *update point*. An update point is essentially a call to a run-time system function `DSU_update()`. Update points can be inserted manually, by the programmer, or automatically, by the compiler. Our safety analyses will annotate these points with constraints as to how definitions are allowed to change at each particular point. The runtime system will check that these constraints are satisfied by the current update, and if so, it "glues" the dynamic patch into the running program. In our experience, finding suitable update points in long-lived server programs is quite straightforward, and the analysis provides useful feedback as to whether the chosen spots are free from restrictions. Sections 3.2, 3.3, and 3.4 describe these features of Ginseng in detail.

A practical DSU system must strive to provide strong update safety guarantees without affecting update availability (the time from when an update becomes available to when it is applied).

Long-running programs amenable to dynamic updating are usually structured around event processing loops, where one loop iteration handles one event. For the single-threaded programs we have updated, we placed update points (calls to `DSU_update`) manually, at the completion of a top-level event-handling loop. While the manual enumeration of a few update points works well for single-threaded programs, in a multi-threaded program, an update can only be applied when *all* threads have reached a safe update point. Since this situation is unlikely to happen naturally, we could imagine interpreting each occurrence of `DSU_update()` as part of a *barrier*—when a thread reaches a safe update point, it blocks until all other threads have done likewise, and the last thread to reach the barrier applies the update and releases the blocked threads.

Unfortunately, because all threads must reach safe points, this approach may fail to apply an update in a timely fashion. Therefore, we must allow updates in the middle of the loop while still ensuring update safety.

1.2.4 Version Consistency

Performing an update in the middle of a loop can potentially lead to problems, even if the update is type-safe, because the update violates what we call *version consistency*: when programmers write the event processing code they assume the loop body will execute code belonging to the same version. An update could violate that assumption.

We solved this problem by allowing programmers to designate blocks of code as *transactions* whose execution must always be attributable to a single program version. An example of a transaction would be a loop iteration, which corresponds to processing an event. In Chapter 5 we present a formalism called *contextual effects* that can be used to reason about the past and future computation at each program point. Using a static analysis based on contextual effects we can enforce version consistency even when an update is performed inside a transaction. Version consistency is a desirable property, but many systems designed to support long-term evolution [103, 12, 11, 6, 1, 23, 24] do not implement it.

Ginseng provides multi-threaded DSU support that is as flexible and safe as the single-threaded approach, while ensuring updates can be applied in a timely fashion. A key concept introduced in this dissertation, explained in Chapter 4, is that of *induced update points*. Induced update points helps us accomplish our goal of balancing safety and availability. We allow programmers to designate update points in multi-threaded programs where global state is consistent, and writing an update is straightforward, as the global invariants hold at those points. The code between two update points constitutes a transaction. The update, however, can take place in between programmer-specified update points, at an induced update point. Our system enforces that an update *appears* to execute at an update point: if a code update takes place in between two update points, the execution trace can be attributed to exactly one program version. In other words, an update can be applied in the middle of a transaction, but the execution of a transaction is still attributable to a single program version. This flexibility is crucial in being able to update multi-threaded programs in a timely manner, without requiring all threads to reach a programmer-inserted update point simultaneously.

1.3 Evaluation

Ginseng's support for a broad range of changes to programs, along with safety and automation, has enabled us to implement long-term updates to single- and multi-threaded programs. We updated three open-source, single-threaded server programs with three to four years' worth of releases: Vsftpd (the Very Secure FTP daemon), the Sshd daemon from the OpenSSH suite, and the Zebra server from the GNU Zebra routing software package, for a total of 27 updates (Chapter 3). We were also able to perform type-safe, version-consistent updates to three multi-threaded programs: the Icecast streaming server, Memcached (a distributed memory object caching system) and the Space Tyrant game server. We considered one year worth of releases for Icecast and Space Tyrant, and ten months for Memcached, for a total of 13 updates (Chapter 4).

Though these programs were not designed with updating in mind, we had to make only a handful of changes to their source code to make them safely updateable. Each dynamic update we performed was based on an actual release, and for each application, we applied updates corresponding to up to four years' worth of releases, totaling as many as twelve different patches in one case. To achieve these results, we developed several new implementation techniques, including new ways to handle the transformation of data whose type changes, to allow dynamic updates to infinite loops and active code, and to allow updates to take effect in programs with function pointers. Details are in Sections 3.2 and 4.3.1. Overhead due to updating is modest: application performance usually degrades by 0–10%, though for one of the programs, the overhead was 32%. Memory footprint for updateable applications is 0–10% larger, compared to unmodified applications, except for one application, where it is 46%.

The updates we performed to the six servers present a substantial demonstration that DSU can be practical: it can support on-line updates over a long period based on actual releases of real-world programs. These servers are similar in that they keep long-lived, in-memory state, and rebooting the server is disruptive for the clients. However, they only constitute one category of long-running programs. Other long-running systems keep short-lived in-memory state (e.g., web servers), or store their state on disk (e.g., database systems). In Sections 7.1 and 7.2 we talk about how DSU would apply to these other categories of systems, and what are the trade-offs between using DSU and using traditional high-availability techniques.

1.4 Contributions

Based on our experience, we believe Ginseng makes significant headway toward meeting the DSU practicality criteria we have set forth above:

- *Flexibility.* Ginseng permits updates to single- and multi-threaded C programs. The six test programs are realistic, substantial and most of them are widely used in constructing real-world Internet services. Ginseng supports changes to functions, types, and global variables, and as a result we could perform all the updates in the 10 months–4 years time frame we considered. Patches were based on actual releases, even though the developers made changes without having dynamic updating in mind.

- *Efficiency.* We had to make very few changes to the application source code. Despite the fact that differences between releases were non-trivial, generating and testing patches was relatively straightforward. We developed tools to generate most of a dynamic patch automatically by comparing two program versions, reducing programmer work. We found that DSU overhead is modest for I/O bound applications, but more pronounced for CPU-bound applications. Our novel version consistency property improves update availability, resulting in a smaller delay between the moment an update is available and the moment the update is applied.

- *Safety.* Updates cannot be applied at arbitrary points during a program's execution, because that could lead to safety violations. Ginseng performs a suite of static safety analyses to determine times during the running program's execution at which an update can be performed safely.

In summary, this dissertation makes the following contributions:

1. A practical framework to support dynamic updates to single- and multi-threaded C programs. Ours is the most flexible, and arguably the most safe, implementation of a DSU system to date.

2. A substantial study of the application of our system to six sizable C server programs, three single-threaded, and three multi-threaded, over long periods of time ranging from 10 months to 4 years worth of releases.

3. A novel type-theoretical system that generalizes standard effect systems, called *contextual effects*; contextual effects are useful when the past or future computation of the program is relevant at various program points, and have applications beyond DSU. We also present a formalism and soundness proof for our novel update correctness property, version consistency, which permits us to provide certain update safety guarantees for single- and multi-threaded programs

4. An approach for comparing the source code of different versions of a C program, as well as a software evolution study of various versions of popular open source programs, including BIND, OpenSSH, Apache, Vsftpd and the Linux kernel.

Chapter 2

Software Evolution

To effectively support dynamic updating, we first need to understand how software evolves. This chapter presents an approach to characterizing the evolution of C programs, along with a study that analyzes how several substantial open-source programs have changed over years-worth of releases.

2.1 Introduction

We have developed a tool called ASTdiff that can quickly compute and summarize simple changes to successive versions of C programs by partially matching their abstract syntax trees. ASTdiff identifies the changes, additions, and deletions of global variables, types, and functions, and uses this information to report a variety of statistics. The Ginseng patch generator uses the ASTdiff output to determine the contents of a dynamic patch.

Our approach is based on the observation that for C programs, function names are relatively stable over time. We analyze the bodies of functions of the same name and match their abstract syntax trees structurally. During this process, we compute a bijection between type and variable names in the two program versions, which will help us determine changes to types and variables. If the old and new ASTs fail to match (modulo name changes), we consider this a change to that function's body,

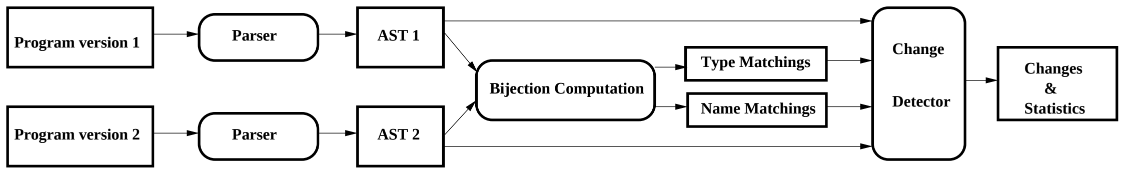

Figure 2.1: High level view of ASTdiff.

and will replace the entire function at the next update.

We have used ASTdiff to study the evolution history of a variety of popular open source programs, including Apache, Sshd, Vsftpd, Bind, and the Linux kernel. This study has revealed trends that we have used to inform our design for DSU. In particular, we observed that function, type and global variable additions are far more frequent than deletions. We also found that function bodies change frequently over time; function prototypes change as well, but not as frequently as function bodies do. Finally, type definitions (such as **struct** and **union** declarations) do change, but infrequently, and often in simple ways.

2.2 Approach

Figure 2.1 provides an overview of ASTdiff. We begin by parsing the two program versions to produce abstract syntax trees (ASTs), which we traverse in parallel to collect type and name mappings; these mappings will help us avoid reporting spurious changes due to renamings. With the mappings at hand, we detect and collect changes to report to the user, either directly or in summary form. In this section, we describe the matching algorithm, illustrate how changes are detected and reported, and describe our implementation and its performance.

```c
typedef int sz_t;

int count;

struct foo {
  int i;
  float f;
  char c;
};

int baz(int a, int b) {
  struct foo sf;
  sz_t c = 2;
  sf.i = a + b + c;
  count++;
}

```

Version 1

```c
typedef int size_t;

int counter;

struct bar {
  int i;
  float f;
  char c;
};

int baz(int d, int e) {
  struct bar sb;
  size_t g = 2;
  sb.i = d + e + g;
  counter++;
}
void biff(void) { }
```

Version 2

Figure 2.2: Two successive program versions.

2.2.1 AST Matching

Figure 2.2 presents an example of two successive versions of a program. Assuming the example on the left is the initial version, ASTdiff discovers that the body of baz is unchanged—which is what we would like, because even though every line has been syntactically modified, the function in fact is structurally the same, and produces the same output. ASTdiff also determines that the type sz_t has been renamed size_t, the global variable count has been renamed counter, the structure foo has been renamed bar, and the function biff has been added.

To report these results, ASTdiff must find a mapping between the old and new names in the program, even though functions and type declarations have been reordered and modified. To do this, ASTdiff begins by finding function names that are common between program versions; our assumption is that function names do

```
procedure GenerateMaps($Version1, Version2$)
 $F_1 \leftarrow$ set of all functions in Version 1
 $F_2 \leftarrow$ set of all functions in Version 2
 global $TypeMap \leftarrow \emptyset$
 global $GlobalNameMap \leftarrow \emptyset$
 for each function $f \in F_1 \cap F_2$
   do { $AST_1 \leftarrow$ AST of $f$ in Version 1
        $AST_2 \leftarrow$ AST of $f$ in Version 2
        Match_Ast($AST_1, AST_2$)

procedure Match_Ast($AST_1, AST_2$)
 local $LocalNameMap \leftarrow \emptyset$
 for each $(node_1, node_2) \in (AST_1, AST_2)$
   do { if $(node_1, node_2) = (t_1\ x_1, t_2\ x_2)$  // declaration
          then { $TypeMap \leftarrow TypeMap \cup \{t_1 \leftrightarrow t_2\}$
                 $LocalNameMap \leftarrow LocalNameMap \cup \{x_1 \leftrightarrow x_2\}$
          else if $(node_1, node_2) = (y_1 := e_1\ op\ e'_1, y_2 := e_2\ op\ e'_2)$  // assignment
          then { Match_Ast($e_1, e_2$)
                 Match_Ast($e'_1, e'_2$)
                 if $isLocal(y_1)$ and $isLocal(y_2)$ then
                   $LocalNameMap \leftarrow LocalNameMap \cup \{y_1 \leftrightarrow y_2\}$
                   else if $isGlobal(y_1)$ and $isGlobal(y_2)$ then
                   $GlobalNameMap \leftarrow GlobalNameMap \cup \{y_1 \leftrightarrow y_2\}$
          else if ... // other syntactic forms
          else break
```

Figure 2.3: Map generation algorithm.

not change very often. ASTdiff then tries to match function bodies corresponding to the same function name in the old and new versions. The function body match helps us construct a *bijection* (i.e., a one-to-one, onto mapping) between names in the old and new versions.

We traverse the ASTs of the function bodies of the old and new versions simultaneously, adding entries to a *LocalNameMap* and a *GlobalNameMap* that map local variable names and global variable names, respectively. Two variables are considered equal if we encounter them in the same syntactic position in the two function bodies. For example, in Figure 2.2, parallel traversal of the two versions of baz results in the *LocalNameMap*:

$$\texttt{a} \leftrightarrow \texttt{d}, \texttt{b} \leftrightarrow \texttt{e}, \texttt{sf} \leftrightarrow \texttt{sb}, \texttt{c} \leftrightarrow \texttt{g}$$

and a *GlobalNameMap* with `count` $\leftrightarrow$ `counter`. Similarly, we form a *TypeMap* between named types (**typedef**s and aggregates) that are used in the same syntactic positions in the two function bodies. For example, in Figure 2.2, the name map pair `sb` $\leftrightarrow$ `sf` will introduce a type map pair `struct foo` $\leftrightarrow$ `struct bar`.

We define a *renaming* to be a name or type pair $j_1 \rightarrow j_2$ where $j_1 \leftrightarrow j_2$ exists in the bijection, j_1 does not exist in the new version, and j_2 does not exist in the old version. Based on this definition, ASTdiff will report `count` $\rightarrow$ `counter` and `structfoo` $\rightarrow$ `structbar` as renamings, rather than additions and deletions. This approach ensures that consistent renamings are not presented as changes, and that type changes are decoupled from value changes, which helps us better understand how types and values evolve.

Figure 2.3 presents the pseudocode for our algorithm. We accumulate global maps *TypeMap* and *GlobalNameMap*, as well as a *LocalNameMap* per function body. We invoke the routine Match_Ast on each function common to the two versions. When we encounter a node with a declaration $t_1\ x_1$ (a declaration of variable x_1 with type t_1) in one AST and $t_2\ x_2$ in the other AST, we require $x_1 \leftrightarrow x_2$ and $t_1 \leftrightarrow t_2$. Similarly, when matching statements, for variables y_1 and y_2 occurring in the same syntactic position we add type pairs in the *TypeMap*, as well as name pairs into *LocalNameMap* or *GlobalNameMap*, depending on the storage class of y_1 and y_2.

```
------- Global Variables  ----------
Version1 :                        1
Version2 :                        1
renamed :                         1

------- Functions  -----------------
Version1 :                        1
Version2 :                        2
added :                           1
locals/formals name changes :     4

------- Structs/Unions  ------------
Version1 :                        1
Version2 :                        1
renamed :                         1

------- Typedefs  -----------------
Version1 :                        1
Version2 :                        1
renamed :                         1
```

Figure 2.4: Summary output produced for the code in Figure 2.2.

LocalNameMap will help us detect functions which are identical up to a renaming of local and formal variables, and *GlobalNameMap* is used to detect renamings for global variables and functions. As long as the ASTs have the same shape, we keep adding pairs to maps. If we encounter an AST mismatch (the **break** statement on the last line of the algorithm), we stop the matching process for that function and use the maps generated from the portion of the tree that did match.

2.2.2 Change Detection and Reporting

With the name and type bijections in hand, ASTdiff visits the functions, global variables, and types in the two programs to detect changes and collect statistics. We categorize each difference that we report either as an addition, deletion, or change.

```
/ : 111
     include/ : 109
          linux/ : 104
               fs.h : 4
               ide.h : 88
               reiserfs_fs_sb.h : 1
               reiserfs_fs_i.h : 2
               sched.h : 1
               wireless.h : 1
               hdreg.h : 7
          net/ : 2
               tcp.h : 1
               sock.h : 1
          asm-i386/ : 3
               io_apic.h : 3
     drivers/ : 1
          char/ : 1
               agp/ : 1
                    agp.h : 1
     net/ : 1
          ipv4/ : 1
               ip_fragment.c : 1
```

Figure 2.5: Density tree for **struct**/**union** field additions (Linux 2.4.20 → 2.4.21).

We report any function names present in one file and not the other as an addition, deletion, or renaming as appropriate. For functions in both files, we report that there is a change in the function body if there is a difference beyond the renamings that are represented in our name and type bijections. This can be used as an indication that the semantics of the function has changed, although this is a conservative assumption (i.e., semantics-preserving transformations such as code motion are flagged as changes). In our experience, whenever ASTdiff detects an AST mismatch, manual inspection has confirmed that the function semantics has indeed changed.

We similarly report additions, deletions and renamings of global variables, and changes in global variable types and static initializers.

For types we perform a deep structural isomorphism check, using the type bijection to identify which types should be equal. We report additions, deletions, or changes in fields for aggregate types; additions, deletions, or changes to base types for typedefs; and additions, deletions, or changes in item values for enums.

ASTdiff can be configured to either report this detailed information or to produce a summary. For the example in Figure 2.2, the summary output is presented in Figure 2.4. In each category, `Version1` represents the total number of items in the old program, and `Version2` in the new program. For brevity we have omitted all statistics whose value was 0.

ASTdiff can also present summary information in the form of a *density tree*, which shows how changes are distributed in a project. Figure 2.5 shows the density tree for the number of **struct** and **union** fields that were added between Linux versions 2.4.20 and 2.4.21. In this diagram, changes reported at the leaf nodes (source files) are propagated up the branches, making clusters of changes easy to visualize. In this example, the `include/linux/` directory and the `include/linux/ide.h` header file have a high density of changes.

A potential over-conservatism of our matching algorithm is that having insufficient name or type pairs could lead to renamings being reported as additions/deletions. The two reasons why we might miss pairs are partial matching of functions and function renamings. As mentioned previously, we stop adding pairs to maps when we detect an AST mismatch, so when lots of functions change their bodies, we miss name and type pairs. This could be mitigated by refining our AST comparison to recover from a mismatch and continue matching after detecting an AST change. Because renamings are detected in the last phase of the process, functions that are renamed don't have their ASTs matched, another reason for missing pairs. In order to avoid this problem, the bijection computation and function body matching would have to be iterated until a fixpoint is reached. Note that reporting spurious changes due to renamings do not affect the correctness of our DSU implementation. For example, reporting a function is as added and deleted instead of renamed would only cause more code to be loaded.

In practice, however, we found the approach to be reliable. For the case studies in Section 2.3, we have manually inspected the ASTdiff output and the source code for renamings that are improperly reported as additions and deletions due to lack of constraints. We found that a small percentage (less than 3% in all cases) of the reported deletions were actually renamings. The only exception was an early version of Apache (versions 1.2.6-1.3.0) which had significantly more renamings, with as many as 30% of the reported deletions as spurious.

2.2.3 Implementation

ASTdiff is constructed using CIL, an OCaml framework for C code analysis [80] that provides ASTs as well as some other high-level information about the source code. We have used it to analyze all releases of Vsftpd[1] from inception (Nov. 2001) to March 2005; all releases of the OpenSSH Sshd daemon[2] from inception (Oct 1999)

[1]`http://vsftpd.beasts.org/`

[2]`http://www.openssh.com/`

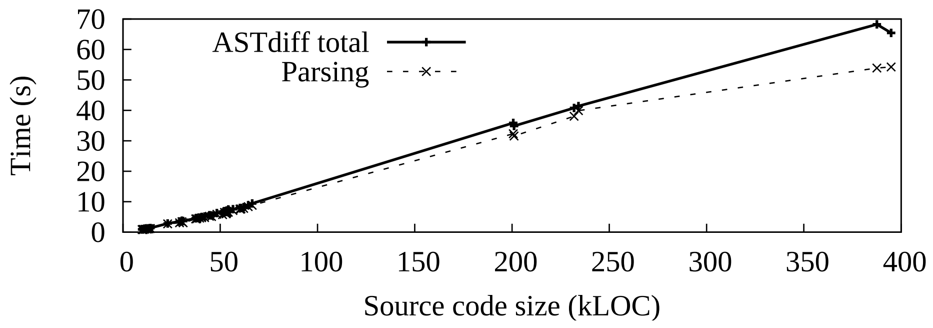

Figure 2.6: ASTdiff running time for various program sizes.

to March 2005; 8 snapshots in the lifetime of Apache 1.x[3] (Feb. 1998 to Oct. 2003); and portions of the lifetimes[4] of the Linux kernel[5] (versions 2.4.17, Dec. 2001 to 2.4.21, Jun. 2003) and BIND[6] (versions 9.2.1, May 2002 to 9.2.3, Oct. 2003).

The running time of ASTdiff is linear in the size of the input programs' ASTs. Figure 2.6 shows the running time of ASTdiff on our test applications, plotting source code size versus running time. Times are the average of 5 runs; the system used for experiments was a dual Xeon@2GHz with 1GB of RAM running Fedora Core 3. The top line is the total running time while the bottom line is the portion of the running time that is due to parsing, provided by CIL. The difference between the two lines is our analysis time. Computing changes for two versions of the largest test program takes slightly over one minute. The total time for running the analysis on the full repository (i.e., all the versions) for Vsftpd was 21 seconds (14 versions), for Sshd was 168 seconds (25 versions), and for Apache was 42 seconds (8 versions).

[3]`http://httpd.apache.org/`

[4]Analyzing earlier versions would have required older versions of `gcc`.

[5]`http://kernel.org/`

[6]`www.isc.org/products/BIND/`

2.3 Implications for Dynamic Software Updating

This section explains how we used ASTdiff to characterize software changes and to guide the way we designed Ginseng. We are mainly interested in three aspects of software evolution: how often do definitions get deleted, how often do function signatures change, and how do type definitions change. The reason we consider these aspects important is that implementing deletion and supporting type changes safely is problematic for DSU systems. We present our findings as structured around asking and answering three research questions:

Are function and variable deletions frequent, relative to the size of the program? When a programmer deletes a function or variable, we would expect a DSU implementation to delete that function from the running program when it is dynamically updated. However, implementing on-line deletion is difficult, because it is not safe to delete functions or variables that are currently in use (or will be in the future). Therefore, if definitions are rarely deleted over a long period, the benefit of cleaning up dead code may not be worth the cost of implementing a safe mechanism to do so. For simplicity, Ginseng does not unload unused functions and variables after they have been replaced and are no longer in use (Section 3.4).

Figure 2.7 illustrates how Sshd, Vsftpd, and Apache have evolved over their lifetime. The x-axis plots time, and the y-axis plots the number of function and global variable definitions for various versions of these programs. Each graph shows the total number of functions and global variables for each release, the cumulative number of functions/variables added, and the cumulative number of functions/variables deleted (deletions are expressed as a negative number, so that the sum of deletions, additions, and the original program size will equal its current size).[7] The rightmost points show the current size of each program, and the total number of additions and deletions to variables and functions over the program's lifetime.

According to ASTdiff, Vsftpd and Apache delete almost no functions, but Sshd deletes them steadily. For the purposes of our DSU question, Vsftpd and Apache could therefore reasonably avoid removing dead code, while doing so for Sshd would have a more significant impact (assuming functions are similar in size).

Are changes to function prototypes frequent? Many DSU methodologies do not update a function whose type has changed. While it is easy, technically, to load or replace a function, a change to a function's prototype can lead to type safety violations (Section 3.3). Figure 2.8 presents graphs similar to those in Figure 2.7. For each program, we graph the total number of functions, the cumulative number of functions whose body has changed, and the cumulative number of functions whose prototype has changed.[8] As we can see from the figure, changes in prototypes are relatively infrequent for Apache and Vsftpd, especially compared to changes more generally. In contrast, functions and their prototypes have changed in Sshd far more rapidly, with the total number of changes over five years roughly four times the current number of functions, with a fair number of these resulting in changes in prototypes. In all cases we can see *some* changes to prototypes, meaning that

[7]We use cumulative figures to show that additions are much more frequent than deletions.

[8]We use cumulative figures to show that body changes are much more frequent than prototype changes.

supporting prototype changes in DSU is a good idea.

Are changes to type definitions relatively simple? In most DSU systems, changes to type definitions (which include **struct**, **union**, **enum**, and **typedef** declarations in C programs) require an accompanying *type transformer function* to be supplied with the dynamic update. Each existing value of a changed type is converted to the new representation using this transformer function. Of course, this approach presumes that such a transformer function can be easily written. If changes to type definitions are fairly complex, it may be difficult to write a transformer function.

Figure 2.9 plots the relative frequency of changes to **struct**, **union**, and **enum** definitions (the y-axis) against the number of fields (or enumeration elements for **enum**s) that were added or deleted in a given change (the x-axis). The y-axis is presented as a percentage of the total number of type changes across the lifetime of the program. We can see that most type changes affect predominantly one or two fields; an exception is Sshd, where changing more than two fields is common. We also used ASTdiff to learn that fields do not change type frequently (not shown in the figure).

2.4 Conclusion

We have presented an approach to finding differences between program versions based on partial abstract syntax tree matching. Our algorithm uses AST matching to determine how types and variable names in different versions of a program correspond. We have constructed ASTdiff, a tool based on our approach and

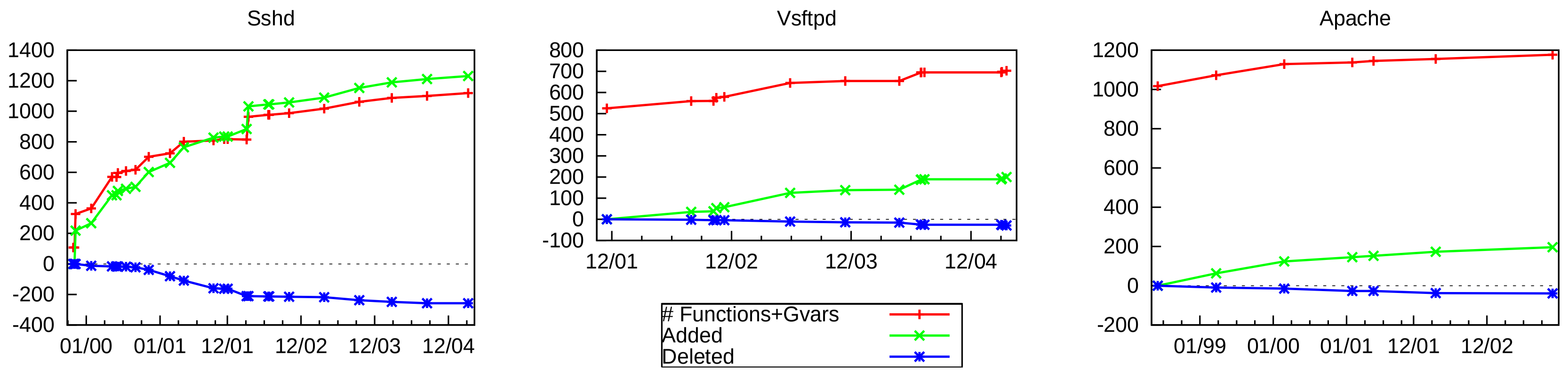

Figure 2.7: Function and global variable additions and deletions.

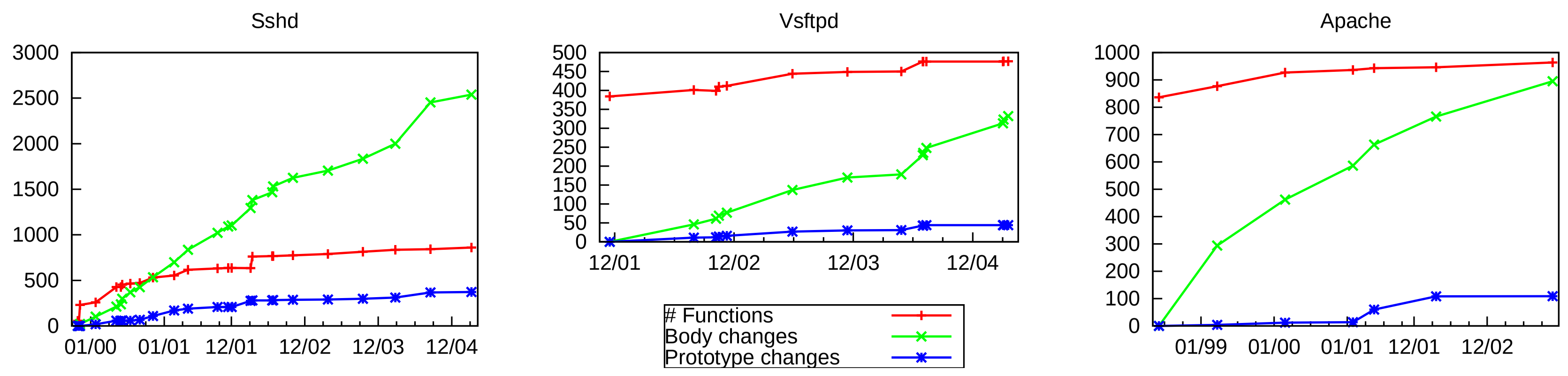

Figure 2.8: Function body and prototype changes.

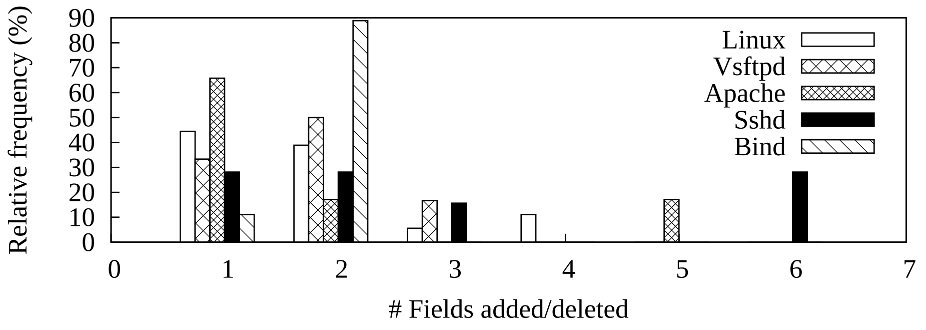

Figure 2.9: Classifying changes to types.

used it to analyze several popular open source projects over a few years in their lifetime. The software evolution insights we have gained from using ASTdiff, e.g., the way types and functions change have helped us in the design and implementation of Ginseng, our DSU system for C programs.

Chapter 3

Single-threaded Implementation and Evaluation

This chapter presents the implementation of Ginseng, an approach and tool suite for dynamically updating C programs, along with its evaluation on single-threaded programs.[1] Chapter 4 will discuss Ginseng's support for multi-threaded programs and its evaluation on multi-threaded programs.

3.1 Introduction

Our primary considerations for designing Ginseng follow the three practicality criteria described in Chapter 1 (efficiency, flexibility, and safety). We believe these features are necessary for any DSU system aiming to support long-term evolution for realistic programs:

Efficiency. DSU should permit writing applications in a natural style: while an application writer should anticipate that software will be upgraded, she should not have to know what form that update will take. Similarly, writing dynamic updates should be as easy as possible. The performance of updateable applications should be in line with that of normally-compiled applications; if support for update imposes

[1]The design, implementation and evaluation of Ginseng on single-threaded programs are the result of joint efforts with Gareth Stoyle, Michael Hicks, Manuel Oriol, Gavin Bierman, and Peter Sewell; we present details on their contributions in Section 3.7.

a high overhead, DSU is not likely to be adopted.

Flexibility. The power and appeal of DSU is to permit applications to change on the fly at a fine granularity. Thus, programmers should be able to change data representations, change function prototypes, reorganize subroutines, etc. as they normally would.

Safety. Dynamic updates should not be hard to establish as correct. The harder it is to develop applications that use DSU and prove their correctness, the more its benefits of finer granularity and control is diminished.

To evaluate single-threaded Ginseng, we have used it to dynamically upgrade three single-threaded servers: Vsftpd (the Very Secure FTP daemon), the Sshd daemon from the OpenSSH suite, and the Zebra server from the GNU Zebra routing software package.

Based on our experience, we believe Ginseng squarely meets the first two criteria for the class of single-threaded server applications we considered, and makes significant headway toward the third. These programs are realistic, substantial, and in common use. Though they were not designed with updating in mind, we had to make only a handful of changes to their source code to make them safely updateable. Each dynamic update we performed was based on an actual release, and for each application, we applied updates corresponding to at least three years' worth of releases, totaling as many as twelve different patches in one case. To achieve these results, we developed several new implementation techniques, including new ways to handle the transformation of data whose type changes, to allow dynamic updates to active code, and to allow updates to take effect in programs with function pointers. Though we have not optimized our implementation, overhead due to updating is modest: between 0 and 32% on the programs we tested.

Despite the fact that changes were non-trivial, generating and testing patches was relatively straightforward. We developed tools to generate most of a dynamic patch automatically by comparing two program versions, reducing programmer work. More importantly, Ginseng performs two safety analyses to determine times during the running program's execution at which an update can be performed safely. The theoretical development of our first analysis, called the *updateability analysis* [106], is not a contribution of this dissertation. We present an implementation of that analysis for the full C programming language, along with some practical extensions for handling some of the low-level features of C. These safety analyses assist assurance of correctness, though the programmer needs a clear "big picture" of the application, e.g., the interactions between application components, and establishing and maintaining global invariants.

A high-level overview of Ginseng's components was presented in Section 1.2. The next three sections describe these components in detail, while Sections 3.5 and 3.6 describe our experience using Ginseng and evaluate its performance.

3.2 Enabling On-line Updates

To make programs dynamically updateable we address two main problems. First, existing code must be able to call new versions of functions, whether via a direct call or via a function pointer. Second, the state of the program must be transformed to be compatible with the new code. For a type whose definition has changed, existing values of that type must be transformed to conform to the new definition.

Ginseng employs two mechanisms to address these two problems, respectively: *function indirection* and *type-wrapping*. We discuss them in turn below, and show how they can be combined to update active code.

3.2.1 Function Indirection

Function indirection is a standard technique [50] that permits old code to call new function versions by introducing a level of indirection between a caller and the called function, so that its implementation can change. For each function `f` in the program, Ginseng introduces a global variable `f_ptr` that initially points to the first version of `f`.[2] Ginseng encodes version information through name mangling, renaming the initial version of `f` to `f_v0`, the subsequent version `f_v1` and so on. Each direct call to `f` within the program is replaced with a call through `*f_ptr`. Ginseng also handles function pointers in an interesting way: if the program passes `f` as data

[2]Ginseng is more careful than we are in these examples about generating non-clashing variable names.

(i.e., as a function pointer), Ginseng generates a wrapper function that calls `*f_ptr` and passes this wrapper instead. To dynamically update `f` to version 1, the runtime system dynamically loads the new version `f_v1` and then stores the address of `f_v1` in `f_ptr`. While function indirection is not new, the idea of generating function wrappers to permit updates to a function whose address is taken is, to our knowledge, first introduced in this dissertation.

3.2.2 Type Wrapping

The Ginseng updating model enforces what we call *representation consistency* [106], in which all values of type `T` in the program at a given time must logically be members of `T`'s most recent version. The alternative would be to allow multiple versions of a type to coexist, where code and values of old and new type could interact freely within the program. (Hjálmtýsson and Gray [52] and Duggan [32] refer to these approaches as *global update* and *passive partitioning*, respectively.) Representation consistency is a useful property because it more closely models the "forward march" of a program's on-line evolution, making it easier to reason about.

To enforce representation consistency, Ginseng must ensure that when a particular type `T`'s definition is updated, values of that type in the running program are updated as well. To do this, a dynamic patch defines a *type transformer function* used to transform a value $v_{\texttt{T}}$ from `T`'s old definition to its new one. Just like functions, types are associated with a version, and the type transformer $c_{\texttt{T}n \to n+1}$

converts values of type $\mathtt{T}_n$ (i.e., the representation of `T` in version n) to be those of type $\mathtt{T}_{n+1}$. As we explain later, much of a type transformer function can be generated automatically via a simple comparison of the old and new definitions.

Given this basic mechanism, we must address two questions. First, when are type transformers to be used? Second, how is updateable data represented?

Applying Type Transformers. To transform existing $v_{\mathtt{T}_n}$ values the runtime system must find them all and apply $c_{\mathtt{T}n \to n+1}$ to each. One approach would be to do this eagerly, at update-time; this would require either implementing a garbage-collector-style tracing algorithm [43], or maintaining a registry of pointers to every (live) value of type $\mathtt{T}_n$ during execution [12]. More simply, we could restrict type transformation to only those data reachable from global variables, and require the programmer to implement the tracer manually [50]. Finally, we could do it lazily, as the program executes following the update [32, 17, 7].

Ginseng uses the lazy approach. The compiler renames version n of the user's definition of `T` to be $\mathtt{T}_n$, where the definition of `T` simply wraps that of $\mathtt{T}_n$, adding a version field. Given a value $v_{\mathtt{T}}$ (of wrapped type `T`), Ginseng inserts a *coercion* function called $con_{\mathtt{T}}$ (for $\underline{\text{con}}$cretization of `T`) that returns the underlying representation. This coercion is inserted wherever $v_{\mathtt{T}}$ is used concretely, i.e., in a way that depends on its definition. For example, this would happen when accessing a field in a **struct**. Whenever $con_{\mathtt{T}}$ is called on $v_{\mathtt{T}}$, the coercion function compares $v_{\mathtt{T}}$'s version n with the latest version m of `T`. If $n < m$, then the necessary type transformer functions are composed and applied to $v_{\mathtt{T}}$ changing it in-place. That is, Ginseng automatically invokes the entire type transformer chain $c_{\mathtt{T}n \to n+1}, c_{\mathtt{T}n+1 \to n+2}, \ldots, c_{\mathtt{T}m-1 \to m}$ to yield the up-to-date $v_{\mathtt{T}_m}$ (of type $\mathtt{T}_m$).

The lazy approach has a number of benefits. First, it is not limited to processing only values that are reachable by global variables; stack-allocated values, or those reachable from stack-allocated values, are handled easily. Second, it amortizes transformation costs, reducing the potential pause at update-time that would be required to transform all data in the program. The drawback is that per-type access during normal program execution is more expensive (due to the calls to $con_{\mathtt{T}}$), and the programmer has little control over when type transformers are invoked, since this is determined by the program's execution. Therefore, transformers must be written to be timing-independent. In our experience, type transformers are used rarely, and so it may be sensible to use a combination of eager and lazy application to reduce total overhead.

Without care, it could be possible for a transformed value to end up being processed by old code, violating representation consistency. This could lead a $con_{\mathtt{T}}$ coercion to discover that the version n on $v_{\mathtt{T}}$ is actually *greater* than the version m of the type `T` expected by the code. A similar situation arises when function types change: old code might end up calling the new version of a function assuming it has the old signature. We solve these problems with some novel safety analyses, described in more detail in Section 3.3.

Type Representations. While lazy type updating is not new [7], there has been little or no exploration of its implementation, particularly for a low-level language such as C. Based on our experience, a given type is likely to grow in size over time, so the representation of the wrapped type T must accommodate this. One approach is to define the wrapper type to use a fixed space, larger than the size of T_0 (padding). This strategy allows future updates to T that do not expand beyond the preallocated padding. The main advantage of the padding approach is that the allocation strategy for wrapped data is straightforward: stack-allocated data in the source program is still stack-allocated in the compiled program, and similarly for malloced data. This is because type transformation happens *in place*: the transformed data overwrites the old data in the same storage.

On the other hand, a data type cannot grow beyond the initial padding, hampering on-line evolution. Padding also changes the cache locality of data. For example, if a two-word structure in the original program is expanded to four words, then half as many elements can fit in a cache line.

An alternative approach would be to use indirection, and represent the wrapped type as a pointer to a value of the underlying type. This mechanism is used in the K42 operating system [60], which supports updating objects. The indirection approach solves the growth problem by allowing the size of the wrapped type to grow arbitrarily, but introduces an extra dereference per access. More importantly, the indirection approach makes memory management more challenging: how should storage for the transformed data be allocated, and what is to happen to the now-unneeded old data? Also, when data is copied, the indirected data must be copied as well, to preserve the sharing semantics of the application. The simplest solution would be to have the compiler malloc new representations and free (or garbage collect) the old ones; this is less performance-friendly than stack allocation. Another alternative would be to use *regions* [109], which have lexically-scoped lifetimes (as with stack frames), but support dynamic allocation. Of course, a hybrid approach is also possible: data could start out with some padding, and an indirection is only added if the padding is ever exceeded. Nevertheless, for simplicity, Ginseng employs the padding approach.

3.2.3 Example

Figure 3.1 presents a simple C program and how we compile it to be updateable. The main program is in function `call`: it creates a value `t` of type **struct** `T` and calls function `foo` (via `apply`) to set its `.x` field to `1`. The original program is on the left, and the resulting updateable program is in the middle and right columns. The comments can be ignored; these are the results of the safety analysis, explained in the next section.

First, we can see that all function definitions have been renamed to include a version, and that Ginseng has introduced a `_ptr` variable for each function, to keep a pointer to the most current version. Calls to functions are indirected through these pointers. Second, we can see that the definition of **struct** `T` is now a wrapper for **struct** `__T0`, the original definition. The `__con_T` function unwraps a **struct** `T`, potentially converting it to the latest representation via a call to `__DSU_transform` (which invokes the type transformer if the value must be updated). The `__con_T` function is called twice in `__call_v0` to extract the underlying value of `t`. Finally, we can see that

```c
struct T {
  int x; int y;
};

void foo(int* x) {
  *x = 1;
}
void apply(void (*fp)(int*),
           int* x) {
  fp(x);
}
void call () {
  struct T t = {1,2};
  apply(foo,&t.x);
  t.y = 1;
}
```

Original program

```c
struct T {
  unsigned int version;
  union { struct __T0 data;
          char padding[X]; } udata;
};
struct __T0* __con_T(struct T* abs) {
  __DSU_transform(abs);
  return &abs→udata.data;
}

void * foo_ptr = &__foo_v0;
void * apply_ptr = &__apply_v0;
void * call_ptr = &__call_v0;

void __foo_wrap(int* x) {
  (*foo_ptr)(x);
}
```

```c
struct __T0 { int x; int y; };

/* D=D'={T}, L={T}, x:T */
void __foo_v0(int* x) { *x = 1; }

/* D={foo,T}, D'={T}, L={}, x:T */
void __apply_v0(void (*fp)(int*),
                int *x) {
  fp(x);
}

/* D={T,apply}, D'={}, L={} */
void __call_v0 () {
  struct T t = { 0, {.data={1,2}}};
  /* D={T,apply} */
  (*apply_ptr)(__foo_wrap,
               &(__con_T(&t))→x);
  /* D={T} */
  &(__con_T(&t))→y = 1;
  /* D={} */

}
```

Updateable program

Figure 3.1: Compiling a program to be dynamically updateable.

Ginseng has generated __foo_wrap to wrap an indirected call to foo; this is passed as a function pointer to apply.

3.2.4 Loops

When a function f is updated, in-flight calls are unaffected, but all subsequent calls, including recursive ones, invoke the new f. In general, this makes reasoning about the timeline of an update simpler. On the other hand, it presents a problem for functions that implement long-running or infinite loops: if an update occurs to such a function while the old version is active, then the new version may not take effect for some time, or may never take effect. This is a disadvantage of any updating system that prevents updates to active functions (Section 3.3.4).

We solve this problem by a transformation we call *code extraction*. To illustrate how this work, we present an example of updating a long-running loop by extracting the loop body into a separate function.If the function containing the block is later changed, then this extracted function will notice the changes to the loop on the next iteration. As the code and state preceding the loop might have changed as well, the loop function must be parametrized by some *extracted code state*. This state will be transformed using our standard type transformer mechanism on the next iteration of the loop. Code extraction using a separate function parametrized by state is a technique similar to prior work on functional and parallel compilers (lambda lifting [59], procedure splitting [91], function outlining [115]) and on-stack replacement in optimizing VMs [22, 2].

```
#pragma DSU_extract("L1")

int foo(float g) {
  int x = 2;
  int y = 3;
  while (1) {
    L1: {
     x = x+1;
      if (x == 8) break;
      else continue;
      if (x == 9) return 42;
    }
  }
  return 1;
}
```

Original program

```
struct L1_xs {
  float *g;  int *x;  int *y;
};

int L1_extract(int *ret,
               struct L1_xs *xs) {
  *(xs->x) = *(xs->x) + 1;
  if (*(xs->x) == 8) {
    return 0; // break
  } else {
    return 1; // continue
  }
  if (*(xs->x) == 9) {
    *ret = 42;
    return 2; // return
  }
  return 1; // implicit continue
}
```

```
int foo(float g) {
  int x = 2;
  int y = 3;
  struct L1_xs xs;
  int retval;
  int code;
  xs.g = & g; // init extracted code state
  xs.x = & x;
  xs.y = & y;
  while (1) {
    code = L1_extract(&retval, &xs);
    if (code == 0) break;
    else if (code == 1) continue;
    else return retval;
  }
  return 1;
}
```

Updateable program

Figure 3.2: Updating a long-running loop using code extraction.

For illustration, consider the code in the left column of Figure 3.2. The programmer directs Ginseng that the code block labeled L1 should be extracted. The result is shown in the middle and right columns. In the middle is the extracted function, L1_extract, and on the right side is the rewritten original function foo. The function L1_extract takes two arguments: **struct** L1_xs *xs, and **int** *ret. The first argument, xs, is the "extracted state", which contains pointers to all of the local variables and parameters referenced in foo that might be needed by the code in L1; we can see in foo where this value is created. Within L1_extract, references to local variables (x) or parameters (g) have been changed to refer to them through *(xs).

Within the function foo, L1_extract is called on each loop iteration. Within L1_extract, expressions that would have exited the loop—notably **break**, **continue**, and **return** statements—are changed to **return** x, where x is 0 for **break**, 1 for **continue** and 2 for **return**. In foo, this return code is checked and the correct action is taken.

If in a subsequent program version the loop in foo were to change, the extracted versions of the two loop bodies would be different, with the new one updating the old one. The new version will be invoked on the loop's next iteration, and if the new loop requires additional state (e.g., new local variables or parameters were added to foo), then this is handled by the type transformer function for **struct** L1_xs. This type transformer might perform side-effecting initialization as well, for code that would have preceded the execution of the current loop. Note that foo's callers are neither aware nor affected by the loop extraction inside the body of foo.

When extracting infinite loops, nothing else needs to be done. However, if the loop might terminate, we must extract the code that follows the loop as well, so that an updated loop does not execute a stale post-amble when it completes; we accomplish this by simply marking the post-amble for code extraction as we did with `L1` above. The annotations the programmer needs to add for code extraction are described in detail in Section A.1.2.

A similar technique to code extraction, called *stack reconstruction*, is used in UpStare, another dynamic updating system [68]. Stack reconstruction allows the update developer to define a correspondence between program points in the old and new versions, and, at update time, the stacks of all active functions are converted into new-version stacks via user-specified functions. The advantage of stack reconstruction is that programmers do not need to identify in advance the code blocks, or loops, that need to be extracted.

Replacing arbitrary code on the stack was critical for supporting two of our three benchmark applications, Vsftpd and Sshd (Section 3.5). Both applications are structured around event loops: a parent process accepts incoming connection requests, and forks. The forked child breaks out of the loop and executes the loop postamble. If the loop body and loop postamble change in later versions, this will translate into updates to both extracted functions, hence both the parent and the children will get to execute the most up to date version.

3.3 Safety Analysis

When developing software with Ginseng, programmers designate points in the program where an update should take place; to indicate an update point, the programmer adds a call to function `DSU_update`. Update points are usually placed at program points where global state is consistent, e.g., at the end of an iteration of a long-running loop (Section 3.2.4). Placing update points where global invariants hold simplifies reasoning about update safety and writing the update. However, correct update point placement raises an issue for the programmer, since the form of future updates cannot be predicted. Therefore, the programmer needs to know whether an update that occurs in the future could create problems if they take effect at a certain update point.

To illustrate this, let us look again at the example in Figure 3.1. Suppose the program has just entered the `call` function—is it safe to update the type `T`? Generally speaking the answer is no, because code `t.x` assumes that `t` is a structure with field `x`, and a change to the representation of `t` could violate this assumption, leading to unexpected behavior. In this section we look at how Ginseng helps the programmer avoid choosing bad update points like this one using static analysis.

3.3.1 Tracking Changes to Types

The example given above illustrates what could happen when old code accesses new data, essentially violating representation consistency. To prevent this situation from happening, Ginseng applies a constraint-based, flow-sensitive *updateability analysis* [106] that annotates each update point with the set of types that may not be updated if representation consistency is to be preserved. This set is called the *capability* because it defines those types that *can* be used by old code that might be on the call stack during execution. Of course, the capability is a conservative approximation, as it approximates all possible "stack shapes." It is computed by propagating concrete uses of data backwards along the control flow of the program to possible update points.

Statically-approximated capabilities are illustrated in Figure 3.1, where the sets labeled D in the comments define the current capability; on functions, D defines the input capability (capability at the start of the function) and D' defines the output capability (capability at the end of the function). When T appears in D, it means that the program has the *capability* to use data of type T concretely. An update must not revoke this capability when it is needed.

We will now explain the capabilities for each function in the program (third column of Figure 3.1). For function foo (line 22), the input capability, D, contains T for two reasons: 1) because T has a live pointer into it for the duration of the function (live pointers are captured by the set L and are explained in Section 3.3.2), and because T appears in the output capability D' (i.e., is used concretely in foo's continuation, in call). For function apply (line 25), the input and output capabilities contain T due to its concrete use in apply's continuation; foo appears in the input capability D because we call foo in apply via the function pointer fp; the live pointer set L is empty because there is no live pointer into a type for the entire duration of apply—the last live pointer to T is dereferenced on line 28. For function call (line 31), the output capability D' is empty because there is nothing left on the stack after call exits; the live pointer set L is empty because there is no live pointer into a type for the entire duration of call; finally, the input capability D contains T and apply because they are used concretely (accessed and called, respectively) in the body of call.

At each program point, the capability D imposes a restriction on the functions and type that can be updated. For example, if we update apply at line 34, its type must either remain unchanged or the new type be a subtype of the old type [106], because apply appears in the capability D at that point. At line 37 we can perform an update that changes the type of apply or foo because there is no call to them on left on the stack; however, we cannot perform an update that changes the definition of T, because T is used concretely on the next line.

Programmers indicate where updates may occur in the program text by inserting a call to a special runtime system function DSU_update. When our analysis sees this function, it "annotates" it with the current capability. At run-time this annotation is used to prevent updates that would violate the static determination of the analysis. Moreover, the runtime system ensures that if a type *is* updated, then any functions in the current program that use the type concretely are updated with it; that is, even though ASTdiff finds no difference in the ASTs of a function in the old and new program versions, we will still load the new function version. This allows the static analysis to be less conservative. In particular, although the constraints on the form of capabilities induced by concrete usage are propagated backwards in the control flow, propagation does not continue into the callers of a function [106]. This propagation is not necessary because the update-time check ensures that all function calls are always compatible with any changed type representations.

The formalization and soundness proof of the updateability analysis are not part of this dissertation, and are presented elsewhere [106]. However, the implementation of this analysis for the full C language is one of the contributions of this dissertation.

Our implementation extends the basic analysis to also track concrete uses of functions and global variables, which permits more flexible updates to them. In the former case, by considering a call as a concrete use of a function, and function names as types, we can use the analysis to safely support a change to the type of the function. Similarly, in the latter case, by taking reads and writes of global variables as concrete uses, and the name of a global variable as a type, we can support representation changes to global variables. As shown in Section 2.3, the types of functions and global variables do change over time, so this extension has been critical to making DSU work for real programs.

The implementation also properly accounts for both signals and non-local control transfers via setjmp/longjmp, albeit quite conservatively. Since signal handlers can fire at any point in the program, we disallow occurrences of DSU_update inside a signal handler (or any function that handler might call), to avoid violating assumptions of the analysis (we could allow updates to occur, but prevent updates that would change type representations, function signatures, etc.). We model setjmp/longjmp as non-local **goto**; that is, the updateability analysis assumes that any longjmp in the program could go to any setjmp. The six server programs presented in Sections 3.5 and 4.4 do not employ setjmp/longjmp, but all of them use signals.

3.3.2 Abstraction-Violating Aliases

C's weak type system and low level of abstraction sometimes make it difficult for us to maintain the illusion that a wrapped type is the same as its underlying type. In particular, the use of unsafe casts and the address-of (&) operator can reveal a type's representation through an alias. An example of this can be seen in Figure 3.1 where apply is called passing the address of field x of t. Within foo, called by apply with this pointer, the statement $*x = 1$ is effectively a concrete use of T, but this fact is not clear from x's type, which is simply **int** $*$. An update to the representation of **struct** T while within foo could lead to a runtime error. We have a similar situation when using a pointer to a **typedef** as a pointer to its concrete representation. We say that these aliases are *abstraction violating*.

One extreme solution would be to mark **struct**s whose fields have their address taken as non-updateable. However, this solution can be relaxed by observing that only as long as an alias into a value of type T exists is it dangerous to update T. Thus if we know, at each possible update point, those types whose values might have live *abstraction-violating aliases* (AVAs), we can prevent those types from being changed. We discover this set of types using a *abstraction violating alias analysis*, an analysis that follows the general approach of effect reconstruction [67, 21, 5]. This analysis is described in Stoyle's dissertation [105].

The comments in Figure 3.1 illustrate the AVA analysis results for the example, where L is the set of types having live abstraction-violating aliases. L's contents are shown for each function, and the effect associated with variable x in functions foo and apply is shown to be T via the notation x:T. Looking at the example, we can see the call function violates T's abstraction by taking the address of t.x, and then passes this pointer to apply. This pointer is not used concretely in call, so does not effect subsequent computation in this function: call's environment has no abstraction violating pointers. As call is the only caller of apply, its associated L is empty. However, the environment of the body of apply does contain an abstraction-violating pointer, namely the parameter x. Thus when apply calls foo via the pointer fp, T's abstraction is violated and the L annotation for foo must contain T. In the example, we consider all statements as possible update points, and so extend D according to the results of the AVA analysis. This is why, for example, T appears in the capability of both foo and apply. In both cases T is in L or in the effect of a free variable in the environment (i.e., x). We do not show an annotation for foo_wrap because it is an auto-generated function (though Ginseng's safety analysis handles it properly).

3.3.3 Unsafe Casts and Polymorphism

To ensure that the program operates correctly, many representation-revealing casts are disallowed. For example, if we had a declaration **struct** S { **int** x; **int** y; **int** z; }, a C programmer might use this as a subtype of **struct** T from Figure 3.1, by casting a **struct** S $*$ to a **struct** T $*$. Given the way that we represent updateable types, permitting this cast would be unsafe, since **struct** S and **struct** T might have distinct type transformers and version numbers and treating one as the other may result in incorrect transformation. As a result, when our analysis discovers such a cast, it rules both types as non-updateable.

However, it would be too restrictive to handle all casts by rendering the types non-updateable. For example, C programmers often use **void** * to program generic types. One might write a "generic" container library in which a function to insert an element takes a **void** * as its argument, while one that extracts an element returns a **void** *. The programmer would cast the inserted element to **void** * and the returned **void** * value back to its assumed type. This idiom corresponds to *parametric polymorphism* in languages like ML and Haskell. Programmers also encode *existential types* using **void** * to build constructs like callback functions, and use upcasts and downcasts when creating and using callbacks, respectively. For example:

```
struct callback {
  void *env;
  void (*fp)(void *env, int arg);
};
void invoke(struct callback *cb, int arg) {
  cb→fp(cb→env,arg);
}
```

In this case, the env field of callback is existentially quantified: users can construct callbacks where there *exists* some consistent type τ that can be given to the env field and the first argument of fp field, but the invoke function is indifferent to this type's actual identity. Because τ can be different for different callbacks, C programmers must use the type **void** *, using upcasts and downcasts when creating and using callbacks, respectively.

If these idioms are used correctly, then they pose no problem to Ginseng's compilation approach since they do not reveal anything about a type's representation. However, we cannot treat casts to and from **void** $*$ as legal in general, because **void** $*$ could be used to "launder" an unsafe cast. For example, we might cast **struct** S $*$ to **void** $*$, and then the **void** $*$ to **struct** T $*$. Each cast may seem benign on its own, but becomes unsafe in combination. To handle this situation, our analysis annotates each **void** $*$ type in the program with the set of concrete types that might have been cast to it, e.g., casting a **struct** T $*$ to a **void** $*$ would add **struct** T to the set. When casting a **void** $*$ to **struct** S $*$, the analysis ensures the annotation on the **void** $*$ contains a single element, which matches **struct** S. If it does not, then this is a potentially unsafe cast and both **struct** T and **struct** S are made non-updateable. Since our analysis is not context-sensitive, some legal downcasts will be forbidden, for example when a container library is used twice in the program to hold different object types. Fortunately, such context-sensitivity is rarely used by the programs we have considered. In the worst case, we inspect the program manually to decide whether a cast is safe or not, and override the analysis results in this case with a pragma. The annotations the programmer needs to add for overriding the analysis, along with some examples of their use are presented in Section A.1.4.

3.3.4 Ginseng's Type Safety vs Activeness Check

One popular way for ensuring proper timing is to restrict an update from taking place if it affects code that is actively executing, i.e., is referenced by the stack of a running thread [23, 24, 1]. We call this restriction the "activeness check."[3] Unfortunately, while the activeness check precludes many problematic update times, not *all* problematic update times are ruled out.

Ginseng's safety check is comparable to the activeness check, though there are some differences. Our check permits updates to the body or signature of the current function, whereas the activeness check doesn't. However, since we take into account abstraction-violating aliases, we are more restrictive as to what types may be updated. For example, an alias p into a field of **struct** T can lead to a type safety violation if p is dereferenced after the definition of **struct** T changes. Ginseng's safety analysis only permits updates to **struct** T after the alias p is no longer live.

3.3.5 Choosing Update Points

In Section 3.3 we mentioned that programmers choose where to place update points. Placing update points where global invariants hold simplifies reasoning about update safety and writing the update. We define such points *quiescent points*, i.e., points in the program is one at which there are no partially-completed operations, and all global state is consistent (i.e., global invariants are satisfied). Dynamic updates are best applied at such quiescent points, so that writing an update is straightforward.

[3]A common criticism of the activeness check is that it is too strong: it precludes updates to code that never becomes inactive, e.g., the body of an infinite loop. In our experience, such updates are relatively rare, and in any case can be supported using techniques such as loop extraction (Section 3.2.4).

Ginseng adds constraints on types that can change at a programmer-inserted update point, so an update does not violate type safety. However, Ginseng does not provide guidance on where an update point should be placed—it only ratifies the programmer's decision in terms of type safety. A problem that can arise from bad update placement is best illustrated by the following example. The code on the left is the old program version, while the code on the right is the new version. The only change is moving the call to `g` from the body of `h` into the body of `f`.

```
1  void g() { ... }
2  void f() { ... }
3
4  void h () {
5
6    f();
7    g();
8  }
```

```
1  void g() { ... }
2  void f() { g(); }
3
4  void h () {
5
6    f();
7
8  }
```

While the old and new program essentially "do the same thing", a badly timed update can lead to unexpected behavior, even though the update is type safe. Suppose the update occurs on line 5 in the old program. The call to `f` will be to the new version that calls `g`, but then returns to its caller, the *old* `h`, which then calls `g` (line 7) again. Note that despite the update being type safe, we ended up calling `g` twice, which is problematic. If `g` is a memory deallocation function such as `free`, we end up freeing a location twice. If `g` is a logging function, we end up with a duplicated log entry. We can construct a symmetrical scenario where `g` is moved from `f` into `h`, and as a result of the update, we fail to call it.

This example illustrates the importance of update timing, and its impact on update correctness. In Chapter 5 we will show how programmers can designate code blocks that "go together" (e.g., the body of function h in our example, or one iteration of an event processing loop). Based on this programmer indication, Ginseng enforces a property named *version consistency*: all the functions and global variables in such a block are accessed at the same program version. In our example, Ginseng would prevent an update that changes both f and h from being applied at line 5, because this leads to a version-inconsistent execution for the code in the body of h.

Note that a quiescent point is related to, but not identical with, a point with empty capability (Section 3.3); its capability may not necessarily be empty, although it is usually small. On the other hand, an empty capability does not imply quiescence, but rather indicates there are no concrete uses of types beyond the current point.

3.4 Dynamic Patches

Patch Generation. For each new release we need to generate a dynamic patch, which consists of new and updated functions and global variables, type transformers and state transformers. The Ginseng patch generator generates most of a dynamic patch automatically in three steps. First, it compares the old and new versions of a program using ASTdiff (Section 2.2) to discover the new and modified definitions. Second, it adds the new and changed definitions to the patch file, where unchanged definitions are made **extern**. Third, it generates type transformers for all changed types by attempting to construct a conversion from the old type into the new type [50]. For example, if a **struct** type had been extended by an extra field, the generator would produce code to copy the common fields and add a default initializer for the added one. This simplistic approach to patch generation is surprisingly effective, requiring few manual adjustments; in Section A.2 we present some concrete examples of how the programmer writes state transformers and adjusts the auto-generated type transformer.

After the patch is generated and the state and/or type transformers are written, we pass the resulting C file to Ginseng, and the final result is compiled to a shared library so that it can be linked into the running program. Ginseng compiles the patch just as it does the initial version of a program, but also introduces initialization code to be run at update-time. The initialization code will effectively "glue" the dynamic patch into the running program, as explained next.

Dynamic Patch Example. Figure 3.3 presents the source code for an actual dynamic patch, corresponding to the update from Zebra 0.92a to 0.93a. All the code, except the state transformer (DSU_state_xform on lines 6–7) and programmer-adjusted part of type transformers (DSU_tt_x on lines 1–4) is auto-generated.

The first part (lines 1–7) contains type and state transformers. The second part (lines 10–22) contains new and changed functions and global variables. Note how Ginseng performs name mangling by renaming each function definition according to function version: access_list_standard is a new function, hence its name ends in v0, whereas vty_serv_sock and config_write_access are now at the second version. The function DSU_install_patch is the auto-generated "glue code" that installs the latest

```
 1  void DSU_tt_vty_v0( struct  vty_old  *xin,
 2                      struct vty_new *xout,
 3                      struct DSU_wrapper_struct_vty *xnew )
 4  { ... }
 5
 6  void DSU_state_xform(void)
 7  { ... }
 8
 9
10  /* New functions */
11  int   access_list_standard_v0  (...)
12  { ... }
13
14  /* Changed functions */
15  extern void  vty_serv_sock_v0 (unsigned short port, char *path ) ;
16  void  vty_serv_sock_v1 (char const *hostname, unsigned short port, char *path ) ;
17  { ... }
18
19  extern int   config_write_access_ipv4_v0  (DSU_wrapper_struct_vty *vty )
20  int   config_write_access_ipv4_v1  ( wrapper_struct_vty  *vty )
21  { ... }
22
23  void  DSU_install_patch (void)
24  {
25     DSU_latest_tt_struct_vty   = & tt_vty_v0;
26
27     vty_serv_sock_ptr  = & vty_serv_sock_v1;
28      config_write_access_ipv4_ptr   = &  config_write_access_ipv4_v1 ;
29      access_list_standard_host_ptr   = &  access_list_standard_host_v0 ;
30
31    DSU_state_xform();
32  }
33
34  char *DSU_update_contents =  " struct_vty   vty_serv_sock   config_write_access_ipv4   ... ";
```

Figure 3.3: Ginseng dynamic patch for the update Zebra 0.92a → 0.93a.

version of the type transformer for types that have changed (line 25), sets the function pointers for new and changed functions (lines 27–29) and finally invokes the state transformer (line 31). Ginseng also includes the update contents (line 34)—a string containing the set of functions, types and global variables changed by the update.

Runtime System. To perform an update, the user sends a signal to the running program, which alerts the runtime system. When the program reaches a possible update point (i.e., the first call to DSU_update for single-threaded programs, or the first thread to reach an induced update point for multi-threaded programs), the runtime system will try to perform the update. First, Ginseng loads the shared library containing the dynamic patch into memory, using `dlopen` [64]. Then, the runtime system retrieves the update contents DSU_update_contents.

Now the runtime system is ready to perform the update safety check. For single-threaded programs, DSU_update_contents is checked against the capability of the update point (Section 3.3.1). For multi-threaded programs, DSU_update_contents is checked against each thread's capability and contextual effects (Section 4.2.1). The check prevents Ginseng from applying an update at an unsafe point. If the check fails, the runtime system gives the control back to the program, and will try to apply the update at a later update point. If the check succeeds, the runtime system will install the patch.

Patch installation is very simple: Ginseng calls DSU_install_patch which installs the type transformers for the updated types, redirects changed functions to the new versions, and finally invokes the state transformer if the user has provided one. Type transformers are not called at update time; they are invoked lazily, when the values to be updated are accessed.

Our current runtime system has two main limitations. We do not support patch unloading, so old code and data will persist following an update. This memory leak has been minimal in practice—between 21% and 40% after three years' worth of patches for our benchmark applications because, as explained in Section 2.3, deletions are infrequent. Second, dynamic updates cannot be rolled back transactional. If, during an update, an error is encountered in Ginseng-generated glue code, the runtime system or the user-supplied state transformer, we do not yet have a safe mechanism to abort the update and restore the state to the pre-update one. Possible approaches to solving this problem are saving the values of global variables prior to the update and restoring them upon failure [50], or using speculations [108]/transactional memory [15] to roll back the effects of a failed update. We leave these problems to future work.

3.5 Experience

We now present our experience with dynamically updating three single-threaded open-source programs: Vsftpd, the Very Secure FTP daemon, the OpenSSH Sshd daemon, and the Zebra routing daemon from the GNU Zebra routing package;[4] in Chapter 4 we will present our experience with updating multi-threaded programs.

We chose these programs because they are long-running, maintain soft state that could be usefully preserved across updates, and are in wide use. For each program we downloaded releases spanning several years and then applied the methodology shown in Figure 1.1. In particular, we compiled the earliest release to be updateable and started running it. Then we generated dynamic patches for subsequent releases and applied them on-the-fly in release order, while the program was

[4] `http://www.zebra.org`

actively performing work (serving files, establishing connections, etc.).

With this process, we identified key application features that make updating the applications easy or hard. We also identified strong points of our approach (that enabled most of the updates to be generated automatically), along with issues that need to be addressed in order to make the updating process easier, more flexible and applicable to a broad category of applications. In the rest of this section, we describe the applications and their evolution history, and the manual effort required to dynamically update them; identify application characteristics and Ginseng features that make updating feasible; and conclude by reviewing factors that enabled us to meet the challenges set forth in Section 3.1.

3.5.1 Applications

Table 3.1 shows release and update information for each program. Columns 2–4 show the version number, release date and program size for each release. Column 5 contains the nature of individual releases.[5] To give a sense of programmer effort for each update, column 6 shows the number of type transformers for that specific update, while column 7 presents the size of the state transformer in lines of code; '-' means no type or state transformers were needed for a particular release.

We now briefly discuss each application, then describe how the applications changed over a three year period, and finally discuss the manual effort required to dynamically update them.

Vsftpd stands for "Very Secure FTP Daemon" and is now the *de facto* FTP

[5]As described at `http://freshmeat.net/`

Program	**Release**	**Date**	**Size**	**Description**	**Type xform**	**State xform**
			(LOC)		**(count)**	**(LOC)**
Vstfpd	1.1.0	07/02	10,141			
	1.1.1	10/02	10,245	Minor bug/security fixes	2	-
	1.1.2	10/02	10,540	Major feature enhanc.	5	-
	1.1.3	11/02	10,723	Minor feature enhanc.	3	-
	1.2.0	05/03	12,027	Major feature enhanc.	7	1
	1.2.1	11/03	12,662	Minor feature enhanc.	6	-
	1.2.2	04/04	12,691	Major bug/security fixes	3	-
	2.0.0	07/04	13,465	Major feature enhanc.	4	-
	2.0.1	07/04	13,478	Major bug/security fixes	-	-
	2.0.2pre2	07/04	13,531	Other	-	-
	2.0.2pre3	03/05	14,712	Other	-	-
	2.0.2	03/05	17,386	Major bug/security fixes	-	-
	2.0.3	03/05	17,424	Major bug/security fixes	1	-
Sshd	3.5p1	10/02	47,424			
	3.6.1p1	04/03	49,120	Minor bug/security fixes	3	-
	3.6.1p2	04/03	49,134	Major bug/security fixes	-	-
	3.7.1p1	09/03	51,133	Major bug/security fixes	8	-
	3.7.1p2	04/03	51,145	Major bug/security fixes	-	-
	3.8p1	02/04	52,547	Other	8	-
	3.8.1p1	04/04	52,549	Minor bug/security fixes	1	-
	3.9p1	08/04	53,979	Major feature enhanc.	8	-
	4.0	03/05	56,803	Minor feature enhanc.	9	-
	4.1	05/05	56,840	Minor bug/security fixes	3	-
	4.2p1	09/05	58,104	Minor bug/security fixes	7	-
Zebra	0.92a	08/01	41,630			
	0.93a	07/02	40,649	Major bugfixes	11	30
	0.93b	09/02	40,679	Minor fixes	1	1
	0.94	11/03	45,447	Minor security fixes	3	17
	0.95	03/05	45,546	Major bugfixes	7	-
	0.95a	09/05	45,586	Other	1	-

Table 3.1: Application releases.

Program	Functions				Types			Global variables		
	Add	Del.	Proto changes	Body changes	Add	Del.	Chg.	Add	Del.	Chg.
Vstfpd	97	21	33	308	12	2	6	72	9	15
Sshd	131	19	85	752	27	2	19	70	19	29
Zebra	134	44	13	321	24	6	4	56	11	52

Table 3.2: Changes to applications.

server in major Unix distributions. For our study, we considered the 13 versions from 1.1.0 through 2.0.3. As can be seen in Table 3.1, in the time frame we considered there were 3 major feature enhancements, 4 major bugfixes, 2 minor feature enhancements and 1 minor bugfix.

Sshd is the SSH daemon from the OpenSSH suite, which is the standard open-source release of the widely-used secure shell protocols. We upgraded Sshd 10 times, corresponding to 11 OpenSSH releases (version 3.5p1 to 4.2p1) over three years.

GNU Zebra is a TCP/IP routing software package for building dedicated routers that support the RIP, OSPF, and BGP protocols on top of IPv4 or IPv6. It consists of protocol daemons (RIPd, OSPFd, BGPd) and a Zebra daemon which acts as a mediator between the protocol daemons and the kernel (Figure 3.4), storing and managing acquired routes. Storing routes in Zebra allows protocol daemons to be stopped and restarted without discarding and re-learning routes (which can be a time consuming process). We upgraded Zebra 5 times, corresponding to 6 releases (version 0.92a to 0.95a) over 4 years.

Evolution History. Table 3.2 contains the cumulative number of changes that occurred to the software over that span, computed using ASTdiff. “Types” refers to **struct**s, **union**s and **typedef**s together. Global variable changes consists of changes to either global variable types or to global variable static initializers. As an example reading of the table, notice that for Vsftpd, 97 functions were added, 21 were deleted, 33 functions had their prototype changed, and 308 functions had the bodies changed. For Sshd, 19 types changed; for Zebra, there were 52 global variable changes. We mentioned in Chapter 2 that a dynamic software updating system must support changes, additions, and deletions for functions, types and global variables if it is to handle realistic software evolution. Ginseng supports all these changes, therefore we have been able to dynamically update the three applications from the earliest to the latest versions we considered.

3.5.2 Changes To Original Source Code

To safely update these applications with Ginseng required making a few small changes and additions to their source code. These changes amount to around 50 lines of code for Vsftpd and Sshd and 40 lines for Zebra, for each program version. The changes consisted of introducing named types for some global variables (we need to introduce such types for global variables whose addresses are taken, to support changes to these variables' types and static initializers), directives to the compiler (for adding update points, analysis, code extraction—described in detail in Section A.1) and in one case (Vsftpd), instantiating an existential use of **void** $*$ (see Section 3.3.3). Another change to Vsftpd is discussed in the next subsection.

For each new release, we would use the Ginseng patch generator to generate the initial patch, and then verify or complete the auto-generated type transformers

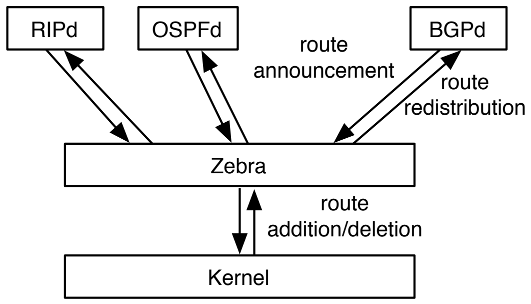

Figure 3.4: Zebra architecture.

and write state transformers (where needed, which was rare, as can be seen in columns 6–7 of Table 3.1). This effort was typically minimal. Table 3.3 presents the breakdown of patches, across all releases, into manual and auto-generated source code: the first column shows the number of source code lines we had to write for type and state transformers, the second column shows code lines we had to write to cope with changes in global variables' types or static initializers, and the third column shows the amount of code coming out of the patch generator. The code dealing with changes in static initializers for global variables is frequently a mere copy-paste of the variable's static initializer.

3.5.3 Dynamic Updating Catalysts

In the process of updating the three applications, we discovered four factors that make programs amenable to dynamic updating.

Quiescence. As mentioned in Section 3.3.5, dynamic updates are best applied at quiescent points, so that writing an update is straightforward. However, it is the programmer's responsibility to find such points and indicate them to the compiler

```
int main() {
  init ();
  conn = accept_loop();
  L1: {
    init_conn (conn);
    handle_conn(conn);
  }
}
```

```
int accept_loop() {
  L2: while (1) {
    fd = accept();
    if (! fork ())
      return fd;
  }
}
```

```
void handle_conn(fd) {
  L3: while (1) {
    read(cmd,fd);
  }
}
```

Figure 3.5: Vsftpd: simplified structure.

via DSU_update(). Fortunately, each application was structured around an event processing loop, where the end of the loop defines a stable quiescent point: there are no pending function calls, little or no data on the stack, and the global state is consistent. At update time, new versions of the functions are installed and global state is transformed so the next iteration of the loop will be effectively executing the new program.

For instance, Vsftpd is structured around two infinite loops: one for accepting new client connections, and one for handling commands in existing connections (Figure 3.5). Each time a connection is accepted, the parent forks a new process and returns from the accept loop within the child process. The main function then initializes the connection and calls handle_con to process user commands. To be able to update the long running loops, and to handle updates following the accept loop in main, we used loop extraction (Section 3.2.4) at each of the three labeled locations so that they could be properly updated. Note that although L1 is not a loop, by using loop extraction we were able to update code on main's stack (the continuation of accept_loop()) without replacing main itself. For each of the three applications we used one programmer-inserted update point, in the main process, at the end of one

Program	**Source code (LOC)**		
	Type + state xform (manual)	Gvar changes (manual)	Patch generator auto
Vsftpd	162	930	83,965
Sshd	125	659	248,587
Zebra	49	244	43,173

Table 3.3: Patch source code breakdown.

iteration of the accept loop (hence both the parent process, and each new child process spawned as result of accepting a connection will always execute the latest version).

Functional State Transformation. Our mechanisms for transforming global state (state transformers) and local state (type transformers) assume that we can write a function that transforms old program state into new program state. Unfortunately, sometimes it is not possible to impose the semantics of the new application on the existing state. We encountered two such cases in our test applications. In the upgrade from Sshd 3.7.1p2 to Sshd 3.8p1, a new security feature was introduced: the user's Unix password is checked during the authentication phase and if the password has expired, port forwarding will be not be allowed on the SSH connection. However, when dynamically updating a live connection from version 3.7.1p2 to 3.8p1, the authentication phase has passed already, so the new policy is not enforced for existing connections (though they could be shut down forcibly). For new connections requests coming in after the update, the new check is, of course, performed.

A similar situation arose in going from Vsftpd 1.1.1 to 1.1.2. The new release introduced per-IP address connection limits by mapping the ID of each connection process with a count related to remote IP address. These counts are increased when a process is forked and decremented in a signal handler when a process dies. Unfortunately, following an update, any current processes will not have been added to the newly introduced map, and so the signal handler will not execute properly. In effect, the new state is not a function of the old state. In this case, the easy remedy is to modify the 1.1.2 signal handler to not decrement the count if the process ID is not known.

When transforming some value, a type transformer can only refer to the old version of the value and the latest version of global variables, which means that in principle some transformations may be difficult or impossible to carry out. In practice we did not find this to be a problem: for all the 29 type transformers we had to write, the programmer effort was limited to initializing newly added **struct** fields.

Type-safe Programs. As mentioned in Section 3.3, low-level programming idioms might result in types being marked non-updateable by the analysis. Since having a non-updateable type restricts the range of possible updates, we would like to maximize the number of updateable types, so the solution is to either have a more precise analysis, or inspect specific type uses by hand and override the analysis for that particular type. For the programs we have considered, the techniques presented in Sections 3.3.2 and 3.3.3 increased the precision of the analysis and thereby greatly reduced the need to inspect the program manually.

For instance, in vsftpd, strings are represented by a **struct** mystr that carries the proper string along with length and the allocated size. The address of the string field is passed to functions, hence revealing **struct** mystr's representation, but our abstraction violation analysis was able to detect that the aliases were temporary and did not escape the scope of the callee, hence the type was updateable at the conclusion of the call. Polymorphism is employed in all three programs; using the **void** $*$ analysis (Section 3.3.3) we were able to detect type-safe uses of **void** $*$ and reduce the number of casts that have to be manually inspected. Inline assembly can compromise type safety as well: we do not know how global variables, types and functions are used when passed to assembly code. Our analysis treats inline assembly conservatively by preventing changes to type definitions and types of functions or global variables used in inline assembly. However, when a manual inspection confirms that such uses are safe, we can decide to override the analysis; we only had one such situation in Sshd. In the end, we manually overrode the analysis only for a handful of types: 0 for Vsftpd, 1 for Zebra, and 4 for Sshd.

Our type wrapping scheme relies on the fact that programs rarely rely on how types are physically laid out in memory, i.e., that they are treated abstractly in this respect. Fortunately, this was a good assumption for these programs. We could not type wrap some "low level" types, e.g., Vsftpd's representation of an IP address, since its layout is ultimately fixed by the OS syscall API. On the other hand, this and low-level structures like this one rarely change, since they are tied to external specifications.

Robust Design. We wanted our DSU approach to be general enough to be applied to off-the-shelf software, written without dynamic updates in mind (as was the case with our test applications). However, there are measures developers can take to make applications more update-friendly. Apart from features mentioned above (quiescent points, type safety, and abstract types), we have also found defensive programming and extensive test cases to be helpful in developing and validating the updates. All three programs we looked at were written defensively using `assert` liberally, which facilitated error detection and helped us spot Ginseng bugs relatively easily. By looking at the assertions in the code, we were able to detect the invariants the programs relied on, and preserve them across updates. Sshd comes with a rigorous test suite that provides extensive code coverage, and for Zebra and Vsftpd we created our own suites to test a broad range of features.

3.5.4 Summary

We believe we have addressed the DSU challenges set forth in Section 3.1. We did not have to change the applications extensively to render them updateable. Patch generation was mostly automatic, and writing the manual parts was easy.

We were able to support a large variety of changes to applications; as can be seen in Tables 3.1 and 3.2, the applications have changed significantly during the three-four years time-frame we considered. Once we became familiar with the application structure (e.g., interaction between components, global invariants), writing patches was easy, with all the infrastructure generated automatically; the only manual task was to initialize newly added fields, write state transformers, or make some small code changes.

A combination of factors have helped us address these challenges: (1) programs were amenable to dynamic updating (easily identifiable quiescence points the application, application changes that allowed updates to be written as functions from the old state to the new state, robust application design and moderate use of type-unsafe, low-level code), and (2) Ginseng, especially analysis refinements and support for automation, has made the task of constructing and validating updates easy, even for applications in the range of 50-60 KLOC.

3.6 Performance

In this section, we evaluate the impact of our approach on updateable software. We analyzed the overhead introduced by DSU by subjecting the instrumented applications to a variety of 'real world' tests. We considered the following aspects:

1. *Application performance.* We measured the overhead that updateability imposes on an application's performance by running stress tests. We found that DSU overhead is modest for I/O bound applications, but significant for CPU-bound ones.

2. *Memory footprint.* Type wrapping, extra version checks and dynamic patches result in an increased memory footprint for DSU applications. We found the increase to be negligible for updateable and updated applications, but after stacking multiple patches, the memory footprint increase is detectable.

3. *Service disruption.* We measure the cost of performing an actual update while the application is in use. The update will cause a delay in the application's processing, while the patch is loaded and applied, and will result in an amortized overhead as data is transformed. In all the updates we performed, even for large patches, we found the update time to be less than 5 ms.

4. *Type wrapping overhead.* In order to measure the impact of type wrapping on CPU-bound applications, we instrumented an application that performs computations on named types exclusively—KissFFT. We found type wrapping to introduce a significant overhead, in terms of both performance and memory footprint.

We also measured the running time of Ginseng to compile our benchmark programs, to measure the overhead of compilation and our analyses.

We conducted our experiments on dual Xeon@2GHz servers with 1GB of RAM, connected by a 100Mbps Fast Ethernet network. The systems ran Fedora Core 3, kernel version 2.6.10. All C code, generated by Ginseng or otherwise, was compiled with `gcc` 3.4.2 at optimization level `-O2`. We have compiled and run the experiments with optimization level `-O3`, but apart form a slight increase in memory footprint (less than 1%), there was no detectable difference in performance. Unless otherwise noted, we report the median of 11 runs.

Program	Connection time (ms)			
	stock	updateable	updated once	streak
Vsftpd	6.71	6.9 (2.83%)	7.04 (4.91%)	8.4 (25.18%)
Sshd	47.62	49.26 (3.44%)	49.5 (3.94%)	62.89 (32.06%)
Zebra	0.63	0.65 (3.17%)	0.65 (3.17%)	0.67 (6.34%)

Program	Transfer rate (MB/s)			
	stock	updateable	updated once	streak
Vsftpd	7.95	7.95 (0%)	7.97 (0.25%)	7.98 (0.37%)
Sshd	7.85	7.84 (−0.12%)	7.83 (−0.25%)	7.84 (−0.12%)

Table 3.4: Server performance, in absolute numbers and relative to the stock version.

3.6.1 Application Performance

To assess the impact of updateability on application performance, we tried different stress tests on the updateable applications. For each application, we measure the performance of its most recent version under four configurations. The *stock* configuration is the application compiled normally, without updating. The *updateable* configuration is the application compiled with updating support. The *updated once* configuration is the application after performing one update, whereas the *updated streak* configuration is the application compiled from its oldest version and then dynamically updated multiple times to bring it to the most recent version; this configuration is useful for considering any longer-term effects on performance due to updating.

Vstfpd. We tested Vsftpd performance using two metrics: connection time and transfer rate. For connection time, we measured the time it took `wget` to sequentially request 500 files of size 0, and divided by 500. Since `wget` opens a new connection for each file, and disk transfers are not involved, we get a picture of the overhead

DSU imposes on FTP clients. As seen in Table 3.4, the updateable, updated and streak-updated versions were 3%, 5% and 25% slower than the stock server. With a difference of at most 1.7 ms, we do not believe this to be a problem for FTP users.

These measurements seem to suggest a progressive slowdown due to updating. The primary reason for this appears to be poorer spatial locality. Using `OProfile`,[6] we measured the total cycles, instructions retired, and cache and TLB misses during benchmark runs of the one-update and streak-updated versions. We found that the effective CPI of the streak-updated version was consistently higher, and that this was attributable to cache and TLB misses. Such misses are understandable: code and data that were close together in the original program are now spread across multiple shared libraries.

We also measured the median transfer rate of a single 600 MB file to a single client. The results are shown in Table 3.4; since file transfer is a network-bound operation, the transfer rates of the different configurations are virtually identical.

Sshd. For Sshd we measured the same indicators as for Vsftpd, connection time and transfer rate. For the former, we loaded the server with 1000 concurrent requests and measured the total elapsed time, divided by 1000. (Client-server authentication was based on public key hence no manual intervention was needed.) Each client connection immediately exited after it was established (by running the `exit` command). The measured connection time is shown in Table 3.4. The updateable, updated and streak-updated versions were 3%, 4% and 32% slower than the

[6] `http://oprofile.sourceforge.net`

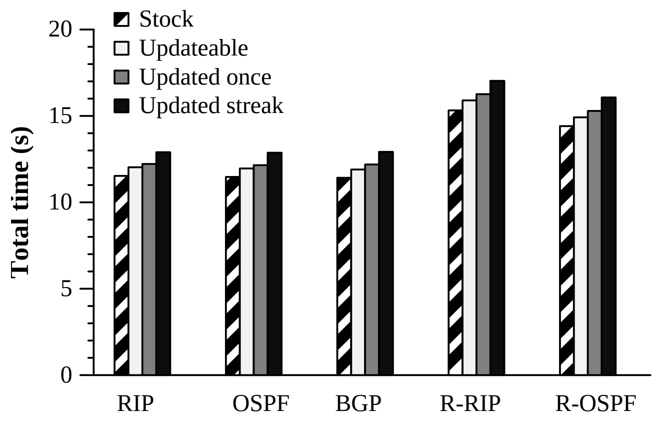

Figure 3.6: Zebra performance.

stock server. Again, we do not think the 15ms difference is going to be noticed in practice. The CPU-intensive nature of authentication and session key computation accounts for SSH connection time being almost 10 times larger than for FTP. To measure the sustained transfer rate over SSH we used `scp` to copy a 600MB file. As shown in Table 3.4, the results are similar to the Vsftpd benchmark—the transfer is network-bound and the DSU overhead is undetectable

Zebra. Since Zebra is primarily used for route proxying and redistribution, the focus of Zebra experiments was different than for Vsftpd and Sshd. First, we measured the overhead DSU imposes on route addition and deletion. We started each protocol daemon alone with Zebra, and programmed the protocol daemon to add and delete 100,000 routes. When passing routes through the updateable, updated and streak-updated versions of the Zebra daemon, the DSU overhead was 4%, 6% and 12%, compared to the stock case (first three clusters in Figure 3.6). Second, we measured route redistribution performance. We started the RIP daemon, turned on

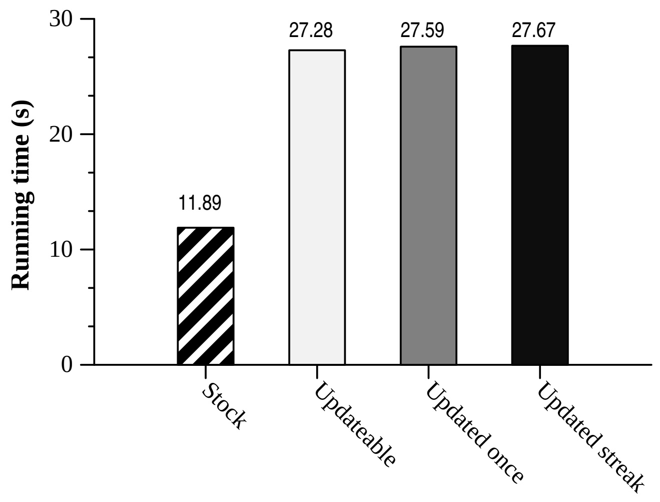

Figure 3.7: KissFFT: DSU impact on performance.

redistribution to OSPF and BGP daemons, programmed the RIP daemon to add and delete 100,000 routes, and measured the time it took until the route updates were reflected back into the OSPF and BGP routing tables. Similarly, we timed redistribution of OSPF routes to RIP and BGP daemons. BGP redistribution is not supported by Zebra. The DSU overhead in the route redistribution case (last two clusters in Figure 3.6) is the same as for the "no redistribution" case above: 4%, 6% and 12% respectively.

Zebra offers a command line interface for remote administration, so as a sanity check only, we measured the connection time for Zebra as well. We wrote a simple client that connects to the Zebra daemon, authenticates, executes a simple command ("`show version`") and then exits. We measured (Table 3.4) a 3%, 3%, and 6% increase in connection times for the updateable, updated once and streak-updated Zebra versions, respectively.

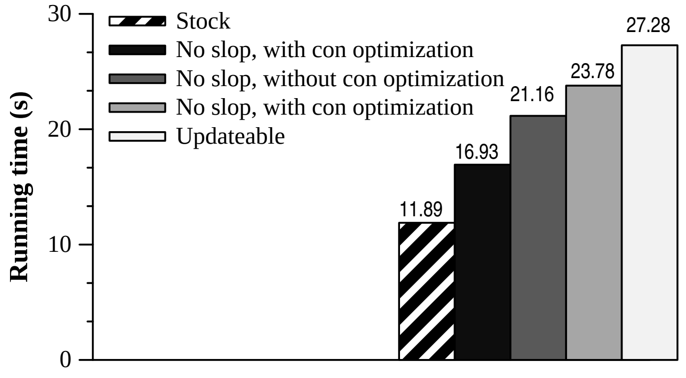

Figure 3.8: KissFFT: impact of optimizations on running time.

KissFFT. The overhead of DSU is dwarfed by I/O costs in our experiments. On the one hand, this is good because illustrates that for a relevant class of applications, DSU is not cost-prohibitive. On the other hand, it does not give a sense of the costs of DSU for more compute-bound applications. To get a sense of this cost, we instrumented KissFFT[7], a Fast Fourier Transform library. Figure 3.7 shows the total time to perform 100,000 Fast Fourier Transforms on 10,000 points. The updateable, updated once and updated streak versions were on average more than twice as slow (a factor of 2.29x) than the stock version.

We analyzed KissFFT to understand the source of the overhead. The program operates on a large array of complex numbers, and each complex number is represented as a **struct** complex. Therefore, before accessing fields a __con_complex has to be performed. Moreover, each complex number will have some slop to accommodate future growth.

Together, these two overheads can make a significant difference, as shown in Figure 3.8. First, the compiler does not attempt to optimize away redundant __cons;

[7]`http://sourceforge.net/projects/kissfft`

that is, KissFFT will perform consecutive __cons for data that could not have been updated in between. As shown in the figure, hand-optimizing away redundant cons in the main loop yielded some improvement. Second, the added slop results in poor cache behavior, as far fewer complex numbers in the array would be hot in the cache. The figure shows the effect of setting the slop to 0, effectively just adding the version field to the **struct**. Avoiding redundant __cons reduces the DSU penalty to 100%, eliminating the slop reduces the DSU penalty to 78%, and combining the two techniques yields a final DSU overhead of only 42%.

We believe that in the future we could leverage static analysis in order to avoid introducing redundant __cons, and could explore different updateable type representations (such as the hybrid solution described in Section 3.2.2) for reducing the overhead of the slop.

Note however, that KissFFT belongs to a category of performance-critical applications where the cost of DSU might outweigh the benefits; we discuss other such applications in Section 7.1. The point of our KissFFT experiments is to explore the cost of DSU in CPU-intensive applications in which uses of named types abound.

3.6.2 Memory Footprint

Type wrapping, function indirection, version checking and loop extraction all consume extra space, so updateable applications have larger memory footprints. Figure 3.9 reports memory footprints for the four scenarios, with quartiles as error

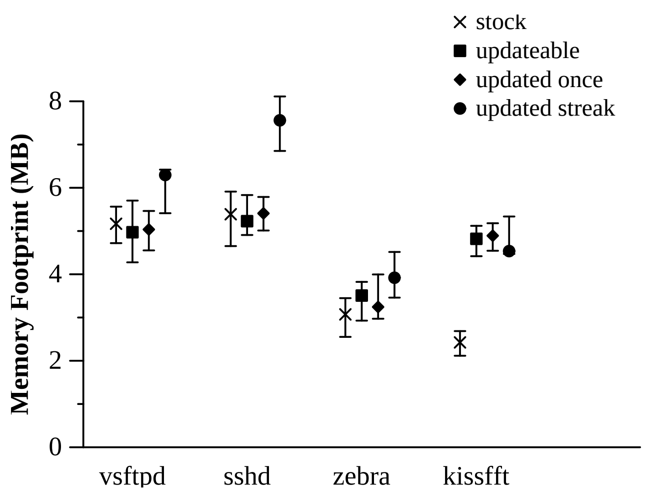

Figure 3.9: Memory footprints.

bars.[8] Measurements were made using `pmap` at the conclusion of each throughput benchmark.

For the updateable and updated cases, the only significant increase is displayed by KissFFT. The explanation is quite simple: KissFFT uses a large number of **struct**s whose size grows by a factor > 2 due to type wrapping. The footprint increases for Vsftpd, Sshd and Zebra are overshadowed by OS variability.

However, for the streak updates, the median footprint increase (relative to the stock version) is 21%, 40% and 27% for Vsftpd, Sshd and Zebra, respectively. The larger footprint increase for streak updates is expected, since dynamic patches for three years worth of updates are added into the memory space of the running program and never unloaded (Section 3.4).

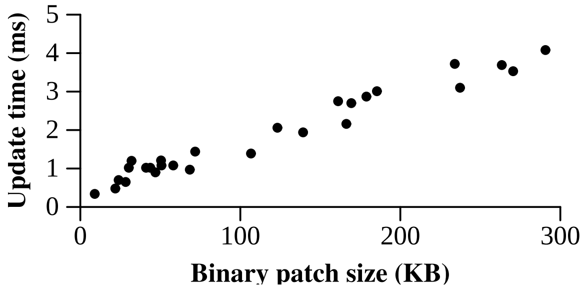

Figure 3.10: Patch application times.

3.6.3 Service Disruption

One of the goals of DSU is to avoid service interruption due to the need to apply software patches. By applying these patches on-line, we preserve useful application state, leave connections open, and sustain service. However, the service will still be paused while new patch files are loaded, and service could be degraded somewhat due to the application of type transformers at patch time and thereafter.

Figure 3.10 illustrates the delay introduced by applying a patch; the delay includes loading the shared object, performing the dynamic linking and running the state transformer (type transformation time was hard to measure, and likely very small, and so is not included). The figure presents measurements for every patch to all of our program versions, and graphs the elapsed time against the size of the patch object files. We can see that patch application time increases linearly with the size of the patch. In terms of service interruption, DSU is minimally intrusive: in all cases, the time to perform an update was under 5 milliseconds.

[8]We present memory footprint data using error bars (as opposed to just numbers as we did for performance experiments) because there is a lot of variability in the data.

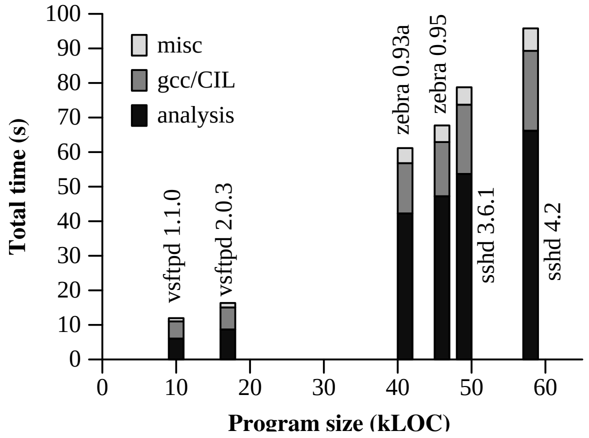

Figure 3.11: DSU compilation time breakdown for updateable programs.

3.6.4 Compilation

The time to compile the first and last versions of our benchmarks is shown in Figure 3.11. The times are divided according to the analysis time, parsing and compilation time, and remaining tasks. In general, the majority of the overhead is due to the safety analyses (whole program, constraint-based analyses). The overhead consists of time spent in the updateability analysis, the AVA analysis and solving constraints introduced in these analyses using Banshee [61].

Given that Ginseng is only needed in the final stages of development, i.e., when the application is about to be deployed or when a patch needs to be generated and compiled, this seems reasonable.

3.7 Acknowledgments

The design, implementation and evaluation of Ginseng on single-threaded programs are the result of joint work. Gareth Stoyle, Michael Hicks, Gavin Bierman,

and Peter Sewell developed Proteus [106], an imperative calculus that forms the base of Ginseng's update type safety. Gareth Stoyle and I worked together on implementing Proteus for the full C language, and on the initial design and implementation of the Ginseng compiler. Gareth Stoyle and Michael Hicks designed and implemented the abstraction-violating alias analysis. Michael Hicks wrote the initial support for code extraction. Manuel Oriol wrote the code that generates type transformers and helped test the preliminary versions of Ginseng. Michael Hicks and Manuel Oriol made the source code changes required to make Vsftpd updateable, and wrote the updates for Vsftpd.

3.8 Conclusion

This chapter has presented the implementation of single-threaded Ginseng, a system for updating C programs while they run. We have shown that Ginseng can be used to easily update realistic C programs over long stretches of their lifetimes, with only a modest performance decrease. Our system is arguably the most flexible of its kind, and our novel static analyses make it one of the most safe. Our results suggest that dynamic software updating can be practical for upgrading running systems. In Chapter 4 we show how we have extended Ginseng to handle multi-threaded programs, along with our experience updating several multi-threaded servers.

Chapter 4

Multi-threaded Implementation and Evaluation

Chapter 3 presented Ginseng's implementation for single-threaded programs and its evaluation on several server programs. In this chapter we show how Ginseng supports multi-threaded programs. Our goal is to provide DSU support that is as flexible and safe as Ginseng's single-threaded approach.

4.1 Introduction

While some prior work on DSU has considered multi-threaded programs, no prior system considers the question of safety in any depth—either no automatic support is provided, leaving the problem entirely to the programmer, or the automatic support is insufficient to establish safety (see Section 6.1.3).

The main technical challenge we address is to ensure updates can be applied in a timely fashion, while providing certain safety guarantees. The key concept that helps us accomplish our goal of balancing safety and availability is that of *induced update points*. Similar to our approach for updating single-threaded programs (Section 3.3.5), we allow programmers to designate update points via `DSU_update()`. We call these *semantic update points*, because they are points chosen by the programmer so that writing an update is straightforward, e.g., where the global state is consistent. The update, however, can take place in between semantic update points, at an induced update point. Our system enforces that an update *appears* to execute at an semantic update point, and that execution is *version-consistent*. This means that even if a code update takes place in between two semantic update points, the execution trace can be attributed to exactly one program version.

We find that semantic update points serve as a useful mechanism for reasoning about update safety, while induced update points permit far greater flexibility in choosing when an update takes place. In particular, programmers can think of updates as possibly occurring at semantic update points only, while the run-time system can actually apply the update at any time so long as it maintains this illusion. This flexibility is crucial in being able to update multi-threaded programs, since it allows us to apply updates without imposing many synchronization constraints on threads.

Section 4.2 introduces our idea of semantic update points and explains how we implement them using the contextual effects analysis described in Chapter 5. Section 4.3 provides further details about our implementation in Ginseng and strategies for reaching a safe update point, and in particular demonstrates that it our system is able to apply an update quickly without compromising safety. Section 4.4 describes our experience using our implementation to update three multi-threaded servers. Section 4.5 measures the performance of our approach, and shows that overhead introduced by update support is detectable using micro-benchmarks but negligible in more realistic scenarios.

4.2 Induced Update Points

The manual enumeration of a few update points works well for single-threaded programs. However, in a multi-threaded program, an update can only be applied when *all* threads have reached a safe update point. Since this situation is unlikely to happen naturally, we could imagine interpreting each occurrence of `DSU_update()` as part of a *barrier*—when a thread reaches a safe update point, it blocks until all other threads have done likewise, and the last thread to reach the barrier applies the update and releases the blocked threads.

Unfortunately, because all threads must reach safe points, this approach may fail to apply an update in a timely fashion. With only one or two update points per thread, each thread may take a while to reach its safe point, and with many threads and few processors, it will take even longer for all threads to do so. More problematic are threads that are blocked, on I/O or on a condition variable, say, since they may take a while to get unblocked. In the worst case, a thread could be delayed indefinitely by the update protocol itself. For example, a thread t might reach the barrier while holding a lock that another thread t', yet to reach the barrier, is blocked on. To avoid deadlock we could imagine causing threads to sleep, rather than block, at update points. But then there is no guarantee the protocol will converge (essentially resulting in a kind of livelock). In all these cases, the update protocol degrades the normal application's performance as its threads are blocked or delayed.

Note that several systems that support dynamic updates to multi-threaded programs employ the activeness check we explained in Section 3.3.4: an update cannot take place if it affects code that is actively executing, i.e., is referenced by the stack of a running thread [23, 24, 1]. The problem is that programmer-specified update points are usually in active code, e.g., a long-running loop. Our approach combines programmer annotations with static analysis and a runtime protocol that permits updates at programmer-specified update points, but increases the chances that a safe point can be reached in the presence of multiple threads.

Given a program in which the programmer has designated safe update points using DSU_update() calls, we will insert some more update points in between,[1] which we call *induced update points* (these can be viewed as calls to a function DSU_induced_update()). The additional update points will provide more opportunities for threads to reach safe points, and thus the update should be able to take effect more quickly. *The key feature of induced update points, as enforced by the run-time system and a compile-time static analysis, is that if an update takes place while a thread is stopped at an induced update point, the program's execution will still appear as if the update took place at a semantic update point instead.* More precisely, it will appear the update took place at the previously-reached semantic update point, or one of the semantic update points that could occur subsequently.

The key benefit of induced update points is that the programmer is able to write the update code as if it will be applied at semantic update points, even though it could happen at potentially many more program points. This allows us to better

[1]At the moment, induced update points are manually inserted, but with some more engineering the compiler could insert them automatically.

balance safety with availability (in Section 5.3.6 we present experimental results that show how induced update points increase availability).

We implement induced update points by ensuring they preserve *version consistency* between semantic update points. That is, we can view each semantic update point as beginning (or ending) a *transaction* such that the execution of the transaction is version-consistent: even if an update takes place while the transaction executes, the execution nevertheless can be attributed to either the old or new version. In Chapter 5 we explain transactions and version consistency in detail.

Given these high-level ideas, we explore compiler and run-time system support for implementing induced update points. We consider two possible approaches, which we dub the "barrier approach" and the "relaxed approach". In the former, we still require all threads to be at update points (semantic or induced) when the update takes effect, which we can force by performing barrier synchronization, for example.

The relaxed approach is similar to barrier approach, but we no longer require threads to actually be stopped at an update point when an update takes effect. Instead, a thread can "check in" its effects (restriction on the form of the update at a particular program point) and then proceed—thus, `DSU_update()` and `DSU_induced_update()` no longer block. An available update may proceed, as long as it does not conflict with the combined effects of all the checked-in threads.

We now proceed to describing the barrier and relaxed approaches in detail, along with the static analysis and runtime support required to implement them.

```
DSU_update(); // semantic up. point
              //  { } = α       {f,g,h,i,k} = ω
f ();         //  {f }          {g,h,i,k}
g = 42;       //  {f,g}         {h,i,k}
h ();         //  {f,g,h}       {i,k}

i ();         //  {f,g,h,i}     {k}
k ();         //  {f,g,h,i,k}   { }

DSU_update(); // semantic up. point
```

(a) Original program
and contextual effects

```
DSU_update(); // semantic up. point
DSU_induced_update({},{f,g,h,i,k});
f ();
g = 42;
h ();
DSU_induced_update({f,g,h},{i,k});
i ();
k ();
DSU_induced_update({f,g,h,i,k},{});
DSU_update(); // semantic up. point
```

(b) Barrier approach

```
DSU_update(); // semantic up. point
DSU_checkin({f,g,h},{f,g,h,i,k});
f ();
g = 42;
h ();
DSU_checkin({f,g,h,i,k},{i,k});
i ();
k ();
DSU_checkin({f,g,h,i,k},{});
DSU_update(); // semantic up. point
```

(c) Relaxed approach

Figure 4.1: Contextual effects and their use for implementing induced update points.

4.2.1 Barrier Approach

In the barrier approach, after an update has been requested, a thread blocks at an update point (induced or semantic) if it does not conflict with the update. When all threads are blocked, we can perform the update. The question is how to determine, once a thread reaches an induced (or semantic) whether the update is safe; our definition of safe is that the update looks like it was applied at a semantic update point for each thread. We make this determination using information from a static analysis that takes into account programmer-designated semantic update points.

The static analysis first performs a standard *effect inference* on the program. In a traditional effect system, the *effect* ε of some program expression e characterizes an aspect of e's non-functional behavior, for example the names of locks e acquires, or the abstract names of memory locations e dereferences. For enforcing version consistency, an effect consists of the names of functions that are called and the names of global variables that are read or written. For example, the effect of the block { `f(); g = 1 ; h();` } where `g` is a global variable would contain {`f,g,h`} because functions `f` and `h` are called, and `g` is written to. Functions `f`, and `h` may contain additional effects, in which case these effects would be included in the effect of the entire block.

Next, we compute a generalization of traditional effects which we call *contextual effects* (explained in more detail in Section 5.2).The contextual effect of an expression e consists of a three-tuple $[\alpha;\varepsilon;\omega]$, where ε is the effect of e, as above;

α is the *prior effect*, which characterizes the computation since the last semantic update point up to (but not including) e; and ω is the *future effect*, which characterizes the computation following e, up until the next semantic update point. Thus, the contextual effect of statement g = 1; within block { f(); g = 1; h(); } would be $[\{\mathsf{f}\};\{\mathsf{g}\};\{\mathsf{h}\}]$. Here, $\alpha = \{\mathsf{f}\}$ because f is called prior to the write to g, and $\omega = \{\mathsf{h}\}$ because h is called following the write to g.

In Figure 4.1 (a) we present a sample program with two semantic update points and several uses of functions and global variables in between. Contextual effects for the sample code are listed in comments, on the right hand side of Figure 4.1 (a). If the listed code appears in a function foo, then foo will be included in prior and future effects for the scope of the function but for clarity, we omit the name of the enclosing function from our presentation.

We can use contextual effects to enforce version consistency as follows. First, the compiler computes the contextual effect of each induced update point, and passes the resulting prior and future effects to DSU_induced_update at run-time (i.e., the DSU_induced_update() call is changed to be DSU_induced_update(α, ω, D), where D is the capability required for enforcing type-safety as described in Section 3.3.1). This is illustrated in Figure 4.1 (b). Second, when an update that changes definitions u becomes available, the system barrier-synchronizes all threads on safe update points. An update point is safe in one of two situations:

1. For all threads t_i, $u \cap \omega_i = \emptyset$ where ω_i is the future effect of thread t_i. In this case, the update will appear as if it took place at the next semantic update

Program	Trace	Program	Trace
DSU_update();		DSU_update();	
f();	f, {1}	f();	f, {1,2}
g = 42;	g, {1}	g = 42;	g, {1,2}
h();	h, {1}	h();	h, {1,2}
$\leadsto$ *update* f,g		$\leadsto$ *update* i,k	
i();	i, {1,2}	i();	i, {2}
k();	k, {1,2}	k();	k, {2}
DSU_update();		DSU_update();	
(a) Roll forward update		(b) Rollback update	

Figure 4.2: Examples of version consistent updates.

point.

2. For all threads t_i, $u \cap \alpha_i = \emptyset$ where α_i is the prior effect of thread t_i. In this case, the update has not changed any definitions the thread has already accessed, and thus, even if it changes definitions that the thread will access subsequently, the entire execution will appear as if the update took place at the prior semantic update point.

In database parlance, the first condition results in a *roll forward* semantics, since the execution is as if due to the old version, while the second condition results in a *rollback* semantics, since the execution is as if due to the new version. We will heretofore refer to the prior and future effects together as *VC effects* and to the safety conditions 1. and 2. above as the *VC check*.

To illustrate how these conditions enforce version consistency, in Figure 4.2 we show two possible updates to the program in Figure 4.1, along with a program *trace* that shows the version at which a function is executed or a global variable is accessed; this helps us determine whether the execution is version consistent. Figure 4.2 (a) shows the execution trace if an update to function f and variable g is applied after the call to h. We see that f, g and h appear in the trace with version set {1}, because they are accessed prior to the update. Functions i and k appear in the trace with version set {1,2} because they are not changed by the update, so their definition is the same in both versions. We can now illustrate the VC check. Figure 4.1(a) shows the contextual effects at each program point, and we can see that at line 6 we have $\omega =$ {i,k}. So the VC check $u \cap \omega = \emptyset$ is satisfied, because $u =$ {f,g} and $\omega =$ {i,k}. Therefore, the update is version consistent because the execution of the block between the two semantic update points can be attributed to a single version—version 1. We call this update a roll forward update, because the update appears to have been applied at the end of the block (second semantic update point).

Similarly, Figure 4.2 (b) shows the execution trace if an update to functions i and k is applied after the call to h. We see that the trace is consistent again, since the execution can be attributed to version 2. At the point where the update is applied, the VC check $u \cap \alpha = \emptyset$ is satisfied, because $u =$ {i,k} and $\alpha =$ {f,g,h}. Therefore, the update is version consistent because the execution of the block in between semantic update points can be attribute to a single version—version 2. We call this update a roll back update, because the update appears to have been applied at the beginning of the block (first semantic update point).

In Figure 4.3 we present an example where version consistency is violated. At the point where the update is applied, we have $\alpha =$ {f,g,h}, $\omega =$ {i,k} and $u =$ {h,i}.

Program	Trace
DSU_update();	
f();	f, {1}
g = 42;	g, {1}
h();	h, {1}
$\leadsto$ *update* h,i	
i();	i, {2}
k();	k, {1,2}
DSU_update();	

Figure 4.3: Example of a version inconsistent update.

Therefore, the VC check fails (we have $u \cap \omega \neq \emptyset$ and $u \cap \alpha \neq \emptyset$) and the update is deemed unsafe. Indeed, we can see that if we apply the update, the trace cannot be attributed to a single program version.

To understand how this approach performs in practice, we have implemented a thread synchronization protocol that builds on it. In this protocol, semantic update points and induced update points are no-ops if an update has not been requested. If an update is requested and the current thread's restriction is compatible with the update, the thread blocks until all other threads have reached a semantic update point or an induced update point. After all threads are blocked, we apply the update. The results of running the experiments on our three test applications are discussed in detail in Section 4.5.1.

4.2.2 Relaxed Approach

The main problem with the barrier approach is that blocking all threads until they reach safe update points may create an undue delay, and even deadlock. For example, if thread T blocks at an induced update point while holding lock L, then any other thread that wants to acquire L cannot make progress and reach one of its semantic update points to allow the update to proceed.

To avoid blocking, we can adapt the barrier approach as follows. Instead of calling DSU_induced_update() with its prior and future effect, we call a different function, DSU_checkin(), that registers the union of the contextual effects computed at all program points between this and the next call to DSU_checkin(). In effect, DSU_checkin() stands in for a series of induced update points from its call site until the next DSU_checkin() or semantic update point. After performing the registration, the thread may continue running—even if an update becomes available prior to reaching the next update point, the update may take effect so long as it satisfies the safety condition described above: for all threads t_i, $u \cap \omega_i = \emptyset$ or $u \cap \alpha_i = \emptyset$. If a thread reaches a check-in point or a semantic update point while an update is in progress, it pauses at that point until the update is finished, and then continues (at the new version).

To see how check-ins work, consider the example in Figure 4.1 (c), which shows the check-in points explicitly. The prior effect of the first DSU_checkin call on line 2 is $\alpha = \{\}$, but notice that we check in $\alpha = \{\mathsf{f,g,h}\}$ instead—this is because we now allow the possibility that an update could occur on line 3, 4, 5, or prior to the call to DSU_checkin on line 6, and thus the checked-in α must approximate the prior effect at these program points. Generally speaking, at each check-in point we register prior effect $\alpha \cup \varepsilon$, where α is the prior effect at that check-in point, and ε is the effect of the code between that point and the next check-in point, or the next semantic update point, whichever comes first. The second argument to DSU_checkin, the future effect, contains the future effect ω only, because it over-approximates the future effects of subsequent statements (up to the next semantic update point). For example, the future effect $\omega = \{\mathsf{h,k}\}$ at line 6 is a sound approximation of future execution, should an update be applied at lines 7, 8, or prior to the check-in on line 9.

4.2.3 Discussion

An important aspect of both barrier and relaxed approaches is where, and how often, should induced update points (DSU_induced_update, and DSU_checkin, respectively) be placed. Having fewer induced update points reduces runtime overhead, but might impact liveness. Also, in the relaxed approach, because of over-provisioning on future effects, fewer check-in points means stronger restrictions on what can be updated, which is detrimental to liveness as well. Currently, we manually place induced update points at the beginning and end of each stage in a thread loop body (Section 4.3.1). This strategy results in 3-4 induced update points per thread.

Note that we could also use an automated, adaptive scheme for choosing induced update points. If the runtime system observes that the current induced update point granularity is not sufficient, we can easily construct a "gratuitous" update whose only purpose is to replace the current function with one having more induced update points (Section 4.4).

We now present the protocol that implements the check-in based relaxed ap-

```
// per−thread restrictions
typedef struct {
   set $\alpha$; set $\omega$; set $D$;
} thread_restr ;

thread_restr restriction [];
rwlock restriction_mutex ;
mutex update_mutex;
volatile bool update_requested = 0;

// elements changed by the update
set update_contents;

bool conflicts ( rst [], u)
{
   bool res = false ;
   for (i = 0; i < n_threads; i++) {
      if ( rst [i ]. $D \cap$ u $\neq \emptyset$ ||
            ( rst [i ]. $\alpha \cap$ u $\neq \emptyset$ &&
            rst [i ]. $\omega \cap$ u $\neq \emptyset$))
      { res = true; break; }
   }
   return res ;
}

void DSU_checkin($\alpha$, $\omega$, $D$)
{
   i = thread_self ();

   read_lock ( restriction_mutex );
   restriction [i] = {$\alpha$, $\omega$, $D$};
   unlock( restriction_mutex );

   leader_selection ();
}

void leader_selection ()
{
   if (update_requested) {
      if ( trylock (update_mutex) == OK){
         // thread 'i' is leader
         leader ();
      }
      else {
         // thread 'i' is follower
         follower ();
      }
   }
}

void leader ()
{
   write_lock ( restriction_mutex );
   if (! conflicts ( restriction ,
                        update_contents)) {
      apply_update ();
      update_requested = 0;
   }
   // else : couldn't apply the update yet
   // give other threads a chance to be leader

   // update protocol finished
   unlock( restriction_mutex );
   unlock(update_mutex );
}

void follower ()
{
   // do nothing; leader will check for
   // safety and apply the update
}
```

Figure 4.4: Pseudo-code for safety check and update protocol routines.

proach described above. The pseudo-code for this protocol is shown in Figure 4.4. We keep a global restriction array protected by a reader-writer lock (lines 2–7). If an update is signaled, the flag update_requested (defined on line 8) is set, and the names of the patch elements are written into update_contents (defined on line 12). This set is used by the safety check in function conflicts (lines 14–24). A thread reaching a check-in point (line 25) checks-in its restriction first(lines 29–31), and, if an update has become available, calls leader_selection to initiate or joins the update protocol. We use the update_mutex to ensure only one thread is leader. If an update has been requested and no other thread has taken the lead (by acquiring update_mutex), the thread is declared a *leader*; it will possibly perform the update. If update_mutex is taken, the current thread will be a *follower*. The leader code (lines 50–64) checks whether the update is safe for all threads by comparing the update contents with each thread's restrictions (effects and capability). The follower (lines 66–70) simply waits until the leader is done, so as to avoid changing a thread's restriction while the leader is busy performing the update safety check. Note that this protocol does not guarantee progress, i.e., it is possible that the leader's conflict check fails. To alleviate this, Ginseng performs a static conflict check at patch compilation time. Check-in points in each thread are verified against update contents and if, for a certain thread, the safety check would always fail at runtime (i.e., all check-in points in that thread conflict with the update), Ginseng notifies the user. In practice, however, we were always able to reach a safe update point within 2 seconds, and typically we could do so in under 8 ms. The results of running the experiments on our three test applications are presented in Section 4.5.1.

A potential shortcoming of our approach is the difficulty of writing state transformers in the presence of I/O. We use contextual effects to implement induced update points and provide the "illusion" that an update is applied at a semantic update point. This illusion, however, is only maintained if the program does not store state outside the process (e.g., via I/O), because contextual effects might fail to capture such state. Since the update could be applied at induced update points in the middle of several I/O operations, the programmer might need to adjust the state transformer based on whether certain I/O operations have completed or not. We have not encountered this situation in practice.

4.3 Implementation

The compiler and runtime system are built on top of single-threaded Ginseng, presented in Chapter 3. We will now talk about how we extended Ginseng to handle multi-threaded programs, and what the programmer has to do to prepare multi-threaded programs for compilation with Ginseng.

4.3.1 Replacing Active Code

Updating long-running programs raises the issue of having to update active code. Performing an activeness check (Section 3.3.4) to determine which functions can be updated is problematic because it prohibits updates to functions which contain active code. To cope with this, in Section 3.2.4 we introduced a technique called *code extraction* that permits updates to code on the stack (e.g., long-running loops). Ginseng extracts a programmer-indicated block (loop body), into a separate function, so to apply an update, we only need to wait for the current iteration to finish, rather than having to wait for the loop to terminate.

However, multi-threading complicates things further: since multi-threaded server programs employ threads each running its own loop, to apply an update we would have to wait until each thread has reached the end of its current loop iteration. This condition is difficult to meet, if not impossible. An example would be producer/consumer threads where one thread is blocked while the other is doing work. To solve this problem, we have to permit updates to a loop body *before the thread has completed the iteration.* We accomplish this by partitioning long-running loops into "stages" that are designated for code extraction and hence can be updated independently.

In Section 4.4.1 we show an example of using code extraction to permit updates to the Icecast server, and in Section 4.4.4 we list the number of instances of code and loop body extraction in our test programs.

4.3.2 Concurrency Issues

Since type wrapping changes the representation of updateable (named) types, we have to be careful not to introduce races. A read–read access in a multi-threaded program is race-free without wrapping, but could be problematic once wrappers are introduced. Since `con` functions can potentially call the type transformer to update a value to the current version, suddenly a read–read access can become a write–

Program	**Release**	**Date**	**Size**	**Description**	**Type xform**	**State xform**
			(LOC)		**(count)**	**(LOC)**
Icecast	2.2.0	12/04	25,349			
	2.3.0rc1	08/05	28,593	Major feature enhanc.	23	5
	2.3.0rc2	09/05	28,788	Other	4	1
	2.3.0rc3	09/05	28,796	Other	-	-
	2.3.1	11/05	29,079	Other	7	1
Memcached	1.2.2	05/07	5,743			
	1.2.3	07/07	5,732	Other	1	-
	1.2.4	02/08	6,144	Major bugfixes	3	2
	1.2.5	03/08	6,345	Major bugfixes	2	-
Space Tyrant	0.307	10/06	18,738			
	0.316	10/06	19,077	Minor feature enhanc.	11	-
	0.319	10/06	19,399	Minor feature enhanc.	2	-
	0.331	04/07	19,526	Minor bugfixes	-	-
	0.335	05/07	19,753	Minor feature enhanc.	-	6
	0.347	08/07	19,979	Minor bugfixes/enhanc.	1	2
	0.351	10/07	20,223	Minor feature enhanc.	2	1

Table 4.1: Application releases.

write access. To prevent this from happening, `con` functions use per-type locks and double-checked locking to make the version check fast while guaranteeing that type transformers are invoked atomically. Another way to avoid this problem is to convert data eagerly, at update time, an approach we might consider in future work.

4.4 Experience

We used Ginseng to dynamically update three open-source multi-threaded programs: the Icecast streaming media server,[2] Memcached, a high-performance, distributed memory object caching system,[3] and the Space Tyrant multiplayer gaming server.[4] We chose these programs because they are long-running, maintain soft state that could be usefully preserved across updates, and exhibit a variety of threading

[2]`http://www.icecast.org`

[3]`http://www.danga.com/memcached/`

[4]`http://spacetyrant.com/`

Program	**Changes**			
	Functions		Types	Global variables
	Proto	Body		
Icecast	10	292	25	1
Memcached	14	118	6	6
SpaceTyrant	0	107	11	5

Table 4.2: Changes to applications.

models. In the remainder of this section, we describe the evolution of these programs during the period we considered, as well as changes we had to make to prepare the programs for compilation with Ginseng.

Table 4.1 shows release and update information for each program. Columns 2–4 show the version number, release date and program size for each release. Column 5 contains the nature of individual releases.[5] Column 6 shows the number of type transformers for that specific update, while column 7 presents the size of the state transformer (in lines of code); '-' means no type or state transformers were needed for a particular release.

Table 4.2 shows the cumulative number of changes that occurred to the software over that span. "Types" refers to **struct**s, **union**s and **typedef**s. These statistics reinforce the findings of Chapters 2 and 3: a dynamic software updating system must support changes, additions, and deletions for functions, types and global variables if it is to handle realistic software evolution.

For each program, we downloaded several releases, converted the programs to updateable applications, wrote dynamic patches, applied all patches in release order

[5]As described at `http://freshmeat.net/` in the case of Icecast and Memcached; for Space Tyrant we provide our own characterization.

and performed testing/benchmarks. In Section 3.5 we have laid out guidelines for preparing C programs for conversion into updateable C programs; we inform the Ginseng compiler about long-running loops and, if necessary, override Ginseng's safety analysis via user-inserted annotations. In this chapter we will focus on programmer effort and annotations that are specific to multi-threaded programs: identifying semantic update points and picking check-in points. We will discuss how we chose semantic update points and check-in points for each program later (Sections 4.4.1, 4.4.2, and 4.4.3).

Our experience with the multi-threaded servers we have considered is that, just like the single-threaded servers we have presented in Chapter 3, they perform a few high-level operations whose boundaries are easily identified as semantic update points. Examples of such operations are processing one event, accepting and dispatching a client connection, etc. In Section 4.4.1 we will present a concrete illustration of how we picked semantic update points and check-in points in the Icecast server. We have also found that semantic update points tend to remain stable within a program's overall structure, even as the overall implementation changes. Nevertheless, the programmer does not have to pick the right induced update point at the outset. If, for example, it turns out that in a function, more check-in points are needed, or code extraction boundaries are too coarse, it is easy to construct a "gratuitous" dynamic update of the function with more check-in points and finer-grained code extraction boundaries.

4.4.1 Icecast

Icecast is a streaming media server—a popular solution for building Internet radio stations. Updating Icecast on the fly enables media content providers to keep their streams alive 24/7, yet be protected with the latest security fixes. We considered five consecutive Icecast releases spanning 49 weeks (Table 4.1).

Icecast uses a fixed number of threads, each performing separate duties: accept connection, handling incoming connections, reading from a media source, keeping statistics, etc. The principal threads are presented in Figure 4.5. Lines marked with '*' are annotations we had to insert; these are used by the Ginseng compiler, and denote long-running loops, semantic update points, check-in points, etc.

For each thread, we placed a semantic update point at the beginning of that thread's long-running loop (lines 3, 20, and 49, respectively). In the loop bodies, we use check-ins (DSU_checkin) to snapshot the thread's current effect, and code extraction (DSU_extract) to permit updates to code on the stack. We placed check-in points and extraction boundaries around each separate *stage* in loop bodies.

The *connection accept* thread's operation is split into two stages: checking for termination requests (lines 6–7) and waiting for incoming connections (lines 11–14). When a new connection is opened, it packs the information in a con structure, and passes it to the pool of connection handler threads. A *connection handler* thread takes an accepted connection, uses an HTTP parser to parse the client's request, and dispatches the request according to the request type, con_type. If, for instance, the client requests some statistics, handle_stats_request () will fire a new thread that sends statistics information to the client (line 35). If the client requests a file, handle_get_request () (inlined here for clarity, lines 38–41) will create a new fclient structure and add it to the file serving thread's working queue. The *file serving* thread's operation is also split into two stages: 1) tending to active clients (lines 52–57), i.e., sending listeners chunks of file contents via HTTP connections, and 2) moving new clients which were generated by the connection handler and added to pending_list (lines 63–64) into the active client list, active_list (lines 65-66).

4.4.2 Memcached

Memcached is a high-performance, distributed memory object caching system that is used on popular sites such as Slashdot or Wikipedia to store pre-rendered HTML pages without having to access the database and render pages individually for each client (since the database is a bottleneck in these situations).

Updating Memcached on-the-fly is essential to maintaining a high web server throughput; taking Memcached down to install the next version will flush the in-memory cache and cause degraded operation while the cache fills up again in the new version. We considered four consecutive Memcached releases spanning 10 months (Table 4.1). Multi-threading was introduced in version 1.2.2, so we did not consider releases prior to that.

Memcached uses a homogeneous threading model, where all application threads (a user-configurable number) perform the same fixed task. Memcached uses the *libevent* library [95] to process client requests; each thread is associated with a

```
  // accepts connections
  while (1) {
*   DSU_update();
*   DSU_checkin();

    if (! global_running ())
      break;

*   DSU_checkin();
*   DSU_extract {
      con = _accept_connection ();
        if (con) {
          _add_connection(con);
        }
*   }
*   DSU_checkin();
  }
```

(a) Connection accept thread

```
  // parses and handles connections
  while (1) {
*   DSU_update();
*   DSU_checkin();

    if (! global_running ())
      break;

*   DSU_checkin();
*   DSU_extract {
      con = _get_connection ();
      header = read_header(con→ sock);
      parser = httpp_create_parser (header);
      con_type = httpp_getvar( parser ,
                                    " protocol" );
      switch(con_type) {
        case STATS:
          handle_stats_request (con, parser );
          break;
        case GET:
          //handle_get_request (con, parser )
          fclient = client_create (con, parser );
          fclient → next = pending_list;
          pending_list = fclient ;
          break;
      }
*   }
*   DSU_checkin();
  }
```

(b) Connection handler thread

```
  // sends data to clients
  while ( run_fserv ) {
*   DSU_update();
*   DSU_checkin();
*   DSU_extract {
      fclient = active_list ;
      while( fclient != NULL) {
        client_send_bytes ( fclient , ...);
        fclient = fclient → next;
      }
*   }
*   DSU_checkin();
*   DSU_extract {
      while ( run_fserv ) {
        if ( pending_list ) {
          to_move = pending_list ;
          pending_list = pending_list → next;
          to_move→ next = active_list;
          active_list = to_move;
        }
        if ( poll ( client_file_desc , ...)>0)
          break;
      }
*   }
*   DSU_checkin();
  }
```

(c) File serving thread

Figure 4.5: Icecast structure. DSU annotations are marked with '*'.

separate libevent instance called *base*; events belonging to the same base will be processed by the same thread. To ensure that the execution associated with processing an event is version-consistent, we placed a semantic update point just prior to the start of processing an event, and another semantic update point after finishing processing the event.

4.4.3 Space Tyrant

Space Tyrant is a multi-threaded gaming server. According to a report by In-Stat, a market research firm, on-line gaming is a multi-billion dollar industry, and expected to grow rapidly [56]. Therefore, continuous game server operation is essential for companies providing on-line gaming services. Since Space Tyrant has a release model based on very frequent, incremental releases (a release every couple of days/weeks), we considered a year in its lifetime, corresponding to versions 0.307 to 0.351. Given the small magnitude of per-release changes in this model, instead of mapping one update per release, we decided to consolidate several adjacent releases, so we could have type and/or global variable changes for each update, resulting in 7 releases (Table 4.1).

Space Tyrant uses a mixed threading model: three fixed threads (for managing the game state, accepting new connections and performing backups) and two threads per each client, one dealing with user input, one dealing with output from the server to the client.

Similar to Icecast (Section 4.4.1), we associated one iteration of a thread's processing loop with two semantic update points, and used 3–4 check-in points per loop, where check-in points delimit stages within the loop.

4.4.4 Source Code Changes

When building updateable applications with Ginseng, the programmer might need to intervene at two stages: when preparing the source code for compilation with Ginseng, and when creating dynamic patches. We present details on programmer effort (annotations or lines of code) for each of these stages, in turn, for our three test programs.

Changes to applications. Constructing updateable versions of each application required inserting some annotations and making a few changes to its source code, mainly indications to the Ginseng compiler. These changes have to be applied to each program version, but once we figured out the changes we had to make to the first version, we were able to automate the process and use `patch` to change the following versions.

The changes and annotations are presented in detail in Table 4.3. The second column shows the number of long-running loops designated for extraction. Identifying long-running loops is easy, as each long-running thread essentially executes a loop. We identified 11 such loops in Icecast and 7 in Space Tyrant; loop body extraction was not necessary for Memcached because iteration is done externally in libevent. The reason why the number of extracted loops is higher than the number of runtime threads is that in Icecast, some threads are short-lived, and only used under certain configurations (we omitted these short-running threads from Figure 4.5 for brevity). In Space Tyrant, which has five distinct kinds of threads, we used loop extraction to extract two nested loops, so the total number of extracted loops was 7.

The third column shows the number of code blocks designated for extraction; the procedure we used to identify such blocks was to find long-running stages within the bodies of thread loops, and mark each stage for extraction.

The fourth column shows the number of semantic update points. We placed a semantic update point at the beginning of each thread loop body for Icecast and Space Tyrant, and in the case of Memcached, the two semantic update points delimit the processing of one event. The fifth column shows the number of check-in points. The rest of the changes (the number of lines of code changed is presented in column 6) consisted of:

- Directives to the compiler to override the conservativity of the safety analysis. Ginseng's analysis does not model existentially quantified types, so even though they are safely used, Ginseng reports a possible safety violation. We had to add two such directives in Icecast, three in Memcached and two in Space Tyrant.
- Changing four low-level, type unsafe, field access macros in Memcached into function calls, so Ginseng's compiler and safety analysis can reason about them.

Program	**Annotations**				**Source code changes (LOC)**
	Extract		Semantic upd. points	Check-in	
	loop	code			
Icecast	11	18	11	34	42
Memcached	0	0	2	2	23
SpaceT	7	16	5	32	19

Table 4.3: Annotations and changes to source code.

Adjusting auto-generated patches. Manual intervention was also required to inspect (and complete, where necessary) the auto-generated type transformers and write state transformers, when needed; across all patches, we had to write 80 lines of code for Icecast, 12 for Memcached and 81 lines for Space Tyrant.

4.5 Experiments

We performed a suite of experiments to evaluate how our approach balances safety and liveness, and to measure the overhead of using our approach to build and run updateable software. We considered the following aspects:

1. *Update availability.* To see how the protocols described in Sections 4.2.1 and 4.2.2 perform in practice, we measured, for each of the updates we have considered (13 in total), the time from the moment an update was signaled to the time it could safely be applied. Our main findings are that the relaxed approach provides higher availability than the barrier approach, and that performing only the activeness check improves availability even more, though at the expense of safety.

2. *Application performance.* We measured the overhead that update support imposes on a application performance, by running stress tests on unmodified and Ginseng-compiled versions of the same program. We also measured the additional overhead that Ginseng imposes on multi-threaded programs, as opposed to single-threaded Ginseng. We found that DSU overhead is modest for the applications we considered.

3. *Memory footprint.* We also measured the update support overhead in terms of memory footprint, again for unmodified applications, and updatable applications compiled with single- and multi-threaded versions of Ginseng. We found the increase to be negligible for Icecast and Memcached (less than 1% and 4%, respectively), but up to 46% for Space Tyrant.

4. *Build time.* We also measured the running time of Ginseng to build our test programs using Ginseng, to measure the overhead of compilation and our analyses. In all cases, the time to compile and link the programs was less than 30 seconds.

We conducted our experiments using a client-server setup, where the updateable applications ran on a quad-core Xeon 2.66GHz server with 4GB of RAM running Red Hat Enterprise Linux AS release 4, kernel version 2.6.9. The clients ran on a two-way SMP Xeon 2.8GHz machine with 3.6GB of RAM running Red Hat Enterprise Linux WS release 3, kernel version 2.4.21. The client and server systems were connected by a 100Mbps network. All C code (generated by Ginseng or otherwise), was compiled with `gcc` 3.4.6 at optimization level `-O2`.

		Protocol											
		P-Relaxed		P-Relaxed-*D*		P-OpWait		P-PostRelaxed		P-Barrier		P-Barrier-*D*	
	#Clients →	4	16	4	16	4	16	4	16	4	16	4	16
	Upd. id ↓												
Icecast	0	1,750	1,068	103	102	X	X	17,037	10,243	1,062	945	506	945
	1	2.1	2.1	1.2	1.3	X	X	805	1,233	976	942	971	937
	2	0.7	0.7	0.6	0.6	X	X	603	1,000	478	937	975	940
	3	2.3	2.3	1.4	1.4	X	X	691	1,303	484	941	963	935
Memc- 4	0	0.6	0.7	0.5	0.6	150	21	1.1	1.2	0.9	1	0.8	0.9
	1	1.2	1.2	0.7	0.8	373	29	1.5	1.5	1.7	1.8	1	1
	2	0.8	0.8	0.6	0.6	275	28	1.3	1.4	1	1	0.8	0.9
Memc- 16	0	1	1.1	0.6	0.6	X	1,163	X	2.3	X	1.7	X	1.2
	1	2.8	2.8	0.9	1	X	1,255	X	3.6	X	4.5	X	1.8
	2	1.4	1.4	0.8	0.8	X	3,465	X	2.9	X	2	X	1.3
Space Tyrant	0	3.4	5.2	1.5	1.6	X	X	3,513	3,492	3,505	3,439	3,503	3,472
	1	4.7	9.4	1.3	1.4	X	X	3,524	3,511	3,506	3,456	3,525	3,469
	2	1	3.4	0.4	0.4	X	X	3,545	3,477	3,526	3,514	3,523	3,444
	3	3.9	4.2	2.9	3	X	X	3,621	3,461	3,524	3,445	3,524	3,414
	4	1.4	1.5	1.6	6	X	X	3,553	3,469	3,508	3,459	3,521	3,450
	5	4.4	3.7	0.8	0.8	X	X	3,504	3,482	3,504	3,426	3,528	3,423

Table 4.4: Time to reach a safe update point using various protocols (ms).

4.5.1 Update Availability

To gain additional insights into the trade-offs of different thread synchronization protocols, in addition to P-Barrier and P-Relaxed, the two protocols presented in Section 4.2.1 and Section 4.2.2, we have implemented two other protocols, P-OpWait and P-PostRelaxed. We now proceed to describing these protocols in more detail.

P-OpWait. This is a simple, optimistic protocol. Whenever a thread reaches a check-in point, if an update has been requested, that thread yields by calling sched_yield (alternatively, we could have the thread sleep by calling nanosleep). This gives the other threads a chance to reach their check-in points as well. If the last thread discovers that all threads are at check-in point, we do the safety check and apply the update if it is safe to do so. As we will see shortly, this protocol performs poorly in practice; it usually times out without being able to reach a safe update point.

P-PostRelaxed. In this protocol, check-ins are no-ops if an update has not been requested. Once an update is requested, threads will check-in their restrictions and continue running. Once all threads have checked-in their restrictions at least once, the update protocol is the leader/follower used in P-Relaxed. The advantage of this protocol is low runtime overhead: we only perform check-ins when an update has actually been requested.

In addition to the four protocols presented so far, we also implemented type-safety only versions of P-Relaxed and P-Barrier; we denote these P-Relaxed-*D* and P-Barrier-*D*. The reason we considered these *D*-only protocols was to compare our approach against one based on the activeness check.

The time to reach a safe update point (in ms), is presented for each protocol in Table 4.4. The first column shows the program, while the second column shows the update sequence number, e.g., entry ‘0’ for Icecast corresponds to the update Icecast 2.2.0 → Icecast 2.3.0rc1, entry ‘0’ for Memcached corresponds to the update Memcached 1.2.2 → Memcached 1.2.3, etc.

We ran experiments for each update and measured the time it took the system from the moment the update was signaled to the moment it could safely be applied. We tested two configurations, 4 and 16 concurrent clients. The number of server-side threads varied, depending on the application and number of clients, as we explain below. Icecast has a fixed number of threads (in our configuration this number was 6 in Icecast 2.2.0, and 7 in later versions, respectively), regardless of the number of clients.[6] Memcached has a thread pool with a configurable number of handler threads, independent of the number of clients. We present results for two configurations, one with four server threads (Memc-4), and one with 16 server threads (Memc-16). Space Tyrant uses two threads per connected client, plus three fixed threads that perform housekeeping. To summarize, the number of server-side threads were:

[6]In the first version, the number of connection handler threads is configurable, but in later versions there is only one connection handling thread, so for update # 0 we fixed the number of handlers to one, to keep things consistent with later versions.

- Icecast: 6 for update # 0, and 7 for updates # 1–3.
- Memcached: 4 and 16 for Memc-4 and Memc-16 respectively.
- Space Tyrant: 11 (8 client handlers + 3 fixed) for the 4-client configuration, and 35 (32 client handlers + 3 fixed) for the 16-client configuration.

We are interested in measuring update availability (time between an update is signaled until it can be applied safely), using the six protocols, while the server is under load (since thread activity is likely to obstruct updates from taking effect).

The methodology for each program was to start the server, connect 4 (or 16) clients that are constantly asking for data, and while the server is performing work, send an update request. We then measured the time from the moment an update was requested to the moment the update could be safely applied, or time out after 30 seconds[7]. We performed each experiment 11 times; we report the median time to reach a safe update point for terminating runs. An 'X' entry means that for that specific configuration, none of the 11 runs could reach such a point within 30 seconds. We also measured update loading times (time to load a dynamic patch after reaching a safe point). Since this is not our focus, we omit showing the loading times; they are proportional to patch size, and in all cases were less than 3 ms.

Overall, P-Relaxed (columns 3 and 4 of Table 4.4) performs best. As expected, P-Relaxed-*D* (columns 5 and 6 of Table 4.4) reaches a safe point faster

[7]We have chosen 30 seconds as a time-out value because we want to provide a reasonable update availability guarantee, and reduce the vulnerability window for security fixes (i.e., the time programs run without being patched).

than P-Relaxed because it performs a less strict safety check; however, using P-Relaxed-D is potentially unsafe since it can lead to version consistency violations. P-OpWait (columns 7 and 8 of Table 4.4) performs poorly because it requires all threads to reach check-in points simultaneously, a condition difficult to meet when the server is under load.

Protocols P-PostRelaxed, P-Barrier and P-Barrier-D (columns 9–14 of Table 4.4) require threads to check-in only after an update has been requested, so it takes some time until all the threads have checked in their effects. In the Memc-16 scenario, we have 16 server threads, and activity from only 4 clients prevents each of the 16 thread from being scheduled within the 30 seconds time-out window, hence the 'X' entries. Note that P-Relaxed and P-Relaxed-D do not have this problem because the runtime system is aware of all thread effects at all times: if an update is compatible with each thread, then the first thread to reach a check-in point becomes leader and applies the update. A possible fix for P-PostRelaxed and P-Barrier would be to force a check-in when 1) a thread is being preempted, or 2) prior to a thread entering a blocking system call, so the runtime system can inspect each thread's effects without having to first wait for all the threads to run and reach a check-in point after an update has been signaled. Solution 1 is difficult to implement without modifying the scheduler. Solution 2 can be implemented by inserting check-ins only prior to blocking I/O. However, our performance experiments in Section 4.5.2 show that, in practice, the cost of always doing check-ins is modest, so P-Relaxed provides a good balance between overhead and availability.

The only situation where P-Relaxed takes longer to reach a safe point than a comparatively safe protocol (P-PostRelaxed or P-Barrier) is the update #0 to Icecast, presented in the first Icecast row. P-Relaxed takes 1.75 and 1.06 seconds to reach a safe update point, compared to 1.06 and 0.94 for P-Barrier. The reason why P-Relaxed takes longer to reach safety for this particular update is due to numerous induced update points conflicting with this (particularly large) update. In the relaxed approach, for a term e delimited by check-ins, whose contextual effects are $[\alpha; \varepsilon; \omega]$ we check-in $(\alpha \cup \varepsilon, \omega \cup \varepsilon)$. This effectively prevents anything in ε from being updated while e is being evaluated. For the Icecast update #0, the relaxed approach has to "skip" many induced update points because the safety check fails due to conflicts between u, the update contents, and the ε_i associated with each thread i. In contrast, using the barrier approach, the safety check is more precise, which permits us to reach a non-conflicting update point faster. However, in all other cases P-Relaxed reaches a safe point faster (sometimes orders of magnitude faster) than P-PostRelaxed or P-Barrier.

We can see that performing only the type safety check increases availability, as the time to reach a safe update point is lower for P-Relaxed-D compared to P-Relaxed and for P-Barrier-D compared to P-Barrier. However, in practice we want to enforce version consistency, and the point of this experiment is only to compare our approach to other multi-threaded DSU systems that employ the activeness check. Performing the activeness/type safety check only essentially means we "pretend" that an induced update point is a semantic update point. This is clearly not what the programmer had intended, since global invariants might not be satisfied at an induced update point. As a consequence, applying the update

Config.		Completion time (sec)					
	# Clients →	4			16		
	Compilation →	Stock	Ginseng-ST	Ginseng-MT	Stock	Ginseng-ST	Ginseng-MT
	Application ↓						
Remote	Icecast	11.09	11.08 (-0.09)	11.09 (0.00)	40.50	40.58 (0.20)	40.84 (0.84)
	Memcached	31.64	31.30 (-1.07)	31.58 (-0.19)	86.32	87.08 (0.88)	87.66 (1.55)
	SpaceT	44.54	44.48 (-0.13)	44.49 (-0.11)	44.62	44.51 (-0.25)	44.60 (-0.04)
Local	Icecast	1.64	1.66 (1.22)	1.75 (6.71)	2.65	2.63 (-0.75)	2.70 (1.89)
	Memcached	7.58	7.89 (4.09)	7.92 (4.49)	31.37	32.13 (2.42)	32.55 (3.76)
	SpaceT	34.60	34.62 (0.06)	34.61 (0.03)	44.52	44.65 (0.29)	44.62 (0.22)

Table 4.5: Impact of update support on performance (absolute time, in seconds, and in % relative to the stock server).

based on the activeness check only can lead to errors at update time or later, in the execution of the new program version.

4.5.2 Application Performance

We evaluated the impact of dynamic update support on application performance using two metrics: 1) application-specific benchmarks, and 2) memory footprint. We report the performance results in Table 4.5 and memory results in Table 4.6.

For each application, we measured the performance of its most recent version under three configurations. The stock configuration forms our base for benchmarking, and consists of the application compiled normally, without support for updating. The *Ginseng-ST* configuration is the application compiled with the single-threaded Ginseng compiler, and using a single-threaded runtime system. The *Ginseng-MT* configuration is the application compiled with the multi-threaded Ginseng compiler, and using a multi-threaded runtime system. Comparing the Ginseng-ST and Ginseng-MT configurations is useful for considering the additional overhead that multi-threading support imposes on applications compiled with Ginseng, e.g., check-ins or locking in `con` functions.

For each application, we ran a specific benchmark and measured the completion time and memory footprint (at the completion of the benchmark) in all three configurations: stock, Ginseng-ST, and Ginseng-MT. The memory footprint was measured at the completion of each benchmark.

For Icecast, we measured the time it took the streaming server to serve eight mp3 files to a client, using `wget` as a client. Each file has size 1, 2, ... 8 MB. To eliminate jitter due to disk I/O, we directed `wget` to send both its output, and the downloaded file, to `/dev/null`.

For Memcached, we ran a "slap" test that comes bundled with the server. The test program spawns multiple clients in parallel, each client inserting key/value pairs into Memcached's hash table. We measured the time it took the test program to complete insertion of 50,000 key/value pairs.

For Space Tyrant, we created a scenario file that simulates a client performing 500 random moves across the universe, and spawn concurrent clients running this scenario. We measured the time it took the server to complete serving all the clients.

In Tables 4.5 and 4.6 we report the median completion time and median memory footprint across 11 runs. We ran each benchmark in two setups. The first setup, **remote**, shows the results of running the clients and server on separate machines, a scenario that models how the updatable servers would be used in practice. The second setup, **local**, shows the results of running the clients and server on the same machine (we used the quad-core machine mentioned above). The point of measuring overhead in a local configuration is to factor out network latency and bandwidth

Config.		Memory footprint (MB)					
	# Clients →	4			16		
	Compilation →	Stock	Ginseng-ST	Ginseng-MT	Stock	Ginseng-ST	Ginseng-MT
	Application ↓						
Remote	Icecast	69.83	70.08 (0.36)	72.32 (3.56)	70.36	70.09 (-0.39)	72.84 (3.52)
	Memcached	99.94	100.21 (0.27)	99.93 (-0.00)	223.74	223.42 (-0.14)	223.91 (0.08)
	SpaceT	62.75	90.15 (43.66)	92.04 (46.68)	63.42	90.18 (42.19)	91.81 (44.76)
Local	Icecast	69.71	70.33 (0.89)	72.23 (3.62)	69.86	70.03 (0.24)	73.18 (4.75)
	Memcached	99.39	99.14 (-0.25)	99.67 (0.28)	222.67	223.23 (0.25)	223.05 (0.17)
	SpaceT	63.73	89.85 (40.99)	91.96 (44.29)	62.97	89.44 (42.03)	91.73 (45.68)

Table 4.6: Impact of update support on memory footprint (absolute footprint, in MB, and in % relative to the stock server).

issues (while reducing parallelism of the server). Similar to the update protocol experiments in Section 4.5.1, we report figures for 4 and 16 clients, respectively.

Benchmark completion times are presented in columns 3–8 of Table 4.5. In the remote setup, for Icecast and Space Tyrant, the completion time is similar to the stock server. Memcached is however slower in the 16-thread configuration, with the multi-threaded updatable version 1.6% slower. In the local setup, impact of update support on completion time is higher than in the remote setting; this is because, as expected, update support (e.g., check-ins, function and type indirection) slows down the application and the slow-down cannot be masked by network latency. However, even in this scenario the slowdown is small.

4.5.3 Memory Footprint

Memory footprint overhead is presented in columns 3–8 of Table 4.6. As expected, local or remote setups exhibit the same memory footprint. Update support (function and type indirections, and the Ginseng runtime) increases the memory footprint of application compiled with Ginseng. To quantify this impact, we measured the virtual memory footprint in the stock and updatable configurations using `pmap`. For Memcached the difference is almost imperceptible. For Icecast, the memory footprint increases by up to 4.8% compared to the stock server. For Space Tyrant, the increase is at most 46.7%; the reason why the increase is so large has to do with Space Tyrant and Ginseng interaction. The median memory footprint for the stock version is around 63 MB, while the median memory footprint for Ginseng-compiled versions is around 92 MB. Space Tyrant uses a pre-allocated global array that keeps the game map, divided into 100,000 sectors, and the size of this array is 7.6 MB. The Ginseng compilation scheme allows room for growth in each structure (Section 3.2.2); by default, room for growth is equal to the initial structure size. Because the large array resides within two nested structures, its size becomes 31.2 MB, hence the growth to 92 MB and the 46% increase. This problem could be solved by keeping updatable data by reference or by allocating sector data on demand. Since in our experiments we used at most 8 concurrent clients, each patrolling at most 500 sectors, the actual memory overhead would be at most 11.7% for the 4-threads case, and 13.3% for the 16-threads case.

In almost all cases, the multi-threaded setup (Ginseng-MT) has a slightly higher overhead on the application than the single-threaded setup (Ginseng-ST): around 3% more for Icecast and 4% more for Space Tyrant. This increase is due to extra work and extra memory requirements, e.g., check-ins (Section 4.2.3), and locks for con functions (Section 4.3.2).

Program	**Size (LOC)**	**Build time (sec)**	
		gcc	Ginseng
Icecast	25,452	2.3	27
Memcached	5,752	0.7	4.2
Space Tyrant	10,480	2.6	20.1

Table 4.7: Time to build (compile and link) the test programs.

4.5.4 Compilation Time

To provide a sense of Ginseng's compilation overhead, in Table 4.7 we present the time to compile and link each test program in two configurations. The first, normal configuration, without update support, using `gcc` (which in turn calls `ld`) is presented in column 3. The second configuration shows build time for updateable applications using Ginseng; it consists of compiling C code into updateable C code using Ginseng, then followed by `gcc` and linking together with the Ginseng runtime system. The build times for this case are presented in column 4; the bulk of the time is spent in the safety analyses (Sections 3.3 and 5.3.5), and we imagine the figures could be improved with some more engineering.

4.6 Conclusion

In this chapter, we presented an approach to updating multi-threaded programs while they run, and show how we have implemented this approach in Ginseng. Updating multi-threaded programs is more difficult than updating single-threaded code because of the tension between update safety and update availability. We solve this tension using a novel concept of induced update points, that allow us to perform updates to multi-threaded while providing the same safety guarantees as in the single-threaded case, while still allowing the update to be applied promptly. We evaluated our approach on three realistic multi-threaded servers. We found that programmer effort for building updateable versions of these applications was modest, and experiments show that update support does not significantly impact application performance.

Chapter 5

Version Consistency

In Chapter 4 we showed how we can perform timely updates to multi-threaded programs by using version consistency—ensuring that certain code blocks are atomic with respect to updating. In this chapter we provide formalisms for reasoning about, and soundness proofs for, version consistency.

5.1 Introduction

As mentioned in Section 4.1, to update multi-threaded programs, the user must designated semantic update points. Semantic update points serve as a useful mechanism for reasoning about update safety, while induced update points permit flexibility in choosing when an update takes place. In particular, programmers can think of updates as possibly occurring at semantic update points only, while the run-time system can actually apply the update at any time so long as it maintains this illusion. The crucial property that allow us to make updates appear as having taken place at a semantic update point is version consistency, a property that Ginseng enforces using static analysis information and runtime checks.

To implement version consistency, we developed an extension of type and effect systems called *contextual effects*, a formalism that permits reasoning about the effects of past and future computation. In Section 5.2 we present our contextual effects calculus and sketch its soundness proof. We then extend contextual effects with support for updates and transactions (version-consistent lexically scoped code blocks), and prove that under certain conditions updates can be safely performed inside transactions, while preserving version consistency (Section 5.3). Next, we extend our update calculus with *check-ins* (Section 5.4) that help us model relaxed updates. Finally, in Section 5.5 we add multi-threading support to our relaxed update calculus and prove that relaxed updates to multi-threaded programs preserve version consistency.

5.2 Contextual effects

Type and effect systems provide a framework for reasoning about the possible side effects of a program's executions. Effects traditionally consider assignments or allocations, but can also track other events, such as functions called or operations performed. A standard type and effect system [66, 83] proves judgments $\varepsilon; \Gamma \vdash e : \tau$, where ε is the effect of the expression e. For many applications, knowing the effect of the *context* in which e appears is also useful. For example, if e includes a security-sensitive operation, then knowing the effect of execution prior to evaluating e could be used to support history-based access control [4, 100]. Conversely, knowing the effect of execution following e could be used for some forms of static garbage collection, e.g., to free initialization functions once initialization is complete [40].

In this section we introduce our core contextual effects calculus. Our contextual effect system proves judgments of the form $\Phi; \Gamma \vdash e : \tau$, where Φ is a tuple

$[\alpha; \varepsilon; \omega]$ containing ε, the standard effect of e, and α and ω, the *prior effect* and *future effect*, respectively, of e's context. For example, in an application $e_1\ e_2$, the prior effect of e_2 includes the effect of e_1, and likewise the future effect of e_1 includes the effect of e_2. We believe that contextual effects have many other uses, in particular any application in which the past or future computation of the program is relevant at various program points.

5.2.1 Syntax

Figure 5.1 presents our source language, which contains expressions e that consist of values v (integers or functions); variables; let binding; function application; and the conditional $\mathsf{if0}$, which tests its integer-valued guard against 0. Our language also includes updateable references $\mathsf{ref}^L\ e$ along with dereference and assignment. Here we annotate each syntactic occurrence of ref with a *label* L, which serves as the abstract name for the locations allocated at that program point. We use labels to define contextual effects. For simplicity we do not model recursive functions directly in our language, but they can be encoded using references.

Our system uses two kinds of effect information. An *effect*, written α, ε, or ω, is a possibly-empty set of labels, and may be 1, the set of all labels. A *contextual effect*, written Φ, is a tuple $[\alpha; \varepsilon; \omega]$. In our system, if e' is a subexpression of e, and e' has contextual effect $[\alpha; \varepsilon; \omega]$, then

- The *current effect* ε is the effect of evaluating e' itself.

- The *prior effect* α is the effect of evaluating e up until we begin evaluating e'.

$$\begin{array}{lrcl}
\text{Expressions} & e & ::= & v \mid x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid e\ e \\
 & & \mid & \mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
 & & \mid & \mathsf{ref}^{L}\ e \mid\ !\,e \mid e := e \\
\text{Values} & v & ::= & n \mid \lambda x.e \\
\text{Effects} & \alpha, \varepsilon, \omega & ::= & \emptyset \mid 1 \mid \{L\} \mid \varepsilon \cup \varepsilon \\
\text{Contextual Effs.} & \Phi & ::= & [\alpha; \varepsilon; \omega] \\
\text{Types} & \tau & ::= & \mathit{int} \mid \mathit{ref}^{\,\varepsilon}\ \tau \mid \tau \longrightarrow^{\Phi} \tau \\
\text{Labels} & L & &
\end{array}$$

Figure 5.1: Contextual effects source language

- The *future effect* ω is the effect of the remainder of the evaluation of e after e' is fully evaluated.

Thus ε is the effect of e' itself, and $\alpha \cup \omega$ is the effect of the context in which e' appears—and therefore $\alpha \cup \varepsilon \cup \omega$ contains all locations accessed during the entire reduction of e.

To make contextual effects easier to work with, we introduce some shorthand. We write Φ^{α}, Φ^{ε}, and Φ^{ω} for the prior, current, and future effect components, respectively, of Φ. We also write $\Phi_{\emptyset}$ for the empty effect $[1; \emptyset; 1]$—by subsumption, discussed below, an expression with this effect may appear in any context. In what follows, we refer to contextual effects simply as *effects*, for brevity.

5.2.2 Typing

We now present a type and effect system to determine the contextual effect of every subexpression in a program. Types τ, listed at the end of Figure 5.1, include the integer type *int*; reference types $\mathit{ref}^{\,\varepsilon}\ \tau$, which denote a reference to memory of type τ where the reference itself is annotated with a label $L \in \varepsilon$; and function types

$$\boxed{\text{Typing}}$$

$$(\textsc{TInt})\frac{}{\Phi_\emptyset;\Gamma\vdash n : \mathit{int}} \qquad (\textsc{TVar})\frac{\Gamma(x)=\tau}{\Phi_\emptyset;\Gamma\vdash x:\tau}$$

$$(\textsc{TLet})\frac{\Phi_1;\Gamma\vdash e_1:\tau_1 \quad \Phi_2;\Gamma,x:\tau_1\vdash e_2:\tau_2 \quad \Phi_1\rhd\Phi_2\hookrightarrow\Phi}{\Phi;\Gamma\vdash \mathsf{let}\ x=e_1\ \mathsf{in}\ e_2:\tau_2}$$

$$(\textsc{TIf})\frac{\Phi_1;\Gamma\vdash e_1:\mathit{int} \quad \Phi_2;\Gamma\vdash e_2:\tau \quad \Phi_2;\Gamma\vdash e_3:\tau \quad \Phi_1\rhd\Phi_2\hookrightarrow\Phi}{\Phi;\Gamma\vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3:\tau} \qquad (\textsc{TRef})\frac{\Phi;\Gamma\vdash e:\tau}{\Phi;\Gamma\vdash \mathsf{ref}^L\ e:\mathit{ref}^{\{L\}}\ \tau}$$

$$(\textsc{TDeref})\frac{\Phi_1;\Gamma\vdash e:\mathit{ref}^{\varepsilon}\ \tau \quad \Phi_2^{\varepsilon}=\varepsilon \quad \Phi_1\rhd\Phi_2\hookrightarrow\Phi}{\Phi;\Gamma\vdash\ !e:\tau}$$

$$(\textsc{TAssign})\frac{\Phi_1;\Gamma\vdash e_1:\mathit{ref}^{\varepsilon}\ \tau \quad \Phi_2;\Gamma\vdash e_2:\tau \quad \Phi_3^{\varepsilon}=\varepsilon \quad \Phi_1\rhd\Phi_2\rhd\Phi_3\hookrightarrow\Phi}{\Phi;\Gamma\vdash e_1:=e_2:\tau}$$

$$(\textsc{TLam})\frac{\Phi;\Gamma,x:\tau'\vdash e:\tau}{\Phi_\emptyset;\Gamma\vdash\lambda x.e:\tau'\longrightarrow^{\Phi}\tau} \qquad [\textsc{TApp}]\frac{\Phi_1;\Gamma\vdash e_1:\tau_1\longrightarrow^{\Phi_f}\tau_2 \quad \Phi_2;\Gamma\vdash e_2:\tau_1 \quad \Phi_1\rhd\Phi_2\rhd\Phi_f\hookrightarrow\Phi}{\Phi;\Gamma\vdash e_1\ e_2:\tau_2}$$

$$(\textsc{TSub})\frac{\Phi';\Gamma\vdash e:\tau' \quad \tau'\le\tau \quad \Phi'\le\Phi}{\Phi;\Gamma\vdash e:\tau}$$

$$\boxed{\text{Effect combinator}}$$

$$(\textsc{XFlow-Ctxt})\frac{\Phi_1=[\alpha_1;\varepsilon_1;(\varepsilon_2\cup\omega_2)] \quad \Phi_2=[(\varepsilon_1\cup\alpha_1);\varepsilon_2;\omega_2] \quad \Phi=[\alpha_1;(\varepsilon_1\cup\varepsilon_2);\omega_2]}{\Phi_1\rhd\Phi_2\ \hookrightarrow\ \Phi}$$

$$\boxed{\text{Subtyping}}$$

$$(\textsc{SInt})\frac{}{\mathit{int}\le\mathit{int}} \qquad (\textsc{SRef})\frac{\tau\le\tau' \quad \tau'\le\tau \quad \varepsilon\subseteq\varepsilon'}{\mathit{ref}^{\varepsilon}\ \tau\le\mathit{ref}^{\varepsilon'}\ \tau'}$$

$$(\textsc{SFun})\frac{\tau_1'\le\tau_1 \quad \tau_2\le\tau_2' \quad \Phi\le\Phi'}{\tau_1\longrightarrow^{\Phi}\tau_2\le\tau_1'\longrightarrow^{\Phi'}\tau_2'} \qquad (\textsc{SCtxt})\frac{\alpha_2\subseteq\alpha_1 \quad \varepsilon_1\subseteq\varepsilon_2 \quad \omega_2\subseteq\omega_1}{[\alpha_1;\varepsilon_1;\omega_1]\le[\alpha_2;\varepsilon_2;\omega_2]}$$

Figure 5.2: Contextual effects type system

$\tau \longrightarrow^{\Phi} \tau'$, where τ and τ' are the domain and range types, respectively, and the function has contextual effect Φ.

Figure 5.2 presents our contextual type and effect system. The rules prove judgments of the form $\Phi;\Gamma\vdash e:\tau$, meaning in type environment Γ, expression e has type τ and contextual effect Φ. The first two rules, (TInt) and (TVar), assign the expected types and the empty effect, since values have no effect.

(TLet) types subexpressions e_1 and e_2, which have effects Φ_1 and Φ_2, respectively, and requires that these effects combine to form Φ, the effect of the entire expression. We use a call-by-value semantics, and hence the effect of the let should be the effect of e_1 followed by the effect of e_2. We specify the sequencing of effects with the combinator $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$, defined by (XFlow-Ctxt) in the middle part of Figure 5.2. Since e_1 happens before e_2, this rule requires that the future effect of e_1 be $\varepsilon_2 \cup \omega_2$, i.e., everything that happens during the evaluation of e_2, captured by ε_2, plus everything that happens after, captured by ω_2. Similarly, the past effect of e_2 must be $\varepsilon_1 \cup \alpha_1$, since e_2 happens just after e_1. Lastly, the effect Φ of the entire expression has α_1 as its prior effect, since e_1 happens first; ω_2 as its future effect, since e_2 happens last; and $\varepsilon_1 \cup \varepsilon_2$ as its current effect, since both e_1 and e_2 are evaluated. We write $\Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi$ as shorthand for $(\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi') \wedge (\Phi' \rhd \Phi_3 \hookrightarrow \Phi)$.

(TIf) requires that its branches have the same type τ and effect Φ_2, which can be achieved with subsumption (below), and uses $\rhd$ to specify that Φ_1, the effect of the guard, occurs before either branch. (TRef) types memory allocation, which has no effect but places the annotation L into a singleton effect $\{L\}$ on the output type. This singleton effect can be increased as necessary by using subsumption.

(TDeref) types the dereference of a memory location of type $\mathit{ref}^{\varepsilon}\ \tau$. In a standard effect system, the effect of $!\,e$ is the effect of e plus the effect ε of accessing the pointed-to memory. Here, the effect of e is captured by Φ_1, and because the dereference occurs after e is evaluated, (TDeref) puts Φ_1 in sequence just before some Φ_2 such that Φ_2's current effect is ε. Therefore by (XFlow-Ctxt), Φ^{ε} is

$\Phi_1^\varepsilon \cup \varepsilon$, and e's future effect Φ_1^ω must include ε and the future effect of Φ_2. On the other hand, Φ_2^ω is unconstrained by this rule, but it will be constrained by the context, assuming the dereference is followed by another expression. (TAssign) is similar to (TDeref), combining the effects Φ_1 and Φ_2 of its subexpressions with a Φ_3 whose current effect is ε.

(TLam) types the function body e and sets the effect on the function arrow to be the effect of e. The expression as a whole has no effect, since the function produces no run-time effects until it is actually called. (TApp) types function application, which combines Φ_1, the effect of e_1, with Φ_2, the effect of e_2, and Φ_f, the effect of the function.

The last rule in our system, (TSub), introduces subsumption on types and effects. The judgments $\tau' \leq \tau$ and $\Phi' \leq \Phi$ are defined at the bottom of Figure 5.2. (SInt), (SRef), and (SFun) are standard, with the usual co- and contravariance where appropriate. (SCtxt) defines subsumption on effects, which is covariant in the current effect, as expected, and contravariant in both the prior and future effects. To understand the contravariance, first consider an expression e with future effect ω_1. Since future effects should soundly approximate (i.e., be a superset of) the locations that may be accessed in the future, we can use e in any context that accesses *at most* locations in ω_1. Similarly, since past effects approximate locations that were accessed in the past, we can use e in any context that accessed at most locations in α_1.

5.2.3 Semantics and Soundness

The semantics of the contextual effects system, the formal definitions of its key soundness properties, and its soundness proof are due to Pratikakis and can be found in his dissertation [94]. We include here the contextual effects semantics and the formal definitions of its key soundness properties for completeness.

The top of Figure 5.3 gives some basic definitions needed for our operational semantics. We extend values v to include the form r_L, which is a run-time heap location r annotated with label L. We need to track labels through our operational semantics to formulate and prove soundness, but these labels need not exist at run-time. We define heaps H to be maps from locations to values. Finally, we extend typing environments Γ to assign types to heap locations.

The bottom part of Figure 5.3 defines a big-step operational semantics for our language. The reduction rules are straightforward. [Id] reduces a value to itself without changing the state or the effects. [Call] evaluates the first expression to a function, the second expression to a value, and then the function body with the formal argument replaced by the actual argument. [Ref] generates a fresh location r, which is bound in the heap to v and evaluates to r_L. [Deref] reads the location r in the heap and adds L to the standard evaluation effect. This rule requires that the future effect after evaluating e have the form $\omega' \cup \{L\}$, i.e., L must be in the capability after evaluating e, but prior to dereferencing the result. Then L is added to α' in the the output configuration of the rule. Notice that $\omega' \cup \{L\}$ is a standard union, and so L may also be in ω'. This allows the same location can be accessed

$$\begin{array}{llll} \text{Values} & v & ::= & \ldots \mid r_L \\ \text{Heaps} & H & ::= & \emptyset \mid H, r \mapsto v \\ \text{Environments} & \Gamma & ::= & \emptyset \mid \Gamma, x : \tau \mid \Gamma, r : \tau \end{array}$$

$$[\textsc{Id}]\frac{}{\langle \alpha, \omega, H, v \rangle \longrightarrow_{\emptyset} \langle \alpha, \omega, H, v \rangle}$$

$$[\textsc{Call}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e_1 \rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, \lambda x.e \rangle \\ \langle \alpha_1, \omega_1, H_1, e_2 \rangle \longrightarrow_{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2 \rangle \\ \langle \alpha_2, \omega_2, H_2, e[x \mapsto v_2] \rangle \longrightarrow_{\varepsilon_3} \langle \alpha', \omega', H', v \rangle\end{array}}{\langle \alpha, \omega, H, e_1\ e_2 \rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3} \langle \alpha', \omega', H', v \rangle}$$

$$[\textsc{Ref}]\frac{\langle \alpha, \omega, H, e \rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega', H', v \rangle \qquad r \notin \mathrm{dom}(H')}{\langle \alpha, \omega, H, \mathsf{ref}^L\ e \rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega', (H', r \mapsto v), r_L \rangle}$$

$$[\textsc{Deref}]\frac{\langle \alpha, \omega, H, e \rangle \longrightarrow_{\varepsilon} \langle \alpha', \omega' \cup \{L\}, H', r_L \rangle \qquad r \in \mathrm{dom}(H')}{\langle \alpha, \omega, H, !\, e \rangle \longrightarrow_{\varepsilon \cup \{L\}} \langle \alpha' \cup \{L\}, \omega', H', H'(r) \rangle}$$

$$[\textsc{Assign}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e_1 \rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, r_L \rangle \\ \langle \alpha_1, \omega_1, H_1, e_2 \rangle \longrightarrow_{\varepsilon_2} \langle \alpha_2, \omega_2 \cup \{L\}, (H_2, r \mapsto v'), v \rangle\end{array}}{\begin{array}{l}\langle \alpha, \omega, H, e_1 := e_2 \rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2 \cup \{L\}} \\ \qquad\qquad \langle \alpha_2 \cup \{L\}, \omega_2, (H_2, r \mapsto v), v \rangle\end{array}}$$

$$[\textsc{If-T}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e_1 \rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, v_1 \rangle \qquad v_1 = 0 \\ \langle \alpha_1, \omega_1, H_1, e_2 \rangle \longrightarrow_{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v \rangle\end{array}}{\langle \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2} \langle \alpha_2, \omega_2, H_2, v \rangle}$$

$$[\textsc{If-F}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e_1 \rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, v_1 \rangle \qquad v_1 = n \neq 0 \\ \langle \alpha_1, \omega_1, H_1, e_3 \rangle \longrightarrow_{\varepsilon_3} \langle \alpha_3, \omega_3, H_3, v \rangle\end{array}}{\langle \alpha, \omega, H, \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_3} \langle \alpha_3, \omega_3, H_3, v \rangle}$$

$$[\textsc{Let}]\frac{\begin{array}{c}\langle \alpha, \omega, H, e_1 \rangle \longrightarrow_{\varepsilon_1} \langle \alpha_1, \omega_1, H_1, v_1 \rangle \\ \langle \alpha_1, \omega_1, H_1, e_2[x \mapsto v_1] \rangle \longrightarrow_{\varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2 \rangle\end{array}}{\langle \alpha, \omega, H, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rangle \longrightarrow_{\varepsilon_1 \cup \varepsilon_2} \langle \alpha_2, \omega_2, H_2, v_2 \rangle}$$

Figure 5.3: Contextual effects operational semantics (partial)

multiple times. [Assign] behaves similarly to [Deref].

Lastly, [If-T] and [If-F] give the two cases for conditionals, and [Let] binds x to the result of evaluating e_1 inside of e_2. Our semantics also includes rules (not shown) that produce **err** when the program tries to access a location that is not in the input capability, or when values are used at the wrong type.

Given this operational semantics, we can now prove that the contextual effect system in Figure 5.2 is sound. We now state our main lemmas and theorems.

We begin with a standard definition of heap typing.

Definition 5.2.1 (Heap Typing). *We say heap H is well-typed under Γ, written $\Gamma \vdash H$, if $\mathrm{dom}(\Gamma) = \mathrm{dom}(H)$ and if for every $r \in \mathrm{dom}(H)$, we have $\Phi_\emptyset; \Gamma \vdash H(r) : \Gamma(r)$.*

Given this definition, we show the standard effect soundness theorem, which states that the program does not go wrong and that the standard effect Φ^ε captures the effect of evaluation.

Theorem 5.2.2 (Standard Effect Soundness). *If $\Phi; \Gamma \vdash e : \tau$ and $\Gamma \vdash H$ and $\langle 1, 1, H, e\rangle \longrightarrow_\varepsilon \langle 1, 1, H', R\rangle$, then there is a $\Gamma' \supseteq \Gamma$ such that R is a value v for which $\Phi_0; \Gamma' \vdash v : \tau$ where $\Gamma' \vdash H'$ and $\varepsilon \subseteq \Phi^\varepsilon$.*

Next, we show the operational semantics is adequate, in that it moves effects from the future to the past during evaluation.

Lemma 5.2.3 (Adequacy of Semantics). *If $\langle \alpha, \omega, H, e\rangle \longrightarrow_\varepsilon \langle \alpha', \omega', H', v\rangle$ then $\alpha' = \alpha \cup \varepsilon$ and $\omega = \omega' \cup \varepsilon$.*

Next we must define what it means for the statically-ascribed contextual effects of some expression e to be sound with respect to the effects of e's evaluation. Suppose that e_p is a program that is well-typed according to typing derivation $\mathcal{T}$ and evaluates to some value v as witnessed by an evaluation derivation D. Observe that each term

e_1 that is reduced in a subderivation of D is either a subterm of e_p, or is derived from a subterm e_2 of e_p via reduction; in the latter case it is sound to give e_1 the same type and effect that e_2 has in $\mathcal{T}$. To reason about the soundness of the effects, therefore, we must track the static effect of expression e_2 as it is evaluated.

We do this by defining a new *typed operational semantics* that extends standard configurations with a typing derivation of the term in that configuration. The key property of this semantics is that it preserves the effect Φ of a term throughout its evaluation, and we prove that given standard evaluation and typing derivations of the original program, we can always construct a corresponding typed operational semantics derivation.

Finally, we prove that given a typed operational semantics derivation, the effect Φ in the typing in each configuration conservatively approximates the actual prior and future effect.

Theorem 5.2.4 (Prior and Future Effect Soundness)**.** *If*

$$E :: \langle T, \alpha, \omega, H, e\rangle \longrightarrow_{\varepsilon} \langle T_v, \alpha', \omega', H', v\rangle$$

where $T :: \Phi; \Gamma \vdash e : \tau$, $\alpha \subseteq \Phi^{\alpha}$ *and* $\omega' \subseteq \Phi^{\omega}$ *then for all sub-derivations* E_i *of* E, $E_i :: \langle T_i, \alpha_i, \omega_i, H_i, e_i\rangle \longrightarrow_{\varepsilon} \langle T_{v_i}, \alpha'_i, \omega'_i, H'_i, v_i\rangle$ *where* $T_i :: \Phi_i; \Gamma_i \vdash e_i : \tau_i$, *it will hold that* $\alpha_i \subseteq \Phi^{\alpha}_i$ *and* $\omega'_i \subseteq \Phi^{\omega}_i$.

The proof of the above theorem is by induction on the derivation, starting at the root and working towards the leaves, and relying on Theorem 5.2.2 and Lemma 5.2.3.

Finally, the soundness of the Contextual Effects system follows as a corollary.

Theorem 5.2.5 (Contextual Effect Soundness). *Given a program e_p with no free variables, its typing $\mathcal{T}$ and its canonical evaluation $\mathcal{D}$, we can construct a typed evaluation $\mathcal{E}$ such that for every sub-derivation $E :: \langle T, \alpha, \omega, H, e\rangle \longrightarrow_\varepsilon \langle T_v, \alpha', \omega', H', v\rangle$ in $\mathcal{E}$, where $T :: \Phi; \Gamma \vdash e : \tau$, it is always the case that $\alpha \subseteq \Phi^\alpha$, $\varepsilon \subseteq \Phi^\varepsilon$ and $\omega \subseteq \Phi^{\omega'}$.*

5.2.4 Contextual Effect Inference

The typing rules in Figure 5.2 form a checking system, but we would prefer to infer effect annotations rather than require the programmer to provide them. Here we sketch the inference process, which is straightforward and uses standard constraint-based techniques.

We change the rules in Figure 5.2 into inference rules by making three modifications. First, we make the rules syntax-driven by integrating (TSub) into the other rules [76]; second, we add variables χ to represent as-yet-unknown effects; and third, we replace implicit equalities with explicit equality constraints.

The resulting rules are mostly as expected, with one interesting difference for (TApp). We might expect inlining subsumption into (TApp) to yield the following rule:

$$(*)\,\frac{\begin{array}{cc}\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 & \Phi_2; \Gamma \vdash e_2 : \tau_1' \\ \tau_1' \leq \tau_1 & \Phi_1 \rhd \Phi_2 \rhd \Phi_f \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

However, this would cause the inferred Φ_f effect to be larger than necessary if there are multiple calls to the same function. For example, consider the following code, where f is some one-argument function, x, y, and z are references, and A and B identify two program points:

```
(if0 ... then /*A*/ (f 1; !x) else /*B*/ (f 2; !y)); !z
```

If we used rule (*), then from branch A, we would require $\{\mathsf{x}, \mathsf{z}\} \subseteq \Phi^{\omega}_{f}$, and from branch B, we would require $\{\mathsf{y}, \mathsf{z}\} \subseteq \Phi^{\omega}_{f}$, where Φ_f is the effect of function f. Putting these together, we would thus have $\Phi^{\omega}_{f} = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$. This result is appropriate, since any of those locations may be accessed after some call to f. However, consider the future effect Φ^{ω}_{A} at program point A. By (XFlow-Ctxt), Φ^{ω}_{A} would contain Φ^{ω}_{f}, and yet y will not be accessed once we reach A, since that access is on another branch. The analogous problem happens at program point B, whose future effect is polluted by x.

The problem is that our effect system conflates all calls to f. One solution would be to add Hindley-Milner style parametric polymorphism, which would address this particular example. However, even with Hindley-Milner polymorphism we would suffer the same problem at indirect function calls, e.g., in C, calls through function pointers would be monomorphic.

The solution is to notice that inlining subsumption into (TApp) should not yield (*), but instead results in the following rule:

$$(\textsc{TApp}')\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \qquad \Phi_2; \Gamma \vdash e_2 : \tau_1' \\ \Phi_f \leq \Phi_f' \qquad\quad \tau_1' \leq \tau_1 \qquad\quad \Phi_f' \text{ fresh} \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_f' \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

Applied to the above example, (TApp$'$) results in two constraints on the future effect of Φ_f:

$$\Phi^{\omega}_{f} \supseteq \Phi^{\omega}_{fA} = \{\mathsf{x}, \mathsf{z}\} \qquad \Phi^{\omega}_{f} \supseteq \Phi^{\omega}_{fB} = \{\mathsf{y}, \mathsf{z}\}$$

Here Φ_{fA} and Φ_{fB} are the fresh function effects at the call to f in A and B, re-

Definitions	d	$::=$	$\mathsf{main}\ e$
		$\mid$	$\mathsf{var\ g} = v\ \mathsf{in}\ d$
		$\mid$	$\mathsf{fun\ f}(x) = e\ \mathsf{in}\ d$
Expressions	e	$::=$	$v \mid x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid e\ e$
		$\mid$	$\mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$
		$\mid$	$\mathsf{ref}\ e \mid !\,e \mid e := e$
		$\mid$	$\mathsf{tx}\ e \mid \mathsf{update}^{\alpha,\omega}$
Values	v	$::=$	$n \mid \mathsf{z}$
Effects	$\alpha, \omega, \varepsilon$	$::=$	$\emptyset \mid 1 \mid \{\mathsf{z}\} \mid \varepsilon \cup \varepsilon$
Global symbols	$\mathsf{f}, \mathsf{g}, \mathsf{z}$	$\in$	$GSym$
Dynamic updates	upd	$::=$	$\{chg, add\}$
Additions	add	$\in$	$GSym \rightharpoonup (\tau \times b)$
Changes	chg	$\in$	$GSym \rightharpoonup (\tau \times b)$
Bindings	b	$::=$	$v \mid \lambda x.e$

Figure 5.4: Proteus-tx syntax, effects, and updates

spectively. Notice that we have $\Phi^{\omega}_{f} = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$, as before, since f is called in both contexts. But now Φ^{ω}_{fA} need not contain y, and Φ_{fB} need not contain x. Thus with (TAPP′), a function's effect summarizes all of its contexts, but does not cause the prior and future effects from different contexts to pollute each other.

To perform type inference, we apply our inference rules, viewing them as generating the *constraints* C in their hypotheses, given by the following grammar:

$$C ::= \tau \leq \tau' \mid \Phi \leq \Phi' \mid \Phi_1 \rhd \Phi_2 \ \hookrightarrow\ \Phi$$

We can then solve the constraints by performing graph reachability to find, for each variable χ, the set of base effects $\{L\}$ or 1 that reach it. In practice, these constraints can be solved very efficiently using a toolkit such as Banshee [61].

5.3 Single-threaded Transactional Version Consistency

In Chapter 4 we showed how we allow programmers to specify semantic update points and how our runtime system permits updates in between semantic update points (i.e., at induced update points) while still maintaining the illusion that code between semantic update points executes at the same version. In this section we present a formalism and proof of version consistency.

We model semantic and induced update points using two syntactic elements: *transactions* $\mathsf{tx}\ B$ and *update points* $\mathsf{update}^{\alpha,\omega}$. A transaction $\mathsf{tx}\ B$ designates a lexically scoped code block B whose execution should be version consistent. Transactions are a restricted form of semantic update point placement: instead of allowing arbitrary semantic update point placement, we require that semantic update points are paired, and delimit lexical scopes. In other words, the beginning and end of transaction correspond to two semantic update points. We require lexical scoping because it makes modeling and reasoning about version consistency easier, without hampering flexibility in choosing update points—instead of inserting a semantic update point at the end of a loop iteration, the programmer simply designates the loop body as a transaction. An update point $\mathsf{update}^{\alpha,\omega}$ in our calculus represents an induced update point, where α and ω are the contextual effects at that point. As we will see in Section 5.3.3, contextual effects impose constraints on updates, to ensure that version consistency is preserved.

Transactional version consistency for DSU is similar to the property of isolation in database-style transactions: just as in the ACID model other operations can either see the entire effect of a transaction, or no effect at all, in DSU the execution of a lexical scope delimited by semantic update points can either be attributed to the old version, or the new version, but not both. Transactions allow us to reason easier about version consistency without placing undue burden on the programmer: instead of having one update point at the beginning of a loop iteration (Section 4.4.4), we simply designate the loop body as a transaction. The formal property our calculus establishes is called *transactional version consistency* (TVC), meaning that transactions execute as if they were entirely the old version or entirely the new version, no matter where an update actually occurs.

5.3.1 Syntax

Figure 5.4 presents Proteus-tx, which extends the language from Section 5.2 to model transactional version-consistent dynamic updates, adapting the ideas of Proteus, our prior dynamic updating calculus [107]. A Proteus-tx program is a definition d, which consists of an expression $\mathsf{main}\ e$, possibly preceded by definitions of *global symbols*, written f, g, or z and drawn from a set *GSym*. The definition $\mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d$ binds mutable variable g to v within the scope of d, and the definition $\mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d$ binds f to a (possibly-recursive) function with formal parameter x and body e.

Expressions e in Proteus-tx have several small differences from the language of Figure 5.1. We add global symbols z to the set of values v. We also remove anonymous lambda bindings to keep things simpler, for reasons discussed in Section 5.3.3.

To mark transactions, we add a form $\mathsf{tx}\ e$ for a transaction whose body is e.

We specify program points where dynamic updates may occur with the term $\mathsf{update}^{\alpha,\omega}$, where the annotations α and ω specify the prior and future effects at the update point, respectively. When evaluation reaches $\mathsf{update}^{\alpha,\omega}$, an available update is applied if its contents do not conflict with the future and prior effect annotations; otherwise evaluation proceeds without updating.

A *dynamic update upd* consists of a pair of partial functions *chg* and *add* that describe the changes and additions, respectively, of global symbol bindings. The range of these functions is pairs (τ, b), where b is the new or replacement value (which may be a function $\lambda x.e$) and τ is its type. Note that Proteus-tx disallows type-altering updates, though Section 5.3.5 explains how they can be supported by employing ideas from our earlier work [107]. Also, although Ginseng allows state initialization functions, for simplicity we do not model them in Proteus-tx.

Finally, effects in Proteus-tx consist of sets of global symbol names z, which represent either a dereference of or assignment to z (if it is a variable) or a call to z (if it is a function name). Because updates in Proteus-tx can only change global symbols (and do not read or write through their contents), we can ignore the effects of the latter (we use syntax $\mathsf{ref}\ \ e$ instead of $\mathsf{ref}^{L}\ e$).

5.3.2 Typing

Figure 5.5 extends the core contextual effect typing rules from Figure 5.2 to Proteus-tx. The first three rules define the judgment $\Gamma \vdash d$, meaning definition d is

$$(\textsc{TMain})\frac{\Phi;\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{main}\ e} \qquad (\textsc{TDVar})\frac{\Phi_\emptyset;\Gamma \vdash v : \tau \qquad \Gamma, \mathsf{g} : \mathit{ref}^{\{\mathsf{g}\}}\ \tau \vdash d}{\Gamma \vdash \mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d}$$

$$(\textsc{TDFun})\frac{\begin{array}{cc}\Gamma' = \Gamma, \mathsf{f} : \tau \longrightarrow^{\Phi} \tau' & \Phi;\Gamma', x : \tau \vdash e : \tau' \\ \Gamma' \vdash d & \{\mathsf{f}\} \subseteq \Phi\end{array}}{\Gamma \vdash \mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d}$$

$$(\textsc{TGVar})\frac{\Gamma(\mathsf{f}) = \tau}{\Phi_\emptyset;\Gamma \vdash \mathsf{f} : \tau} \qquad (\textsc{TUpdate})\frac{\Phi^\alpha \subseteq \alpha' \qquad \Phi^\omega \subseteq \omega'}{\Phi;\Gamma \vdash \mathsf{update}^{\alpha',\omega'} : \mathit{int}}$$

$$(\textsc{TTransact})\frac{\Phi_1;\Gamma \vdash e : \tau \qquad \Phi^\alpha \subseteq \Phi_1^\alpha \qquad \Phi^\omega \subseteq \Phi_1^\omega}{\Phi;\Gamma \vdash \mathsf{tx}\ e : \tau}$$

Figure 5.5: Proteus-tx typing (extends Figure 5.2)

well-typed in environment Γ. (TMain) types e in Γ, where e may have any effect and any type. (TDVar) types the value v, which has the empty effect (since it is a value), and then types d with g bound to a reference to v labeled with effect $\{\mathsf{g}\}$. The last definition rule, (TDFun), constructs a new environment Γ' that extends Γ with a binding of f to the function's type. The function body e is type checked in Γ', to allow for recursive calls. This rule also requires that f appear in all components of the function's effect Φ, written $\{\mathsf{f}\} \subseteq \Phi$. We add f to the prior effect because f must have been called for its entry to be reached. We add f to the current effect so that it is included in the effect at a call site. Lastly, we add f to the future effect because f is on the call stack and we consider its continued execution to be an effect. Note that this prohibits updates to main(), which is always on the stack. However, we can solve this problem by extracting portions of main() into separate functions, which can then be updated; Ginseng provides support to automate this process via loop and code extraction (explained in Sections 3.2.4 and 4.3.1). The next rule, (TGVar), types global variables, which are bound in Γ. The last two rules type the dynamic updating-related elements of Proteus-tx. (TUPDATE) types `update` by checking that its prior and future effect annotations are supersets of (and thus conservatively approximate) the prior and future effects of the context.

Finally, (TTRANSACT) types transactions. A key design choice here is deciding how to handle nested transactions. In (TTRANSACT), we include the prior and future effects of Φ, from the outer context, into those of Φ_1, from the transaction body. This ensures that an update within a child transaction does not violate version consistency of its parent. However, we do not require the reverse—the components of Φ_1 need not be included in Φ. This has two consequences. First, sequenced transactions are free to commit independently. For example, consider the following code

```
tx { tx { /*A*/ }; /*B*/ tx { /*C*/ } }
```

According to (TTRANSACT), the effect at B is included in the prior and future effects of C and A, respectively, but not the other way around. Thus neither transaction's effect propagates into the other, and therefore does not influence any update operations in the other.

The second consequence is that version consistency for a parent transaction ignores the effects of its child transactions. This resembles *open nesting* in concurrency transactions [81]. For example, suppose in the code above that A and C contain calls to a hash table T. Without the inner transaction markers, an update to T available at B would be rejected, because due to A it would overlap with the prior effect, and due to C it would overlap with the future effect. With the inner transactions in place, however, the update would be allowed. As a result, the parent transaction could use the old version of the hash table in A and the new version in C.

This treatment of nested transactions makes sense when inner transactions contain code whose semantics is largely independent of the surrounding context, e.g., the abstraction represented by a hash table is independent of where, or how often, it is used. Baumann et al.[11] have applied this semantics to successfully partition dynamic updates to the K42 operating system into independent, object-sized chunks. While we believe open nesting makes sense, we can see circumstances in which closed nesting might be more natural, so we expect to refine our approach in future work.

5.3.3 Operational Semantics

Figures 5.6 and 5.7 define a small-step operational semantics that reduces configurations $\langle n; \Sigma; H; e\rangle$, where n defines the current program version (a successful dynamic update increments n), Σ is the *transaction stack* (explained shortly), H is the heap, and e is the active program expression. Reduction rules have the form $\langle n; \Sigma; H; e\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e'\rangle$, where the event η on the arrow is either μ, a dynamic update that occurred (discussed below), or ε, the effect of the evaluation step.

In our semantics, heaps map references r and global variables z to triples (τ, b, ν) consisting of a type τ, a binding b (defined in Figure 5.4), and a *version*

Definitions

Heaps	H	$::=$	$\emptyset \mid r \mapsto (\cdot, b, \nu), H$
		$\mid$	$\mathsf{z} \mapsto (\tau, b, \nu), H$
Version sets	ν	$::=$	$\emptyset \mid \{n\} \cup \nu$
Traces	σ	$::=$	$\emptyset \mid (\mathsf{z}, \nu) \cup \sigma$
Transaction stacks	Σ	$::=$	$\emptyset \mid (n, \sigma), \Sigma$
Expressions	e	$::=$	$\ldots \mid r \mid \mathsf{intx}\ e$
Events	η	$::=$	$\varepsilon \mid \mu$
Update Direction	dir	$::=$	$bck \mid fwd$
Update Bundles	μ	$::=$	(upd, dir)

Compilation

$\mathcal{C}(H; \mathsf{main}\ e)$	$= H; e$
$\mathcal{C}(H; \mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d)$	$= \mathcal{C}(H, \mathsf{f} \mapsto (\tau \longrightarrow^{\Phi} \tau', \lambda x.e, \{0\}); d)$
$\mathcal{C}(H; \mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d)$	$= \mathcal{C}(H, \mathsf{g} \mapsto (\tau, v, \{0\}); d)$

Evaluation Contexts

$$
\begin{array}{lll}
\mathbb{E} & ::= & [\,] \mid \mathbb{E}\ e \mid v\ \mathbb{E} \mid \mathsf{let}\ x = \mathbb{E}\ \mathsf{in}\ e \\
 & \mid & \mathsf{ref}\ \mathbb{E} \mid !\,\mathbb{E} \mid \mathbb{E} := e \mid r := \mathbb{E} \mid \mathsf{g} := \mathbb{E} \\
 & \mid & \mathsf{if0}\ \mathbb{E}\ \mathsf{then}\ e\ \mathsf{else}\ e
\end{array}
$$

Computation

[LET]	$\langle n; (n', \sigma); H; \mathsf{let}\ x = v\ \mathsf{in}\ e\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; e[x \mapsto v]\rangle$	
[REF]	$\langle n; (n', \sigma); H; \mathsf{ref}\ v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H[r \mapsto (\cdot, v, \emptyset)]; r\rangle$	$r \notin \mathrm{dom}(H)$
[DEREF]	$\langle n; (n', \sigma); H; !\,r\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; v\rangle$	$H(r) = (\cdot, v, \emptyset)$
[ASSIGN]	$\langle n; (n', \sigma); H; r := v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H[r \mapsto (\cdot, v, \emptyset)]; v\rangle$	$r \in \mathrm{dom}(H)$
[IF-T]	$\langle n; (n', \sigma); H; \mathsf{if0}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; e_1\rangle$	
[IF-F]	$\langle n; (n', \sigma); H; \mathsf{if0}\ n''\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; e_2\rangle$	$n'' \neq 0$
[CONG]	$\langle n; \Sigma; H; \mathbb{E}[e]\rangle \longrightarrow_{\eta} \langle n'; \Sigma'; H'; \mathbb{E}[e']\rangle$	$\langle n; \Sigma; H; e\rangle \longrightarrow_{\eta} \langle n'; \Sigma'; H'; e'\rangle$
[GVAR-DEREF]	$\langle n; (n', \sigma); H; !\,\mathsf{z}\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu)); H; v\rangle$	$H(\mathsf{z}) = (\tau, v, \nu)$
[GVAR-ASSIGN]	$\langle n; (n', \sigma); H; \mathsf{z} := v\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu)); H[\mathsf{z} \mapsto (\tau, v, \nu)]; v\rangle$	$H(\mathsf{z}) = (\tau, v', \nu)$
[CALL]	$\langle n; (n', \sigma); H; \mathsf{z}\ v\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu)); H; e[x \mapsto v]\rangle$	$H(\mathsf{z}) = (\tau, \lambda x.e, \nu)$
[TX-START]	$\langle n; (n', \sigma); H; \mathsf{tx}\ e\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma), (n, \emptyset); H; \mathsf{intx}\ e\rangle$	
[TX-CONG-1]	$\langle n; (n'', \sigma), \Sigma; H; \mathsf{intx}\ e\rangle \longrightarrow_{\mu} \langle n'; \mathcal{U}[(n'', \sigma)]^{\mu}_{n'}, \Sigma'; H'; \mathsf{intx}\ e'\rangle$	$\langle n; \Sigma; H; e\rangle \longrightarrow_{\mu} \langle n'; \Sigma'; H'; e'\rangle$
[TX-CONG-2]	$\langle n; \Sigma; H; \mathsf{intx}\ e\rangle \longrightarrow_{\emptyset} \langle n'; \Sigma'; H'; \mathsf{intx}\ e'\rangle$	$\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon} \langle n'; \Sigma'; H'; e'\rangle$
[TX-END]	$\langle n; ((n', \sigma'), (n'', \sigma'')); H; \mathsf{intx}\ v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma'); H; v\rangle$	$traceOK(n'', \sigma'')$
[UPDATE]	$\langle n; (n', \sigma); H; \mathsf{update}^{\alpha,\omega}\rangle \longrightarrow_{(upd,dir)} \langle n+1; \mathcal{U}[(n', \sigma)]^{upd,dir}_{n+1}; \mathcal{U}[H]^{upd}_{n+1}; 1\rangle$	$updateOK(upd, H, (\alpha, \omega), dir)$
[NO-UPDATE]	$\langle n; (n', \sigma); H; \mathsf{update}^{\alpha,\omega}\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma); H; 0\rangle$	

Figure 5.6: Proteus-tx operational semantics

Update Safety

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{updateOK}(\mathit{upd}, H, (\alpha,\omega), \mathit{dir}) =} \\
 & \mathit{dir} = \mathit{bck} \Rightarrow \alpha \cap \mathrm{dom}(\mathit{upd}^{\mathit{chg}}) = \emptyset \\
\wedge & \mathit{dir} = \mathit{fwd} \Rightarrow \omega \cap \mathrm{dom}(\mathit{upd}^{\mathit{chg}}) = \emptyset \\
\wedge & \Gamma = \mathit{types}(H) \\
\wedge & \Gamma_{\mathit{upd}} = \Gamma, \mathit{types}(\mathit{upd}^{\mathit{add}}) \\
\wedge & \forall \mathsf{z} \mapsto (\tau, b, \cdot) \in \mathit{upd}^{\mathit{chg}}. \\
 & \quad \big(\Phi_\emptyset; \Gamma_{\mathit{upd}} \ \vdash \ b : \tau \ \wedge \ \mathit{heapType}(\tau, \mathsf{z}) = \Gamma(\mathsf{z})\big) \\
\wedge & \forall \mathsf{z} \mapsto (\tau, b, \cdot) \in \mathit{upd}^{\mathit{add}}. \\
 & \quad \big(\Phi_\emptyset; \Gamma_{\mathit{upd}} \ \vdash \ b : \tau \ \wedge \ \mathsf{z} \notin \mathrm{dom}(H)\big)
\end{array}$$

Trace Safety

$$\mathit{traceOK}(n, \sigma) = (\forall (\mathsf{z}, \nu) \in \sigma.\ n \in \nu)$$

Heap Updates

$$\mathcal{U}[(\mathsf{z} \mapsto (\tau, b, \nu), H)]^{\mathit{upd}}_n = \begin{cases} \mathsf{z} \mapsto (\tau, b', \{n\}), \mathcal{U}[H]^{\mathit{upd}}_n \\ \quad \text{if } \mathit{upd}^{\mathit{chg}}(\mathsf{z}) \mapsto (\tau, b') \\ \mathsf{z} \mapsto (\tau, b, \nu \cup \{n\}), \mathcal{U}[H]^{\mathit{upd}}_n \\ \quad \text{otherwise} \end{cases}$$

$$\mathcal{U}[(r \mapsto (\cdot, b, \emptyset), H)]^{\mathit{upd}}_n = (r \mapsto (\cdot, b, \emptyset)), \mathcal{U}[H]^{\mathit{upd}}_n$$

$$\mathcal{U}[\emptyset]^{\mathit{upd}}_n = \{\mathsf{z} \mapsto (\tau, b, \{n\}) \mid \mathsf{z} \mapsto (\tau, b) \in \mathit{upd}^{\mathit{add}}\}$$

Heap Typing Environments

$$\begin{array}{rcll}
\mathit{types}(\emptyset) & = & \emptyset & \\
\mathit{types}(\mathsf{z} \mapsto (\tau, b, \nu), H') & = & \mathsf{z} : \mathit{heapType}(\tau, \mathsf{z}), \mathit{types}(H') & \\
\mathit{heapType}(\tau_1 \longrightarrow^{\Phi} \tau_2, \mathsf{z}) & = & \tau_1 \longrightarrow^{\Phi} \tau_2 & \mathsf{z} \in \Phi \\
\mathit{heapType}(\tau, \mathsf{z}) & = & \mathit{ref}^{\{\mathsf{z}\}}\ \tau & \tau \neq (\tau_1 \longrightarrow^{\Phi} \tau_2)
\end{array}$$

Trace Stack Updates

$$\begin{array}{lcl}
\mathcal{U}[(n', \sigma)]^{\mathit{upd},\mathit{fwd}}_n & = & (n', \sigma) \\
\mathcal{U}[(n', \sigma)]^{\mathit{upd},\mathit{bck}}_n & = & (n, \mathcal{U}_t[\sigma]^{\mathit{upd}}_n) \\
\mathcal{U}_t[\sigma]^{\mathit{upd}}_n & = & \{(\mathsf{z}, \nu \cup \{n\} \mid \mathsf{z} \notin \mathrm{dom}(\mathit{upd}^{\mathit{chg}})\} \\
 & \cup & \{(\mathsf{z}, \nu) \mid \mathsf{z} \in \mathrm{dom}(\mathit{upd}^{\mathit{chg}})\}
\end{array}$$

Figure 5.7: Proteus-tx update safety

set ν. The first and last components are relevant only for global symbols; the type τ is used to ensure that dynamic updates do not change the types of global bindings, and the version set ν contains all the program versions up to, and including, the current version since the corresponding variable was last updated. When an update occurs, new or changed bindings are given only the current version, while all other bindings have the current version added to their version set (i.e., we preserve the fact that the same binding was used in multiple program versions).

As evaluation proceeds, we maintain a transaction stack Σ, which is a list of pairs (n, σ) that track the currently active transactions. Here n is the version the program had when the transaction began, and σ is a *trace*. A trace is a set of pairs (z, ν), each of which represents a global symbol access paired with its version set at the time of use. The traces act as a log of dynamic events, and we track them in our semantics so we can prove that all global symbols accessed in a transaction come from the same version.

To evaluate a program d, we first compute $\mathcal{C}(\emptyset, d)$ using the function $\mathcal{C}$ shown at the top of Figure 5.6, which yields a pair $H; e$. This function implicitly uses the types derived by typing d using the rules in Figure 5.5. Then we begin regular evaluation in the configuration $\langle 0; (0, \emptyset); H; e\rangle$, i.e., we evaluate e at version 0, with initial transaction stack $(0, \emptyset)$, and with the declared bindings H. This causes the top-level expression e in $\mathsf{main}\ e$ to be treated as if it were enclosed in a transaction block.

The first several reduction rules in Figure 5.6 are straightforward. [LET], [REF], [DEREF], [ASSIGN], [IF-T], and [IF-F] are small-step versions of the rules in Figure 5.3, though normal references no longer have effects. None of these rules affects the current version or transaction stack. [CONG] is standard.

[GVAR-DEREF], [GVAR-ASSIGN], and [CALL] each have effect $\{\mathsf{z}\}$ and add (z, ν) to the current transaction's trace, where ν is z's current version set. Notice that [CALL] performs dereference and application in one step, finding z in the heap and performing substitution. Since dynamic updates modify heap bindings, this ensures that every function call is to the most recent version. Notice that although both functions and variables are stored in the heap, we assign regular function types to functions ((TDFun) in Figure 5.5) so that they cannot be assigned to within a program. Including λ-terms in the expression language would either complicate function typing or make it harder to define function updates so we omit them to keep things simpler.

The next several rules handle transactions. [tx-start] pushes the pair $(n, \emptyset)$ onto the right of the transaction stack, where n is the current version and $\emptyset$ is the empty trace. The expression $\mathsf{tx}\ e$ is reduced to $\mathsf{intx}\ e$, which is a new form that represents an actively-evaluating transaction. The form $\mathsf{intx}\ e$ does not appear in source programs, and its type rule matches that of $\mathsf{tx}\ e$ (see Figure 5.8).

Next, [tx-cong-1] and [tx-cong-2] perform evaluation within an active transaction $\mathsf{intx}\ e$ by reducing e to e'. The latter rule applies if e's reduction does not include an update, in which case the effect ε of reducing e is treated as $\emptyset$ in the outer transaction. This corresponds to our model of transaction nesting, which does not consider the effects of inner transactions when updating outer transactions. Otherwise, if an update occurs, then [tx-cong-1] applies, and we use the function $\mathcal{U}$ to update version numbers on the outermost entry of the transaction stack. $\mathcal{U}$ is discussed shortly.

The key property guaranteed by Proteus-tx, that transactions are version consistent, is enforced by [tx-end], which gets stuck unless $traceOK(n'', \sigma'')$ holds. This predicate, defined just below the reduction rules, states that every element (z, ν) in the transaction's trace σ'' satisfies $n'' \in \nu$, meaning that when z was used, it could be attributed to version n'', the version of the transaction. If this predicate is satisfied, [tx-end] strips off intx and pops the top (rightmost) entry on the transaction stack.

The last two rules handle dynamic updates. When $\mathsf{update}^{\alpha,\omega}$ is in redex position, these rules try to apply an available update bundle μ, which is a pair (upd, dir) consisting of an update (from Figure 5.4) and a *direction* dir that indicates whether we should consider the update as occurring at the beginning or end of the transaction, respectively. If $updateOK(upd, H, (\alpha, \omega), dir)$ is satisfied for some dir, then [UPDATE] applies and the update occurs. Otherwise [NO-UPDATE] applies, and the update must be delayed.

If [UPDATE] applies, we increment the program's version number and update the heap using $\mathcal{U}[H]^{upd}_{n+1}$, defined in the middle-right of Figure 5.6. This function replaces global variables and adds new bindings according to the update. New and replaced bindings' version sets contain only the current version, while unchanged bindings add the current version to their existing version sets.

The $updateOK()$ predicate is defined just below the reduction rules in Figure 5.6. The first two conjuncts enforce the update safety requirement discussed in Section 5.3. There are two cases. If $dir = bck$, then we require that the update not intersect the prior effects, so that the update will appear to have happened at the beginning of the transaction. In this case, we need to update the version number of the transaction to be the new version, and any elements in the trace not modified by the update can have the new version added to their version sets, i.e., the past effect can be attributed to the new version. To do this, [UPDATE] applies the function $\mathcal{U}[(n', \sigma)]^{upd,dir}_{n+1}$, defined on the bottom right of Figure 5.6, with $dir = bck$. The update applies to outer transactions as well, and thus [TX-CONG-1] applies this same version number replacement process across the transaction stack.

$$\text{TIntrans}\ \frac{\Phi_1; \Gamma \vdash e : \tau \qquad \Phi^\alpha \subseteq \Phi_1^\alpha \qquad \Phi^\omega \subseteq \Phi_1^\omega}{\Phi; \Gamma \vdash \mathsf{intx}\ e : \tau}$$

$$\frac{\begin{array}{l} \mathrm{dom}(\Gamma) = \mathrm{dom}(H) \\ \forall \mathsf{z} \mapsto (\tau \longrightarrow^{\Phi} \tau', \lambda x.e, \nu) \in H. \\ \quad \Phi; \Gamma, x : \tau \vdash e : \tau' \ \wedge\ \Gamma(\mathsf{z}) = \tau \longrightarrow^{\Phi} \tau' \ \wedge\ \mathsf{z} \in \Phi \\ \forall \mathsf{z} \mapsto (\tau, v, \nu) \in H. \quad \Phi_\emptyset; \Gamma \vdash v : \tau \ \wedge\ \Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon}\ \tau \ \wedge\ \mathsf{z} \in \varepsilon \\ \forall r \mapsto (\cdot, v, \nu) \in H. \quad \Phi_\emptyset; \Gamma \vdash v : \tau \ \wedge\ \Gamma(r) = \mathit{ref}^{\,\varepsilon}\ \tau \\ \forall \mathsf{z} \mapsto (\tau, b, \nu) \in H. \quad n \in \nu \end{array}}{n; \Gamma \ \vdash \ H}$$

$$(\text{TC1})\ \frac{\begin{array}{c} \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \Rightarrow n \in \mathit{ver}(H, \mathsf{f}) \end{array}}{[\alpha; \varepsilon; \omega], \cdot; H \vdash (n, \sigma)}$$

$$(\text{TC2})\ \frac{\begin{array}{c} \Phi', \mathcal{R}; H \vdash \Sigma \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \Rightarrow n \in \mathit{ver}(H, \mathsf{f}) \end{array}}{[\alpha; \varepsilon; \omega], \Phi', \mathcal{R}; H \vdash (n, \sigma), \Sigma}$$

$$\text{where} \quad \mathit{ver}(H, \mathsf{f}) = \nu \ \mathit{iff}\ H(\mathsf{f}) = (\tau, b, \nu)$$

Figure 5.8: Proteus-tx typing extensions for proving soundness

In the other case, if $dir = fwd$, we require that the remainder of the transaction not be affected by the update, so the update will appear to have happened at the end of the transaction. In this case we need not modify the transaction stack, and hence $\mathcal{U}[(n', \sigma)]_n^{upd,dir}$ with $dir = fwd$ simply returns (n', σ).

The remaining premises of $updateOK()$ determine whether the update itself is well-formed: each replacement binding must have the same type as the original, and new and added bindings must type check in the context of the updated heap.

5.3.4 Soundness

We have proven that well-typed Proteus-tx programs are version-consistent. The main result is that a well-typed, well-formed program either reduces to a value or evaluates indefinitely while preserving typing and version consistency. To prove this we need two additional judgments, shown in Figure 5.8. Heap typing $n; \Gamma \vdash H$ extends Definition 5.2.1 from the core system, where the additional conditions ensure that global symbols are well-typed, have well-formed effects, and include version n (presumed to be the current version) in their version sets.

Stack well-formedness $\mathcal{R}; H \vdash \Sigma$ checks that a transaction stack Σ is correctly approximated by a *transaction effect* $\mathcal{R}$, which consists of a list of contextual effects Φ, one for each nested transaction. $\mathcal{R}$ is computed from a typing derivation in a straightforward way according to the function $[\![\Phi; \Gamma \vdash e : \tau]\!] = \mathcal{R}$, extracting Φ_1 from each occurrence of (TIntrans) recursively; the rules are not shown due to space constraints. Stack well-formedness ensures two properties. First, it ensures that each element in the trace σ is included in the corresponding prior effect α (i.e., $\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha$). As a result, we know that *bck* updates that rewrite the stack will add the new version to all elements of the trace, since none have changed. Second, it ensures that elements in each transaction's current effect (i.e., the part yet to be executed) have the version of that transaction: $\mathsf{f} \in \varepsilon \Rightarrow n \in \mathit{ver}(H, \mathsf{f})$.

With this we can prove the core result:

Theorem 5.3.1 (Single-step Soundness). *If $\Phi; \Gamma \vdash e : \tau$ where $[\![\Phi; \Gamma \vdash e : \tau]\!] = \mathcal{R}$; and $n; \Gamma \vdash H$; and $\Phi, \mathcal{R}; H \vdash \Sigma$; and $\mathit{traceOK}(\Sigma)$, then either e is a value, or there exist n', H', Σ', Φ', e', and η such that $\langle n; \Sigma; H; e \rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e' \rangle$ and $\Phi'; \Gamma' \vdash e' : \tau$ where $[\![\Phi'; \Gamma' \vdash e' : \tau]\!] = \mathcal{R}'$; and $n'; \Gamma' \vdash H'$; and $\Phi', \mathcal{R}'; H' \vdash \Sigma'$; and $\mathit{traceOK}(\Sigma')$ for some $\Phi', \Gamma', \mathcal{R}'$.*

The proof is based on progress and preservation lemmas, as is standard. Details are in Appendix B.

From this lemma we can prove soundness:

Corollary 5.3.2 (Soundness). *If* $\Phi; \Gamma \vdash e : \tau$ *and* $0; \Gamma \vdash H$ *then* $\langle 0; (0, \emptyset); H; e \rangle \rightsquigarrow_A \langle n'; (n'', \sigma); H'; v \rangle$ *for some value* v *or else evaluates indefinitely, where* $\rightsquigarrow_A$ *is the reflexive, transitive closure of the* $\longrightarrow_\eta$ *relation such that* A *is a set of events* η*.*

5.3.5 Implementing Version Consistency for C Programs

Ginseng implements transactional version consistency for C using contextual effects. The programmer indicates transactional blocks as lexically scoped blocks delimited by semantic update points, and the compiler annotates these points with contextual effects information.

To perform effect inference, we first compute a context-sensitive points-to analysis using CIL [80]. Then we generate (context-insensitive) effect constraints (as described in Section 5.2.4) using labels derived from the points-to analysis, and we solve the constraints with Banshee [61].

After computing the contextual effects, Ginseng transforms the program to make it updateable, and transforms each occurrence of $\mathsf{update}^{\alpha,\omega}$ into a call to a function $\mathsf{DSU\_induced\_update}(\alpha, \omega, D)$. Here α and ω are the prior and future effects at the update point, pre-computed by our contextual effect inference, and D is the capability (Section 3.3.1), i.e., the set of type names whose definitions cannot be modified and variable or functions whose types cannot be modified.

When DSU_induced_update is called at run time, it checks to see whether an update is available and, if so, applies the update if it is both type safe (i.e., no variable or type in D has been changed by the update to have a different type) and version consistent (given α and ω). If an update is not safe, it is delayed and execution continues at the old version.

State Transformation. Our version consistency condition is slightly more complicated in practice due to state transformers (described in Section 3.4). The programmer writes the state transformer as if it will be run at the beginning or end of a transaction, and our system must ensure that this appearance is true. That is, to allow an update to occur within a transaction, we must ensure that (1) the writes performed by the state transformer do not violate the version consistency of the current program transactions, and (2) the effects of the current transactions do not violate the version consistency of the state transformer itself. We achieve both ends by considering the update changes ($\mathrm{dom}(upd^{chg})$) *and* the state transformer's current effect ε_{xf} as the effect of the update when performing the usual checks for version consistency.

For example, if an update point DSU_induced_update(α, ω, D) is reached within a transaction and $\omega \cap (\varepsilon_{xf} \cup \mathrm{dom}(upd^{chg})) = \emptyset$, then the remaining actions of the transaction will not affect the state transformer, and vice versa. Therefore, the update appears to have occurred at the end of the transaction. Likewise, if $\alpha \cap (\varepsilon_{xf} \cup \mathrm{dom}(upd^{chg})) = \emptyset$ then the effect of the transaction to this point has no bearing on the execution of the state transformer, and vice versa, so it is as if the update occurred at the beginning of the transaction. Note that because state transformers can also access the heap from global variables we need to include accesses to standard heap references (i.e., names L as in Section 5.2) in our effects.

A similar complication arises from the use of type transformers (Section 3.2.2). If type transformers have effects (e.g., call functions or access global variables), these effects need to be taken into account for our version consistency condition. We can do this by simply adding the effects of all type transformers to $\mathrm{dom}(upd^{chg})$, as we do with the effects of state transformers above. In our current implementation, we do not consider type transformer effects—we have manually inspected our type transformers and found them to be safe, since the type transformer code for our test applications is very simple. However, we plan to add this feature in future work.

5.3.6 Experiments

We measured the potential benefits of transactional version consistency by analyzing 12 dynamic updates to Vsftpd. When updating Vsftpd (Section 3.5.3), we manually placed one semantic update point at the end of an iteration of the long-running accept loop. Placing the update point there ensured that updates were *de facto* version consistent, as the entire loop iteration executes at the same (old or new) version. However, having a single update point hampers update availability, as we need to wait until the end of an iteration to apply the update. In this section, we try to determine how induced update points can improve availability by allowing updates to be applied inside transactions, while preserving version consistency.

We first designated the two long-running loops in Vsftpd (the accept loop and the command processing loop) as transactions. Then, we modified Ginseng to seed the code used in transactions with candidate update points (i.e., induced update points). While we could conceivably insert induced update points at every statement, we found through manual examination that inserting them just before the `return` statement of any function reachable from within a transaction provides good coverage. Finally, we used Ginseng to infer the contextual effects and type modification restrictions at each induced update point, and computed at how many of them we could safely apply the update.

We conducted our experiments on an Athlon 64 X2 dual core 4600 machine with 4GB of RAM, running Debian, kernel version 2.6.18. Figure 5.9 summarizes our results. For each version, we list its size, the time Ginseng takes to pre-compute contextual effects and type modification restrictions, and the number of candidate update points that were automatically inserted. The analysis takes around 10 minutes for the largest example, and we expect that time could be reduced with more engineering effort. The last two columns indicate how many update points are type safe, and how many are both type safe and version consistent, with respect to the update from the version in that row to the next version. Note that determining whether an update is type safe and version consistent is very fast, and so we do not report the time for that computation.

From the table, we can see that several induced update points are type safe and version consistent. We manually examined all of these points. For all program versions except 1.1.0, 1.2.1, and 2.0.2pre2, we found that roughly one-third of the

Version	**Size (LOC)**	**Time (sec)**	**Candidate upd. points**	**Type-safe**	**VC-safe**
1.1.0	10,157	193	344	300	33
1.1.1	10,245	196	346	19	9
1.1.2	10,540	234	350	25	8
1.1.3	10,723	238	354	19	8
1.2.0	12,027	326	413	31	9
1.2.1	12,662	264	438	368	146
1.2.2	12,691	278	439	32	9
2.0.0	13,465	440	471	392	9
2.0.1	13,478	420	471	459	9
2.0.2pre2	13,531	632	471	471	9
2.0.2pre3	14,712	686	484	484	8
2.0.2	17,386	649	471	468	9

Figure 5.9: Version consistency analysis results.

VC-safe induced update points occur somewhere in the middle of a transaction, providing better potential update availability. Another third occur close to or just before the end of a transaction, and the last third occur in dead code, providing no advantage. For the remaining versions, 1.1.0, 1.2.1, and 2.0.2pre2, we found that roughly 10% of the induced update points are in the middle of transactions, and almost all the remaining ones are close to the end of a transaction, with a few more in dead code.

One reason so many safe induced update points tend to occur toward the end of the transaction is due to the field-insensitivity of the alias analysis we used. In Vsftpd, the type vsf_session contains a multitude of fields and is used pervasively throughout the code. The field-insensitive analysis causes spurious conflicts when one field is accessed early in the transaction but others are accessed later on, as is typical. This pushes the safe induced update points to the end of the transaction, following vsf_session's last use. We plan to integrate a field-sensitive alias analysis into Ginseng to remedy this problem.

Interestingly, there are generally far more updates that are exclusively type safe than those that are both type safe and version consistent. We investigated some of these, and we found that the reasons for this varied with the update. For example, the updates that do not change `vsf_session` (e.g., 1.1.0) have a high number of type-safe update points, while those that do (e.g., 1.1.1) have far fewer. This makes sense, given `vsf_session`'s frequent use.

In summary, these results show that many induced update points are both type safe and version consistent, providing greater availability of updates than via manual placement. We expect still more update availability with a more accurate alias analysis.

5.4 Relaxed Updates

The barrier approach for updating multi-threaded programs (Section 4.2.1) performs a *synchronous* safety check at update points: the contextual effects (α, ω) of a thread are computed statically, and a runtime check verifies whether they conflict with the update contents `u`.

To avoid barrier synchronization, in Section 4.2.2 we proposed a relaxed approach where threads periodically *check-in* their effects. We call these *relaxed updates*, because an update does not necessarily take place at an update point. When an update becomes available, the runtime system does not have to barrier synchronize to do the safety check—all it has to do is inspect the checked-in effects of each thread. To ensure update safety, we need a means to prove that check-ins are safe,

Definitions	d	$::=$	$\mathsf{main}\ e$
		$\mid$	$\mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d$
		$\mid$	$\mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d$
Expressions	e	$::=$	$v \mid x \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid e\ e$
		$\mid$	$\mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$
		$\mid$	$\mathsf{ref}\ \ e \mid !\, e \mid e := e$
		$\mid$	$\mathsf{tx}\ e \mid \mathsf{checkin}^{\alpha,\omega}$
Values	v	$::=$	$n \mid \mathsf{z}$
Effects	$\alpha, \omega, \varepsilon, \delta$	$::=$	$\emptyset \mid 1 \mid \{\mathsf{z}\} \mid \varepsilon \cup \varepsilon$
Cntxt Effs.	Φ	$::=$	$[\alpha; \varepsilon; \omega; \delta_i ; \delta_o]$
Global symbols	$\mathsf{f}, \mathsf{g}, \mathsf{z}$	$\in$	GSym
Dynamic updates	upd	$::=$	$\{\mathit{chg}, \mathit{add}\}$
Additions	add	$\in$	$\mathit{GSym} \rightharpoonup (\tau \times b)$
Changes	chg	$\in$	$\mathit{GSym} \rightharpoonup (\tau \times b)$
Bindings	b	$::=$	$v \mid \lambda x.e$

Figure 5.10: Source language for relaxed updates.

i.e., at runtime, they over-approximate the actual thread effects.

5.4.1 Syntax

The language syntax is presented in Figure 5.10; it extends the calculus in Figure 5.4 with support for check-ins (all additions and changes are highlighted). We only discuss these additions. Check-ins $\mathsf{checkin}^{\alpha,\omega}$ "snapshot" a thread's prior (α) and future (ω) effects. To account for effects of code blocks between check-in points, we use two extra components in the type-level contextual effects. The contextual effect Φ of a term e is $[\alpha; \varepsilon; \omega; \delta_i; \delta_o]$ where α, ε, and ω are the prior, normal and future effects, as described in Section 5.3. The output check-in effect, δ_o is the effect of the program after evaluating e, up to the next check-in point. The input check-in effect, δ_i contains the effect from the start of e's evaluation up to the first check-in term in e, or in e's continuation. The relation $\delta_i = \delta_o \cup \varepsilon$ holds for terms that do not contain check-ins.

5.4.2 Typing

We present the relevant type rules in Figure 5.11. The rules not shown are identical to the ones in Figure 5.5, and we highlight the differences for the ones we present. The fundamental rule for contextual effects is the (XFlow-Ctxt); if an expression e consists of two sub-expressions e_1 and e_2 where e_1 is evaluated first, (XFlow-Ctxt) captures this by adding e_1's normal effect (ε_1) to e_2's prior effect, and adding e_2's normal effect (ε_2) to e_1's future effect; the combined effect Φ is a tuple whose prior effect is e_1's prior effect, normal effect is the union of normal effects of e_1 and e_2, and future effect is the future effect of e_2. Note how the e_1's output check-in effect, δ_{o1}, is the same as e_2's input effect δ_{i2}; this ensures proper chaining for check-in effects. The subtyping rule, (TSub), allows an expression e, whose contextual effect is Φ', to be type-checked in a context Φ if $\Phi' \leq \Phi$; intuitively, this means that the prior and future components of Φ make *fewer* assumptions about the prior and future computations, but we can pretend that e's normal effect is larger. The rules for deference (TDeref) and assignment (TAssign) ensure proper check-in effect chaining from Φ_1 to Φ_2 in the premise $\Phi_2^{\delta_i} = \Phi_2^{\delta_o} \cup \varepsilon$. The application rule, (TApp), pushes the prior and future effects of the caller into the callee, and adds the effect of the body of the callee to the effect of the call. The transaction rule, (TTransact), pushes Φ, the effect of the transaction's enclosing scope, into the transaction body. The rationale for this is to disallow an update in an inner transaction that could potentially break version consistency for an outer transaction. (TCheckin) is the key rule that makes the static contextual effects available to the runtime system. Note how the runtime prior effect, α' contains not only the effects of the program so far, α, but also the effects of the evaluation up to the next check-in point, δ_o. This is necessary because we do not have control over when an update is applied in the interval from the current check-in point up to the next check-in point. The runtime future effect ω' is a safe approximation of the type-level future effect, ω'. We provide an example of how the check-in effects work in practice in Section 4.2.2.

5.4.3 Operational Semantics

Our operational semantics extends the Proteus-tx semantics from Section 5.3.3 with support for checkin-ins; additions and changes are highlighted. The evaluation rules (Figure 5.12) are transitions between configurations $\langle n; \Sigma; H; e\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e'\rangle$ where n is a global program version, Σ is a transaction stack, and H is the heap. A stack element has the form (n', σ, κ) where n' is the program version associated with the transaction, σ (trace) contains the bindings accessed so far in the transaction, and κ is a runtime restriction. When a function is called or a global variable is dereferenced [GVAR-DEREF], the name is added to the trace. When we start a transaction [TX-START], we push a new element on the transaction stack, with an empty trace and an initial restriction κ, and we mark the fact that we are evaluating inside a transaction using an intx marker. Reductions inside a

$$(\textsc{XFlow-Ctxt})\,\frac{\begin{array}{c}\Phi_1 = [\alpha_1; \varepsilon_1; (\varepsilon_2 \cup \omega_2);\ \delta_{i1}\ ;\ \delta_{i2}\ ] \\ \Phi_2 = [(\varepsilon_1 \cup \alpha_1); \varepsilon_2; \omega_2;\ \delta_{i2}\ ;\ \delta_{o2}\ ] \\ \Phi = [\alpha_1; (\varepsilon_1 \cup \varepsilon_2); \omega_2;\ \delta_{i1}\ ;\ \delta_{o2}\ ]\end{array}}{\Phi_1 \rhd \Phi_2 \ \hookrightarrow\ \Phi}$$

$$(\textsc{SCtxt})\,\frac{\begin{array}{c}\alpha_2 \subseteq \alpha_1 \qquad \varepsilon_1 \subseteq \varepsilon_2 \qquad \omega_2 \subseteq \omega_1 \\ \delta_{i1} \subseteq \delta_{i2} \qquad \delta_{o2} \subseteq \delta_{o1}\end{array}}{[\alpha_1; \varepsilon_1; \omega_1;\ \delta_{i1}\ ;\ \delta_{o1}\ ] \le [\alpha_2; \varepsilon_2; \omega_2;\ \delta_{i2}\ ;\ \delta_{o2}\ ]}$$

$$(\textsc{TSub})\,\frac{\begin{array}{c}\Phi'; \Gamma \vdash e : \tau' \\ \tau' \le \tau \qquad \Phi' \le \Phi\end{array}}{\Phi; \Gamma \vdash e : \tau} \qquad (\textsc{TDeref})\,\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e : ref^{\,\varepsilon}\ \tau \\ \Phi_2^{\varepsilon} = \varepsilon \qquad \Phi_2^{\delta_i} = \Phi_2^{\delta_o} \cup \varepsilon \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash\ !\,e : \tau}$$

$$(\textsc{TAssign})\,\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : ref^{\,\varepsilon}\ \tau \qquad \Phi_2; \Gamma \vdash e_2 : \tau \\ \Phi_3^{\varepsilon} = \varepsilon \qquad \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \varepsilon \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1 := e_2 : \tau}$$

$$(\textsc{TApp})\,\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \qquad \Phi_2; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \Phi_f^{\varepsilon} \qquad \Phi_3^{\delta_o} \subseteq \Phi_f^{\delta_o}\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

$$(\textsc{TTransact})\,\frac{\Phi_1; \Gamma \vdash e : \tau \qquad \Phi^{\alpha} \subseteq \Phi_1^{\alpha} \qquad \Phi^{\omega} \subseteq \Phi_1^{\omega}}{\Phi; \Gamma \vdash\ \mathsf{tx}^{(\Phi_1^{\alpha} \cup \Phi_1^{\delta_i}, \Phi_1^{\omega} \cup \Phi_1^{\varepsilon})}\ e\ : \tau}$$

$$(\textsc{TCheckin})\,\frac{\alpha \cup \delta_o \subseteq \alpha' \qquad \omega \subseteq \omega'}{[\alpha; \emptyset; \omega; \emptyset; \delta_o]; \Gamma \vdash \mathsf{checkin}^{\alpha', \omega'} : int}$$

Figure 5.11: Selected type rules for relaxed updates.

transaction can proceed normally [TX-CONG-1], or perform an update [TX-CONG-2], in which case we update the stack using the $\mathcal{U}[]$ function. If the expression in the transaction body has been reduced to a value, we can exit the transaction via the [TX-END] rule. Updates [UPDATE] can only take place if the runtime restriction κ

does not conflict with the update contents *upd*. The [CHECKIN] rule will set a new runtime restriction κ.

The definitions of update safety check, heap updates and stack updates are presented in Figure 5.13; they are straightforward extensions of the definitions used in synchronous updates (see Figure 5.7 and Section 5.3.3).

5.4.4 Soundness

The goal of our formal system is to prove that relaxed updates are version consistent. We first need to introduce some auxiliary definitions (Figure 5.14). Configuration typing and (TINTRANS) are the same as the one used in Section 5.3.3.

The definition of a well-formed transaction stack, $\mathcal{R}; H \vdash \Sigma$, differs from the one used in the synchronous case to account for κ, the runtime approximation for contextual effects. (TC1) shows the well-formedness condition for one stack element. The first premise is unchanged; it ensures that each element in the trace σ is included in the corresponding prior effect α. The second premise ensures that elements in each transaction's current effect (i.e., the part yet to be executed, up to the first checkin) have the version of that transaction: $\mathsf{f} \in (\varepsilon \cap \delta_i) \Rightarrow n \in ver(H, \mathsf{f})$. The third premise ensures that the runtime approximation of the prior effect, κ^α, covers the prior execution, and the rest of the execution up to the next check-in point. It is necessary that κ^α include not only the past execution, but part of the future execution as well, because we do not know where exactly an update will be applied in the evaluation from the current redex to the next check-in point. The

Definitions

$$\begin{array}{llll}
\text{Heaps} & H & ::= & \emptyset \mid r \mapsto (\cdot, b, \nu), H \\
 & & \mid & \mathsf{z} \mapsto (\tau, b, \nu), H \\
\text{Version sets} & \nu & ::= & \emptyset \mid \{n\} \cup \nu \\
\text{Traces} & \sigma & ::= & \emptyset \mid (\mathsf{z}, \nu) \cup \sigma \\
\text{Restrictions} & \kappa & ::= & (\alpha, \omega) \\
\text{Transaction stacks} & \Sigma & ::= & \emptyset \mid (n, \sigma, \kappa), \Sigma \\
\text{Expressions} & e & ::= & ... \mid r \mid \mathsf{tx}^{(\alpha,\omega)}\ e \mid \mathsf{intx}\ e \\
 & & & \mid \mathsf{checkin}^{(\alpha,\omega)} e \\
\text{Events} & \eta & ::= & \varepsilon \mid \mu \\
\text{Update Direction} & dir & ::= & bck \mid fwd \\
\text{Update Bundles} & \mu & ::= & (upd, dir)
\end{array}$$

Compilation

$$\begin{array}{ll}
\mathcal{C}(H; \mathsf{main}\ e) & = H; e \\
\mathcal{C}(H; \mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d) & = \mathcal{C}(H, \mathsf{f} \mapsto (\tau \longrightarrow^{\Phi} \tau', \lambda x.e, \emptyset); d) \\
\mathcal{C}(H; \mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d) & = \mathcal{C}(H, \mathsf{g} \mapsto (\tau, v, \emptyset); d)
\end{array}$$

Evaluation Contexts

$$\begin{array}{lll}
\mathbb{E} & ::= & [\,] \mid \mathbb{E}\ e \mid v\ \mathbb{E} \mid \mathsf{let}\ x = \mathbb{E}\ \mathsf{in}\ e \\
 & \mid & \mathsf{ref}\ \mathbb{E} \mid !\ \mathbb{E} \mid \mathbb{E} := e \mid r := \mathbb{E} \mid \mathsf{g} := \mathbb{E} \\
 & \mid & \mathsf{if0}\ \mathbb{E}\ \mathsf{then}\ e\ \mathsf{else}\ e
\end{array}$$

Computation

$$\begin{array}{rll}
[\text{LET}] & \langle n; (n', \sigma, \kappa); H; \mathsf{let}\ x = v\ \mathsf{in}\ e\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma, \kappa); H; e[x \mapsto v]\rangle & \\
[\text{REF}] & \langle n; (n', \sigma, \kappa); H; \mathsf{ref}\ v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma, \kappa); H[r \mapsto (\cdot, v, \emptyset)]; r\rangle & r \notin \mathrm{dom}(H) \\
[\text{DEREF}] & \langle n; (n', \sigma, \kappa); H; !\, r\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma, \kappa); H; v\rangle & H(r) = (\cdot, v, \emptyset) \\
[\text{ASSIGN}] & \langle n; (n', \sigma, \kappa); H; r := v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma, \kappa); H[r \mapsto (\cdot, v, \emptyset)]; v\rangle & r \in \mathrm{dom}(H) \\
[\text{IF-T}] & \langle n; (n', \sigma, \kappa); H; \mathsf{if0}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma, \kappa); H; e_1\rangle & \\
[\text{IF-F}] & \langle n; (n', \sigma, \kappa); H; \mathsf{if0}\ n''\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma, \kappa); H; e_2\rangle & n'' \neq 0 \\
[\text{CONG}] & \langle n; \Sigma; H; \mathbb{E}[e]\rangle \longrightarrow_{\eta} \langle n'; \Sigma'; H'; \mathbb{E}[e']\rangle & \langle n; \Sigma; H; e\rangle \longrightarrow_{\eta} \langle n'; \Sigma'; H'; e'\rangle \\
[\text{GVAR-DEREF}] & \langle n; (n', \sigma, \kappa); H; !\,\mathsf{z}\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu), \kappa); H; v\rangle & H(\mathsf{z}) = (\tau, v, \nu) \\
[\text{GVAR-ASSIGN}] & \langle n; (n', \sigma, \kappa); H; \mathsf{z} := v\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu), \kappa); H[\mathsf{z} \mapsto (\tau, v, \nu)]; v\rangle & H(\mathsf{z}) = (\tau, v', \nu) \\
[\text{CALL}] & \langle n; (n', \sigma, \kappa); H; \mathsf{z}\ v\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu), \kappa); H; e[x \mapsto v]\rangle & H(\mathsf{z}) = (\tau, \lambda x.e, \nu) \\
[\text{TX-START}] & \langle n; (n', \sigma, \kappa'); H; \mathsf{tx}^{\kappa}\ e\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma, \kappa'), (n, \emptyset, \kappa); H; \mathsf{intx}\ e\rangle & \\
[\text{TX-CONG-1}] & \langle n; (n'', \sigma, \kappa), \Sigma; H; \mathsf{intx}\ e\rangle \longrightarrow_{\mu} \langle n'; \mathcal{U}\Big[(n'', \sigma, \kappa')\Big]^{\mu}_{n'}, \Sigma'; H'; \mathsf{intx}\ e'\rangle & \langle n; \Sigma; H; e\rangle \longrightarrow_{\mu} \langle n'; \Sigma'; H'; e'\rangle \\
[\text{TX-CONG-2}] & \langle n; \Sigma; H; \mathsf{intx}\ e\rangle \longrightarrow_{\emptyset} \langle n'; \Sigma'; H'; \mathsf{intx}\ e'\rangle & \langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon} \langle n'; \Sigma'; H'; e'\rangle \\
[\text{TX-END}] & \langle n; ((n', \sigma', \kappa'), (n'', \sigma'', \kappa'')); H; \mathsf{intx}\ v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma', \kappa'); H; v\rangle & traceOK(n'', \sigma'', \kappa'') \\
[\text{UPDATE}] & \langle n; (n', \sigma, \kappa); H; e\rangle \longrightarrow_{(upd, dir)} \langle n+1; \mathcal{U}\big[(n', \sigma, \kappa)\big]^{upd, dir}_{n+1}; \mathcal{U}[H]^{upd}_{n+1}; e\rangle & updateOK(upd, H, (\kappa^{\alpha}, \kappa^{\omega}), dir) \\
[\text{CHECKIN}] & \langle n; (n', \sigma, \kappa'); H; \mathsf{checkin}^{\kappa}\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma', \kappa); H; 1\rangle &
\end{array}$$

Figure 5.12: Relaxed updates: operational semantics.

fourth premise ensures that the runtime approximation of the future effect, κ^ω, covers the current term and its continuation.

With this we can prove the core result:

Theorem 5.4.1 (Single-step Soundness). *If* $\Phi; \Gamma \vdash e : \tau$ *where* $[\![\Phi; \Gamma \vdash e : \tau]\!] = \mathcal{R}$*; and* $n; \Gamma \vdash H$*; and* $\Phi, \mathcal{R}; H \vdash \Sigma$*; and* $\mathit{traceOK}(\Sigma)$*, then either* e *is a value, or there exist* n'*,* H'*,* Σ'*,* Φ'*,* e'*, and* η *such that* $\langle n; \Sigma; H; e\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e'\rangle$ *and* $\Phi'; \Gamma' \vdash e' : \tau$ *where* $[\![\Phi'; \Gamma' \vdash e' : \tau]\!] = \mathcal{R}'$*; and* n'*;* $\Gamma' \vdash H'$*; and* $\Phi', \mathcal{R}'; H' \vdash \Sigma'$*; and* $\mathit{traceOK}(\Sigma')$ *for some* $\Phi', \Gamma', \mathcal{R}'$*.*

The proof is based on progress and preservation lemmas, as is standard. Details are in Appendix C.

5.5 Multi-threaded Version Consistency

The formal system presented in Section 5.4 ensures version consistency for relaxed updates to single-threaded programs. In this section, we extend the formalism to be able to prove version consistency for multi-threaded programs. This is essential for ensuring safety in our multi-threaded Ginseng implementation (Chapter 4). Our multi-threaded calculus extends the calculus presented in Section 5.4 with the ability to model multiple threads. Again, we only present newly-introduced rules, or rules that are different from the single-threaded calculus.

5.5.1 Syntax

The additions to the syntax are presented in Figure 5.15. In this calculus, each thread has its own transaction stack Σ and its own evaluation context e. We use $\mathcal{T}$ to denote the sequence of all threads' evaluation contexts as a sequence of pairs (Σ, e). The new expression $\mathsf{fork}^{\kappa}\ e$ models spawning a new thread: the current, parent, thread forks a new child thread whose body is e, where κ is the child thread's initial runtime effect.

5.5.2 Typing

The only new type rule is (TFork)—the rule for spawning a thread (Figure 5.16). The child thread's initial restriction has two components. The prior effect, $\alpha' \cup {\delta_i}'$, ensures that an update in the child thread takes into account the prior execution in the parent (α') and the effect of the evaluation up to the first check-in point in the child (${\delta_i}'$). The future effect consists solely of the effect of the evaluation in the child, ε'; this is modeling the "exit" after the child thread has finished execution.

5.5.3 Operational semantics.

Configuration typing (Figure 5.17) is the same as single-threaded configuration typing in Figure 5.8, $n; \Gamma \vdash H$. This is because the heap H is shared among threads, there is only one global version n, and we require well-typing for each thread.

The operational semantics extends the semantics from Section 5.4.3 with support for multiple threads. The multi-threaded semantics (Figure 5.18) models an abstract machine that non-deterministically chooses a thread and lets that thread take a step; this is similar to multi-threading formal models used by others [36, 46]. Depending on the expression in redex position, the step taken is either one from the single-threaded semantics, or a concurrency-specific step such as thread creation ([FORK]) or thread exit ([RETURN]). The evaluation rules consist of transitions of form:

$$\langle n; H; \mathcal{T} \rangle \ \rightrightarrows_{(\eta,j)} \ \langle n'; H'; \mathcal{T}' \rangle$$

where $\mathcal{T}$ is the sequence of thread contexts, j is the thread taking a step, and η is the effect of the evaluation. There is a single global version n and a single heap H, just like in the single-threaded semantics. Rule [FORK] spawns a new thread by pushing a new thread context onto $\mathcal{T}$; note that the new thread's restriction comes from the (TFork) typing rule. Rule [RETURN] does just the opposite: when a thread has finished evaluation (i.e., its evaluation context is a single value), it is removed from the context sequence. Rule [MT-UPDATE] states that we can perform a multi-threaded update if each thread can take an [UPDATE] step (Figure 5.12); this implies that no thread can conflict with the update. The crucial aspect of setting the [UPDATE] and [MT-UPDATE] rules this way is that the system can perform an *asynchronous* update, at any point, without the need to have an update in redex position (compare these rules with the synchronous [UPDATE] rule in Figure 5.6).

5.5.4 Soundness

The goal of our formal system is to prove that updating a multi-threaded program preserves version consistency. In our system, a program is version-consistent if each thread is version-consistent. Albeit simple, this definition of version consistency essentially says that transactions in different threads are unrelated, though this might be considered too lax. With this we can prove the core result:

Theorem 5.5.1 (Multi-threading Soundness)**.** *Let* $\mathcal{T} = (\Sigma_1, e_1).(\Sigma_2, e_2)\ldots(\Sigma_{|\mathcal{T}|}, e_{|\mathcal{T}|})$*. Suppose we have the following:*

1. $n \vdash H, \mathcal{T}$
2. $\forall i \in 1..|\mathcal{T}|.\ \Phi_i, \mathcal{R}_i; H \vdash \Sigma_i$
3. $\forall i \in 1..|\mathcal{T}|.\ \mathit{traceOK}(\Sigma_i)$

Then for all Σ_i *such that* $\Phi_i, \mathcal{R}_i; H \vdash \Sigma_i$*, and* $\mathit{traceOK}(\Sigma_i)$*, either* e_i *is a value, or there exist* $n', H', \mathcal{T}', j, n', \Gamma'$ *such that:*

$$n; H; \mathcal{T} \ \Rrightarrow_{(\varepsilon, j)}\ n'; H'; \mathcal{T}'$$

$$\mathcal{T}' = (\Sigma'_1, e'_1).(\Sigma'_2, e'_2)\ldots(\Sigma'_{|\mathcal{T}|}, e'_{|\mathcal{T}|})$$

and we have:

1. $n' \vdash H', \mathcal{T}'$ *(such that* $n'; \Gamma' \vdash H'$ *and* $\forall i \in 1..|\mathcal{T}|'.\ \Phi'_i; \Gamma' \vdash e'_i : \tau \rightsquigarrow \mathcal{R}'_i\ \Gamma' \supseteq \Gamma$ *and some* Φ'_i *such that*
   - $\Phi'_i = \Phi_i$*, if* $i \neq j$

- $\Phi'_i \equiv [\Phi^\alpha_i \cup \varepsilon_0; \varepsilon'_i; \Phi^\omega_i; \Phi^{\delta_i}_i; \Phi^{\delta_o}_i]$, $\varepsilon'_i \cup \varepsilon_0 \subseteq \Phi^\varepsilon_i$, *if* $i = j$

2. $\forall i \in 1..|\mathcal{T}|'.\ \Phi'_i, \mathcal{R}'_i; H' \vdash \Sigma'_i$

3. $\forall i \in 1..|\mathcal{T}|'.\ \mathit{traceOK}(\Sigma'_i)$

The proof is based on progress and preservation lemmas, as is standard. Details are in Appendix D.

5.6 Conclusion

This chapter first introduces contextual effects, which extend standard effect systems to capture the effect of the context in which each subexpression appears. Then, we present transactional version consistency, a new correctness condition for dynamic software updates. Finally, we present two formalisms based on contextual effects (relaxed updates and a multi-threaded calculus) that help us safely update multi-threaded programs.

Update Safety

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{updateOK}(\mathit{upd}, H, (\alpha,\omega), \mathit{dir}) =} \\ & \mathit{dir} = \mathit{bck} \Rightarrow \alpha \cap \mathrm{dom}(\mathit{upd}^{chg}) = \emptyset \\ \wedge & \mathit{dir} = \mathit{fwd} \Rightarrow \omega \cap \mathrm{dom}(\mathit{upd}^{chg}) = \emptyset \\ \wedge & \Gamma = \mathit{types}(H) \\ \wedge & \Gamma_{upd} = \Gamma, \mathit{types}(\mathit{upd}^{add}) \\ \wedge & \forall \mathsf{z} \mapsto (\tau, b, \cdot) \in \mathit{upd}^{chg}. \\ & \quad (\Phi_\emptyset; \Gamma_{upd} \ \vdash\ b : \tau \ \wedge\ \mathit{heapType}(\tau, \mathsf{z}) = \Gamma(\mathsf{z})) \\ \wedge & \forall \mathsf{z} \mapsto (\tau, b, \cdot) \in \mathit{upd}^{add}. \\ & \quad (\Phi_\emptyset; \Gamma_{upd} \ \vdash\ b : \tau \ \wedge\ \mathsf{z} \notin \mathrm{dom}(H)) \end{array}$$

Heap Typing Environments

$$\begin{array}{rcl} \mathit{types}(\emptyset) & = & \emptyset \\ \mathit{types}(\mathsf{z} \mapsto (\tau, b, \nu), H') & = & \mathsf{z} : \mathit{heapType}(\tau, \mathsf{z}), \mathit{types}(H') \\ \mathit{heapType}(\tau_1 \longrightarrow^{\Phi} \tau_2, \mathsf{z}) & = & \tau_1 \longrightarrow^{\Phi} \tau_2 \quad \mathsf{z} \in \Phi \\ \mathit{heapType}(\tau, \mathsf{z}) & = & \mathit{ref}^{\{\mathsf{z}\}}\ \tau \qquad \tau \neq (\tau_1 \longrightarrow^{\Phi} \tau_2) \end{array}$$

Trace Safety

$$\mathit{traceOK}(n, \sigma, \kappa) = (\forall (\mathsf{z}, \nu) \in \sigma.\ n \in \nu)$$

Heap Updates

$$\mathcal{U}[(\mathsf{z} \mapsto (\tau, b, \nu), H)]_n^{upd} = \begin{cases} \mathsf{z} \mapsto (\tau, b', \{n\}), \mathcal{U}[H]_n^{upd} \\ \quad \text{if } \mathit{upd}^{chg}(\mathsf{z}) \mapsto (\tau, b') \\ \mathsf{z} \mapsto (\tau, b, \nu \cup \{n\}), \mathcal{U}[H]_n^{upd} \\ \quad \text{otherwise} \end{cases}$$

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

$$\mathcal{U}[\emptyset]_n^{upd} = \{\mathsf{z} \mapsto (\tau, b, \{n\}) \mid \mathsf{z} \mapsto (\tau, b) \in \mathit{upd}^{add}\}$$

Trace Stack Updates

$$\begin{array}{lcl} \mathcal{U}[(n', \sigma, \kappa)]_n^{upd,fwd} & = & (n', \sigma, \kappa) \\ \mathcal{U}[(n', \sigma, \kappa)]_n^{upd,bck} & = & (n, \mathcal{U}_t[\sigma]_n^{upd}, \kappa) \\ \mathcal{U}_t[\sigma]_n^{upd} & = & \{(\mathsf{z}, \nu \cup \{n\} \mid \mathsf{z} \notin \mathrm{dom}(\mathit{upd}^{chg})\} \\ & \cup & \{(\mathsf{z}, \nu) \mid \mathsf{z} \in \mathrm{dom}(\mathit{upd}^{chg})\} \end{array}$$

Figure 5.13: Relaxed updates: heap typing, trace and update safety.

$$\text{TIntrans}\frac{\Phi_1; \Gamma \vdash e : \tau \qquad \Phi^\alpha \subseteq \Phi_1^\alpha \qquad \Phi^\omega \subseteq \Phi_1^\omega}{\Phi; \Gamma \vdash \mathsf{intx}\ e : \tau}$$

$$\frac{\begin{array}{l} \mathrm{dom}(\Gamma) = \mathrm{dom}(H) \\ \forall \mathsf{z} \mapsto (\tau \longrightarrow^{\Phi} \tau', \lambda x.e, \nu) \in H. \\ \quad \Phi; \Gamma, x : \tau \vdash e : \tau' \ \wedge \ \Gamma(\mathsf{z}) = \tau \longrightarrow^{\Phi} \tau' \ \wedge \ \mathsf{z} \in \Phi \\ \forall \mathsf{z} \mapsto (\tau, v, \nu) \in H. \quad \Phi_\emptyset; \Gamma \vdash v : \tau \ \wedge \ \Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon}\ \tau \ \wedge \ \mathsf{z} \in \varepsilon \\ \forall r \mapsto (\cdot, v, \nu) \in H. \quad \Phi_\emptyset; \Gamma \vdash v : \tau \ \wedge \ \Gamma(r) = \mathit{ref}^{\,\varepsilon}\ \tau \\ \forall \mathsf{z} \mapsto (\tau, b, \nu) \in H. \quad n \in \nu \end{array}}{n; \Gamma \ \vdash \ H}$$

$$\text{(TC1)}\frac{\begin{array}{c} \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\varepsilon \cap \delta_i\,) \Rightarrow n \in \mathit{ver}(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\alpha \cup \delta_i) \\ \kappa^\omega \supseteq (\omega \cup \varepsilon) \end{array}}{[\alpha; \varepsilon; \omega;\ \delta_i\ ;\ \delta_o\ ], \cdot; H \vdash (n, \sigma,\ \kappa\ )}$$

$$\text{(TC2)}\frac{\begin{array}{c} \Phi', \mathcal{R}; H \vdash \Sigma \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\varepsilon \cap \delta_i\,) \Rightarrow n \in \mathit{ver}(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\alpha \cup \delta_i) \\ \kappa^\omega \supseteq (\omega \cup \varepsilon) \end{array}}{[\alpha; \varepsilon; \omega;\ \delta_i\ ;\ \delta_o\ ], \Phi', \mathcal{R}; H \vdash (n, \sigma,\ \kappa\ ), \Sigma}$$

$$\text{where} \quad \mathit{ver}(H, \mathsf{f}) = \nu \ \mathit{iff} \ H(\mathsf{f}) = (\tau, b, \nu)$$

Figure 5.14: Relaxed updates: typing extensions for proving soundness

Thread Sequences	$\mathcal{T}$	$::=$	$(\Sigma, e) \mid (\Sigma, e).\mathcal{T}$
Expressions	e	$::=$	$\ldots \mid \mathsf{fork}^{\kappa}\ e$

Figure 5.15: Multi-threaded syntax

$$\text{TFork}\frac{\begin{array}{c} \Phi'; \Gamma \vdash e : \tau \rightsquigarrow \cdot \\ \Phi' \equiv [\alpha'; \varepsilon'; \omega'; \delta_i{}'; \delta_o{}'] \end{array}}{\Phi; \Gamma \vdash \mathsf{fork}^{\alpha' \cup \delta_i{}',\ \varepsilon'}\ e : \mathit{int} \rightsquigarrow \cdot}$$

Figure 5.16: Multi-threaded additions for expression typing

$$\frac{\begin{array}{c} n; \Gamma \ \vdash \ H \\ \mathcal{T} = (\Sigma_1, e_1).(\Sigma_2, e_2) \ldots (\Sigma_{|\mathcal{T}|}, e_{|\mathcal{T}|}) \\ \forall i \in 1..|\mathcal{T}|.\ \Phi_i; \Gamma \vdash e_i : \tau \rightsquigarrow \mathcal{R}_i \qquad \text{where } \Phi_i \equiv [\alpha_i; \varepsilon_i; \omega_i; \delta_{ii}; \delta_{oi}] \end{array}}{n \vdash H; \mathcal{T}}$$

Figure 5.17: Multi-threaded configuration typing

$$[\textsc{fork}] \quad \langle n; H; \mathcal{T}_1.(\Sigma_i, \mathbb{E}[\mathsf{fork}^{\kappa}\ e]).\mathcal{T}_2\rangle \ \rightrightarrows_{(\emptyset,i)} \ \langle n; H; \mathcal{T}_1.(\Sigma_i, \mathbb{E}[0]).((n,\emptyset,\kappa),e).\mathcal{T}_2\rangle$$

$$[\textsc{return}] \quad \langle n; H; \mathcal{T}_1.((n'',\sigma'',\kappa''),v).\mathcal{T}_2\rangle \ \rightrightarrows_{\emptyset} \ \langle n; H; \mathcal{T}_1.\mathcal{T}_2\rangle$$

$$[\textsc{mt-update}] \quad \frac{\begin{array}{c}\langle n; \Sigma_1; H; e_1\rangle \ \longrightarrow_{(upd,dir)} \ \langle n+1; \mathcal{U}[\Sigma_1]^{upd,dir}_{n+1}; \mathcal{U}[H]^{upd}_{n+1}; e_1\rangle \\ \langle n; \Sigma_2; H; e_2\rangle \ \longrightarrow_{(upd,dir)} \ \langle n+1; \mathcal{U}[\Sigma_2]^{upd,dir}_{n+1}; \mathcal{U}[H]^{upd}_{n+1}; e_2\rangle \\ \dots \\ \langle n; \Sigma_{|\mathcal{T}|}; H; e_{|\mathcal{T}|}\rangle \ \longrightarrow_{(upd,dir)} \ \langle n+1; \mathcal{U}\left[\Sigma_{|\mathcal{T}|}\right]^{upd,dir}_{n+1}; \mathcal{U}[H]^{upd}_{n+1}; e_{|\mathcal{T}|}\rangle\end{array}}{\begin{array}{l}\langle n; H; (\Sigma_1,e_1).(\Sigma_2,e_2)\dots(\Sigma_{|\mathcal{T}|},e_{|\mathcal{T}|})\rangle \ \rightrightarrows_{(upd,dir)} \\ \langle n+1; \mathcal{U}[H]^{upd}_{n+1}; (\mathcal{U}[\Sigma_1]^{upd,dir}_{n+1}, e_1).(\mathcal{U}[\Sigma_2]^{upd,dir}_{n+1}, e_2)\dots(\mathcal{U}\left[\Sigma_{|\mathcal{T}|}\right]^{upd,dir}_{n+1}, e_{|\mathcal{T}|})\rangle\end{array}}$$

$$[\textsc{mt-cong}] \quad \frac{\langle n; \Sigma_i; H; e\rangle \ \longrightarrow_{\eta} \ \langle n'; \Sigma'_i; H'; e'\rangle}{\langle n; H; \mathcal{T}_1.(\Sigma_i, \mathbb{E}[e]).\mathcal{T}_2\rangle \ \rightrightarrows_{(\eta,i)} \ \langle n'; H'; \mathcal{T}_1.(\Sigma'_i, \mathbb{E}[e']).\mathcal{T}_2\rangle}$$

Figure 5.18: Multi-threaded operational semantics rules.

Chapter 6

Related Work

Over the past thirty years, a variety of approaches have been proposed for dynamically updating running software. In this section we compare our approach with a few past systems, focusing on differences in functionality, safety, and updating model.

6.1 Dynamic Software Updating

6.1.1 Update Support

A large number of compiler- or library-based systems have been developed for C [42, 47, 20, 6], C++ [52, 60], Java [17, 86, 31, 70], and functional languages like ML [32, 43] and Erlang [8]. Many do not support all of the changes needed to make dynamic updates in practice. For example, updates cannot change type definitions or function prototypes [86, 31, 52, 60, 6], or else only permit such changes for abstract types or encapsulated objects [60, 43]. In many cases, updates to active code (e.g., long-running loops) are disallowed [43, 70, 42, 47, 60], and data stored in local variables may not be transformed [50, 47, 42, 52]. Some approaches are intentionally less full-featured, targeting "fix and continue" development [58, 44] or dynamic instrumentation [20]. On the other hand, Erlang [8] and Boyapati et al. [17] are both quite flexible, and have been used to build and upgrade significant applications.

Many systems employ the notion of type or state transformer, as we do. Boyapati et al. [17] improve on our interface by letting one type transformer look at the *old* representation of an encapsulated object, to allow both the parent and the child to be transformed at once. In our setting, the child will always have to be transformed independent of the parent, which can make writing transformers more complicated or impossible (e.g., if a field was moved from a child object into the parent), though we have not run into this problem as yet. Duggan [32] also proposes lazy dynamic updates to types using type transformers, using *fold*/*unfold* primitives similar to our $con_{\mathrm{T}}/abs_{\mathrm{T}}$. Ours is the first work to explore the implementation of such primitives.

The most similar system to ours is DLpop—Hicks's work on providing dynamic updating in a type-safe C-like language called Popcorn [50]. While that system is fairly flexible, this work makes three substantial improvements. First, DLpop could not transform data in local variables, could not automatically update function pointers, and had no support for updating long-running loops. We have found all of these features to be important in the server programs, and are part of our current work. Second, while DLpop ensured that all updates were type-safe, it did not ensure they were *representation-consistent* (Section 3.3), as it permitted multiple versions of a type to coexist in the running program. In particular, when a type definition changed, it required making a *copy* of existing data having the old type, opening the possibility that old code could operate on stale data. Finally, DLpop only experimented with a single program (a port of the Flash web server, about

8,000 LOC), and all updates to it were crafted by the author, rather than being official releases.

6.1.2 Correctness of Dynamic Software Updating

Several systems for on-line updates have been proposed. In this section we focus on how prior work controls update timing to assure its effects are correct. Recall that in our approach, the programmer can control update timing by placement of update points (Section 3.3.5), and Ginseng adds constraints on types that can change at an update point, to preserve type safety.

Most DSU systems disallows updates to code that is active, i.e., actually running or referenced by the call stack. The simplest approach to updating active code, taken by several recent systems [43, 69, 23], is to passively wait for it to become inactive. This can be problematic for multi-threaded programs, since there is a greater possibility that active threads reference a to-be-updated object. To address this problem, the K42 operating system [60, 103, 12, 11] employs a quiescence protocol. Once an update for an object is requested, an adapter object is interposed to block subsequent callers of the object. Once the active threads have exited, the object is upgraded and the blocked callers are resumed. The danger is that dependencies between updated objects could result in a deadlock. While applying updates based on code inactivity is useful, activeness is not sufficient for ensuring correctness—update timings allowed by the this approach can result in incorrect combinations of old and new behavior. In particular, version consistency may require delaying an update if to-be-updated objects are not currently active but were during the transaction.

Lee [63] proposed a generalization of the quiescence condition by allowing programmers to specify timing constraints on when elements of an update should occur; recent work by [24] is similar. As an example, the condition `update P, Q when P, M, S idle` specifies that procedures `P` and `Q` should be updated only when procedures `P`, `M`, and `S` are not active. Lee provides some guidance for using these conditions. For example, if procedure `P`'s type has changed, then an update to it and its callers should occur when all are inactive. Our work relies on programmer-designated transactions (which are higher-level and arguably easier to find out and specify), and uses program analysis to discover conditions that enforce transactional version consistency.

Ginseng's *updatability analysis* (Section 3.3) gathers type constraints imposed by the active (old) code at each program point and only allows an update to take place if it satisfies the constraints. This is more fine-grained than Lee's constraints—if the type of a function changes, we can update it even when its callers are active so long as they will not call the updated function directly. Our current work is complementary to this work, as a type-safe update will not necessarily be version-consistent, and depending on how transactions are specified the reverse may also be true.

Our use of transactions to ensure version consistency resembles work by [17] on lazily upgrading objects in a *persistent object store* (POS). Using a type system that guarantees object encapsulation, their system ensures that an object's transformation function, used to initialize the state of a new version based on old state, sees only objects of the old version, which is similar to our version consistency property. How updates interact with application-level transactions is less clear to us. The assumption seems to be that updates to objects are largely semantically-independent, so there is less concern about version-related dependencies between objects within a transaction.

6.1.3 Multi-threaded Systems

Several existing systems permit updates to multi-threaded programs. They tend to be either less flexible than Ginseng, or if flexible, no automatic safety support is provided, leaving the problem entirely to the programmer.

First, there are systems that do not permit updates to currently-running functions, and rely on the activeness check for safety, such as the K42 operating system [103, 12, 11], OPUS [6], or Ksplice [1]. The problem with these approaches is two-fold: 1) the activeness check does not preclude badly timed updates (see Section 4.5.1), and 2) updating long-running multi-threaded programs requires updates to active functions; our approach permits updates to running code using code extraction (Section 3.2.4).

The second category is systems that do not employ the activeness check, such as Lucos [23], Polus [24], and UpStare [68]. Lucos and Polus employ binary rewriting in function prologues to redirect calls to the new function versions, and permit updates to active code, but active functions continue to execute at the old version. This is problematic as it can lead to type safety violations and precludes updates to long-running loops.

UpStare permits dynamic updates to multi-threaded C programs using a technique called *stack reconstruction*: an update can be applied at any point in any thread, and the stack state for all threads is reconstructed according to the new stack layout in the new program version; this resembles our code extraction technique for supporting sub-function level updates to code, but UpStare's technique is more general since it does not require explicitly annotations for code to be extracted. Albeit very flexible, this approach permits updates at points where the global state is likely not consistent, e.g., in the middle of a loop, without making the update appear at the beginning or end of the iteration. Another disadvantage of UpStare is that threads need to cooperate with the update coordinator thread, in other words, to apply an update, all threads but one must be blocked; detecting whether threads are blocked is difficult. Our system uses check-ins and does not have this requirement, as the runtime system always keeps a safe approximation of each thread's effects; if a thread does not conflict with the update, it can continue executing while an update is applied.

6.1.4 Operating Systems

Chen et al. [23] have developed LUCOS, an approach to updating Linux using Xen-based virtualization. Xen is used for hardware decoupling, as well as update management (initiation, rollback). Detecting changes between versions, as well as update construction is completely manual. To update functions, stack-walking and binary rewriting will find and fix all references to old functions, and redirect them to the new versions. To update data, paging is set up to detect when an access to an old type value occurs, and, upon such accesses, a transformer function will convert the old type value to a new type value.

Although the authors present some realistic updates from the 2.6.10 → 2.6.11 patch, the approach has some room for improvement. First, LUCOS does not require quiescence for an update to be applied, but this can be problematic, as the update could be applied while updated type values are (or will be) used concretely, might have live pointers into them, updated functions are still on the stack, or function signatures change. These issues can all lead to type safety violations. Manual patch construction (finding instances of changed data, finding all the functions that changed) is also unlikely to scale, especially in a large system as the Linux kernel.

Baumann et al. [103, 10, 12] have worked on implementing dynamic updates in K42, an operating system developed at IBM Research. K42 is almost entirely object-oriented, and is written in C++. In K42, dynamic updates are performed at class level: since all the code is encapsulated behind class interfaces, a dynamic update to code or data consist of updates to one or more classes. All classes that might be subject to dynamic updating have to provide state import and state export methods, thus upon update, the new version imports the old version's exported state. This is similar to our type transformers, but in K42 import and export methods are written manually, whereas in Ginseng type transformers are generated mostly automatically. Quiescence detection [103] is based on a mechanism similar to RCU [72]. The update is split into two phases: first, while existing calls finish, all the incoming calls are tracked; second, after all untracked calls have finished, incoming calls are blocked, while waiting for the tracked calls to finish. At the end of the second phase the object is quiescent and can be dynamically updated. All objects of a certain type are accounted for using factories. Object updates can be performed either lazily, or at update time. Changes to class interfaces are supported via adapters; to avoid changing callers when the interface of an objects changes, an adapter will expose the old class interfaces to objects using the old interface. Encapsulation makes the task of updating K42 easier than updating C code: ADTs are explicit, so types and functions operating on those types change together; because all accesses go via indirection (OTT), abstraction-violating pointers are ruled out.

Ksplice [1] is a dynamic update approach for updating the Linux kernel. It supports changes to functions only (no changes to types or function signatures) by loading the new functions as a kernel module, and inserting trampolines at the entry points of old functions to redirect the callers to the new function versions. Ksplice uses the activeness check to only allow function updates if the changed function does not appear on any thread's stack, and exhibits the deficiencies of activeness-check based DSU systems.

6.1.5 DSU in Languages Other Than C

DVM [70] and DUSC [86] add dynamic updating support to Java; both systems preserve update type safety by requiring that a class update keep the class interface unchanged. DVM changes the Java virtual machine to add type safety checks upon patch loading, and uses a split phase mark-and-sweep algorithm for marking to-be-update classes at update time, and performing the update lazily. DUSC does not require changes to the JVM, because it uses a source-to source transformation tools (called proxy-, and implementation-class builders) to redirect all accesses to classes that could potentially be updated to wrapper classes. Wrapper classes use delegation to direct the access to the most recent version of an updatable class. Upon update, the wrapper class will instantiate an updated class object for each existing old instance, passing the state of the old instance as a parameter to the constructor of the new instance. Naming issues restrict the developer of updatable classes from using public or private fields directly—accesses must use methods instead. Since neither DVM nor DUSC support changes to actively-running methods or class interfaces, the scope of possible updates is limited.

DynamicML [43] uses encapsulation to permit updates to ML modules. Updates cannot change function signatures, but the new interface can add functions, types and values. DynamicML uses the ML garbage collector in an elegant fashion to support updates to values. All updateable values are tagged with a runtime type, and when the generational garbage collector performs a $from \rightarrow to$ copy operation, all tagged values are updated if needed. The garbage-collector approach can also help with rolling back an update: if during an update an exception is thrown, the system simply reverts to the $from$ space, effectively canceling any modifications performed during the update. Like K42, DynamicML uses encapsulation to permit whole-module updates without worry about module's implementation internals, at the expense of having to wait for the module to be quiescent before proceeding with the update. C has no modules or encapsulation boundaries, which allow us to update types, function signatures and active code in an unrestricted fashion; however, the burden of reasoning about safety guarantees (beyond type safety) and appropriate update points falls on the programmer.

Stewart and Chakravarty [104] provide a solution for dynamic reconfiguration and dynamic updating of Haskell applications. Their approach is based on changing the software architecture such that configuration values, application state and application code are separated; configuration and state are optional arguments to a modified application main function that is ready to accept exiting state. Dynamic update or reconfiguration is simply a matter of reinvoking main with a (possibly changed) configuration and the old state. Changes in type representations are dealt with prior to the update: before saving the global state, the old state values are converted to the new format, so that upon reloading, the new code will directly see new data. Unlike Ginseng, this approach requires changes to the software architecture; it is also not clear to us how easy it is to separate data from code, given a legacy application.

6.1.6 Edit and Continue Development

Microsoft's Visual Studio IDE allows programmers to make (and observe the effects of) small-scale changes to their applications after having started the application in the debugger. This feature, called "Edit and Continue", is available for several programming languages such as C++, C# and Visual Basic [74]. Though convenient, Edit and Continue only permits a limited set of changes to code and data, so this approach is suitable for incremental development rather than long-term software evolution. For example, the documentation for the C++ version of Edit and Continue does not support “most changes to global or static data”, or “Changes to a data type that affect the layout of an object, such as data members of a class”, or “Adding more than 64k bytes of new code or data”. In contrast, Ginseng permits such changes, but requires programs to be compiled specially, and offers no smooth integration with a debugger/IDE.

Java supports a limited class replacement mechanism, primarily meant to be used by debuggers, IDEs (for incremental development) and profilers [30]. The JVM Tool Interface (JVMTI [3]) is a programming interface that development and monitoring tools can use to communicate with (and control) the JVM. JVMTI provides a `RedefineClasses()` method that allows runtime class redefinition, but the only thing the new class can change is method bodies. Threads having old methods on the stack continue to execute the old code, but all new invocations will use the new method (JVMTI also supports a `PopFrame()` operation that can be used to discard currently running old methods by effectively popping the old method’s stack frame and returning to the call site). Method replacement in Java is hence similar to the approach we take in Ginseng—following an update, all calls to replaced functions are to new versions. Although easy to implement [29], this method replacement scheme can lead to violations of version consistency.

6.1.7 Dynamic Code Patching

Dynamic code patching is a helpful technique for debugging, instrumentation, or small-scale updates, e.g., fixing a limited-scope security bug. Dyninst [20] uses binary rewriting to replace code in a process with programmer-defined code. Dyninst supports fine-grained changes to code within a function, and works on unmodified binaries. It is difficult however, to achieve long-term code evolution using code patching only: updating data, and providing safety guarantees become necessary for non-trivial, realistic updates and multi-year changes.

6.2 Alternative (Non-DSU) Approaches to Online Updating

A typical approach to upgrading on-line systems is to use a load-balancer [18, 85]. It redirects requests away from a to-be-updated application until it is idle, at which point it can be halted and replaced with a new version. Such approaches typically employ redundant hardware, which, while required for fault-tolerance, is undesirable in some settings (e.g., upgrading a personal OS).

Microvisor [65] employs a virtual-machine monitor (VMM) to follow this basic methodology on a single node. When an application or OS on a server node is to be upgraded, a second OS instance is started concurrently on the same node and upgraded. When the original instance becomes idle, applications are restarted on the new instance and the machine is devirtualized. While Microvisor avoids the need for extra hardware, it shares the same drawbacks as the load-balancing approach: applications must be stateless (so they can be stopped and restarted) or they must be able to save their state under the old version, and then restore the state under the new version. While checkpointing [19] or process migration [102, 75] can be used to stop and restart the same version of an application, it cannot support version changes. DSU handles application state changes naturally. Since all state is visible to an update, it can be changed as necessary to be compatible with the new code.

Zap [87] combines checkpoint/restart with virtualization to support process migration in Linux. Autopod [82, 93] builds on Zap to provide OS upgrades without disrupting user applications by checkpointing user processes, starting a new OS instance, upgrading it, and reinstating the user processes on the updated machine. The range of OS upgrades is limited, however, since all the virtualization metadata layout has to be kept the same across upgrades, or converted upon process restart.

Another similar model is dependency-agnostic upgrades [33] at the system level. In this model, the new software version is installed on a different machine; old and new versions continue to run in parallel, and the data is converted gradually. While the data is migrated and converted, writes to old data are invalidated to prevent inconsistency. It is less clear to us whether the upgrade can be totally dependency-agnostic due to the presence of system-level state (e.g., open files, OS and application data in memory) which might need to be transferred as well.

While checkpointing [92, 19] or process migration [102] can be used to stop and restart the same version of an application, it cannot support version changes. DSU handles application state changes naturally. Since all state is visible to an update, it can be changed as necessary to be compatible with the new code. Indeed, one can imagine composing our approach with checkpointing to combine updating with process migration.

6.3 Software Evolution

A number of systems for identifying differences between programs have been developed. We discuss a few such systems briefly.

Yang [114] developed a system for identifying "relevant" syntactic changes between two versions of a program, filtering out irrelevant ones that would be produced by `diff`. Yang's solution matches parse trees (similar to ASTdiff) and can even match structurally different trees using heuristics. In contrast, ASTdiff stops at the very first node mismatch in order not to introduce spurious name or type bijections. Yang's tool cannot deal with variable renaming or type changes, and in general focuses more on finding a *maximum syntactic similarity* between two parse trees. We take the semantics of AST nodes into account, distinguish between different program constructs (e.g., types, variables and functions) and specific changes associated with them.

Horwitz [53] proposed a system for finding *semantic*, rather than syntactic, changes in programs. Two programs are semantically identical if the sequence of observable values they produce is the same, even if they are textually different. For example, with this approach semantics-preserving transformations such as code motion or instruction reordering would not be flagged as a change, while they would in our approach. Horwitz's algorithm runs on a limited subset of C that does not include functions, pointers, or arrays.

Binkley [14] proposes using semantic differencing to reduce the cost of regression testing—for a new version of a program, regression tests only need to be run for components whose behavior has changed. A semantic difference is defined as a difference in program's behavior, e.g., values assigned to variables, values of boolean predicates, or return values for procedures. Regression testing is performed only for components whose behavior has changed. The target language used is a simplified one that does not contain pointers, arrays, or global variables.

Jackson and Ladd [57] developed a differencing tool that analyzes two versions of a procedure to identify changes in dependencies between formals, locals, and globals. Their approach is insensitive to local variable names, like our approach, but their system performs no global analysis, does not consider type changes, and sacrifices soundness for the sake of suppressing spurious differences.

6.4 Contextual Effects

Contextual effects extend standard effect systems to capture the effect of the context in which each subexpression appears, i.e., the effect of evaluation both before and after the evaluation of the subexpression. We use contextual effects to determine the effects of past and future computation at each program point, and enforce transactional version consistency while allowing updates to occur more frequently within programs.

Several researchers have proposed extending standard effect systems [66, 83] to model more complex properties. One common approach is to use traces of actions for effects rather than sets of actions. These traces can be used to check that resources are accessed in the correct order [55], to statically enforce history-based access control [100], and to check communication sequencing [83].

Skalka et al.'s trace effects include sequencing, choice, and recursion operators to precisely describe an execution history, in contrast to standard effects which are sets of unordered events. We believe we could construct contextual versions of trace effects (and thus derive prior and future trace effects), at least for sequencing and choice, via extension to our $\rhd$ operator.

Nielson et al.'s communication analysis [83] uses a type and effect system to characterize the actions of a concurrent program and the temporal order among them. Their effects resemble our normal effects, but there are no prior or future effects in their system. However, our prior and future effects consists of sets, and do not capture order; we could imagine adding ordering to our system by using a sequencing operation on effects (;), and defining composition as $(\Phi_1 \rhd \Phi_2)^\varepsilon = \Phi_1^\varepsilon ; \Phi_2^\varepsilon$ instead of $(\Phi_1 \rhd \Phi_2)^\varepsilon = \Phi_1^\varepsilon \cup \Phi_2^\varepsilon$.

While these systems can model the ordering of events, they do not compute the prior or future effect at a program point. We believe we could combine trace effects with our approach to create a contextual trace effect system, which we leave for future work.

Hicks et al. [51] introduced *continuation effects* γ, which resemble the union $\varepsilon \cup \omega$ of our standard and future effects. Judgments in this system have the form $\gamma; \Gamma \vdash e : \tau; \gamma'$, where γ' describes the effect of e's continuation in the remainder of the program, and γ is equivalent to $\varepsilon \cup \gamma'$ where ε is the standard effect of e. The drawback of this formulation is that the standard effect ε of e cannot be recovered from γ, since $(\varepsilon \cup \gamma') - \gamma' \neq \varepsilon$ when $\varepsilon \cap \gamma' \neq \emptyset$. This system also does not include prior effects.

Capabilities in Alias Types [101] and region systems like CL [110] are likewise related to our standard and future effects. A capability consists of static approximation of the memory locations that are live in the program, and thus may be dereferenced in the current expression or in later evaluation. Because these systems assume their inputs are in continuation passing style (CPS), the effect of a continuation is equivalent to our future effects. The main differences are that we compute future effects at every program point (rather than only for continuations), that we compute prior effects, and that we do not require the input program to be CPS-converted.

Chapter 7

Future Work

In this dissertation we have shown that DSU can be practical for updating realistic applications as they are written now, and as they evolve in practice. However, the three dimensions of DSU practicality (flexibility, safety, and efficiency) could be further explored. In this section we lay out some possible directions for future work.

7.1 DSU for Other Categories of Applications

DSU is useful for long-running programs that do not tolerate reboots. In Figure 7.1 we present a spatio-temporal categorization of long-running applications, based on how long a history they keep, and where state is stored. In this dissertation we have focused on programs in quadrant 1, i.e., programs that keep long-lived, in-memory state. We now discuss the other categories of applications (quadrants 2–4) from a DSU perspective, trying to answer questions such as "Does DSU provide significant benefits for this application class?", or "What are the costs of making a certain application class updateable on-the-fly?".

1: Long history, volatile state. This is the class of applications we have focused on in this work. DSU is appealing for such programs because restarting the program causes the in-memory state to be lost, which causes disruption at the client, or performance degradation at the server. For example, restarting

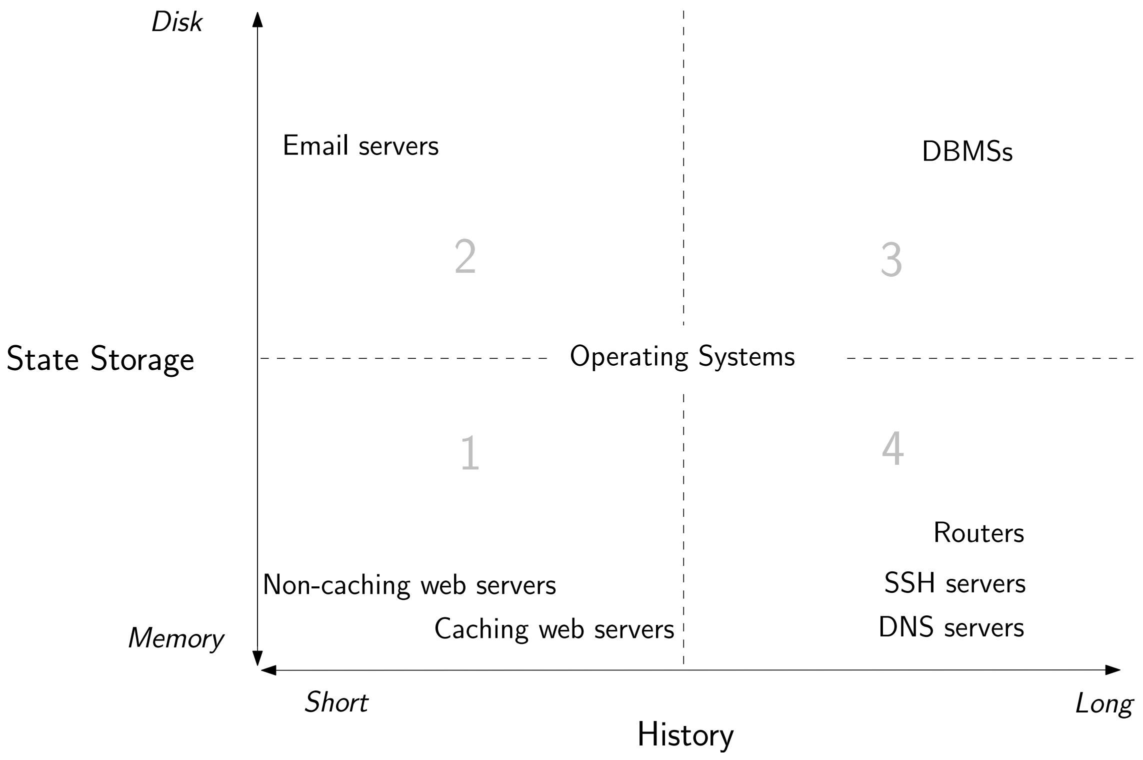

Figure 7.1: Spatio-temporal characteristics of long-running applications.

OpenSSHd shuts down all the long-lived SSH connections, which is disruptive for the clients, while restarting Icecast interrupts live streaming. Restarting Zebra causes all the routing information the server has accumulated to be lost; upon restart, the BGP, RIP and OSPF routing daemons using Zebra will have to take time to re-learn routes. Restarting Memcached will cause the underlying HTTP server to query the database for each web page, which degrades performance while the new instance of Memcached fills up its cache.

2: Short history, volatile state. Applications falling into this category are servers with short-lived connections/requests, e.g., NTP servers or non-caching web servers. The standard approaches to updating web servers are 1) deny access to new clients while the pending requests are completed, followed by update installation and restart at the new version, or 2) taking down a certain percentage of servers and updating them, and redirecting new requests away from the old web server to an updated server running the new version. These stop/restart approaches have certain disadvantages. Shutting down the server while requests are pending is undesirable, especially for e-commerce applications. Waiting for pending requests to complete might have security implications, e.g., if the update fixes a security bug that might be triggered by a pending request. Redirecting new connections to a spare web server requires spare resources. All these issues can make DSU support attractive for web servers. However, if the system is already provisioned with spare hardware or spare virtual machines for fault-tolerance, then upgrades based on redirecting new requests away from the old web server to a server running the new version make sense.

3: Short history, persistent state. An example of such an application is an email server such as Sendmail. The state of an email server (email messages) is persistent, but is under the server's control for only a short period, i.e., while the server is accepting the e-mail and delivering it to other servers or to users' mailboxes. The server's history (emails accepted and delivered in the past) is not significant to the current system state. Shutting down and restarting the server does not cause significant client disruption and slow operation during the warm-up phase. Therefore, DSU is a less compelling case for such short-history, persistent state applications.[1]

4: Long history, persistent state. Database systems make canonical examples of applications with long history and persistent state. Schemas for long-running databases need to evolve, but the current approach is tedious. It involves changing the application to use the new schema, shutting down the database, creating a new table with the new schema, writing code that populates the new table and converts the old elements in the process, and finally restarting the system. ACID semantics guarantee that shutting down a server will not lead to corrupted state. However, being able to perform on-the-fly changes to a database schema without shutting down the DBMS has several advantages [27]: the process presented above is time consuming, and mostly manual, the database is unavailable during the operation, and application soft state (e.g. caches) is lost.

Ultimately, application users have to decide whether the cost of running an updateable (DSU) application with a slight overhead outweighs the cost of having to shut down and restart the application, and bear the associated disruption or warm-up costs. In an enterprise where redundant hardware is used for fault tolerance anyway, the redirect/reboot approach might be less expensive than the cost (hardware resources, power consumption) incurred because of DSU performance

[1]Incidentally, SMTP is designed with support for high availability using mail exchangers. Exchangers act as backup servers that can accept email messages when the primary server is down, and deliver them to recipients later, when the main server comes back up. This scheme makes SMTP even more tolerant of server reboots.

overhead. On the other hand, for low-cost systems (e.g., home routers, PCs, laptops, cell phones), the cost of extra hardware might be prohibitive, and the users prefer to pay a performance penalty in exchange for the convenience of not having to reboot.

7.2 DSU for Operating Systems

It has become commonplace for OS vendors to release their patches online; end-users download the patch, install it, and restart the OS so that the patch can take effect. Experts suggest that high release frequency is needed to reduce vulnerability of end-user systems, but increased patch release frequency is burdensome for patch recipients. The reason why users and administrators are slow in applying OS patches is because they are disruptive and might introduce new bugs [97]. The disruption stems from the OS's position at the bottom of the software stack: upon restart, all the transient state is lost. The lack of confidence in the latest patch is due in part to difficulties in expressing and validating update safety.

Despite a lot of progress toward on-the-fly OS updates, updating a commodity OS's kernel (e.g., updating to the next Linux kernel version) still requires rebooting. The main reason is that OS code is low-level, sizable, complex, highly concurrent and performance-critical. This presents a significant opportunity for improvement, and is an area worth exploring in future work.

Updating an OS requires solving two main problems: finding a safe update point, and after reaching a safe update point, redirecting execution to the new version.

Finding safe update points. As explained in Section 3.3.5, update timing is crucial for update correctness, and picking a right update point is up to the programmer. In Chapter 5 we showed that to preserve version consistency, updates should be applied at program points where they do not conflict with in-flight transactions. Therefore, the challenge is to define what constitutes an OS transaction, and identify transactions in the OS code. One possible direction is to model the operating system as an event processor, where "processing an event" could be serving an interrupt/exception or running a bottom half. If we define processing an event as a transaction, we can compute contextual effects at possible update points in the OS code, and leverage our transactional version consistency framework to determine whether applying the update at a certain program point is safe. Applying static and dynamic analyses to kernel code is not trivial, due to the presence of wild casts, inline assembly, preemption and non-standard control flow [96, 16], etc. Nevertheless, recent results [35] indicate that modeling and reasoning about some of these low-level operations formally is feasible.

Redirecting execution to the new version. One approach to OS updating is to perform an *in-place* update where new code and data are loaded into a running OS. This is the solution used by Ginseng and other systems [69, 23]. An alternative would be a *checkpointing*/VM approach [65]: save the entire OS state at the old version, create a new virtual machine that runs the new OS version, and import the old state into the new machine.

The difference between these two solutions consists in how they reorganize the memory image after an update. For the in-place approach, the concern is finding, converting and accommodating data whose representation has changed, which often requires indirection and imposes some overhead. In the checkpointing/VM approach, at update time, we have to find and checkpoint all the data, convert data whose representation has changed and reinstate it in the new virtual machine, a process that is complicated and time-consuming. However, this approach offers more flexibility in dealing with changes to data representations, and there is no overhead. The ideal solution would be to combine the ease of use of the in-place approach and the low overhead/flexibility of the VM approach.

7.3 Safety of Dynamic Software Updating

As argued throughout this dissertation, safety is a paramount concern for dynamic updating, as the consequences of violating update safety can be severe.[2] In Section 3.3.5 we showed how a type safe update can go wrong due to improper timing. Therefore, we would like a DSU system to provide safety guarantees that

[2]In a recent incident [62], a nuclear power plant was forced to shut down because a software update on a monitoring system reset the data on a control system. The incident was the result of applying an otherwise safe update to only one party in a distributed system: a perfectly safe update was applied on the monitoring system only, instead of both monitoring and control systems being updated simultaneously. Allowing this inconsistency triggered data synchronization and the subsequent reset.

give application developers (and patch developers) assurances that following the update, the program will behave as intended. The question, though, is how to define update correctness.

Gupta [47] proposed the following correctness property: given program P_1 and its new version P_2, an update to the active P_1 will eventually lead to a program state reachable from a normal evaluation of P_2. That is, an update U to P_1 is correct iff for all P_1' such that $P_1 \longrightarrow^* P_1'$ there exists some P_{12}, P_2' such that $P_1' \longrightarrow_U \quad P_{12} \longrightarrow^* P_2'$ implies $P_2 \longrightarrow^* P_2'$. While illuminating, in practice this property is too strong because it cannot account for bug fixes and some kinds of feature enhancements, which are the most frequent reasons a program is changed.

As a consequence, we have to relax the update correctness condition, and define an update as correct if, following an update U to program P_1, the resulting updated program $P_1 \oplus U$ exhibits property Π. Here Π is defined in terms of a specific program verification technique. For example, when using regression testing, satisfying Π means that the updated program passes the new program's regression tests. If using type-based approaches to correctness, Π could mean that the type of a function has not changed as result of an update.

We now present several dynamic and static approaches that aim to satisfy this relaxed update correctness condition.

7.3.1 Dynamic Approaches

An obvious approach to finding update-introduced errors is to apply regression tests after the update, as we would test the new release. The accuracy of finding update correctness errors depends on the existence of a thorough test suite for the application. However, this is not sufficient, because update timing and program state might influence the outcome of the test, so to test the update, we would have to try applying it at various update points, and in various program states. This can quickly lead to an explosion of states and update points that need to be tested. An interesting area for exploration is how to reduce this possible explosion by identifying equivalent update points and equivalent states, so we only apply the update at one representative update point, and in one state, respectively, for each equivalence class.

Another approach to guaranteeing correctness after applying the update is to use checkpointing, speculations, or transactional memory and I/O [15] so we can roll back when applying an update has lead to a program crash, exception, etc. The idea is to take a checkpoint, start a transaction or speculation [108] prior to applying the update, and let the program run for a while. If the program crashes or throws an exception, we restart from the pre-update checkpoint, or abort the speculation/transaction. A similar approach named "Band-aid Patching" [99], based on Dyninst [20], forks extra processes that run the continuation of the update point in parallel, at the old and new version. If the old or new version crashes, it is discarded. However, this approach needs to deal with resource duplication, and the coexistence of old and new versions.

7.3.2 Static Approaches

Several formalisms allow expressing and checking program correctness properties: type systems, model checking, theorem proving, etc. We show how these formalisms could be used to express and check our relaxed notion of update correctness.

Type-based Approaches. Advanced type systems can capture higher-level program correctness properties. For example, if the original application contains lock acquire/release operations, one of the possible specifications for update correctness is that after applying the update, the locks are in the same state as they were in the original program. This could be modeled using *refinement types* [41, 71], *type qualifiers* [39, 25], or *dependent types* [88, 112, 113].

Type-based approaches for update safety have two main advantages. First, Ginseng performs type checking for, and has access to, both the old and new program; therefore, a straightforward comparison between program elements' types in the old and new version could reveal potential errors. Second, type checkers are fast, and with certain restrictions, decidable, so they do not place undue burden on the programmers of updateable applications.

Model Checking and Theorem Proving. Advances in *software model checking* and *theorem prover*-based verifiers have made these techniques increasingly popular.

In the last few years, a variety of checkers and verifiers have shown to be able to efficiently verify substantial, realistic programs [48, 9, 49, 38, 37]. These features make checkers and verifiers a suitable candidate for enforcing update correctness in Ginseng. Returning to our locking example we could model lock operations via a state machine, and lock usage not conforming to the state machine is flagged as an error (e.g., taking a lock twice or releasing an unlocked lock). Again, if the original application contains lock acquire/release operations, we could use model checking to ensure that, as a result of update, the locks keep their state.

7.4 Software Evolution

Understanding how software changes over time can improve our ability to build and maintain it. We have access to the history of many sizable open source programs, but, to be able to tap the potential of large source code repositories, we need to build tools to effectively "mine" repositories and try to paint a clearer image of the software evolution process. Having such tools can have beneficial implications beyond DSU systems. Apart from learning what classes of changes are frequent, and using them to design the next DSU systems, we can inform tool and language writers what artifacts would be needed to facilitate change and make software easier to maintain.

Detecting common classes of changes developers make when writing/maintaining their programs can reveal certain "evolution patterns," such as refactoring. By providing automatic support for effecting such changes in development tools and environments, we cut down on time and errors. Similarly, mining repositories for common fixes enables us to infer “bug patterns” [111] that can be generalized and added to bug detectors [54], allowing us to root out certain classes of defects early in the process.

One possible direction for continuing this work is to extend ASTdiff (Chapter 2) with support for tracking a larger set of program aspects e.g., software complexity metrics, or support for inferring patterns in the observed changes. In fact, coming up with formal models or theories that allow us to understand how software evolves is listed as one of the most important challenges of software evolution [73, 26].

Chapter 8

Conclusions

The central idea of this dissertation is that Dynamic Software Updating can be practical: programs can be updated while they run, with modest programmer effort, while providing certain update safety guarantees, and without imposing a significant performance overhead.

As evidence for this thesis, we present an approach and tool called Ginseng, that permits constructing and applying dynamic updates to C programs, along with an evaluation of Ginseng on six realistic server applications. This dissertation shows that Ginseng makes DSU practical by meeting several criteria we believe to be critical for supporting long-term dynamic updates to C programs:

- *Flexibility.* Ginseng permits updates to single- and multi-threaded C programs. The six test programs are realistic, substantial and most of them are widely used in constructing real-world Internet services. Ginseng supports changes to functions, types, and global variables, and as a result we could perform all the updates in the 10 months–4 years time frame we considered. Patches were based on actual releases, even though the developers made changes without having dynamic updating in mind.
- *Efficiency.* We had to make very few changes to the application source code. Despite the fact that differences between releases were non-trivial, generating

and testing patches was relatively straightforward. We developed tools to generate most of a dynamic patch automatically by comparing two program versions, reducing programmer work. We found that DSU overhead is modest for I/O bound applications, but more pronounced for CPU-bound applications. Our novel version consistency property improves update availability, resulting in a smaller delay between the moment an update is available and the moment the update is applied.

- *Safety.* Updates cannot be applied at arbitrary points during a program's execution, because that could lead to safety violations. Ginseng performs a suite of static safety analyses to determine times during the running program's execution at which an update can be performed safely.

In summary, this dissertation makes the following contributions:

1. A practical framework to support dynamic updates to single- and multi-threaded C programs. Ours is the most flexible, and arguably the most safe, implementation of a DSU system to date.

2. A substantial study of the application of our system to six sizable C server programs, three single-threaded, and three multi-threaded, over long periods of time ranging from 10 months to 4 years worth of releases.

3. A novel type-theoretical system that generalizes standard effect systems, called *contextual effects*; contextual effects are useful when the past or future computation of the program is relevant at various program points, and have ap-

plications beyond DSU. We also present a formalism and soundness proof for our novel update correctness property, version consistency, which permits us to provide certain update safety guarantees for single- and multi-threaded programs

4. An approach for comparing the source code of different versions of a C program, as well as a software evolution study of various versions of popular open source programs, including BIND, OpenSSH, Apache, Vsftpd and the Linux kernel.

Appendix A

Developing Updateable Software Using Ginseng

When developing updateable software with Ginseng, the programmer might need to intervene at two stages: when compiling the initial program, and when generating dynamic patches. We will touch on these two aspects in turn.

A.1 Preparing Initial Sources

To prepare C programs for compilation with Ginseng, developers might have to annotate the source code with **#pragma** directives which indicate where the update points are, which loops ought to be extracted, which functions act like `malloc`, etc.

A.1.1 Specifying Update Points

Update points in single-threaded programs and semantic update points in multi-threaded programs are points in the program at which there are no partially-completed transactions, and all global state is consistent (see Sections 3.5.3 and 4.2). Dynamic updates are best applied at such quiescent points, and preferably those that are stable throughout a system's lifetime. If the application/thread is structured around an event processing loop, the end of the loop defines a stable quiescent point: there are no pending function calls, little or no data on the stack, and the global state is consistent. Once the user has identified such quiescent points in the

```
 ...
f();
g();
 ...
```

Original program

```
#pragma DSU_extract("FOO")

...
  FOO: {
    f();
    g();
  }
...
```

Program prepared for code extraction

Figure A.1: Directing Ginseng to perform code extraction.

program, an update point can be specified by inserting a call to the runtime system: DSU_update().

A.1.2 Code Extraction

Ginseng cannot replace code on the stack, but can replace functions via function indirection (Section 3.2). Thus, to replace a code block on the stack, we can direct Ginseng to extract the block in a separate function, a technique called code extraction (Section 3.2.4). The user can request code extraction as indicated in Figure A.1. The first step is delimiting the code to be extracted, by adding a labeled scope (e.g., FOO), and the second step is adding a **#pragma** DSU_extract("FOO") to the source file.

Ginseng also supports a combination of loop body extraction and automatically-inserted update points at the end of an iteration of long-running loops using the directive **#pragma** DSU_loopupd(''label'').

A.1.3 Memory Allocation Functions

Ginseng treats malloc and other memory allocation functions specially. Since these functions (named abs_{T} [106]) are used to construct wrapped type values, Ginseng has to properly initialize the version field of a type wrapped value (Section 3.2.2). Ginseng recognizes malloc, calloc and alloca by default, but sometimes the applications use custom memory allocators, hence the names of allocation functions have to be communicated to the compiler using **#pragma** DSU_malloc("function_name"). For instance, OpenSSH uses a custom function for memory allocation called "xmalloc", so the user has to notify Ginseng by adding the following line to the original program: **#pragma** DSU_malloc("xmalloc").

A.1.4 Analysis

As described in Section 3.5.3, Ginseng performs safety analyses to detect types used in a representation-dependent way that hampers future changes in a type's representation. For example, uses of **sizeof** or unsafe type casts that are legal in the current program version might become illegal in future versions, once the type representation has changed. A type used in an illegal fashion is deemed *non-updateable*; Ginseng will not use the type wrapping scheme for such a type, and its representation cannot change in future versions.

The programmer might have to guide Ginseng's safety analysis in certain cases. Since the analysis is monomorphic, it will not detect universal or existential uses of types, rendering certain types non-updateable, although they are used in a type-safe, representation-independent fashion. On the other hand, the analysis might deem a type updateable, but the programmer needs to have a fixed, non-wrapped representation for the type in question.

To override the analysis and force a type (non)updateable, Ginseng provides two pragma primitives, **#pragma** DSU_FORCE_NONUPDATABLE and **#pragma** DSU_FORCE_UPDATABLE. Their use is detailed below.

Whenever Ginseng encounters an "illegal" type use, it prints out an error message of the form:

```
(<source file>:<line>) setTypeNonUpdatable(<type name>) (<reason>)
```

This points the programmer to the offending source code line; there are cases when changes to the source code eliminate the offending use (e.g., instantiating an existential). When such changes are not effective, the last resort is forcing types updateable. For example, when updating Sshd we had to use the directive **#pragma** DSU_FORCE_UPDATABLE("struct_Channel") to tell Ginseng that an existential use of **struct** Channel is update-safe. Conversely, when updating Vsftpd we used a **#pragma** DSU_FORCE_NONUPDATABLE("struct_vsf_sysutil_ipv4addr") to prevent Ginseng from wrapping **struct** vsf_sysutil_ipv4addr whose representation must exactly match the IP address format.

C lacks support for universal or existential polymorphism, so programmers have to resort to using **void** $*$ for polymorphism. Ginseng checks all upcasts to **void** $*$ and downcasts from **void** $*$ to ensure no type "laundering" occurs (Section 3.3.3). Ginseng tracks all upcasts from an abstract type pointer T $*$ into **void** $*$ by annotating the **void** $*$ and tracking its subsequent flow. If a **void** $*$ flows to an abstract type pointer S $*$, with $\mathsf{T} \neq \mathsf{S}$, both S and T are set non-updateable, to avoid representation inconsistencies. Whenever a downcast to S $*$ from a **void** $*$ with annotation T $*$, U $*$, V $*$, etc. is encountered, Ginseng emits an error message of the form:

```
(<source file>:<line>) printVoidConstraints <S> <= <T U V>
```

The user can then decide whether this a benign or problematic cast. Note that having non-updateable types in a program limits the range of possible updates, so the developers should strive to reduce the number of such types.

A.1.5 Check-ins

Ginseng supports both synchronous (barrier-based) and asynchronous (relaxed) updates. Synchronous updates take place at an user-specified update point (marked by a user-inserted DSU_update(), as explained in Section 3.3). Asynchronous updates take place at an induced update point or an arbitrary, though safe, point inside a scoped check-in block (Section 4.2). To designate a check-in block, the user simply adds curly braces around the code block and a DSU_CHECKIN_<name> label to the block, as shown in Figure A.2; no **#pragma** is necessary. Scoped check-ins "snapshot" a safe approximation of thread's current restriction plus the effects of executing the block; the result of this is that the effects of the block will appear in both the prior and future restrictions for the entire execution of the block (Section 4.2.2). While, in our example, this prevents f, g, and s from changing, the advantage is that multi-threaded programs can perform updates without the need

```
struct S { int i; };
 ...
struct S s;
 ...
f();
s.i = 0;
g();
 ...
```

```
struct S { int i; };
 ...
struct S s;
 ...
  DSU_CHECKIN_1: {

    // {S,f,g} = α     {S,f,g} = ω

    f();
    s.i = 0;
    g();

    // {S,f,g} = α     {} = ω
  }
 ...
```

Original program   Program annotated with check-ins

Figure A.2: Directing Ginseng to perform check-ins.

for blocking synchronization—as long as all threads have check-in effects that do not conflict with the update, the update can be performed right away.

A.2 Dynamic Patches

Dynamic patches are generated mostly automatically by Ginseng, but (depending on the nature of changes between versions), the programmer might still have to complete the auto-generated type transformers and write state transformers (Section 3.4). Source code for patches consists of two files: a .patch.custom.c file containing state and type transformers, which can be tailored by the programmer, and a .patch.gen.c containing definitions of new (or changed) types and functions. Ginseng generates both these files automatically, but the programmer is only supposed to alter the former.

We now provide examples of completing auto-generated type transformers and writing state transformers.

```
// OLD program, sshd 3.7.1p2
struct Authct_old {
  int             failures ;
  char           *user;
  char           * service ;
  struct passwd  *pw;
  char           * style ;
};

// NEW program, sshd 3.8p1
struct Authct_new {
  int             failures ;
  int             force_pwchange;
  char           *user;
  char           * service ;
  struct passwd  *pw;
  char           * style ;
};
```

(a) Source code

```
void tt_Authct(struct Authct_old *xin,
               struct Authct_new *xout) {
  xout→ failures = xin→ failures;
  xout→ force_pwchange = ??;
  xout→ user = xin→ user;
  xout→ service = xin→ service;
  xout→ pw = xin→ pw;
  xout→ style = xin→ style;
}
```

(c) Type Transformer

Figure A.3: Type transformer example.

A.2.1 Type Transformers

When type representations change, *type transformers* will convert values from the old representation to the new one. Ginseng compiler automatically generates type transformer skeletons containing "best guess" conversion functions between representations, but the programmer still has to intervene in order to verify the auto-generated conversions and add initialization code where needed. For instance, if a **struct** type has changed, the stub consists of code to copy the preserved fields over from the old to the new definition, and the programmer will have to initialize newly added fields. Type transformers have the following signature:

```
void DSU_tt_type(type_old *xin, type_new *xout, wrapped_type *xnew)
```

The arguments are pointers to the concrete type representations (xin and xout) and to the wrapped representation (xnew); most of the time, xin and xout are sufficient for

```
// OLD program, zebra 0.93b
struct route_table * rib_table_ipv4 ;
struct route_table * static_table_ipv4 ;

struct route_table * rib_table_ipv6 ;
struct route_table * static_table_ipv6 ;

// NEW program, zebra 0.94
struct route_table *vrf [4];
```

(a) Source code

```
void DSU_state_xform() {
  vrf [0] = rib_table_ipv4 ;
  vrf [1] = rib_table_ipv6 ;
  vrf [2] = static_table_ipv4 ;
  vrf [3] = static_table_ipv6 ;
}
```

(b) State Transformer

Figure A.4: State transformer example.

writing the conversion function, but when converting linked structures e.g., trees or lists, xnew is needed as well. In most cases type is a **struct**, and the effort consists of initializing newly added fields.

As an example, in Figure A.3 we show the Ginseng-generated type transformer for **struct** Authct in the update from Sshd version 3.7.1p2 to version 3.8p1. The new version adds a field force_pwchange (line 13). Ginseng generates code to copy the existing fields, but the programmer has to write the correct initializer for the newly-introduced field. Depending on when or how the new code uses the newly added fields, writing the type transformer can range from trivial (assigning a default value) to impossible (Section 3.5.3).

If no type has changed, the auto-generated .patch.custom.c will be empty, meaning there are no type transformers to be filled out. Note however that state transformers (described in the next section) might still be necessary.

A.2.2 State Transformers

A state transformer is an optional function supplied by the programmer and invoked by the runtime system run at update time (Section 3.4). The purpose of state transformers is two-fold: 1) to convert global state and establish the invariants the new program version expects, and 2) to run initialization code the new program depends on, but is not part of the old program's initialization code.

Since a state transformer function is optional, it is not included by default in the .patch.custom.c; the programmer has to add it using the following skeleton:

```
void DSU_state_xform() {   ...  }
```

As an example, in Figure A.4 we show the state transformer we had to write for the update from Zebra version 0.93b to version 0.94. We see that the old version keeps routing tables in four different global variables (rib_table_ipv4, static_table_ipv4, rib_table_ipv6, and static_table_ipv6), whereas the new version uses a routing table array, vrf. The state transformer makes the array elements point the associated routing table.

Just like in the type transformer case, state transformer complexity can range from trivial (if at all needed) to impossible (e.g., if at boot time hardware is initialized differently by the old and the new program). The most complicated cases we have encountered were refactorings of global structures where global state had to be transferred between the old and new storage model.

Appendix B
Proteus-tx Proofs

Lemma B.0.1 (Weakening). *If $\Phi; \Gamma \vdash e : \tau$ and $\Gamma' \supseteq \Gamma$ then $\Phi; \Gamma' \vdash e : \tau$.*

Proof. By induction on the typing derivation of $\Phi; \Gamma \vdash e : \tau$. □

Lemma B.0.2 (Flow subtyping). *If $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$ then $\Phi_1 \leq \Phi$ and $\Phi_2 \leq \Phi$.*

Proof. Follows directly from the definitions. □

Lemma B.0.3 (Subtyping reflexivity). *$\tau \leq \tau$ for all τ.*

Proof. Straightforward, from the definition of subtyping in Figure 5.2. □

Lemma B.0.4 (Subtyping transitivity). *For all τ, τ', τ'', if $\tau \leq \tau'$ and $\tau' \leq \tau''$ then $\tau \leq \tau''$.*

Proof. By simultaneous induction on $\tau \leq \tau'$ and $\tau' \leq \tau''$. Notice that subtyping is syntax-directed, and this forces the final rule of each derivation to be the same:

case (SINT,SINT) **:**

From the definition of (SINT), we have $int \leq int$, hence $\tau \leq \tau''$ follows directly.

case (SREF,SREF) **:**

We have:

$$\text{SRef}\frac{\tau \leq \tau' \quad \tau' \leq \tau \quad \varepsilon \subseteq \varepsilon'}{ref^{\varepsilon}\ \tau \leq ref^{\varepsilon'}\ \tau'} \quad \text{SRef}\frac{\tau' \leq \tau'' \quad \tau'' \leq \tau' \quad \varepsilon' \subseteq \varepsilon''}{ref^{\varepsilon'}\ \tau' \leq ref^{\varepsilon''}\ \tau''}$$

We know that $\varepsilon \subseteq \varepsilon' \wedge \varepsilon' \subseteq \varepsilon'' \Rightarrow \varepsilon \subseteq \varepsilon''$, and by induction we have that $\tau \leq \tau' \wedge \tau' \leq \tau'' \Rightarrow \tau \leq \tau''$ and $\tau' \leq \tau \wedge \tau'' \leq \tau' \Rightarrow \tau'' \leq \tau$, respectively.
We can now apply (SREF):

$$\text{SRef}\frac{\tau \leq \tau'' \quad \tau'' \leq \tau \quad \varepsilon \subseteq \varepsilon''}{ref^{\varepsilon}\ \tau \leq ref^{\varepsilon''}\ \tau''}$$

case (SFUN,SFUN) **:**

We have:

$$\text{SFun}\frac{\begin{array}{c}\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \\ \Phi \leq \Phi'\end{array}}{\tau_1 \longrightarrow^{\Phi} \tau_2 \leq \tau_1' \longrightarrow^{\Phi'} \tau_2'} \quad \text{SFun}\frac{\begin{array}{c}\tau_1'' \leq \tau_1' \quad \tau_2' \leq \tau_2'' \\ \Phi' \leq \Phi''\end{array}}{\tau_1' \longrightarrow^{\Phi'} \tau_2' \leq \tau_1'' \longrightarrow^{\Phi''} \tau_2''}$$

We know that $\Phi \leq \Phi' \wedge \Phi' \leq \Phi'' \Rightarrow \Phi \leq \Phi''$, and by induction (see (SREF,SREF) above) we have $\tau_1'' \leq \tau_1$ and $\tau_2 \leq \tau_2''$.

We can now apply (SFUN):

$$\text{SFun}\frac{\begin{array}{c}\tau_1'' \leq \tau_1 \quad \tau_2 \leq \tau_2'' \\ \Phi \leq \Phi''\end{array}}{\tau_1 \longrightarrow^{\Phi} \tau_2 \leq \tau_1'' \longrightarrow^{\Phi''} \tau_2''}$$

□

Lemma B.0.5 (Value typing). *If $\Phi; \Gamma \vdash v : \tau$ then $\Phi'; \Gamma \vdash v : \tau$ for all Φ'.*

Proof. By induction on the typing derivation of $\Phi; \Gamma \vdash v : \tau$.

case (TINT) **:**

Thus $v \equiv n$ and we prove the result as follows:

$$\text{TSub}\frac{\text{TInt}\dfrac{}{\Phi_\emptyset; \Gamma \vdash n : int} \quad int \leq int \quad \text{SCtxt}\dfrac{\Phi_\emptyset \equiv \Phi_\emptyset \quad \Phi' \equiv [\alpha; \varepsilon'; \omega] \quad \emptyset \subseteq \varepsilon'}{\Phi_\emptyset \leq \Phi'}}{\Phi'; \Gamma \vdash n : int}$$

case (TGVAR) **:**

We have

$$\text{TGvar}\frac{\Gamma(\mathsf{f}) = \tau}{\Phi_\emptyset; \Gamma \vdash \mathsf{f} : \tau}$$

We prove the result as follows:

$$\text{TSub}\frac{\text{TGvar}\dfrac{\Gamma(\mathsf{f}) = \tau}{\Phi_\emptyset; \Gamma \vdash \mathsf{f} : \tau} \qquad \tau \leq \tau \qquad \text{SCtxt}\dfrac{\Phi_\emptyset \equiv \Phi_\emptyset \qquad \Phi' \equiv [\alpha; \varepsilon'; \omega] \qquad \emptyset \subseteq \varepsilon'}{\Phi_\emptyset \leq \Phi'}}{\Phi'; \Gamma \vdash \mathsf{f} : \tau}$$

case (TLOC) **:**

Similar to (TGVAR).

case (TSUB) **:**

We have

$$\text{TSub}\frac{\Phi''; \Gamma \vdash v : \tau' \qquad \tau' \leq \tau \qquad \text{SCtxt}\dfrac{\Phi'' \equiv [\alpha; \varepsilon''; \omega] \qquad \Phi \equiv [\alpha; \varepsilon; \omega] \qquad \varepsilon'' \subseteq \varepsilon}{\Phi'' \leq \Phi}}{\Phi; \Gamma \vdash v : \tau}$$

The result follows by induction on $\Phi''; \Gamma \vdash v : \tau'$ and by applying [TSUB].

□

Lemma B.0.6 (Subtyping Derivations)**.** *If $\Phi; \Gamma \vdash e : \tau$ then we can construct a proof derivation of this judgment that ends in one use of* (TSUB) *whose premise uses a rule other than* (TSUB)*.*

Proof. By induction on $\Phi; \Gamma \vdash e : \tau$.

case (TSUB) **:**

We have

$$\text{TSub}\frac{\Phi'; \Gamma \vdash e : \tau' \qquad \tau' \leq \tau \qquad \text{SCtxt}\dfrac{\Phi'^{\varepsilon} \subseteq \Phi^{\varepsilon} \qquad \Phi^{\alpha} \subseteq \Phi'^{\alpha} \qquad \Phi^{\omega} \subseteq \Phi'^{\omega}}{\Phi' \leq \Phi}}{\Phi; \Gamma \vdash e : \tau}$$

By induction, we have

$$\text{TSub}\frac{\Phi''; \Gamma \vdash e : \tau'' \qquad \tau'' \leq \tau' \qquad \text{SCtxt}\dfrac{\Phi''^{\varepsilon} \subseteq \Phi'^{\varepsilon} \qquad \Phi'^{\alpha} \subseteq \Phi''^{\alpha} \qquad \Phi'^{\omega} \subseteq \Phi''^{\omega}}{\Phi'' \leq \Phi'}}{\Phi'; \Gamma \vdash e : \tau'}$$

where the derivation $\Phi''; \Gamma \vdash e : \tau''$ does not conclude with (TSUB). By the transitivity of subtyping (Lemma B.0.4), we have $\tau'' \leq \tau$; we also have $\varepsilon'' \subseteq \varepsilon$ and finally we get the desired result by (TSUB):

$$\text{TSub}\frac{\Phi''; \Gamma \vdash e : \tau'' \qquad \tau'' \leq \tau \qquad \text{SCtxt}\dfrac{\Phi''^{\varepsilon} \subseteq \Phi^{\varepsilon} \qquad \Phi^{\alpha} \subseteq \Phi''^{\alpha} \qquad \Phi^{\omega} \subseteq \Phi''^{\omega}}{\Phi'' \leq \Phi}}{\Phi; \Gamma \vdash e : \tau}$$

case all others **:**

Since we have that the last rule in $\Phi; \Gamma \vdash e : \tau$ is not (TSUB), we have the desired result by applying (TSUB) (where $\tau \leq \tau$ follows from the reflexivity of subtyping, Lemma B.0.3):

$$\text{TSub}\frac{\Phi; \Gamma \vdash e : \tau \qquad \tau \leq \tau \qquad \Phi \leq \Phi}{\Phi; \Gamma \vdash e : \tau}$$

□

Lemma B.0.7 (Flow effect weakening)**.** *If $\Phi; \Gamma \vdash e : \tau$ where $\Phi \equiv [\alpha; \varepsilon; \omega]$, then $\Phi'; \Gamma \vdash e : \tau$ where $\Phi' \equiv [\alpha'; \varepsilon; \omega']$, $\alpha' \subseteq \alpha$, and $\omega' \subseteq \omega$, and all uses of [*TSUB*] applying $\Phi' \leq \Phi$ require $\Phi'^{\omega} = \Phi^{\omega}$ and $\Phi'^{\alpha} = \Phi^{\alpha}$.*

Proof. By induction on $\Phi; \Gamma \vdash e : \tau$.

case (TGvar),(TInt),(TVar) **:**

Trivial.

case (TUpdate) **:**

We have

$$\text{TUpdate}\,\dfrac{\alpha \subseteq \alpha'' \qquad \omega \subseteq \omega''}{(\Phi_\emptyset); \Gamma \vdash \mathsf{update}^{\alpha'',\omega''} : \mathit{int}}$$

Since $\alpha' \subseteq \alpha$ and $\omega' \subseteq \omega$ we can apply (TUpdate):

$$\text{TUpdate}\,\dfrac{\alpha' \subseteq \alpha'' \qquad \omega' \subseteq \omega''}{([\alpha'; \varepsilon; \omega']); \Gamma \vdash \mathsf{update}^{\alpha',\omega'} : \mathit{int}}$$

case (TTransact) **:**

We have

$$\text{TTransact}\,\dfrac{\begin{array}{c}\Phi''; \Gamma \vdash e : \tau \\ \Phi^\alpha \subseteq \Phi''^\alpha \qquad \Phi^\omega \subseteq \Phi''^\omega\end{array}}{\Phi; \Gamma \vdash \mathsf{tx}\; e : \tau}$$

Let $\Phi' = [\alpha'; \varepsilon; \omega']$. Since $\Phi'^\alpha \subseteq \Phi^\alpha$ and $\Phi'^\omega \subseteq \Phi^\omega$ we can apply (TTransact):

$$\text{TTransact}\,\dfrac{\begin{array}{c}\Phi''; \Gamma \vdash e : \tau \\ \Phi'^\alpha \subseteq \Phi''^\alpha \qquad \Phi'^\omega \subseteq \Phi''^\omega\end{array}}{\Phi'; \Gamma \vdash \mathsf{tx}\; e : \tau}$$

case (TIntrans) **:**

Similar to (TTransact).

case (TSub) **:**

We have

$$\text{TSub}\,\dfrac{\Phi'; \Gamma \vdash e : \tau' \qquad \tau' \leq \tau \qquad \dfrac{\Phi'^\varepsilon \subseteq \Phi^\varepsilon \qquad \Phi^\omega \subseteq \Phi'^\omega \qquad \Phi^\alpha \subseteq \Phi'^\alpha}{\Phi' \leq \Phi}}{\Phi; \Gamma \vdash e : \tau}$$

Let $\Phi'' = [\Phi^\alpha; \Phi'^\varepsilon; \Phi^\omega]$. Thus we have

$$\text{TSub}\,\dfrac{\Phi''; \Gamma \vdash e : \tau' \qquad \tau' \leq \tau \qquad \dfrac{\Phi''^\varepsilon \subseteq \Phi^\varepsilon \qquad \Phi^\omega = \Phi''^\omega \qquad \Phi^\alpha = \Phi''^\alpha}{\Phi'' \leq \Phi}}{\Phi; \Gamma \vdash e : \tau}$$

where the first premise follows by induction (which we can apply because $\Phi''^\omega \subseteq \Phi'^\omega$ and $\Phi''^\alpha \subseteq \Phi'^\alpha$ by assumption); the first premise of $\Phi'' \leq \Phi$ is by assumption, and the latter two premises are by definition of Φ''.

case (TRef) **:**

We know that

$$\text{TRef}\,\dfrac{\Phi; \Gamma \vdash e : \tau}{\Phi; \Gamma \vdash \mathsf{ref}\;\; e : \mathit{ref}^{\,\varepsilon}\; \tau}$$

and have $\Phi'; \Gamma \vdash e : \tau$ by induction, hence we get the result by (TRef).

case (TDeref) **:**

We know that

$$\text{TDeref}\,\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash e : \mathit{ref}^{\,\varepsilon}\; \tau \\ \Phi_2^\varepsilon = \varepsilon \qquad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash {!}\,e : \tau}$$

We have $\Phi' \equiv [\alpha'; \Phi_1^\varepsilon \cup \Phi_2^\varepsilon; \omega']$ where $\alpha' \subseteq \Phi^\alpha$ and $\omega' \subseteq \Phi^\omega$. Choose $\Phi_1' \equiv [\alpha'; \Phi_1^\varepsilon; \Phi_2^\varepsilon \cup \omega']$ and $\Phi_2' \equiv [\alpha' \cup \Phi_1^\varepsilon; \Phi_2^\varepsilon; \omega']$, hence $\Phi_1' \longrightarrow \Phi_2'$, $\Phi_2'^\varepsilon = \Phi_2^\varepsilon = \varepsilon$, and $\Phi' \equiv \Phi_1' \rhd \Phi_2'$. We want to prove that $\Phi'; \Gamma \vdash {!}\,e : \tau$. Since $\alpha' \subseteq \alpha$ and $\Phi_2^\varepsilon \cup \omega' \subseteq \Phi_2^\varepsilon \cup \omega$ we can apply induction to get $\Phi_1'; \Gamma \vdash e : \mathit{ref}^{\,\varepsilon}\; \tau$ and we get the result by applying (TDeref):

$$\text{TDeref}\,\dfrac{\begin{array}{c}\Phi_1'; \Gamma \vdash e : \mathit{ref}^{\,\varepsilon}\; \tau \\ \Phi_2'^\varepsilon = \varepsilon \qquad \Phi_1' \rhd \Phi_2' \hookrightarrow \Phi'\end{array}}{\Phi'; \Gamma \vdash {!}\,e : \tau}$$

case (TRet) **:**

Similar to [TDeref].

case (TApp) **:**

We know that

$$\text{TApp}\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \qquad \Phi_2; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega}\end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2}$$

We have $\Phi' \equiv [\alpha'; \Phi_1^{\varepsilon} \cup \Phi_2^{\varepsilon} \cup \Phi_3^{\varepsilon}; \omega']$ where $\alpha' \subseteq \Phi^{\alpha}$ and $\omega' \subseteq \Phi^{\omega}$. Choose $\Phi_1' \equiv [\alpha'; \Phi_1^{\varepsilon}; \Phi_2^{\varepsilon} \cup \Phi_3^{\varepsilon} \cup \omega']$, $\Phi_2' \equiv [\alpha' \cup \Phi_1^{\varepsilon}; \Phi_2^{\varepsilon}; \Phi_3^{\varepsilon} \cup \omega']$, $\Phi_3' \equiv [\alpha' \cup \Phi_1^{\varepsilon} \cup \Phi_2^{\varepsilon}; \Phi_3^{\varepsilon}; \omega']$, hence $\Phi_3'^{\varepsilon} = \Phi_3^{\varepsilon} = \varepsilon_f$, and $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'$. We want to prove that $\Phi'; \Gamma \vdash e_1\ e_2 : \tau_2$. Since $\alpha' \subseteq \alpha$ and $\Phi_2^{\varepsilon} \cup \Phi_3^{\varepsilon} \cup \omega' \subseteq \Phi_2^{\varepsilon} \cup \Phi_3^{\varepsilon} \cup \omega'$ we can apply induction to get $\Phi_1'; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2$. Similarly, since $\alpha' \cup \Phi_1^{\varepsilon} \subseteq \alpha \cup \Phi_1^{\varepsilon}$ and $\Phi_3^{\varepsilon} \cup \omega' \subseteq \Phi_3^{\varepsilon} \cup \omega$, we can apply induction to get $\Phi_2'; \Gamma \vdash e_2 : \tau_1$. We get the get the result by applying (TApp):

$$\text{TApp}\frac{\begin{array}{c}\Phi_1'; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f'} \tau_2 \qquad \Phi_2'; \Gamma \vdash e_2 : \tau_1 \\ \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi' \\ \Phi_3'^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3'^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3'^{\omega} \subseteq \Phi_f^{\omega}\end{array}}{\Phi'; \Gamma \vdash e_1\ e_2 : \tau_2}$$

case (TAssign), (TIf), (TLet) **:**

Similar to (TApp).

□

Definition B.0.8. *If* $\Phi; \Gamma \vdash e : \tau$, $[\![\Phi; \Gamma \vdash e : \tau]\!] = \mathcal{R}$, *and* $\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}'$ *then* $\mathcal{R} \equiv \mathcal{R}'$, *where* $[\![\Phi; \Gamma \vdash e : \tau]\!]$ *is defined in Figure B.1.*

Lemma B.0.9 (Left subexpression version consistency)**.** *If* $\Phi, \mathcal{R}; H \vdash \Sigma$ *and* $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$ *then* $\Phi_1, \mathcal{R}; H \vdash \Sigma$.

Proof. We know that $\Phi_1 \rhd \Phi_2 \equiv [\alpha_1; \varepsilon_1 \cup \varepsilon_2; \omega_2]$. We have two cases:

$\mathcal{R} \equiv \cdot$: Thus $\Sigma \equiv (\beta, \sigma)$ and by assumption we have:

$$\text{TC1}\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \cup \varepsilon_2 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{[\alpha_1; \varepsilon_1 \cup \varepsilon_2; \omega_2], \cdot; H \vdash (\beta, \sigma)}$$

The result follows from [TC1]:

$$\text{TC1}\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{[\alpha_1; \varepsilon_1; \omega_1], \cdot; H \vdash (\beta, \sigma)}$$

$\mathcal{R} \equiv \Phi', \mathcal{R}'$: Thus we must have

$$\text{TC2}\frac{\begin{array}{c}\Phi', \mathcal{R}'; H \vdash \Sigma' \\ \Phi \equiv [\alpha_1; \varepsilon_1 \cup \varepsilon_2; \omega_2] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \cup \varepsilon_2 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{\Phi, \Phi', \mathcal{R}'; H \vdash ((\beta, \sigma), \Sigma')}$$

where $\Sigma \equiv ((\beta, \sigma), \Sigma')$. The result follows by [TC2]:

$$\text{TC2}\frac{\begin{array}{c}\Phi', \mathcal{R}'; H \vdash \Sigma' \\ \Phi_1 \equiv [\alpha_1; \varepsilon_1; \omega_1] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{\Phi_1, \Phi', \mathcal{R}'; H \vdash ((\beta, \sigma), \Sigma')}$$

where the first premise follows by assumption.

□

$$\left[\!\!\left[(\textsc{TInt})\frac{}{\Phi_\emptyset;\Gamma\vdash n : int}\right]\!\!\right] = \cdot \qquad \left[\!\!\left[(\textsc{TVar})\frac{\Gamma(x)=\tau}{\Phi_\emptyset;\Gamma\vdash x : \tau}\right]\!\!\right] = \cdot \qquad \left[\!\!\left[(\textsc{TGvar})\frac{\Gamma(\mathsf{f})=\tau}{\Phi_\emptyset;\Gamma\vdash \mathsf{f} : \tau}\right]\!\!\right] = \cdot$$

$$\left[\!\!\left[(\textsc{TSub})\frac{\begin{array}{c}D :: \Phi';\Gamma\vdash e : \tau' \\ \tau'\le\tau \quad \Phi'\le\Phi\end{array}}{\Phi;\Gamma\vdash e : \tau}\right]\!\!\right] = \mathcal{R},\ \text{where } [\![D]\!] = \mathcal{R}$$

$$\left[\!\!\left[(\textsc{TUpdate})\frac{\Phi^\alpha\subseteq\alpha' \quad \Phi^\omega\subseteq\omega'}{\Phi;\Gamma\vdash \mathsf{update}^{\alpha',\omega'} : int}\right]\!\!\right] = \cdot$$

$$\left[\!\!\left[(\textsc{TRef})\frac{D :: \Phi;\Gamma\vdash e : \tau}{\Phi;\Gamma\vdash \mathsf{ref}\ e : ref^{\varepsilon}\ \tau}\right]\!\!\right] = \mathcal{R},\ \text{where } [\![D]\!] = \mathcal{R}$$

$$\left[\!\!\left[(\textsc{TDeref})\frac{\begin{array}{c}D_1 :: \Phi_1;\Gamma\vdash e : ref^{\varepsilon}\ \tau \\ \Phi_2^{\varepsilon}=\varepsilon \quad \Phi_1\rhd\Phi_2\hookrightarrow\Phi\end{array}}{\Phi;\Gamma\vdash\, !e : \tau}\right]\!\!\right] = \mathcal{R}_1,\ \text{where } [\![D_1]\!] = \mathcal{R}_1$$

$$\left[\!\!\left[(\textsc{TAssign})\frac{\begin{array}{c}D_1 :: \Phi_1;\Gamma\vdash e_1 : ref^{\varepsilon}\ \tau \quad D_2 :: \Phi_2;\Gamma\vdash e_2 : \tau \\ \Phi_3^{\varepsilon}=\varepsilon \qquad \Phi_1\rhd\Phi_2\rhd\Phi_3\hookrightarrow\Phi\end{array}}{\Phi;\Gamma\vdash e_1 := e_2 : \tau}\right]\!\!\right] = \mathcal{R}_1\bowtie\mathcal{R}_2,\ \text{where } \begin{array}{l}[\![D_1]\!]=\mathcal{R}_1\\ [\![D_2]\!]=\mathcal{R}_2\\ e_1\not\equiv v\Rightarrow\mathcal{R}_2=\cdot\end{array}$$

$$\left[\!\!\left[(\textsc{TIf})\frac{\begin{array}{c}D_1 :: \Phi_1;\Gamma\vdash e_1 : int \\ D_2 :: \Phi_2;\Gamma\vdash e_2 : \tau \quad D_3 :: \Phi_2;\Gamma\vdash e_3 : \tau \\ \Phi_1\rhd\Phi_2\hookrightarrow\Phi\end{array}}{\Phi;\Gamma\vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}\right]\!\!\right] = \mathcal{R}_1,\ \text{where } \begin{array}{l}[\![D_1]\!]=\mathcal{R}_1\\ [\![D_2]\!]=\cdot\\ [\![D_3]\!]=\cdot\end{array}$$

$$\left[\!\!\left[(\textsc{TLet})\frac{\begin{array}{c}D_1 :: \Phi_1;\Gamma\vdash e_1 : \tau_1 \quad D_2 :: \Phi_2;\Gamma,x:\tau_1\vdash e_2 : \tau_2 \\ \Phi_1\rhd\Phi_2\hookrightarrow\Phi\end{array}}{\Phi;\Gamma\vdash \mathsf{let}\ x:\tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2}\right]\!\!\right] = \mathcal{R}_1,\ \text{where } \begin{array}{l}[\![D_1]\!]=\mathcal{R}_1\\ [\![D_2]\!]=\cdot\end{array}$$

$$\left[\!\!\left[(\textsc{TApp})\frac{\begin{array}{c}D_1 :: \Phi_1;\Gamma\vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \quad D_2 :: \Phi_2;\Gamma\vdash e_2 : \tau_1 \\ \Phi_1\rhd\Phi_2\rhd\Phi_3\hookrightarrow\Phi \\ \Phi_3^{\varepsilon}=\Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha}\subseteq\Phi_f^{\alpha} \qquad \Phi_3^{\omega}\subseteq\Phi_f^{\omega}\end{array}}{\Phi;\Gamma\vdash e_1\ e_2 : \tau_2}\right]\!\!\right] = \mathcal{R}_1\bowtie\mathcal{R}_2,\ \text{where } \begin{array}{l}[\![D_1]\!]=\mathcal{R}_1\\ [\![D_2]\!]=\mathcal{R}_2\\ e_1\not\equiv v\Rightarrow\mathcal{R}_2=\cdot\end{array}$$

$$\left[\!\!\left[(\textsc{TTransact})\frac{\Phi_1;\Gamma\vdash e : \tau \quad \Phi^\alpha\subseteq\Phi_1^\alpha \quad \Phi^\omega\subseteq\Phi_1^\omega}{\Phi;\Gamma\vdash \mathsf{tx}\ e : \tau}\right]\!\!\right] = \cdot$$

$$\left[\!\!\left[(\textsc{TIntrans})\frac{D_1 :: \Phi_1;\Gamma\vdash e : \tau \quad \Phi^\alpha\subseteq\Phi_1^\alpha \quad \Phi^\omega\subseteq\Phi_1^\omega}{\Phi;\Gamma\vdash \mathsf{intx}\ e : \tau}\right]\!\!\right] = \Phi,\mathcal{R}_1 \text{ where } [\![D_1]\!] = \mathcal{R}_1$$

$$\cdot\bowtie\mathcal{R} = \mathcal{R} \ \wedge\ \mathcal{R}\bowtie\cdot = \mathcal{R}$$

Figure B.1: Transaction effect extraction

Lemma B.0.10 (Subexpression version consistency). *If* $\Phi,\mathcal{R}_1\bowtie\mathcal{R}_2;H\vdash\Sigma$ *and* $\Phi_1\rhd\Phi_2\hookrightarrow\Phi$ *then*

(i) $\mathcal{R}_2\equiv\cdot$ *implies* $\Phi_1,\mathcal{R}_1;H\vdash\Sigma$

(ii) $\mathcal{R}_1\equiv\cdot$ *and* $\Phi_1^{\varepsilon}\equiv\emptyset$ *implies* $\Phi_2,\mathcal{R}_2;H\vdash\Sigma$

Proof. (i) Since $\mathcal{R}_2=\cdot$ by assumption, we have $\mathcal{R}_1=\mathcal{R}_1\bowtie\mathcal{R}_2$. We have two cases:

$\mathcal{R}_1 \equiv \cdot$: Thus we must have

$$\text{TC1}\,\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \cup \varepsilon_2 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{[\alpha_1; \varepsilon_1 \cup \varepsilon_2; \omega_2], \cdot; H \vdash (\beta, \sigma)}$$

where $\Sigma \equiv (\beta, \sigma)$. The result follows from [TC1] :

$$\text{TC1}\,\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{[\alpha_1; \varepsilon_1; \omega_1], \cdot; H \vdash (\beta, \sigma)}$$

$\mathcal{R}_1 \equiv \Phi', \mathcal{R}'$: Thus we must have

$$\text{TC2}\,\frac{\begin{array}{c}\Phi', \mathcal{R}'; H \vdash \Sigma' \\ \Phi \equiv [\alpha_1; \varepsilon_1 \cup \varepsilon_2; \omega_2] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \cup \varepsilon_2 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{\Phi, \Phi', \mathcal{R}'; H \vdash ((\beta, \sigma), \Sigma')}$$

where $\Sigma \equiv ((\beta, \sigma), \Sigma')$. The result follows by [TC2]:

$$\text{TC2}\,\frac{\begin{array}{c}\Phi', \mathcal{R}'; H \vdash \Sigma' \\ \Phi_1 \equiv [\alpha_1; \varepsilon_1; \omega_1] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{\Phi_1, \Phi', \mathcal{R}'; H \vdash ((\beta, \sigma), \Sigma')}$$

where the first premise follows by assumption.

(ii) Since $\mathcal{R}_1 = \cdot$ by assumption, we have $\mathcal{R}_2 = \mathcal{R}_1 \bowtie \mathcal{R}_2$. We have two cases:

$\mathcal{R}_2 \equiv \cdot$: Thus we must have

$$\text{TC1}\,\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \cup \varepsilon_2 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{[\alpha_1; \varepsilon_1 \cup \varepsilon_2; \omega_2], \cdot; H \vdash (\beta, \sigma)}$$

where $\Sigma \equiv (\beta, \sigma)$. Since $\Phi_1^\varepsilon \equiv \emptyset$ and $\Phi_2^\alpha = \Phi_1^\alpha \cup \Phi_1^\varepsilon$ we have $\Phi_2^\alpha = \Phi_1^\alpha$ and the result follows from [TC1]:

$$\text{TC1}\,\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_2 \\ \mathsf{f} \in \varepsilon_2 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{[\alpha_2; \varepsilon_2; \omega_2], \cdot; H \vdash (\beta, \sigma)}$$

$\mathcal{R}_2 \equiv \Phi', \mathcal{R}'$: Thus we must have

$$\text{TC2}\,\frac{\begin{array}{c}\Phi', \mathcal{R}'; H \vdash \Sigma' \\ \Phi \equiv [\alpha_1; \varepsilon_1 \cup \varepsilon_2; \omega_2] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \cup \varepsilon_2 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{\Phi, \Phi', \mathcal{R}'; H \vdash ((\beta, \sigma), \Sigma')}$$

where $\Sigma \equiv ((\beta, \sigma), \Sigma')$. The result follows by [TC2] and $\Phi_2^\alpha = \Phi_1^\alpha$ (because $\Phi_1^\varepsilon \equiv \emptyset$ and $\Phi_2^\alpha = \Phi_1^\alpha \cup \Phi_1^\varepsilon$):

$$\text{TC2}\,\frac{\begin{array}{c}\Phi', \mathcal{R}'; H \vdash \Sigma' \\ \Phi_2 \equiv [\alpha_2; \varepsilon_2; \omega_2] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_2 \\ \mathsf{f} \in \varepsilon_2 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{\Phi_2, \Phi', \mathcal{R}'; H \vdash ((\beta, \sigma), \Sigma')}$$

where the first premise follows by assumption.

□

Lemma B.0.11 (Stack Shapes). *If $\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon_0} \langle n; \Sigma'; H'; e'\rangle$ then $top(\Sigma) = (\beta, \sigma)$ and $top(\Sigma') = (\beta', \sigma')$ where $n' = n''$ and $\sigma \subseteq \sigma'$.*

Proof. By induction on $\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon_0} \langle n; \Sigma'; H'; e'\rangle$. □

Lemma B.0.12 (Update preserves heap safety)**.** *If* $n; \Gamma \vdash H$ *and* $\mathit{updateOK}(\mathit{upd}, H, (\alpha, \omega), \mathit{dir})$ *then* $n{+}1; \mathcal{U}[\Gamma]^{\mathit{upd}} \vdash \mathcal{U}[H]^{\mathit{upd}}_{n+1}$.

Proof. Let $n' = n + 1$ and $\Gamma' \equiv \mathcal{U}[\Gamma]^{\mathit{upd}}$ and $H' \equiv \mathcal{U}[H]^{\mathit{upd}}_{n'}$. From the definition of heap typing (Figure 5.8), to prove $n'; \Gamma' \vdash H'$, we need to show:

1. $\mathrm{dom}(\Gamma') = \mathrm{dom}(H')$
2. $\forall \mathsf{z} \mapsto (\tau, v, \nu) \in H'.\quad \Phi_\emptyset; \Gamma' \vdash v : \tau \ \wedge\ \Gamma'(\mathsf{z}) = \mathit{ref}^{\varepsilon}\ \tau \ \wedge\ \mathsf{z} \in \varepsilon$
3. $\forall \mathsf{z} \mapsto (\tau \longrightarrow^{\Phi} \tau', \lambda(x).e, \nu) \in H'.\quad \Phi; \Gamma', x : \tau \vdash e : \tau' \ \wedge\ \Gamma'(\mathsf{z}) = \tau \longrightarrow^{\Phi} \tau' \ \wedge\ \mathsf{z} \in \Phi^{\alpha} \ \wedge\ \mathsf{z} \in \Phi^{\varepsilon}$
4. $\forall r \mapsto (\cdot, v, \nu) \in H'.\quad \Phi_\emptyset; \Gamma' \vdash v : \tau \ \wedge\ \Gamma'(r) = \mathit{ref}^{\varepsilon}\ \tau$
5. $\forall \mathsf{z} \mapsto (\tau, b, \nu) \in H'.\quad n' \in \nu$

Proof by induction on H.

case $H \equiv \emptyset$ **:**

We have $\mathcal{U}[\emptyset]^{\mathit{upd}}_{n'} = \mathit{upd}^{\mathit{add}}$ (modified to have version set $\{n+1\}$), and thus $\mathrm{dom}(H') = \mathrm{dom}(\mathit{upd}^{\mathit{add}})$. Our assumption $\mathrm{dom}(H) = \mathrm{dom}(\Gamma)$ implies that $\Gamma = \emptyset$, and thus $\Gamma' = \mathcal{U}[\emptyset]^{\mathit{upd}} = \mathit{types}(\mathit{upd}^{\mathit{add}})$.

1. $\mathrm{dom}(\Gamma') = \mathrm{dom}(\mathit{types}(\mathit{upd}^{\mathit{add}})) = \mathrm{dom}(\mathit{upd}^{\mathit{add}}) = \mathrm{dom}(H')$.
2. Since $H' = \mathit{upd}^{\mathit{add}}$, this follows directly from $\mathit{updateOK}(\mathit{upd}, H, (\alpha, \omega), \mathit{dir})$ given the definition of $\Gamma' = \mathcal{U}[\emptyset]^{\mathit{upd}} = \mathit{types}(\mathit{upd}^{\mathit{add}})$.
3. Similar to 2.
4. Vacuously true, since $r \notin \mathrm{dom}(H') = \mathrm{dom}(\mathit{upd}^{\mathit{add}})$ for all r.
5. Holds by the definition of $\mathcal{U}[\emptyset]^{\mathit{upd}}_{n'}$.

case $H \equiv (r \mapsto (\cdot, b, \emptyset), H'')$ **:**

We have $H' \equiv \mathcal{U}[(r \mapsto (\cdot, b, \emptyset), H'')]^{\mathit{upd}}_{n'} = (r \mapsto (\cdot, b, \emptyset)), \mathcal{U}[H'']^{\mathit{upd}}_{n'}$. Our assumption $\mathrm{dom}(H) = \mathrm{dom}(\Gamma)$ implies $\Gamma \equiv (r : \tau, \Gamma'')$ for some Γ'', where $\mathrm{dom}(H'') = \mathrm{dom}(\Gamma'')$ and $\Gamma' \equiv \mathcal{U}[r : \tau, \Gamma'']^{\mathit{upd}} = r : \tau, \mathcal{U}[\Gamma'']^{\mathit{upd}}$.

1. By induction we know $\mathrm{dom}(\mathcal{U}[\Gamma'']^{\mathit{upd}}) = \mathrm{dom}(\mathcal{U}[H'']^{\mathit{upd}}_{n'})$. But $\mathrm{dom}(H') = \mathrm{dom}(r = (\cdot, b, \emptyset)), \mathcal{U}[H'']^{\mathit{upd}}_{n'}) = \{r\} \cup \mathrm{dom}(\mathcal{U}[H'']^{\mathit{upd}}_{n'})$, and $\mathrm{dom}(\Gamma') = \mathrm{dom}(r : \tau, \mathcal{U}[\Gamma'']^{\mathit{upd}}) = \{r\} \cup \mathrm{dom}(\mathcal{U}[\Gamma'']^{\mathit{upd}})$.
2. Follows by induction, since $r \neq \mathsf{z}$ for all z.
3. Same as above.
4. For r, this follows by assumption, since it is clear that $H(r) = \mathcal{U}[H]^{\mathit{upd}}_{n'}(r)$ and $\Gamma(r) = \mathcal{U}[\Gamma]^{\mathit{upd}}(r)$, and for the rest of the heap the property follows by induction.
5. Follows by induction, since $r \neq \mathsf{z}$ for all z.

case $H \equiv (\mathsf{z} = (\tau, b, \nu), H'')$ **:**

We have $H' \equiv \mathcal{U}[(\mathsf{z} \mapsto (\tau, b, \nu), H'')]^{\mathit{upd}}_{n'} = (\mathsf{z} \mapsto (\tau, b', \nu')), \mathcal{U}[H'']^{\mathit{upd}}_{n'}$. Our assumption $\mathrm{dom}(H) = \mathrm{dom}(\Gamma)$ implies $\Gamma \equiv (\mathsf{z} : \mathit{heapType}(\mathsf{z}, \tau), \Gamma'')$ for some Γ'', where $\mathrm{dom}(H'') = \mathrm{dom}(\Gamma'')$ and $\Gamma' \equiv \mathcal{U}[\mathsf{z} : \tau, \Gamma'']^{\mathit{upd}} = \mathsf{z} : \mathit{heapType}(\mathsf{z}, \tau), \mathcal{U}[\Gamma'']^{\mathit{upd}}$.

1. Similar to the argument for the $H \equiv (r \mapsto (...), H'')$ case.
4. This follows by induction, since $\mathsf{z} \neq r$.

Now consider the remaining cases according to z with respect to $\mathit{upd}^{\mathit{chg}}$:

case $\mathsf{z} \notin \mathrm{dom}(\mathit{upd}^{\mathit{chg}})$ **:**

2. For z, this follows by assumption, since it is clear that $H(\mathsf{z}) = \mathcal{U}[H]^{\mathit{upd}}_{n'}(\mathsf{z})$ and $\Gamma(\mathsf{z}) = \mathcal{U}[\Gamma]^{\mathit{upd}}(\mathsf{z})$. The rest of the heap follows by induction.
3. Same as above.

5. We have $\mathcal{U}[(\mathsf{z} \mapsto (\tau, b, \nu), H'')]^{upd}_{n'} = (\mathsf{z} \mapsto (\tau, b, \nu \cup \{n'\}), \mathcal{U}[H'']^{upd}_{n'})$ where $n' \in (\nu \cup \{n'\})$ for z, and the rest follows by induction.

case $\mathsf{z} \in \mathrm{dom}(upd^{chg})$ **:**

2. From the definition of $updateOK(upd, H, (\alpha, \omega), dir)$ we know that (i) $\Phi_\emptyset; \mathcal{U}[\Gamma]^{upd} \vdash v' : \tau$. Considering z, from the definition of $heapType(\tau, \mathsf{z})$ we have (ii) $heapType(\tau, \mathsf{z}) = ref^{\varepsilon}\ \tau$ where $\mathsf{z} \in \varepsilon$. Combining (i) and (ii) yields

$$\Phi_\emptyset; \Gamma' \vdash v : \tau \ \wedge\ \Gamma'(\mathsf{z}) = ref^{\varepsilon}\ \tau\ \wedge \mathsf{z} \in \varepsilon$$

The property holds for the rest of the heap by induction.

3. Similar to the previous.

5. We have $\mathcal{U}[(\mathsf{z} \mapsto (\tau, b, \nu), H'')]^{upd}_{n'} = (\mathsf{z} \mapsto (\tau, b', \{n'\}), \mathcal{U}[H'']^{upd}_{n'})$ and obviously $n' \in \{n'\}$ for z, and the rest by induction.

□

The following lemma states that if we start with a well-typed program and a version-consistent trace and we take an update step, then afterward we will still have a well-typed program whose trace is version-consistent.

Lemma B.0.13 (Update preservation)**.**
Suppose we have the following:

1. $n \vdash H, e : \tau$ *(such that* $\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}$ *and* $n; \Gamma \vdash H$ *for some* Γ, Φ*)*

2. $\Phi, \mathcal{R}; H \vdash \Sigma$

3. $traceOK(\Sigma)$

4. $\langle n; \Sigma; H; e\rangle \longrightarrow_\mu \langle n+1; \Sigma'; H';\ e\rangle$

where $H' \equiv \mathcal{U}[H]^{upd}_{n+1}$*,* $\Gamma' \equiv \mathcal{U}[\Gamma]^{upd}$*,* $\mu = (upd, dir)$*,* $\Sigma' \equiv \mathcal{U}[\Sigma]^{upd,dir}_{n}$*, and* $top(\Sigma') = (\beta', \sigma')$*. Then for some* Φ' *such that* $\Phi'^{\alpha} = \Phi^{\alpha}$*,* $\Phi'^{\omega} = \Phi^{\omega}$*, and* $\Phi'^{\varepsilon} \subseteq \Phi^{\varepsilon}$ *and some* $\Gamma' \supseteq \Gamma$ *we have that:*

1. $n+1 \vdash H', e : \tau$ *where* $\Phi'; \Gamma' \vdash e : \tau \rightsquigarrow \mathcal{R}$ *and* $n+1; \Gamma' \vdash H'$

2. $\Phi', \mathcal{R}; H' \vdash \Sigma'$

3. $traceOK(\Sigma')$

4. $(dir = bck) \Rightarrow n'' \equiv n+1\ \wedge\ (dir = fwd) \Rightarrow (\mathsf{f} \in \omega \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f}))$

Proof. Since $\mathcal{U}[\Gamma]^{upd} \supseteq \Gamma$, $\Phi; \mathcal{U}[\Gamma]^{upd} \vdash e : \tau \rightsquigarrow \mathcal{R}$ follows by weakening (Lemma B.0.1). Proceed by simultaneous induction on the typing derivation of e ($n \vdash H, e : \tau$) and on the evaluation derivation $\langle n; \Sigma; H; e\rangle \longrightarrow_\mu \langle n+1; \Sigma'; H';\ e\rangle$. Consider the last rule used in the evaluation derivation:

case [gvar-deref], [gvar-assign], [call], [let], [tx-start], [tx-end], [ref], [deref], [assign], [if-t], [if-f], [no-update] **:**

Not possible, as these transitions cannot cause an update to occur.

case [update] **:**

This implies that $e \equiv \mathsf{update}^{\alpha,\omega}$ and thus

$$\langle n; (\beta, \sigma); H; \mathsf{update}^{\alpha,\omega}\rangle \longrightarrow_\mu \langle n+1; \mathcal{U}[(\beta, \sigma)]^{upd,dir}_{n+1}; \mathcal{U}[H]^{upd}_{n+1}; 1\rangle$$

where $\mu \equiv (upd, dir)$ and $updateOK(upd, H, (\alpha, \omega), dir)$. By subtyping derivations (Lemma B.0.6) we have

$$\text{TSub}\dfrac{int \leq int \qquad \text{TUpdate}\dfrac{\alpha \subseteq \alpha'' \qquad \omega \subseteq \omega'' \qquad \Phi_u \equiv [\alpha; \emptyset; \omega]}{\Phi_u; \Gamma \vdash \mathsf{update}^{\alpha'',\omega''} : int \rightsquigarrow \cdot} \qquad \Phi_u \leq \Phi \qquad \Phi \equiv [\alpha; \varepsilon; \omega]}{\Phi; \Gamma \vdash \mathsf{update}^{\alpha,\omega} : int \rightsquigarrow \cdot}$$

and by flow effect weakening (Lemma B.0.7) we know that α and ω are unchanged in the use of (TSub). Let $\Phi' = \Phi_u$ (hence $\Phi'^{\alpha} = \Phi^{\alpha}$, $\Phi'^{\omega} = \Phi^{\omega}$, and $\emptyset \subseteq \Phi^{\varepsilon}$ as required) and $(\beta', \sigma') \equiv \mathcal{U}[(\beta, \sigma)]^{upd,dir}_{n+1}$.

To prove 1., we get $n+1; \Gamma' \vdash H'$ by Lemma B.0.12 and $\Phi_u; \Gamma' \vdash 1 : int \leadsto \cdot$ by [TInt].

To prove 2., we must show $\Phi_u, \cdot; H' \vdash (\beta', \sigma')$. By assumption, we have

$$\text{TC1}\,\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{[\alpha; \varepsilon; \omega], \cdot; H \vdash (\beta, \sigma)}$$

We need to prove

$$\text{TC1}\,\frac{\begin{array}{c}\mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \emptyset \Rightarrow n'' \in ver(H', \mathsf{f})\end{array}}{[\alpha; \emptyset; \omega], \cdot; H' \vdash (\beta', \sigma')}$$

We have the first premise by assumption (since $\text{dom}(\sigma) = \text{dom}(\sigma')$ from the definition of $\mathcal{U}[(\beta, \sigma)]^{upd,dir}_{n+1}$). The second premise holds vacuously.

To prove 3., we must show $traceOK(\beta', \sigma')$. Consider each possible update type:

case $dir = bck$ **:**

From the definition of $\mathcal{U}[(\beta, \sigma)]^{upd,bck}_{n+1}$, we know that $n'' = n + 1$. Consider $(\mathsf{f}, \nu) \in \sigma$; it must be the case that $\mathsf{f} \notin \text{dom}(upd^{chg})$. This is because $dir = bck$ implies $\alpha \cap \text{dom}(upd^{chg}) = \emptyset$ and by assumption (from the first premise of [TC1] above) $\mathsf{f} \in \alpha$. Therefore, since $\mathsf{f} \notin \text{dom}(upd^{chg})$, its σ' entry is $(\mathsf{f}, \nu \cup \{n''\})$, which is the required result.

case $dir = fwd$ **:**

Since $\mathcal{U}[(\beta, \sigma)]^{upd,fwd}_{n+1} = (\beta, \sigma)$, the result is true by assumption.

To prove 4., we must show $n'' \equiv n + 1 \vee (\mathsf{f} \in \omega \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f}))$. Consider each possible update type:

case $dir = bck$ **:**

From the definition of $\mathcal{U}[(\beta, \sigma)]^{upd,bck}_{n+1}$, we know that $n'' = n + 1$ so we are done.

case $dir = fwd$ **:**

We have $\mathcal{U}[(\beta, \sigma)]^{upd,fwd}_{n+1} = (\beta, \sigma)$, and from $updateOK(upd, H, (\alpha, \omega), dir)$ we know that $\mathsf{f} \in \omega \Rightarrow \mathsf{f} \notin \text{dom}(upd^{chg})$. From the definition of $\mathcal{U}[H]^{upd}_{n}$ we know that $\mathcal{U}[(\mathsf{f} \mapsto (\tau, b, \nu), H)]^{upd}_{n+1} = \mathsf{f} \mapsto (\tau, b, \nu \cup \{n+1\})$ if $\mathsf{f} \notin \text{dom}(upd^{chg})$. This implies that for $\mathsf{f} \in \omega$, $ver(H, \mathsf{f}) = \nu$ and $ver(H', \mathsf{f}) = \nu \cup \{n+1\}$, and therefore $ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f})$.

case [TX-CONG-1] **:**

We have that $\langle n; ((\beta, \sigma), \Sigma); H; \mathsf{intx}\ e\rangle \longrightarrow_{\mu} \langle n{+}1; (\mathcal{U}[(\beta, \sigma)]^{upd,dir}_{n+1}, \Sigma'); H'; \mathsf{intx}\ e'\rangle$ follows from $\langle n; \Sigma; H; e\rangle \longrightarrow_{\mu}$ $\langle n+1; \Sigma'; H'; e'\rangle$ by [TX-CONG-1], where $\mu \equiv (upd, dir)$. Let $(\beta', \sigma') \equiv \mathcal{U}[(\beta, \sigma)]^{upd,dir}_{n+1}$. By assumption and subtyping derivations (Lemma B.0.6) we have

$$\text{TSub}\,\frac{\text{TIntrans}\,\dfrac{\begin{array}{c}\Phi_e; \Gamma \vdash e : \tau' \leadsto \mathcal{R} \\ \alpha \subseteq \Phi_e^{\alpha} \qquad \omega \subseteq \Phi_e^{\omega}\end{array}}{[\alpha; \emptyset; \omega]; \Gamma \vdash \mathsf{intx}\ e : \tau' \leadsto \Phi_e, \mathcal{R}} \qquad \tau' \leq \tau \qquad [\alpha; \emptyset; \omega] \leq [\alpha; \varepsilon; \omega]}{[\alpha; \varepsilon; \omega]; \Gamma \vdash \mathsf{intx}\ e : \tau \leadsto \Phi_e, \mathcal{R}}$$

and by flow effect weakening (Lemma B.0.7) we know that α and ω are unchanged in the use of (TSub). We have $\Phi_e \equiv [\alpha_e; \varepsilon_e; \omega_e]$, so that $\omega_e \supseteq \omega$ and $\alpha_e \supseteq \alpha$. To apply induction, we must show that $\Phi_e, \mathcal{R}; H \vdash \Sigma$ (which follows by inversion on $\Phi, \Phi_e, \mathcal{R}; H \vdash ((\beta, \sigma), \Sigma)$); $\Phi_e; \Gamma \vdash e : \tau' \leadsto \mathcal{R}$ (which follows by assumption); and $n; \Gamma \vdash H$ (by assumption).

By induction we have:

(i) $\Phi'_e; \Gamma' \vdash e' : \tau' \leadsto \mathcal{R}$ and

(ii) $n + 1; \Gamma' \vdash H'$

(iii) $\Phi'_e, \mathcal{R}; H' \vdash \Sigma'$

(iv) $traceOK(\Sigma')$

(v) $(dir = bck) \Rightarrow n'' \equiv n + 1 \ \wedge\ (dir = fwd) \Rightarrow (\mathsf{f} \in \omega_e \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f}))$

where $\Phi'_e \equiv [\alpha_e; \varepsilon'_e; \omega_e]$, $\varepsilon'_e \subseteq \varepsilon_e$.

Let $\Phi' = [\alpha; \emptyset; \omega]$ (hence $\Phi'^{\alpha} = \Phi^{\alpha}$, $\Phi'^{\omega} = \Phi^{\omega}$, and $\emptyset \subseteq \Phi^{\varepsilon}$ as required). To prove 1., we can show

$$\text{TSub}\frac{\text{TIntrans}\dfrac{\begin{array}{c}\Phi'_e; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R} \\ \alpha \subseteq \Phi'^{\alpha}_e \qquad \omega \subseteq \Phi'^{\omega}_e\end{array}}{\Phi'; \Gamma \vdash \mathsf{intx}\ e' : \tau \rightsquigarrow \Phi'_e, \mathcal{R}} \qquad \tau' \leq \tau \qquad \Phi' \leq \Phi'}{\Phi'; \Gamma \vdash \mathsf{intx}\ e' : \tau \rightsquigarrow \Phi'_e, \mathcal{R}}$$

The first premise of [TIntrans] follows by (i), and the second since $\alpha_e \supseteq \alpha$ and $\omega_e \supseteq \omega$.

To prove 2., we need to show that

$$\text{TC2}\frac{\begin{array}{c}\Phi'_e, \mathcal{R}; H' \vdash \Sigma' \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \emptyset \Rightarrow n'' \in ver(H', \mathsf{f})\end{array}}{[\alpha; \emptyset; \omega], \Phi'_e, \mathcal{R}; H' \vdash ((\beta', \sigma'), \Sigma')}$$

We have the first premise by (iii), the second by assumption (since $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$ from the definition of $\mathcal{U}[(\beta, \sigma)]^{upd,dir}_{n+1}$), and the last holds vacuously.

To prove 3., we must show $traceOK((\beta', \sigma'), \Sigma')$, which reduces to proving $traceOK(\beta', \sigma')$ since we have $traceOK(\Sigma')$ from (iv). We have $traceOK(\beta, \sigma)$ by assumption. Consider each possible update type:

case $dir = bck$ **:**

From the definition of $\mathcal{U}[(\beta, \sigma)]^{upd,bck}_{n+1}$, we know that $n'' = n + 1$. Consider $(\mathsf{f}, \nu) \in \sigma$; it must be the case that $\mathsf{f} \notin \mathrm{dom}(upd^{chg})$. This is because $dir = bck$ implies $\alpha_e \cap \mathrm{dom}(upd^{chg}) = \emptyset$ and by assumption we have $\alpha \subseteq \alpha_e$ (from (TIntrans)) and $\mathsf{f} \in \alpha$ (from the first premise of [TC1] above). Therefore, since $\mathsf{f} \notin \mathrm{dom}(upd^{chg})$, its σ' entry is $(\mathsf{f}, \nu \cup \{n''\})$, which is the required result.

case $dir = fwd$ **:**

Since $\mathcal{U}[(\beta, \sigma)]^{upd,fwd}_{n+1} = (\beta, \sigma)$, the result is true by assumption.

Part 4. follows directly from (v) and the fact that $\omega_e \supseteq \omega$.

case [cong] **:**

We have that $\langle n; \Sigma; H; \mathbb{E}[e]\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H'; \mathbb{E}[e]\rangle$ follows from $\langle n; \Sigma; H; e\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H'; e\rangle$ by [cong], where $\mu \equiv (upd, dir)$. Consider the shape of $\mathbb{E}$:

case _ **:**

The result follows directly by induction.

case $\mathbb{E}\ e_2$ **:**

By assumption, we have $\Phi; \Gamma \vdash (\mathbb{E}\ e_2)[e_1] : \tau \rightsquigarrow \mathcal{R}$. By subtyping derivations (Lemma B.0.6) we know we can construct a proof derivation of this ending in (TSub):

$$\text{TSub}\frac{\tau'_2 \leq \tau_2 \qquad \text{TApp}\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash \mathbb{E}[e_1] : \tau_1 \longrightarrow^{\Phi_f} \tau'_2 \rightsquigarrow \mathcal{R}_1 \qquad \Phi_2; \Gamma \vdash e_2 : \tau_1 \rightsquigarrow \cdot \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi_s \\ \Phi^{\varepsilon}_3 = \Phi^{\varepsilon}_f \qquad \Phi^{\alpha}_3 \subseteq \Phi^{\alpha}_f \qquad \Phi^{\omega}_3 \subseteq \Phi^{\omega}_f \\ \mathbb{E}\ [e_1] \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot\end{array}}{\Phi_s; \Gamma \vdash (\mathbb{E}\ e_2)[e_1] : \tau'_2 \rightsquigarrow \mathcal{R}_1} \qquad \text{SCtxt}\dfrac{\begin{array}{c}\Phi \equiv [\alpha; \varepsilon; \omega] \\ \Phi_s \equiv [\alpha; \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f; \omega] \\ (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f) \subseteq \varepsilon\end{array}}{\Phi_s \leq \Phi}}{\Phi; \Gamma \vdash (\mathbb{E}\ e_2)[e_1] : \tau_2 \rightsquigarrow \mathcal{R}_1}$$

and by flow effect weakening (Lemma B.0.7) we know that α and ω are unchanged in the use of (TSub).

By inversion on $\langle n; \Sigma; H; (\mathbb{E}\ e_2)[e_1]\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H'; (\mathbb{E}\ e_2)[e_1]\rangle$ we have $\langle n; \Sigma; H; e_1\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H'; e'_1\rangle$, and then applying [cong] we have $\langle n; \Sigma; H; \mathbb{E}[e_1]\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H'; \mathbb{E}[e'_1]\rangle$. From $\Phi, \mathcal{R}_1; H \vdash \Sigma$ we know that:

$$\begin{array}{l}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \Rightarrow n' \in ver(H, \mathsf{f})\end{array}$$

where (β, σ) is the top of Σ. Since $\Phi \equiv [\alpha; \varepsilon; \omega]$ and $\Phi_s \equiv [\alpha; \varepsilon_s; \omega]$ and $\varepsilon_s = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$ (where $\varepsilon_3 = \varepsilon_f$), we have

$$\begin{array}{l} \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon_1 \Rightarrow n' \in \mathit{ver}(H, \mathsf{f}) \end{array}$$

but since $\Phi_1 \equiv [\alpha; \varepsilon_1; \omega_1]$, we have $\Phi_1, \mathcal{R}_1; H \vdash \Sigma$. Hence we can apply induction on $\Phi_1; \Gamma \vdash \mathbb{E}[e_1] : \tau_1 \longrightarrow^{\Phi_f} \tau_2' \rightsquigarrow \mathcal{R}_1$, yielding:

(i) $\Phi_1'; \Gamma' \vdash \mathbb{E}[e_1'] : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \mathcal{R}_1$ and

(ii) $n+1; \Gamma' \vdash H'$

(iii) $\Phi_1', \mathcal{R}_1; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

(v) $(\mathit{dir} = \mathit{bck}) \Rightarrow n'' \equiv n+1 \ \wedge \ (\mathit{dir} = \mathit{fwd}) \Rightarrow (\mathsf{f} \in \omega_1 \Rightarrow \mathit{ver}(H, \mathsf{f}) \subseteq \mathit{ver}(H', \mathsf{f}))$

where $\Phi_1' \equiv [\alpha_s; \varepsilon_1'; \omega_1]$ and $\varepsilon_1' \subseteq \varepsilon_1$. Choose $\Phi_2' = [\alpha_1 \cup \varepsilon_1'; \varepsilon_2; \omega_2]$ and $\Phi_3' = [\alpha_1 \cup \varepsilon_1' \cup \varepsilon_2; \varepsilon_f; \omega_s]$ and thus $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi_s'$ and $\Phi_3'^{\varepsilon} = \Phi_f^{\varepsilon}$. Let $\Phi' = [\alpha; \varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_f; \omega]$, where $\varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_f \subseteq \varepsilon$, as required.

To prove 1., we have $n+1; \Gamma' \vdash H'$ by (ii), and apply (TApp):

$$\text{TApp}\frac{\begin{array}{c} \Phi_1'; \Gamma' \vdash \mathbb{E}[e_1'] : \tau_1 \longrightarrow^{\Phi_f} \tau_2' \rightsquigarrow \mathcal{R}_1 \qquad \Phi_2'; \Gamma' \vdash e_2 : \tau_1 \rightsquigarrow \cdot \\ \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi_s' \\ \Phi_3'^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3'^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3'^{\omega} \subseteq \Phi_f^{\omega} \\ \mathbb{E}[e_1'] \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot \end{array}}{\Phi_s'; \Gamma' \vdash (\mathbb{E}\ e_2)[e_1'] : \tau_2' \rightsquigarrow \mathcal{R}_1}$$

The first premise follows by (i), the second because we have $\Phi_2; \Gamma' \vdash e_2 : \tau_1$ by weakening (since $\Gamma' \supseteq \Gamma$) and then $\Phi_2'; \Gamma' \vdash e_2 : \tau_1$ by flow effect weakening (Lemma B.0.7) (which we can apply because $\Phi_2'^{\omega} = \Phi_2^{\omega}$, $\Phi_2'^{\varepsilon} = \Phi_2^{\varepsilon}$, $\Phi_2'^{\alpha} = \alpha_1 \cup \varepsilon_1'$ $\Phi_2^{\alpha} = \alpha_1 \cup \varepsilon_1$ hence $\Phi_2'^{\alpha} \subseteq \Phi_2^{\alpha}$) the third—sixth by choice of Φ_2', Φ_3' and Φ_s', and the last as $\mathcal{R}_2 \equiv \cdot$ by assumption. We can now apply (TSub):

$$\text{TSub}\frac{\begin{array}{c} \Phi'; \Gamma \vdash (\mathbb{E}\ e_2)[e_1'] : \tau_2' \rightsquigarrow \mathcal{R}_1 \\ \tau_2' \leq \tau_2 \qquad \Phi' \leq \Phi' \end{array}}{\Phi'; \Gamma \vdash (\mathbb{E}\ e_2)[e_1'] : \tau_2 \rightsquigarrow \mathcal{R}_1'}$$

To prove part 2., we must show that $\Phi', \mathcal{R}_1; H' \vdash \Sigma'$.

By inversion on $\Phi, \mathcal{R}_1; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$ or $\Sigma \equiv (\beta, \sigma), \Sigma''$. We have two cases:

$\Sigma \equiv (\beta, \sigma)$: By (iii) we must have $\mathcal{R}_1 \equiv \cdot$ such that

$$\text{TC1}\frac{\begin{array}{c} \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon_1' \Rightarrow n'' \in \mathit{ver}(H', \mathsf{f}) \end{array}}{[\alpha; \varepsilon_1'; \omega_1], \cdot; H' \vdash (\beta', \sigma')}$$

To achieve the desired result we need to prove:

$$\text{TC1}\frac{\begin{array}{c} \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_f \Rightarrow n'' \in \mathit{ver}(H', \mathsf{f}) \end{array}}{[\alpha; \varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_f; \omega], \cdot; H' \vdash (\beta', \sigma')}$$

The first premise is by assumption (since $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$ from the definition of $\mathcal{U}[(\beta, \sigma)]_{n+1}^{upd,dir}$). For the second premise, we need to show that for all $\mathsf{f} \in (\varepsilon_2 \cup \varepsilon_f) \Rightarrow n'' \in \mathit{ver}(H', \mathsf{f})$ (for those $\mathsf{f} \in \varepsilon_1'$ the result is by assumption).

Consider each possible update type:

case $\mathit{dir} = \mathit{bck}$ **:**

From the definition of $\mathcal{U}[(\beta, \sigma)]_{n+1}^{upd,bck}$, we know that $n'' = n+1$; from the definition of $\mathcal{U}[H]_n^{upd}$ we know that $n+1 \in \mathit{ver}(H', \mathsf{f})$ for all f, hence $n'' \in \mathit{ver}(H', \mathsf{f})$ for all f.

case $\mathit{dir} = \mathit{fwd}$ **:**

From (v) we have that $\mathsf{f} \in \omega_1 \Rightarrow \mathit{ver}(H, \mathsf{f}) \subseteq \mathit{ver}(H', \mathsf{f})$. Since $(\varepsilon_2 \cup \varepsilon_f) \subseteq \omega_1$ (by $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'$), we have $\mathsf{f} \in (\varepsilon_2 \cup \varepsilon_f) \Rightarrow \mathit{ver}(H, \mathsf{f}) \subseteq \mathit{ver}(H', \mathsf{f})$. By inversion on $\Phi, \mathcal{R}_1; H \vdash \Sigma$ we have $\mathsf{f} \in (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f) \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$, and thus $\mathsf{f} \in (\varepsilon_2 \cup \varepsilon_f) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f})$. We have $\mathcal{U}[(\beta, \sigma)]_{n+1}^{upd,fwd} = (\beta, \sigma)$ hence $n'' = n'$, so finally we have $\mathsf{f} \in (\varepsilon_2 \cup \varepsilon_f) \Rightarrow n'' \in \mathit{ver}(H', \mathsf{f})$.

$\Sigma \equiv (\beta, \sigma), \Sigma''$ By (iii), we must have $\mathcal{R}_1 \equiv \Phi'', \mathcal{R}''$ such that

$$\text{TC2}\,\frac{\begin{array}{c}\Phi'', \mathcal{R}''; H' \vdash \Sigma'' \\ \Phi'_1 \equiv [\alpha; \varepsilon'_1; \omega_1] \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon'_1 \Rightarrow n'' \in ver(H', \mathsf{f})\end{array}}{\Phi'_1, \Phi'', \mathcal{R}''; H' \vdash ((\beta', \sigma'), \Sigma'')}$$

We wish to show that

$$\text{TC2}\,\frac{\begin{array}{c}\Phi'', \mathcal{R}''; H' \vdash \Sigma'' \\ \Phi' \equiv [\alpha; \varepsilon'_1 \cup \varepsilon_2 \cup \varepsilon_f; \omega] \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\varepsilon'_1 \cup \varepsilon_2 \cup \varepsilon_f) \Rightarrow n'' \in ver(H', \mathsf{f})\end{array}}{\Phi', \Phi'', \mathcal{R}''; H' \vdash ((\beta', \sigma'), \Sigma'')}$$

$\Phi'', \mathcal{R}''; H' \vdash \Sigma$ follows by assumption while the third and fourth premises follow by the same argument as in the $\Sigma \equiv (\beta, \sigma)$ case, above.

Part 3. follows directly from (iv).

Part 4. follows directly from (v) and the fact that $\omega_1 \supseteq \omega$ (because $\omega_1 \equiv \varepsilon_2 \cup \varepsilon_f \cup \omega$).

case $v\ \mathbb{E}$:

By assumption, we have $\Phi; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau \rightsquigarrow \mathcal{R}$. By subtyping derivations (Lemma B.0.6) we have:

$$\text{TSub}\,\frac{\tau'_2 \leq \tau_2 \qquad \text{TApp}\,\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash v : \tau_1 \longrightarrow^{\Phi_f} \tau'_2 \rightsquigarrow \cdot \qquad \Phi_2; \Gamma \vdash \mathbb{E}[e_2] : \tau_1 \rightsquigarrow \mathcal{R}_2 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi_s \\ \Phi^{\varepsilon}_3 = \Phi^{\varepsilon}_f \qquad \Phi^{\alpha}_3 \subseteq \Phi^{\alpha}_f \qquad \Phi^{\omega}_3 \subseteq \Phi^{\omega}_f \\ v \not\equiv v' \Rightarrow \mathcal{R}_2 = \cdot\end{array}}{\Phi_s; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau'_2 \rightsquigarrow \mathcal{R}_2} \qquad \text{SCtxt}\,\dfrac{\begin{array}{c}\Phi \equiv [\alpha; \varepsilon; \omega] \\ \Phi_s \equiv [\alpha; \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f; \omega] \\ (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f) \subseteq \varepsilon\end{array}}{\Phi_s \leq \Phi}}{\Phi; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau_2 \rightsquigarrow \mathcal{R}_2}$$

and by flow effect weakening (Lemma B.0.7) we know that α and ω are unchanged in the use of (TSub).

By inversion on $\langle n; \Sigma; H; (v\ \mathbb{E})[e_2]\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H'; (v\ \mathbb{E})[e_2]\rangle$ we have $\langle n; \Sigma; H; e_2\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H'; e'_2\rangle$, and then applying [cong] we have $\langle n; \Sigma; H; \mathbb{E}[e_2]\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H'; \mathbb{E}[e_2]\rangle$. From $\Phi, \mathcal{R}_2; H \vdash \Sigma$ we know that:

$$\begin{array}{l}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \Rightarrow n' \in ver(H, \mathsf{f})\end{array}$$

where (β, σ) is the top of Σ. We have $\Phi \equiv [\alpha; \varepsilon; \omega]$, $\Phi_s \equiv [\alpha_s; \varepsilon_s; \omega_s]$, $\varepsilon_s \subseteq \varepsilon$, $\varepsilon_s = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$ (where $\varepsilon_3 = \varepsilon_f$), $\Phi_2 \equiv [\alpha_2; \varepsilon_2; \omega_2]$, $\alpha_2 \equiv \alpha_1 \cup \varepsilon_1 = \alpha$ (since $\varepsilon_1 = \emptyset$; if it's not $\emptyset$ we can construct a derivation for v that has $\varepsilon_1 = \emptyset$ as argued in preservation (Lemma D.0.36), (TApp)-[Cong], case $v\ \mathbb{E}$). We have

$$\begin{array}{l}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon_2 \Rightarrow n' \in ver(H, \mathsf{f})\end{array}$$

hence $\Phi_2, \mathcal{R}_2; H \vdash \Sigma$ and we can apply induction on $\Phi_2; \Gamma \vdash \mathbb{E}[e_2] : \tau_1 \longrightarrow^{\Phi_f} \tau'_2 \rightsquigarrow \mathcal{R}_2$, yielding:

(i) $\Phi'_2; \Gamma' \vdash \mathbb{E}[e_2] : \tau_1 \rightsquigarrow \mathcal{R}_2$ and

(ii) $n+1; \Gamma' \vdash H'$

(iii) $\Phi'_2, \mathcal{R}_2; H' \vdash \Sigma'$

(iv) $traceOK(\Sigma')$

(v) $(dir = bck) \Rightarrow n'' \equiv n+1\ \wedge\ (dir = fwd) \Rightarrow (\mathsf{f} \in \omega_2 \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f}))$

where $\Phi'_2 \equiv [\alpha_2; \varepsilon'_2; \omega_2]$ and $\varepsilon'_2 \subseteq \varepsilon_2$. Choose $\Phi'_1 = [\alpha; \emptyset; \omega_2 \cup \varepsilon'_2]$ and $\Phi'_3 = [\alpha \cup \varepsilon'_2; \varepsilon_f; \omega]$ and thus $\Phi'_1 \rhd \Phi'_2 \rhd \Phi'_3 \hookrightarrow \Phi'$ and $\Phi'^{\varepsilon}_3 = \Phi^{\varepsilon}_f$.

Let $\Phi' \equiv [\alpha; \varepsilon'_2 \cup \varepsilon_f; \omega]$ and thus $\varepsilon'_2 \cup \varepsilon_f \subseteq \varepsilon$ as required.

To prove 1., we have $n+1; \Gamma' \vdash H'$ by (ii), and apply (TAPP):

$$\text{TApp}\frac{\begin{array}{c}\Phi'_1; \Gamma' \vdash v : \tau_1 \longrightarrow^{\Phi_f} \tau'_2 \rightsquigarrow \cdot \qquad \Phi'_2; \Gamma' \vdash \mathbb{E}[e_2] : \tau_1 \rightsquigarrow \mathcal{R}_2 \\ \Phi'_1 \rhd \Phi'_2 \rhd \Phi'_3 \hookrightarrow \Phi' \\ \Phi'^{\varepsilon}_3 = \Phi^{\varepsilon}_f \qquad\qquad \Phi'^{\alpha}_3 \subseteq \Phi^{\alpha}_f \qquad\qquad \Phi'^{\omega}_3 \subseteq \Phi^{\omega}_f \\ v \not\equiv v' \Rightarrow \mathcal{R}_2 = \cdot\end{array}}{\Phi'; \Gamma' \vdash (v\ \mathbb{E})[e_2] : \tau'_2 \rightsquigarrow \mathcal{R}_2}$$

The first premise follows by value typing, the second by (i), the third—sixth by choice of Φ'_1 and Φ'_3, and the last holds vacuously. We can now apply (TSUB):

$$\text{TSub}\frac{\begin{array}{c}\Phi'; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau'_2 \rightsquigarrow \mathcal{R}_2 \\ \tau'_2 \leq \tau_2 \qquad\qquad \Phi' \leq \Phi'\end{array}}{\Phi'; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau_2 \rightsquigarrow \mathcal{R}_2}$$

To prove part 2., we must show that $\Phi', \mathcal{R}_2; H' \vdash \Sigma'$.
By inversion on $\Phi, \mathcal{R}_2; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$ or $\Sigma \equiv (\beta, \sigma), \Sigma''$. We have two cases:

$\Sigma \equiv (\beta, \sigma)$: By (iii) we must have $\mathcal{R}_2 \equiv \cdot$ such that

$$\text{TC1}\frac{\begin{array}{c}\mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon'_2 \Rightarrow n'' \in ver(H', \mathsf{f})\end{array}}{[\alpha; \varepsilon'_2; \omega_2], \cdot; H' \vdash (\beta', \sigma')}$$

To achieve the desired result we need to prove:

$$\text{TC1}\frac{\begin{array}{c}\mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon'_2 \cup \varepsilon_f \Rightarrow n'' \in ver(H', \mathsf{f})\end{array}}{[\alpha; \varepsilon'_2 \cup \varepsilon_f; \omega], \cdot; H' \vdash (\beta', \sigma')}$$

The first premise follows by assumption (since $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$ from the definition of $\mathcal{U}[(\beta, \sigma)]^{upd,dir}_{n+1}$). For the second premise, we need to show that for all $\mathsf{f} \in \varepsilon_f \Rightarrow n'' \in ver(H', \mathsf{f})$ (for those $\mathsf{f} \in \varepsilon'_2$ the result is by assumption).
Consider each possible update type:

case $dir = bck$ **:**
From the definition of $\mathcal{U}[(\beta, \sigma)]^{upd,bck}_{n+1}$, we know that $n'' = n+1$; from the definition of $\mathcal{U}[H]^{upd}_n$ we know that $n+1 \in ver(H', \mathsf{f})$ for all f, hence $n'' \in ver(H', \mathsf{f})$ for all f.

case $dir = fwd$ **:**
From (v) we have that $\mathsf{f} \in \omega_2 \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f})$. Thus $\varepsilon_f \subseteq \omega_2$ (by $\Phi'_1 \rhd \Phi'_2 \rhd \Phi'_3 \hookrightarrow \Phi'$) implies $\mathsf{f} \in \varepsilon_f \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f})$. By inversion on $\Phi, \mathcal{R}_2; H \vdash \Sigma$ we have $\mathsf{f} \in (\varepsilon_2 \cup \varepsilon_f) \Rightarrow n' \in ver(H, \mathsf{f})$, and thus $\mathsf{f} \in \varepsilon_f \Rightarrow n' \in ver(H', \mathsf{f})$. We have $\mathcal{U}[(\beta, \sigma)]^{upd,fwd}_{n+1} = (\beta, \sigma)$ hence $n'' = n'$, so finally we have $\mathsf{f} \in \varepsilon_f \Rightarrow n'' \in ver(H', \mathsf{f})$.

$\Sigma \equiv (\beta, \sigma), \Sigma''$ By (iii), we must have $\mathcal{R}_2 \equiv \Phi'', \mathcal{R}''$ such that

$$\text{TC2}\frac{\begin{array}{c}\Phi'', \mathcal{R}''; H' \vdash \Sigma'' \\ \Phi'_2 \equiv [\alpha; \varepsilon'_2; \omega_2] \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon'_2 \Rightarrow n'' \in ver(H', \mathsf{f})\end{array}}{\Phi'_2, \Phi'', \mathcal{R}''; H' \vdash ((\beta', \sigma'), \Sigma'')}$$

We wish to show that

$$\text{TC2}\frac{\begin{array}{c}\Phi'', \mathcal{R}''; H' \vdash \Sigma'' \\ \Phi' \equiv [\alpha; \varepsilon'_2 \cup \varepsilon_f; \omega] \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\varepsilon'_2 \cup \varepsilon_f) \Rightarrow n'' \in ver(H', \mathsf{f})\end{array}}{\Phi', \Phi'', \mathcal{R}''; H' \vdash ((\beta', \sigma'), \Sigma'')}$$

$\Phi'', \mathcal{R}''; H' \vdash \Sigma$ follows by assumption while the third and fourth premises follow by the same argument as in the $\Sigma \equiv (\beta, \sigma)$ case, above.

Part 3. follows directly from (iv).

Part 4. follows directly from (v) and the fact that $\omega_2 \supseteq \omega$.

case all others **:**

Similar to cases above.

□

This lemma says that if take an evaluation step that is not an update, the version set of any z remains unchanged.

Lemma B.0.14 (Non-update step version preservation)**.** *If* $\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'; H';\; e'\rangle$ *then for all* $\mathsf{z} \in \mathrm{dom}(H')$*,* $ver(H', \mathsf{z}) = ver(H, \mathsf{z})$*.*

Proof. By inspection of the evaluation rules. □

The following lemma states that if we start with a well-typed program and a version-consistent trace and we can take an evaluation step, then afterward we will still have a well-typed program whose trace is version-consistent.

Lemma B.0.15 (Preservation)**.**
Suppose we have the following:

1. $n \vdash H, e : \tau$ *(such that* $\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}$ *and* $n; \Gamma \vdash H$ *for some* Γ *and* Φ*)*
2. $\Phi, \mathcal{R}; H \vdash \Sigma$
3. $traceOK(\Sigma)$
4. $\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'; H';\; e'\rangle$

Then for some $\Gamma' \supseteq \Gamma$ *and* $\Phi' \equiv [\Phi^{\alpha} \cup \varepsilon_0; \varepsilon'; \Phi^{\omega}]$ *such that* $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^{\varepsilon}$*, we have:*

1. $n \vdash H', e' : \tau$ *where* $\Phi'; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}'$ *and* $n; \Gamma' \vdash H'$
2. $\Phi', \mathcal{R}'; H' \vdash \Sigma'$
3. $traceOK(\Sigma')$

Proof. Induction on the typing derivation $n \vdash H, e : \tau$. By inversion, we have that $\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}$; consider each possible rule for the conclusion of this judgment:

case (TInt-TVar-TGvar-TLoc) **:**

These expressions do not reduce, so the result is vacuously true.

case (TRef) **:**

We have that:

$$(\text{TRef})\frac{\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}}{\Phi; \Gamma \vdash \mathsf{ref}\ e : ref^{\varepsilon}\ \tau \rightsquigarrow \mathcal{R}}$$

There are two possible reductions:

case [ref] **:**

We have that $e \equiv v$, $\mathcal{R} = \cdot$, and $\langle n; (\beta, \sigma); H; \mathsf{ref}\ v\rangle \longrightarrow_{\emptyset} \langle n; (\beta, \sigma); H'; r\rangle$ where $r \notin \mathrm{dom}(H)$ and $H' = H, r \mapsto (\cdot, v, \emptyset)$.

Let $\Gamma' = \Gamma, r : ref^{\varepsilon}\ \tau$ and $\Phi' = \Phi$ (which is acceptable since $\Phi'^{\alpha} = \Phi^{\alpha} \cup \emptyset$, $\varepsilon' \cup \emptyset \subseteq \Phi^{\varepsilon}$, and $\Phi'^{\omega} = \Phi^{\omega}$), and $\mathcal{R}' = \cdot$. We have part 1. as follows:

$$(\text{TSub})\frac{(\text{TLoc})\dfrac{\Gamma'(r) = ref^{\varepsilon}\ \tau}{\Phi_{\emptyset}; \Gamma' \vdash r : ref^{\varepsilon}\ \tau \rightsquigarrow \cdot} \qquad ref^{\varepsilon}\ \tau \leq ref^{\varepsilon}\ \tau \qquad \Phi_{\emptyset} \leq \Phi}{\Phi; \Gamma' \vdash r : ref^{\varepsilon}\ \tau \rightsquigarrow \cdot}$$

Heap well-formedness $n; \Gamma' \vdash H, r \mapsto (\cdot, v, \emptyset)$ holds since $\Phi_{\emptyset}; \Gamma' \vdash v : \tau$ follows by value typing (Lemma B.0.5) from $\Phi; \Gamma' \vdash v : \tau$, which we have by assumption and weakening; we have $n; \Gamma' \vdash H$ by weakening.

To prove 2., we must show $\Phi, \cdot; H' \vdash (\beta, \sigma)$. This follows by assumption since H' only contains an additional location (i.e., not a global variable) and nothing else has changed. Part 3. follows by assumption since $\Sigma' = \Sigma$.

case [CONG] **:**

We have that $\langle n; \Sigma; H; \mathsf{ref}\ \mathbb{E}[e'']\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; \mathsf{ref}\ \mathbb{E}[e''']\rangle$ from $\langle n; \Sigma; H; e''\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e'''\rangle$. By [CONG], we have $\langle n; \Sigma; H; e\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e'\rangle$ where $e \equiv \mathbb{E}[e'']$ and $e' \equiv \mathbb{E}[e''']$.

By induction we have:

(i) $\Phi'; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}'$ and

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi', \mathcal{R}'; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

where $\Phi'^\alpha = \Phi^\alpha \cup \varepsilon_0$, $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^\varepsilon$, and $\Phi'^\omega = \Phi^\omega$. We prove 1. using (ii), and applying [TREF] using (i):

$$(\text{TREF})\dfrac{\Phi'; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}'}{\Phi'; \Gamma' \vdash \mathsf{ref}\ e' : \mathit{ref}^{\,\varepsilon}\ \tau \rightsquigarrow \mathcal{R}'}$$

Part 2. follows directly from (iii), and part 3. follows directly from (iv).

case (TDEREF) **:**

We know that

$$(\text{TDEREF})\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash e : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R} \\ \Phi_2^\varepsilon = \varepsilon_r \qquad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash !\,e : \tau \rightsquigarrow \mathcal{R}}$$

We can reduce using either [GVAR-DEREF], [DEREF], or [CONG].

case [GVAR-DEREF] **:**

Thus we have $e \equiv \mathsf{z}$ such that

$$\langle n; (\beta, \sigma); (H'', \mathsf{z} \mapsto (\tau', v, \nu)); !\,\mathsf{z}\rangle \ \longrightarrow_{\{\mathsf{z}\}}\ \langle n; (\beta, \sigma \cup (\mathsf{z}, \nu)); (H'', \mathsf{z} \mapsto (\tau', v, \nu)); v\rangle$$

(where $H \equiv (H'', \mathsf{z} \mapsto (\tau', v, \nu))$), by subtyping derivations (Lemma B.0.6) we have

$$(\text{TSUB})\dfrac{(\text{TGVAR})\dfrac{\Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon_r'}\ \tau'}{\Phi_\emptyset; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r'}\ \tau' \rightsquigarrow \cdot} \qquad \dfrac{\tau' \le \tau \qquad \tau \le \tau' \qquad \varepsilon_r' \subseteq \varepsilon_r}{\mathit{ref}^{\,\varepsilon_r'}\ \tau' \le \mathit{ref}^{\,\varepsilon_r}\ \tau} \qquad \Phi_\emptyset \le \Phi_1}{\Phi_1; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \cdot}$$

and

$$(\text{TDEREF})\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \cdot \\ \Phi_2^\varepsilon = \varepsilon_r \qquad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash !\,\mathsf{z} : \tau \rightsquigarrow \cdot}$$

(where $\mathcal{R} = \cdot$) and $\Phi \equiv [\Phi_1^\alpha; \Phi_1^\varepsilon \cup \varepsilon_r; \Phi_2^\omega]$. Let $\Gamma' = \Gamma$, $\Phi' = [\Phi_1^\alpha \cup \{\mathsf{z}\}; \emptyset; \Phi_2^\omega]$ and $\mathcal{R}' = \mathcal{R} = \cdot$. Since $\mathsf{z} \in \varepsilon_r$ (by $n; \Gamma \vdash H$) we have $\emptyset \cup \{\mathsf{z}\} \subseteq (\Phi_1^\varepsilon \cup \varepsilon_r)$ hence $\varepsilon' \cup \{\mathsf{z}\} \subseteq \Phi^\varepsilon$. The choice of Φ' is acceptable since $\Phi'^\alpha = \Phi^\alpha \cup \{\mathsf{z}\}$, $\varepsilon' \cup \{\mathsf{z}\} \subseteq \Phi^\varepsilon$, and $\Phi'^\omega = \Phi^\omega$.

To prove 1., we need to show that $\Phi'; \Gamma \vdash v : \tau \rightsquigarrow \cdot$ (the rest of the premises follow by assumption of $n \vdash H, !\,\mathsf{z} : \tau$). $H(\mathsf{z}) = (\tau', v, \nu)$ and $\Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon_r'}\ \tau'$ implies $\Phi'; \Gamma \vdash v : \tau' \rightsquigarrow \cdot$ by $n; \Gamma \vdash H$. The result follows by (TSUB):

$$(\text{TSUB})\dfrac{\Phi'; \Gamma \vdash v : \tau' \rightsquigarrow \cdot \qquad \tau' \le \tau \qquad \Phi' \le \Phi'}{\Phi'; \Gamma \vdash v : \tau \rightsquigarrow \cdot}$$

For part 2., we know $\Phi, \cdot; H \vdash (\beta, \sigma)$:

$$(\text{TC1})\dfrac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \\ \mathsf{f} \in (\Phi_1^\varepsilon \cup \varepsilon_r) \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})\end{array}}{[\Phi_1^\alpha; \Phi_1^\varepsilon \cup \varepsilon_r; \Phi_2^\omega], \cdot; H \vdash (\beta, \sigma)}$$

and need to prove $\Phi', \cdot; H \vdash (\beta, \sigma \cup (\mathsf{z}, \nu))$, hence:

$$(\text{TC1})\dfrac{\begin{array}{c}\mathsf{f} \in (\sigma \cup (\mathsf{z}, \nu)) \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \{\mathsf{z}\} \\ \mathsf{f} \in \emptyset \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})\end{array}}{[\Phi_1^\alpha \cup \{\mathsf{z}\}; \emptyset; \Phi_2^\omega], \cdot; H \vdash (\beta, \sigma \cup (\mathsf{z}, \nu))}$$

The first premise is true by assumption for all $\mathsf{f} \in \sigma$, and for (z, ν) since $\mathsf{z} \in \Phi_1^\alpha \cup \{\mathsf{z}\}$. The second premise is vacuously true.

For part 3., we need to prove $\mathit{traceOK}(n', \sigma \cup (\mathsf{z}, \nu))$; we have $\mathit{traceOK}(n', \sigma)$ by assumption, hence need to prove that $n' \in \nu$. Since by assumption of version consistency we have that $\mathsf{f} \in \Phi_1^\varepsilon \cup \varepsilon_r \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$ and $\mathit{ver}(H, \mathsf{f}) = \mathit{ver}(H', \mathsf{f}) = \nu$ (by Lemma B.0.14), and $\{\mathsf{z}\} \subseteq \varepsilon_r$ (by $n; \Gamma \vdash H$), we have $n' \in \nu$.

case [DEREF] **:**

Follows the same argument as the [GVAR-DEREF] case above for part 1.; parts 2 and 3 follow by assumption since the trace has not changed.

case [CONG] **:**

Here $\langle n; \Sigma; H; !\, e\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; !\, e'\rangle$ follows from $\langle n; \Sigma; H, e\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H', e'\rangle$. To apply induction, we must have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ which follows by Lemma B.0.9 since $\Phi, \mathcal{R}; H \vdash \Sigma$ and $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$. Hence we have:

(i) $\Phi'_1; \Gamma' \vdash e' : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \mathcal{R}'$

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi'_1, \mathcal{R}'; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi'_1 \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1; \Phi_1^\omega]$ where $\varepsilon'_1 \cup \varepsilon_0 \subseteq \Phi_1^\varepsilon$. Let $\Phi'_2 = [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_r; \Phi_2^\omega]$ hence $\Phi_2'^{\varepsilon} = \varepsilon_r$ and $\Phi'_1 \rhd \Phi'_2 \hookrightarrow \Phi'$, where $\Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1 \cup \varepsilon_r; \Phi_2^\omega]$ and $(\varepsilon'_1 \cup \varepsilon_r) \cup \varepsilon_0 \subseteq (\varepsilon_1 \cup \varepsilon_r)$ hence $\Phi'^{\alpha} = \Phi^\alpha \cup \varepsilon_0$, $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^\varepsilon$, and $\Phi'^{\omega} = \Phi^\omega$ as required.

We prove 1. by (ii) and by applying [TDEREF]:

$$(\text{TDEREF})\frac{\begin{array}{c}\Phi'_1; \Gamma' \vdash e' : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \mathcal{R}' \\ \Phi_2'^{\varepsilon} = \varepsilon_r \qquad \Phi'_1 \rhd \Phi'_2 \hookrightarrow \Phi'\end{array}}{\Phi'; \Gamma' \vdash !\, e' : \tau \leadsto \mathcal{R}'}$$

The first premise follows from (i) and the second and third premises follows by definition of Φ' and Φ'_2.

To prove part 2., we must show that $\Phi', \mathcal{R}'; H' \vdash \Sigma'$. We have two cases:

$\Sigma' \equiv (\beta, \sigma)$: By (iii) we must have $\mathcal{R}' \equiv \cdot$ such that

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in \varepsilon'_1 \Rightarrow n' \in \mathit{ver}(H', \mathsf{f})\end{array}}{[\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1; \Phi_1^\omega], \cdot; H' \vdash (\beta, \sigma)}$$

To achieve the desired result we need to prove:

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon'_1 \cup \varepsilon_r) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f})\end{array}}{[\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1 \cup \varepsilon_r; \Phi_1^\omega], \cdot; H' \vdash (\beta, \sigma)}$$

The first premise follows directly from (iii). To prove the second premise, we observe that by Lemma B.0.11, $\mathit{top}(\Sigma) = (n', \sigma')$ where $\sigma' \subseteq \sigma$, and by inversion on $\Phi; \mathcal{R}; H \vdash \Sigma$ we know (a) $\mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \Phi_1^\alpha$, and (b) $\mathsf{f} \in \varepsilon_1 \cup \varepsilon_r \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$. The second premise follows from (iii) and the fact that $\mathsf{f} \in \varepsilon_r \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$ by (b), and for all f, $\mathit{ver}(H, \mathsf{f}) = \mathit{ver}(H', \mathsf{f})$ by Lemma B.0.14.

$\Sigma' \equiv (\beta, \sigma), \Sigma''$: By (iii), we must have $\mathcal{R}' \equiv \Phi''', \mathcal{R}'''$ such that

$$(\text{TC2})\frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi'_1 \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1; \Phi_1^\omega] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in \varepsilon'_1 \Rightarrow n' \in \mathit{ver}(H', \mathsf{f})\end{array}}{\Phi'_1, \Phi''', \mathcal{R}'''; H' \vdash (\beta, \sigma), \Sigma''}$$

We wish to show that

$$(\text{TC2})\frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1 \cup \varepsilon_r; \Phi_2^\omega] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in \varepsilon'_1 \cup \varepsilon_r \Rightarrow n' \in \mathit{ver}(H', \mathsf{f})\end{array}}{\Phi', \Phi''', \mathcal{R}'''; H' \vdash (\beta, \sigma), \Sigma''}$$

The first and third premises follow from (iii), while the fourth premise follows by the same argument as in the $\Sigma' \equiv (\beta, \sigma)$ case, above.

Part 3. follows directly from (iv).

case (TAssign) **:**

We know that:

$$(\text{TAssign})\frac{\begin{array}{cc}\Phi_1;\Gamma \vdash e_1 : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}_1 & \Phi_2;\Gamma \vdash e_2 : \tau \rightsquigarrow \mathcal{R}_2\\ \Phi_3^{\varepsilon} = \varepsilon_r & \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi\\ \multicolumn{2}{c}{e_1 \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot}\end{array}}{\Phi;\Gamma \vdash e_1 := e_2 : \tau \rightsquigarrow \mathcal{R}_1 \bowtie \mathcal{R}_2}$$

From $\mathcal{R}_1 \bowtie \mathcal{R}_2$ it follows that either $\mathcal{R}_1 \equiv \cdot$ or $\mathcal{R}_2 \equiv \cdot$.

We can reduce using [gvar-assign], [assign], or [cong].

case [gvar-assign] **:**

This implies that $e \equiv \mathsf{z} := v$ with

$$\langle n;(\beta,\sigma);(H'',\mathsf{z}\mapsto(\tau,v',\nu));\mathsf{z} := v\rangle \ \longrightarrow_{\{\mathsf{z}\}}\ \langle n;(\beta,\sigma\cup(\mathsf{z},\nu));(H'',\mathsf{z}\mapsto(\tau,v,\nu));v\rangle$$

where $H \equiv (H'',\mathsf{z}\mapsto(\tau,v',\nu))$. $\mathcal{R}_1 \equiv \cdot$ and $\mathcal{R}_2 \equiv \cdot$ (thus $\mathcal{R}_1 \bowtie \mathcal{R}_2 \equiv \cdot$).

Let $\Gamma' = \Gamma$, $\mathcal{R}' = \cdot$, and $\Phi' = [\Phi^{\alpha}\cup\{\mathsf{z}\};\emptyset;\Phi^{\omega}]$. Since $\mathsf{z} \in \varepsilon_r$ (by $n;\Gamma \vdash H$) we have $\emptyset \subseteq (\varepsilon_1\cup\varepsilon_2\cup\varepsilon_r)$, hence $\emptyset\cup\{\mathsf{z}\} \subseteq (\varepsilon_1\cup\varepsilon_2\cup\varepsilon_r)$ which means $\varepsilon'\cup\{\mathsf{z}\} \subseteq \Phi^{\varepsilon}$. The choice of Φ' is acceptable since $\Phi'^{\alpha} = \Phi^{\alpha}\cup\{\mathsf{z}\}$, $\varepsilon'\cup\{\mathsf{z}\} \subseteq \Phi^{\varepsilon}$, and $\Phi'^{\omega} = \Phi^{\omega}$. We prove 1. as follows. Since $\Phi_2;\Gamma \vdash v : \tau \rightsquigarrow \cdot$, by value typing (Lemma B.0.5) we have $\Phi';\Gamma \vdash v : \tau \rightsquigarrow \cdot$. $n;\Gamma \vdash H'$ follows from $n;\Gamma \vdash H$ and $\Phi';\Gamma \vdash v : \tau \rightsquigarrow \cdot$ (since $\Phi^{\varepsilon} = \emptyset$).

Parts 2. and 3. are similar to the (TDeref) case.

case [assign] **:**

Part 1. is similar to (gvar-assign); we have parts 2. and 3. by assumption.

case [cong] **:**

Consider the shape of $\mathbb{E}$:

case $\mathbb{E} := e$ **:**

$\langle n;\Sigma;H;e_1 := e_2\rangle \longrightarrow_{\varepsilon} \langle n;\Sigma';H';e_1' := e_2\rangle$ follows from $\langle n;\Sigma;H;e_1\rangle \longrightarrow_{\varepsilon} \langle n;\Sigma';H';e_1'\rangle$. Since $e_1 \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot$ by assumption, by Lemma B.0.10 we have $\Phi_1,\mathcal{R}_1;H \vdash \Sigma$, hence we can apply induction:

(i) $\Phi_1';\Gamma' \vdash e_1' : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}_1'$ and

(ii) $n;\Gamma' \vdash H'$

(iii) $\Phi_1',\mathcal{R}_1';H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi_1' \equiv [\Phi_1^{\alpha}\cup\varepsilon_0;\varepsilon_1';\Phi_1^{\omega}]$ where $\varepsilon_1'\cup\varepsilon_0 \subseteq \varepsilon_1$ and $\Phi_1^{\omega} \equiv \Phi_2^{\varepsilon}\cup\varepsilon_r\cup\Phi_3^{\omega}$.

Let $\begin{array}{lcl}\Phi_2' & \equiv & [\Phi_1^{\alpha}\cup\varepsilon_1'\cup\varepsilon_0;\Phi_2^{\varepsilon};\varepsilon_r\cup\Phi_3^{\omega}]\\ \Phi_3' & \equiv & [\Phi_1^{\alpha}\cup\varepsilon_1'\cup\varepsilon_0\cup\Phi_2^{\varepsilon};\varepsilon_r;\Phi_3^{\omega}]\end{array}$

Thus $\Phi_3'^{\varepsilon} = \varepsilon_r$ and $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'$ such that $\Phi' \equiv [\Phi_1^{\alpha}\cup\varepsilon_0;\varepsilon_1'\cup\Phi_2^{\varepsilon}\cup\varepsilon_r;\Phi_3^{\omega}]$ The choice of Φ' is acceptable since $\Phi'^{\alpha} = \Phi^{\alpha}\cup\varepsilon_0$, $(\varepsilon_1'\cup\varepsilon_r\cup\varepsilon_2)\cup\varepsilon_0 \subseteq (\varepsilon_1\cup\varepsilon_r\cup\varepsilon_2)$ i.e., $\varepsilon'\cup\varepsilon_0 \subseteq \Phi^{\varepsilon}$ and $\Phi'^{\omega} = \Phi^{\omega}$ as required).

To prove 1., we have $n;\Gamma' \vdash H'$ by (ii), and apply (TAssign):

$$(\text{TAssign})\frac{\begin{array}{cc}\Phi_1';\Gamma' \vdash e_1' : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}_1' & (\text{TSub})\dfrac{\Phi_2;\Gamma' \vdash e_2 : \tau \rightsquigarrow \mathcal{R}_2 \quad \tau \leq \tau \quad \dfrac{\begin{array}{c}\Phi_1^{\alpha}\cup\varepsilon_1'\cup\varepsilon_0 \subseteq \Phi_1^{\alpha}\cup\Phi_1^{\varepsilon}\\ \Phi_2^{\varepsilon} \subseteq \Phi_2^{\varepsilon}\\ \varepsilon_r\cup\Phi_3^{\omega} \subseteq \varepsilon_r\cup\Phi_3^{\omega}\end{array}}{\Phi_2 \leq \Phi_2'}}{\Phi_2';\Gamma' \vdash e_2 : \tau \rightsquigarrow \mathcal{R}_2}\\ \Phi_3'^{\varepsilon} = \varepsilon_r & \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'\\ \multicolumn{2}{c}{e_1' \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot}\end{array}}{\Phi';\Gamma' \vdash e_1' := e_2 : \tau \rightsquigarrow \mathcal{R}_1' \bowtie \mathcal{R}_2}$$

Note that $\Phi_2;\Gamma' \vdash e_2 : \tau$ follows from $\Phi_2;\Gamma \vdash e_2 : \tau$ by weakening (Lemma B.0.1).

To prove part 2., we must show that $\Phi',\mathcal{R}_1';H' \vdash \Sigma'$ (since $\mathcal{R}_1' \bowtie \mathcal{R}_2 = \mathcal{R}_1'$). By inversion on $\Phi,\mathcal{R};H \vdash \Sigma$ we have $\Sigma \equiv (\beta,\sigma)$ or $\Sigma \equiv (\beta,\sigma),\Sigma''$. We have two cases:

$\Sigma' \equiv (\beta,\sigma)$: By (iii) we must have $\mathcal{R}_1' \equiv \cdot$ such that

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^{\alpha}\cup\varepsilon_0\\ \mathsf{f} \in \varepsilon_1' \Rightarrow n' \in \mathit{ver}(H',\mathsf{f})\end{array}}{[\Phi_1^{\alpha}\cup\varepsilon_0;\varepsilon_1';\Phi_1^{\omega}],\cdot;H' \vdash (\beta,\sigma)}$$

To achieve the desired result we need to prove:

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon_1' \cup \Phi_2^\varepsilon \cup \varepsilon_r) \Rightarrow n' \in ver(H', \mathsf{f})\end{array}}{[\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_1' \cup \Phi_2^\varepsilon \cup \varepsilon_r; \Phi_3^\omega], \cdot; H' \vdash (\beta, \sigma)}$$

The first premise follows directly from (iii). To prove the second premise, we observe that by Lemma B.0.11, $top(\Sigma) = (n', \sigma')$ where $\sigma' \subseteq \sigma$, and by inversion on $\Phi; \mathcal{R}; H \vdash \Sigma$ we know (a) $\mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \Phi_1^\alpha$, and (b) $\mathsf{f} \in \Phi_1^\varepsilon \cup \Phi_2^\varepsilon \cup \varepsilon_r \Rightarrow n' \in ver(H, \mathsf{f})$. The second premise follows from (iii) and the fact that $\mathsf{f} \in \varepsilon_r \Rightarrow n' \in ver(H, \mathsf{f})$ by (b), and for all f, $ver(H, \mathsf{f}) = ver(H', \mathsf{f})$ by Lemma B.0.14.

$\Sigma' \equiv (\beta, \sigma), \Sigma''$: By (iii), we must have $\mathcal{R}_1' \equiv \Phi''', \mathcal{R}'''$ such that

$$(\text{TC2})\frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi_1' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_1'; \Phi_1^\omega] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in \varepsilon_1' \Rightarrow n' \in ver(H', \mathsf{f})\end{array}}{\Phi_1', \Phi''', \mathcal{R}'''; H' \vdash (\beta, \sigma), \Sigma''}$$

We wish to show that

$$(\text{TC2})\frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_1' \cup \Phi_2^\varepsilon \cup \varepsilon_r; \Phi_3^\omega] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon_1' \cup \Phi_2^\varepsilon \cup \varepsilon_r) \Rightarrow n' \in ver(H', \mathsf{f})\end{array}}{\Phi', \Phi''', \mathcal{R}'''; H' \vdash (\beta, \sigma), \Sigma''}$$

The first and third premises follow from (iii), while the fourth premise follows by the same argument as in the $\Sigma' \equiv (\beta, \sigma)$ case, above.

Part 3. follows directly from (iv).

case $r := \mathbb{E}$ **:**

$\langle n; \Sigma; H; r := e_2\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; r := e_2'\rangle$ follows from $\langle n; \Sigma; H; e_2\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e_2'\rangle$. Since $e_1 \equiv r$, by inversion $\mathcal{R}_1 \equiv \cdot$. By Lemma B.0.10 (which we can apply because $\Phi_1^\varepsilon \equiv \emptyset$; if $\Phi_1^\varepsilon \not\equiv \emptyset$ we can rewrite the derivation using value typing to make it so) we have $\Phi_2, \mathcal{R}_2; H \vdash \Sigma$, hence we can apply induction to get:

(i) $\Phi_2'; \Gamma' \vdash e_2' : \tau \rightsquigarrow \mathcal{R}_2'$

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi_2', \mathcal{R}_2'; H' \vdash \Sigma'$

(iv) $traceOK(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi_2' \equiv [\Phi_2^\alpha \cup \varepsilon_0; \varepsilon_2'; \Phi_2^\omega]$ where $(\varepsilon_2' \cup \varepsilon_0) \subseteq \Phi_2^\varepsilon$; note $\Phi_2^\alpha \equiv \Phi_1^\alpha$ (since $\Phi_1^\varepsilon \equiv \emptyset$) and $\Phi_2^\omega \equiv \varepsilon_3 \cup \Phi_3^\omega$.

Let $\begin{array}{lcl}\Phi_1' & \equiv & [\Phi_1^\alpha \cup \varepsilon_0; \emptyset; \varepsilon_2' \cup \varepsilon_r \cup \Phi_3^\omega] \\ \Phi_3' & \equiv & [\Phi_1^\alpha \cup \varepsilon_0 \cup \varepsilon_2'; \varepsilon_r; \Phi_3^\omega]\end{array}$

Thus $\Phi_3'^\varepsilon = \varepsilon_r$ and $\Phi_1' \triangleright \Phi_2' \triangleright \Phi_3' \hookrightarrow \Phi'$ such that $\Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_2' \cup \varepsilon_r; \Phi_3^\omega]$ and $(\varepsilon_2' \cup \varepsilon_r) \cup \varepsilon_0 \subseteq (\Phi_2^\varepsilon \cup \varepsilon_r)$. The choice of Φ' is acceptable since $\Phi'^\alpha = \Phi^\alpha \cup \varepsilon_0$, $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^\varepsilon$ and $\Phi'^\omega = \Phi^\omega$ as required).

To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and we can apply [TAssign]:

$$(\text{TAssign})\frac{\begin{array}{cc}\Phi_1'; \Gamma' \vdash r : ref^{\,\varepsilon_r}\ \tau \rightsquigarrow \cdot & \Phi_2'; \Gamma' \vdash e_2' : \tau \rightsquigarrow \mathcal{R}_2' \\ \Phi_3'^{\varepsilon_r} = \varepsilon_r & \Phi_1' \triangleright \Phi_2' \triangleright \Phi_3' \hookrightarrow \Phi' \\ \multicolumn{2}{c}{r \not\equiv v \Rightarrow \mathcal{R}_2' = \cdot}\end{array}}{\Phi'; \Gamma' \vdash r := e_2' : \tau \rightsquigarrow \cdot \bowtie \mathcal{R}_2'}$$

Note that we have $\Phi_1'; \Gamma' \vdash r : ref^{\,\varepsilon_r}\ \tau \rightsquigarrow \cdot$ from $\Phi_1; \Gamma \vdash r : ref^{\,\varepsilon_r}\ \tau \rightsquigarrow \cdot$ by value typing and weakening

To prove part 2., we must show that $\Phi', \mathcal{R}_2'; H' \vdash \Sigma'$ (since $\mathcal{R}_1 \bowtie \mathcal{R}_2 = \mathcal{R}_2'$). By inversion on $\Phi, \mathcal{R}; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$ or $\Sigma \equiv (\beta, \sigma), \Sigma''$. We have two cases:

$\Sigma' \equiv (\beta, \sigma)$: By (iii) we must have $\mathcal{R}_2' \equiv \cdot$ such that

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_2^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in \varepsilon_2' \Rightarrow n' \in ver(H', \mathsf{f})\end{array}}{[\Phi_2^\alpha \cup \varepsilon_0; \varepsilon_2'; \Phi_2^\omega], \cdot; H' \vdash (\beta, \sigma)}$$

To achieve the desired result we need to prove:

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon_r \cup \varepsilon_2') \Rightarrow n' \in ver(H', \mathsf{f})\end{array}}{[\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_2' \cup \varepsilon_r; \Phi_3^\omega], \cdot; H' \vdash (\beta, \sigma)}$$

The first premise follows from (iii) since $\Phi_1^\alpha = \Phi_2^\alpha$.
To prove the second premise, we observe that by Lemma B.0.11, $top(\Sigma) = (n', \sigma')$ where $\sigma' \subseteq \sigma$, and by inversion on $\Phi; \mathcal{R}; H \vdash \Sigma$ we know (a) $\mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \Phi_1^\alpha$, and (b) $\mathsf{f} \in \varepsilon_r \cup \Phi_2^\varepsilon \Rightarrow n' \in ver(H, \mathsf{f})$. The second premise follows from (iii) and the fact that $\mathsf{f} \in \varepsilon_r \Rightarrow n' \in ver(H, \mathsf{f})$ by (b), and for all f, $ver(H, \mathsf{f}) = ver(H', \mathsf{f})$ by Lemma B.0.14.

$\Sigma' \equiv (\beta, \sigma), \Sigma''$: By (iii), we must have $\mathcal{R}_2' \equiv \Phi''', \mathcal{R}'''$ such that:

$$(\text{TC2})\frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi_2' \equiv [\Phi_2^\alpha \cup \varepsilon_0; \varepsilon_2'; \Phi_2^\omega] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_2^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in \varepsilon_2' \Rightarrow n' \in ver(H', \mathsf{f})\end{array}}{\Phi_2', \Phi''', \mathcal{R}'''; H' \vdash (\beta, \sigma), \Sigma''}$$

We wish to show that

$$(\text{TC2})\frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_2' \cup \varepsilon_r; \Phi_3^\omega] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \cup \varepsilon_0 \\ \mathsf{f} \in \varepsilon_2' \cup \varepsilon_r \Rightarrow n' \in ver(H', \mathsf{f})\end{array}}{\Phi', \Phi''', \mathcal{R}'''; H' \vdash (\beta, \sigma), \Sigma''}$$

The first and third premises follow from (iii), while the fourth premise follows by the same argument as in the $\Sigma' \equiv (\beta, \sigma)$ case, above.

Part 3. follows directly from (iv).

case (TUpdate) **:**

case [no-update] **:**

Thus we must have

$$\langle n; (\beta, \sigma); H; \mathsf{update}^{\alpha'', \omega''} \rangle \longrightarrow \langle n; (\beta, \sigma); H; 0 \rangle$$

Let $\Gamma' = \Gamma$ and $\Phi' = \Phi$ (and thus $\varepsilon \cup \emptyset \subseteq \Phi^\varepsilon$, $\Phi'^\alpha = \Phi^\alpha \cup \emptyset$, and $\Phi'^\omega = \Phi^\omega$) as required. For 1., $\Phi; \Gamma \vdash 0 : int \rightsquigarrow \cdot$ follows from (TInt) and value typing and $n; \Gamma \vdash H$ is true by assumption. Parts 2. and 3. follow by assumption.

case (TIf) **:**

We know that:

$$(\text{TIf})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : int \rightsquigarrow \mathcal{R} \\ \Phi_2; \Gamma \vdash e_2 : \tau \rightsquigarrow \cdot \quad \Phi_2; \Gamma \vdash e_3 : \tau \rightsquigarrow \cdot \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \rightsquigarrow \mathcal{R}}$$

We can reduce using [if-t], [if-f] or [cong].

case [if-t] **:**

This implies that $e_1 \equiv v$ hence $\mathcal{R} = \cdot$. We have

$$\langle n; (\beta, \sigma); H; \mathsf{if0}\ v\ \mathsf{then}\ e2\ \mathsf{else}\ e3 \rangle \longrightarrow \langle n; (\beta, \sigma); H; e2 \rangle$$

Let $\Gamma' = \Gamma$ and $\Phi' = \Phi$ (and thus $\varepsilon \cup \emptyset \subseteq \Phi^\varepsilon$, $\Phi'^\alpha = \Phi^\alpha \cup \emptyset$, and $\Phi'^\omega = \Phi^\omega$) as required. To prove 1., we have $n; \Gamma \vdash H$ by assumption, and we have

$$(\text{TSub})\frac{\begin{array}{c}\Phi_2; \Gamma \vdash e_2 : \tau \rightsquigarrow \cdot \quad \tau \leq \tau \\ \Phi_2 \leq \Phi\end{array}}{\Phi; \Gamma \vdash e_2 : \tau \rightsquigarrow \cdot}$$

The first premise holds by assumption, the second by reflexivity of subtyping, and the third by Lemma B.0.2.

case [IF-F] **:**

This is similar to [IF-T].

case [CONG] **:**

$\langle n; \Sigma; H; \mathsf{if0}\ e_1\ \mathsf{then}\ e2\ \mathsf{else}\ e3\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; \mathsf{if0}\ e_1'\ \mathsf{then}\ e2\ \mathsf{else}\ e3\rangle$ follows from $\langle n; \Sigma; H; e_1\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e_1'\rangle$. To apply induction, we must have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ which follows by Lemma B.0.9 since $\Phi, \mathcal{R}; H \vdash \Sigma$ and $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$. Hence we have:

(i) $\Phi_1'; \Gamma' \vdash e_1' : int \rightsquigarrow \mathcal{R}'$ and
(ii) $n; \Gamma' \vdash H'$
(iii) $\Phi_1', \mathcal{R}'; H' \vdash \Sigma'$
(iv) $traceOK(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi_1' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_1'; \Phi_1^\omega]$ where $\varepsilon_1' \cup \varepsilon_0 \subseteq \Phi_1^\varepsilon$. (Note that $\Phi_1^\omega \equiv \Phi_2^\varepsilon \cup \Phi_2^\omega$.) Let $\Phi_2' \equiv [\Phi_1^\alpha \cup \varepsilon_1' \cup \varepsilon_0; \Phi_2^\varepsilon; \Phi_2^\omega]$. Thus $\Phi_1' \rhd \Phi_2' \hookrightarrow \Phi'$ so that $\Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_1' \cup \Phi_2^\varepsilon; \Phi_2^\omega]$ where $\varepsilon_1' \cup \varepsilon_0 \cup \Phi_2^\varepsilon \subseteq \Phi_1^\varepsilon \cup \Phi_2^\varepsilon$ and $\Phi'^\omega = \Phi^\omega$ as required.

To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and can apply (TIF): We prove 1. by (ii) and as follows:

$$\text{(TIF)}\dfrac{\text{(TSUB)}\dfrac{\Phi_2; \Gamma' \vdash e_2 : \tau \rightsquigarrow \cdot \qquad \tau \le \tau \qquad \Phi_2 \le \Phi_2'}{\Phi_2'; \Gamma' \vdash e_2 : \tau \rightsquigarrow \cdot} \quad \text{(TSUB)}\dfrac{\Phi_2; \Gamma' \vdash e_2 : \tau \rightsquigarrow \cdot \qquad \tau \le \tau \qquad \Phi_2 \le \Phi_2'}{\Phi_2'; \Gamma' \vdash e_3 : \tau \rightsquigarrow \cdot} \qquad \Phi_1'; \Gamma' \vdash e_1' : int \rightsquigarrow \mathcal{R}_1' \qquad \Phi_1' \rhd \Phi_2' \hookrightarrow \Phi'}{\Phi'; \Gamma' \vdash \mathsf{if0}\ e_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \rightsquigarrow \mathcal{R}'}$$

Note that $\Phi_2; \Gamma' \vdash e_2 : \tau \rightsquigarrow \mathcal{R}$ follows from $\Phi_2; \Gamma \vdash e_2 : \tau \rightsquigarrow \mathcal{R}$ by weakening (Lemma B.0.1) and likewise for $\Phi_2; \Gamma' \vdash e_3 : \tau \rightsquigarrow \mathcal{R}$.

Parts 2. and 3. follow by an argument similar to (TDEREF)-[CONG] and (TASSIGN)-[CONG].

case (TTRANSACT) **:**

We know that:

$$\text{(TTRANSACT)}\dfrac{\Phi''; \Gamma \vdash e : \tau \rightsquigarrow \cdot \qquad \Phi^\alpha \subseteq \Phi''^\alpha \qquad \Phi^\omega \subseteq \Phi''^\omega}{\Phi; \Gamma \vdash \mathsf{tx}\ e : \tau \rightsquigarrow \cdot}$$

We can reduce using [TX-START]:

$$\langle n; (\beta, \sigma); H; \mathsf{tx}\ e\rangle \longrightarrow \langle n; (n, \emptyset), (\beta, \sigma); H; \mathsf{intx}\ e\rangle$$

Let $\Gamma' = \Gamma$ and $\Phi' \equiv [\Phi^\alpha; \emptyset; \Phi^\omega]$ (and thus $\emptyset \cup \emptyset \subseteq \Phi^\varepsilon$, $\Phi'^\alpha = \Phi^\alpha \cup \emptyset$, and $\Phi'^\omega = \Phi^\omega$, as required). To prove 1., we have $n; \Gamma \vdash H$ by assumption, and the rest follows by (TINTRANS):

$$\text{(TINTRANS)}\dfrac{\Phi''; \Gamma \vdash e : \tau \rightsquigarrow \cdot \qquad \Phi'^\alpha \subseteq \Phi''^\alpha \qquad \Phi'^\omega \subseteq \Phi''^\omega}{\Phi'; \Gamma \vdash \mathsf{intx}\ e : \tau \rightsquigarrow \Phi'', \cdot}$$

The first premise is true by assumption, and the second by choice of Φ'.

We prove 2. as follows:

$$\text{(TC2)}\dfrac{\text{(TC1)}\dfrac{\mathsf{f} \in \emptyset \Rightarrow \mathsf{f} \in \Phi''^\alpha \qquad \mathsf{f} \in \Phi''^\varepsilon \Rightarrow n \in ver(H, \mathsf{f})}{\Phi'', \cdot; H \vdash (n, \emptyset)} \qquad \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi^\alpha \qquad \mathsf{f} \in \emptyset \Rightarrow n' \in ver(H, \mathsf{f})}{[\Phi^\alpha; \emptyset; \Phi^\omega], \Phi'', \cdot; H \vdash (n, \emptyset), (\beta, \sigma)}$$

First premise of [TC1] is true vacuously, and the second is true by $n; \Gamma \vdash H$, which we have by assumption. For [TC2], the first premise holds by inversion of $\Phi, \cdot; H \vdash (\beta, \sigma)$, which we have by assumption, and the second holds vacuously.

Part 3. follows easily: we have $traceOK((\beta, \sigma))$ by assumption, $traceOK((n, \emptyset))$ is vacuously true, hence $traceOK((n, \emptyset), (\beta, \sigma))$ is true.

case (TINTRANS) **:**

We know that:

$$\text{(TINTRANS)}\dfrac{\Phi''; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R} \qquad \Phi^\alpha \subseteq \Phi''^\alpha \qquad \Phi^\omega \subseteq \Phi''^\omega}{\Phi; \Gamma \vdash \mathsf{intx}\ e : \tau \rightsquigarrow \Phi'', \mathcal{R}}$$

There are two possible reductions:

case [TX-END] **:**

We have that $e \equiv v$ and thus $\mathcal{R} \equiv \cdot$; we reduce as follows:

$$\frac{\mathit{traceOK}(n'', \sigma')}{\langle n; ((\beta', \sigma'), (\beta, \sigma)); H; \mathsf{intx}\ v\rangle \ \longrightarrow\ \langle n; (\beta, \sigma); H; v\rangle}$$

Let $\Phi' = \Phi$ and $\Gamma' = \Gamma$ (and thus $\Phi'^{\alpha} = \Phi^{\alpha} \cup \emptyset$, $\varepsilon' \cup \emptyset \subseteq \Phi^{\varepsilon}$, and $\Phi'^{\omega} = \Phi^{\omega}$ as required). To prove 1., we know that $n; \Gamma \vdash H$ follows by assumption and $\Phi; \Gamma \vdash v : \tau \rightsquigarrow \cdot$ by value typing. To prove 2., we must show that $\Phi, \cdot; H \vdash (\beta, \sigma)$, but this is true by inversion on $\Phi, \Phi'', \cdot; H \vdash ((\beta', \sigma'), (\beta, \sigma))$.

For 3., $\mathit{traceOK}((\beta, \sigma))$ follows from $\mathit{traceOK}(((\beta', \sigma'), (\beta, \sigma)))$ (which is true by assumption).

case [TX-CONG-2] **:**

We know that

$$\frac{\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon} \langle n'; \Sigma'; H'; e'\rangle}{\langle n; \Sigma; H; \mathsf{intx}\ e\rangle \ \longrightarrow_{\emptyset}\ \langle n'; \Sigma'; H'; \mathsf{intx}\ e'\rangle}$$

follows from $\langle n; \Sigma; H; e\rangle \longrightarrow_{\eta} \langle n; \Sigma'; H'; e'\rangle$ (because the reduction does not perform an update, hence $\eta \equiv \varepsilon_0$ and we apply [TX-CONG-2]).

We have $\Phi'', \mathcal{R}; H \vdash \Sigma$ by inversion on $\Phi, \Phi'', \mathcal{R}; H \vdash ((\beta, \sigma), \Sigma)$, hence by induction:

(i) $\Phi'''; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}'$ and

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi''', \mathcal{R}'; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some Φ''' such that $\Phi'''^{\alpha} = \Phi''^{\alpha} \cup \varepsilon_0$, $\varepsilon''' \cup \varepsilon_0 \subseteq \Phi''^{\varepsilon}$, and $\Phi'''^{\omega} = \Phi''^{\omega}$.

Let $\Phi' = \Phi$ (hence $\Phi'^{\alpha} = \Phi'^{\alpha} \cup \emptyset$, $\varepsilon' \cup \emptyset \subseteq \Phi^{\varepsilon}$, and $\Phi'^{\omega} = \Phi^{\omega}$ as required) and $\Gamma' = \Gamma$.

To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and we can apply [TIntrans]:

$$(\text{TIntrans})\frac{\Phi'''; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}' \\ \Phi'^{\alpha} \subseteq \Phi'''^{\alpha} \qquad \Phi'^{\omega} \subseteq \Phi'''^{\omega}}{\Phi'; \Gamma' \vdash \mathsf{intx}\ e' : \tau \rightsquigarrow \Phi''', \mathcal{R}'}$$

The first premise follows from (i), and the second holds because $\Phi^{\alpha} \subseteq \Phi''^{\alpha}$ and $\Phi^{\omega} \subseteq \Phi''^{\omega}$ by assumption and we picked $\Phi' = \Phi$ (hence $\Phi'^{\alpha} \subseteq \Phi'''^{\alpha}$ $\Phi'^{\omega} \subseteq \Phi'''^{\omega}$).

Part 2. follows directly from (iii). Part 3. follows directly from (iv).

case (TLet) **:**

We know that:

$$(\text{TLet})\frac{\Phi_1; \Gamma \vdash e_1 : \tau_1 \rightsquigarrow \mathcal{R} \qquad \Phi_2; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow \cdot \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}{\Phi; \Gamma \vdash \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2 \rightsquigarrow \mathcal{R}}$$

We can reduce using either [LET] or [CONG].

case [LET] **:**

This implies that $e_1 \equiv v$ hence $\mathcal{R} \equiv \cdot$. We have:

$$\langle n; (\beta, \sigma); H; \mathsf{let}\ x : \tau = v\ \mathsf{in}\ e\rangle \ \longrightarrow\ \langle n; (\beta, \sigma); H; e[x \mapsto v]\rangle$$

To prove 1., we have $n; \Gamma \vdash H$ by assumption; let $\Gamma' = \Gamma$ and $\Phi' = \Phi$; since $\varepsilon_2 \subseteq (\varepsilon_1 \cup \varepsilon_2)$, we can apply [TSub]:

$$(\text{TSub})\frac{\Phi_2; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow \cdot \qquad \tau_2 \leq \tau_2 \\ \Phi_2 \leq \Phi}{\Phi; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow \cdot}$$

The first premise holds by assumption, the second by reflexivity of subtyping, and the third by Lemma B.0.2. By value typing we have $\Phi; \Gamma \vdash v : \tau_1 \rightsquigarrow \cdot$, so by substitution (Lemma B.0.17) we have $\Phi; \Gamma \vdash e_2[x \mapsto v] : \tau_2 \rightsquigarrow \cdot$.

Parts 2. and 3. hold by assumption.

case [CONG] **:**

Similar to (TIf)-[Cong].

case (TAPP) **:**

We know that:

$$(\text{TApp})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \mathcal{R}_1 \qquad \Phi_2; \Gamma \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ e_1 \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot \end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2 \rightsquigarrow \mathcal{R}_1 \bowtie \mathcal{R}_2}$$

We can reduce using either [CALL] or [CONG].

case [CALL] **:**

We have that

$$\langle n; (\beta, \sigma); (H'', \mathsf{z} \mapsto (\tau, \lambda(x).e, \nu)); \mathsf{z}\ v\rangle \ \longrightarrow_{\{\mathsf{z}\}} \langle n; (\beta, \sigma \cup (\mathsf{z}, \nu)); (H'', \mathsf{z} \mapsto (\tau, \lambda(x).e, \nu)); e[x \mapsto v]\rangle$$

(where $H \equiv (H'', \mathsf{z} \mapsto (\tau, \lambda(x).e, \nu))$), and

$$(\text{TApp})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash \mathsf{z} : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \cdot \qquad \Phi_2; \Gamma \vdash v : \tau_1 \rightsquigarrow \cdot \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ \mathsf{z} \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot \end{array}}{\Phi; \Gamma \vdash \mathsf{z}\ v : \tau_2 \rightsquigarrow \cdot}$$

where by subtyping derivations (Lemma B.0.6) we have

$$(\text{TSub})\frac{(\text{TGVar})\dfrac{\Gamma(\mathsf{z}) = \tau_1' \longrightarrow^{\Phi'_f} \tau_2'}{\Phi_\emptyset; \Gamma \vdash \mathsf{z} : \tau_1' \longrightarrow^{\Phi'_f} \tau_2' \rightsquigarrow \cdot} \qquad \dfrac{\tau_1 \leq \tau_1' \quad \tau_2' \leq \tau_2 \quad \Phi'_f \leq \Phi_f}{\tau_1' \longrightarrow^{\Phi'_f} \tau_2' \leq \tau_1 \longrightarrow^{\Phi_f} \tau_2} \qquad \Phi_\emptyset \leq \Phi_1}{\Phi_1; \Gamma \vdash \mathsf{z} : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \cdot}$$

Define $\Phi_f \equiv [\alpha_f; \varepsilon_f; \omega_f]$ and $\Phi'_f \equiv [\alpha'_f; \varepsilon'_f; \omega'_f]$.

Let $\Gamma' = \Gamma$, $\mathcal{R}' = \cdot$ and choose $\Phi' = [\Phi_1^{\alpha} \cup \{\mathsf{z}\}; \varepsilon_f; \Phi_3^{\omega}]$. Since $\mathsf{z} \in \varepsilon'_f$ (by $n; \Gamma \vdash H$) and $\varepsilon'_f \subseteq \varepsilon_f$ (by $\Phi'_f \leq \Phi_f$) we have $\varepsilon_f \cup \{\mathsf{z}\} \subseteq (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f)$. The choice of Φ' is acceptable since $\Phi'^{\alpha} = \Phi^{\alpha} \cup \{\mathsf{z}\}$, $\Phi'^{\varepsilon} \cup \{\mathsf{z}\} \subseteq \Phi^{\varepsilon}$, and $\Phi'^{\omega} = \Phi^{\omega}$. For 1., we have $n; \Gamma \vdash H'$ by assumption; for the remainder we have to prove $\Phi'; \Gamma \vdash e[x \mapsto v] : \tau_2 \rightsquigarrow \cdot$. First, we must prove that $\Phi'_f \leq \Phi'$. Note that since $\{\mathsf{z}\} \subseteq \alpha_f$ by $n; \Gamma \vdash H'$, from $\Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi$ and choice of Φ' we get $\Phi_3'^{\alpha} \cup \{\mathsf{z}\} \subseteq \alpha_f$. We have:

$$\begin{array}{lcll}
\Phi' & \equiv & [\Phi_1^{\alpha} \cup \{\mathsf{z}\}; \varepsilon_f; \Phi_3^{\omega}] & \text{(by choice of } \Phi'\text{)} \\
\Phi_f & \equiv & [\alpha_f; \varepsilon_f; \omega_f] & \\
\Phi'_f & \equiv & [\alpha'_f; \varepsilon'_f; \omega'_f] & \\
\varepsilon'_f & \subseteq & \varepsilon_f & \text{(by } \Phi'_f \leq \Phi_f\text{)} \\
\alpha_f & \subseteq & \alpha'_f & \text{(by } \Phi'_f \leq \Phi_f\text{)} \\
\omega_f & \subseteq & \omega'_f & \text{(by } \Phi'_f \leq \Phi_f\text{)} \\
\Phi_3'^{\alpha} \cup \{\mathsf{z}\} & \subseteq & \alpha_f & \text{(by assumption and choice of } \Phi'\text{)} \\
\Phi_3'^{\alpha} & = & \Phi_1^{\alpha} \cup \Phi_1^{\varepsilon} \cup \Phi_2'^{\varepsilon} & \text{(by } \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi\text{)} \\
\Phi_3'^{\omega} & \subseteq & \omega_f & \text{(by assumption and choice of } \Phi'\text{)}
\end{array}$$

Thus we have the result by [TSUB]

$$\frac{\Phi'_f; \Gamma \vdash e[x \mapsto v] : \tau_2' \rightsquigarrow \cdot \qquad \tau_2' \leq \tau_2 \qquad \Phi'_f \leq \Phi'}{\Phi'; \Gamma \vdash e[x \mapsto v] : \tau_2}$$

By assumption, we have $\Phi_2; \Gamma \vdash v : \tau_1 \rightsquigarrow \cdot$. By value typing and $\tau_1 \leq \tau_1'$ we have $\Phi'; \Gamma \vdash v : \tau_1' \rightsquigarrow \cdot$. Finally by substitution we have $\Phi'; \Gamma \vdash e[x \mapsto v] : \tau_2 \rightsquigarrow \cdot$.

For part 2., we need to prove $\Phi', \cdot; H \vdash (\beta', \sigma')$ where $\sigma' = \sigma \cup (\mathsf{z}, \nu)$ and $n'' = n'$, hence:

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in (\sigma \cup (\mathsf{z}, \nu)) \Rightarrow \mathsf{f} \in \Phi^{\alpha} \cup \{\mathsf{z}\} \\ \mathsf{f} \in \varepsilon_f \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})\end{array}}{\Phi', \cdot; H \vdash (\beta', \sigma')}$$

The first premise is true by assumption and the fact that $\{\mathsf{z}\} \subseteq \{\mathsf{z}\}$. The second premise is true by assumption.

For part 3., we need to prove $\mathit{traceOK}(\sigma \cup (\mathsf{z}, \nu))$; we have $\mathit{traceOK}(\sigma)$ by assumption, hence need to prove that $n' \in \nu$. Since by assumption we have that $\mathsf{f} \in \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$ and $\{\mathsf{z}\} \subseteq \varepsilon_f$, we have $n' \in \nu$.

case [CONG] **:**

case $\mathbb{E}\ e$ **:**

$\langle n; \Sigma; H; e_1\ e_2\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e_1'\ e_2\rangle$ follows from $\langle n; \Sigma; H; e_1\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e_1'\rangle$.
Since $e_1 \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot$ by assumption, by Lemma B.0.10 we have $\Phi_1, \mathcal{R}_1; H \vdash \Sigma$ hence we can apply induction:

(i) $\Phi_1'; \Gamma' \vdash e_1' : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \mathcal{R}_1'$ and
(ii) $n; \Gamma' \vdash H'$
(iii) $\Phi_1', \mathcal{R}_1'; H' \vdash \Sigma'$
(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi_1' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_1'; \Phi_1^\omega]$ where $\varepsilon_1' \cup \varepsilon_0 \subseteq \varepsilon_1$ and $\Phi_1^\omega \equiv \Phi_2^\varepsilon \cup \varepsilon_f \cup \Phi_3^\omega$.
Let $\begin{array}{lcl} \Phi_2' & \equiv & [\Phi_1^\alpha \cup \varepsilon_1' \cup \varepsilon_0; \Phi_2^\varepsilon; \varepsilon_f \cup \Phi_3^\omega] \\ \Phi_3' & \equiv & [\Phi_1^\alpha \cup \varepsilon_1' \cup \varepsilon_0 \cup \Phi_2^\varepsilon; \varepsilon_f; \Phi_3^\omega] \end{array}$
Thus $\Phi_3'^\varepsilon = \varepsilon_f$, $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'$, $\Phi_3'^\alpha \subseteq \Phi_f^\alpha$ and $\Phi_3'^\omega \subseteq \Phi_f^\omega$ (since $\Phi_3'^\alpha \cup \varepsilon_0 \subseteq \Phi_3^\alpha$ and $\Phi_3'^\omega = \Phi_3^\omega$). We have $\Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_1' \cup \Phi_2^\varepsilon \cup \varepsilon_f; \Phi_3^\omega]$. The choice of Φ' is acceptable since $\Phi'^\alpha = \Phi^\alpha \cup \varepsilon_0$, $(\varepsilon_1' \cup \varepsilon_f \cup \varepsilon_2) \cup \varepsilon_0 \subseteq (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f)$ i.e., $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^\varepsilon$ and $\Phi'^\omega = \Phi^\omega$ as required).
To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and apply (TAPP):

$$\text{(TAPP)}\ \dfrac{\begin{array}{c} \Phi_1'; \Gamma' \vdash e_1' : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \mathcal{R}_1' \\ \text{(TSUB)}\ \dfrac{\Phi_2; \Gamma' \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2 \quad \tau_1 \leq \tau_1 \quad \dfrac{\begin{array}{c} \Phi_1^\alpha \cup \varepsilon_1' \cup \varepsilon_0 \subseteq \Phi_1^\alpha \cup \Phi_1^\varepsilon \\ \Phi_2^\varepsilon \subseteq \Phi_2^\varepsilon \\ \varepsilon_f \cup \Phi_3^\omega \subseteq \varepsilon_f \cup \Phi_3^\omega \end{array}}{\Phi_2 \leq \Phi_2'}}{\Phi_2'; \Gamma' \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2} \\ \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi' \\ \Phi_3'^\varepsilon = \Phi_f^\varepsilon \qquad \Phi_3'^\alpha \subseteq \Phi_f^\alpha \qquad \Phi_3'^\omega \subseteq \Phi_f^\omega \\ e_1' \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot \end{array}}{\Phi'; \Gamma' \vdash e_1'\ e_2 : \tau_2 \rightsquigarrow \mathcal{R}_1' \bowtie \mathcal{R}_2}$$

Note that $\Phi_2; \Gamma' \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2$ follows from $\Phi_2; \Gamma \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2$ by weakening (Lemma B.0.1). The last premise holds vacuously as $\mathcal{R}_2 \equiv \cdot$ by assumption.
To prove part 2., we must show that $\Phi', \mathcal{R}'; H' \vdash \Sigma'$. The proof is similar to the (TASSIGN)-[CONG] proof, case $\mathbb{E} := e$ but substituting ε_f for ε_r.
Part 3. follows directly from (iv).

case $v\ \mathbb{E}$ **:**

$\langle n; \Sigma; H; v\ e_2\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; v\ e_2'\rangle$ follows from $\langle n; \Sigma; H; e_2\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e_2'\rangle$.
For convenience, we make $\Phi_1^\varepsilon \equiv \emptyset$; if $\Phi_1^\varepsilon \not\equiv \emptyset$, we can always construct a typing derivation of v that uses value typing to make $\Phi_1^\varepsilon \equiv \emptyset$. Note that $\Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi$ would still hold since Lemma B.0.7 allows us to decrease Φ_2^α to satisfy $\Phi_2^\alpha = \Phi_1^\alpha \cup \Phi_1^\varepsilon$; similarly, since $\Phi_3^\alpha = \Phi_1^\alpha \cup \Phi_1^\varepsilon \cup \Phi_2^\varepsilon$ we know that $\Phi_3^\alpha \subseteq \Phi_f^\alpha$ would still hold if Φ_3^α was smaller as a result of shrinking Φ_1^ε to be $\emptyset$.
Since $e_1 \equiv v$, by inversion $\mathcal{R}_1 \equiv \cdot$ and by Lemma B.0.10 (which we can apply since $\Phi_1^\varepsilon \equiv \emptyset$), we have $\Phi_2, \mathcal{R}_2; H \vdash \Sigma$; hence by induction:

(i) $\Phi_2'; \Gamma' \vdash e_2' : \tau_1 \rightsquigarrow \mathcal{R}_2'$
(ii) $n; \Gamma' \vdash H'$
(iii) $\Phi_2', \mathcal{R}_2'; H' \vdash \Sigma'$
(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi_2' \equiv [\Phi_2^\alpha \cup \varepsilon_0; \varepsilon_2'; \Phi_2^\omega]$ where $(\varepsilon_2' \cup \varepsilon_0) \subseteq \Phi_2^\varepsilon$; note $\Phi_2^\alpha \equiv \Phi_1^\alpha$ (since $\Phi_1^\varepsilon \equiv \emptyset$) and $\Phi_2^\omega \equiv \varepsilon_3 \cup \Phi_3^\omega$.
Let $\begin{array}{lcl} \Phi_1' & \equiv & [\Phi_1^\alpha \cup \varepsilon_0; \emptyset; \varepsilon_2' \cup \varepsilon_f \cup \Phi_3^\omega] \\ \Phi_3' & \equiv & [\Phi_1^\alpha \cup \varepsilon_0 \cup \varepsilon_2'; \varepsilon_f; \Phi_3^\omega] \end{array}$
Thus $\Phi_3'^\varepsilon = \varepsilon_f$, $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'$, $\Phi_3'^\alpha \subseteq \Phi_f^\alpha$ and $\Phi_3'^\omega \subseteq \Phi_f^\omega$ (since $\Phi_3'^\alpha \cup \varepsilon_0 \subseteq \Phi_3^\alpha$ and $\Phi_3'^\omega = \Phi_3^\omega$). We have $\Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_2' \cup \varepsilon_f; \Phi_3^\omega]$ and $(\varepsilon_2' \cup \varepsilon_f) \cup \varepsilon_0 \subseteq (\Phi_2^\varepsilon \cup \varepsilon_f)$. The choice of Φ' is acceptable since $\Phi'^\alpha = \Phi^\alpha \cup \varepsilon_0$, $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^\varepsilon$ and $\Phi'^\omega = \Phi^\omega$ as required).
To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and we can apply [TApp]:

$$\text{(TAPP)}\ \dfrac{\begin{array}{c} \Phi_1'; \Gamma' \vdash v : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \cdot \qquad \Phi_2'; \Gamma' \vdash e_2' : \tau_1 \rightsquigarrow \mathcal{R}_2' \\ \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi' \\ \Phi_3'^\varepsilon = \Phi_f^\varepsilon \qquad \Phi_3'^\alpha \subseteq \Phi_f^\alpha \qquad \Phi_3'^\omega \subseteq \Phi_f^\omega \\ v \not\equiv v' \Rightarrow \mathcal{R}_2' = \cdot \end{array}}{\Phi'; \Gamma' \vdash v\ e_2' : \tau_2 \rightsquigarrow \cdot \bowtie \mathcal{R}_2'}$$

(Note that $\cdot \bowtie \mathcal{R}'_2 = \mathcal{R}'_2$.)
The first premise follows by value typing and weakening; the second by (i); the third—sixth by choice of Φ', Φ'_1, Φ'_2, Φ'_3; the last holds vacuously since $\mathcal{R}_1 \equiv \cdot$ by assumption.
To prove part 2., we must show that $\Phi', \mathcal{R}'; H' \vdash \Sigma'$. The proof is similar to the (TAssign)-[cong] proof, case $r := \mathbb{E}$ but substituting ε_f for ε_r.
Part 3. follows directly from (iv).

case (TSub) **:**

We have

$$(\textsc{TSub})\frac{\begin{array}{c}\Phi''; \Gamma \vdash e : \tau'' \rightsquigarrow \mathcal{R} \\ \Phi'' \equiv [\alpha; \varepsilon''; \omega] \qquad \Phi \equiv [\alpha; \varepsilon; \omega] \\ \tau'' \leq \tau \qquad\qquad \varepsilon'' \subseteq \varepsilon\end{array}}{\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}}$$

since by flow effect weakening (Lemma B.0.7) we know that α and ω are unchanged in the use of (TSub).

We have $\langle n; \Sigma; H; e\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e'\rangle$. To apply induction we must show that $n; \Gamma \vdash H$, which we have by assumption, $\Phi''; \Gamma \vdash e : \tau'' \rightsquigarrow \mathcal{R}$, which we also have by assumption, and $\Phi'', \mathcal{R}; H \vdash \Sigma$, which follows easily since $\varepsilon'' \subseteq \varepsilon$.

Hence we have:

(i) $\Phi'''; \Gamma' \vdash e' : \tau'' \rightsquigarrow \mathcal{R}'$ and
(ii) $n; \Gamma' \vdash H'$
(iii) $\Phi''', \mathcal{R}'; H' \vdash \Sigma'$
(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$, Φ''' such that $\Phi'''^\alpha = \alpha \cup \varepsilon_0$, $\Phi'''^\varepsilon \cup \varepsilon_0 \subseteq \varepsilon''$ Let $\Phi' \equiv \Phi'''$, and thus $\Phi'^\alpha = \alpha \cup \varepsilon_0$, $\Phi'^\varepsilon \cup \varepsilon_0 \subseteq \varepsilon$ since $\varepsilon'' \subseteq \varepsilon$, and $\Phi'^\omega = \omega$ as required. All results follow by induction.

□

Lemma B.0.16 (Progress)**.** *If $n \vdash H, e : \tau$ (such that $\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}$ and $n; \Gamma \vdash H$) and for all Σ such that $\Phi, \mathcal{R}; H \vdash \Sigma$ and $\mathit{traceOK}(\Sigma)$, then either e is a value, or there exist n', H', Σ', e' such that $\langle n; \Sigma; H;\ e\rangle \longrightarrow_\eta \langle n'; \Sigma'; H';\ e'\rangle$.*

Proof. Induction on the typing derivation $n \vdash H, e : \tau$; consider each possible rule for the conclusion of this judgment:

case (TInt-TGvar-TLoc) **:**

These are all values.

case (TVar) **:**

Can't occur, since local values are substituted for.

case (TRef) **:**

We must have that

$$(\textsc{TRef})\frac{\Phi; \Gamma \vdash e' : \tau \rightsquigarrow \mathcal{R}}{\Phi; \Gamma \vdash \mathsf{ref}\ e' : \mathit{ref}^{\,\varepsilon}\ \tau \rightsquigarrow \mathcal{R}}$$

There are two possible reductions, depending on the shape of e:

case $e' \equiv v$ **:**
By inversion on $\Phi; \Gamma \vdash v : \tau \rightsquigarrow \cdot$ we know that $\mathcal{R} \equiv \cdot$ hence by inversion on $\Phi, \mathcal{R}; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$. We have that $\langle n; (\beta, \sigma); H; \mathsf{ref}\ v\rangle \longrightarrow n; (\beta, \sigma); H'; r$ where $r \notin \mathrm{dom}(H)$ and $H' = H, r \mapsto (\cdot, v, \emptyset)$ by (ref).

case $e' \not\equiv v$ **:**
By induction, $\langle n; \Sigma; H; e'\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e''\rangle$ and thus $\langle n; \Sigma; H; (\mathsf{ref}\ \_)[e']\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; (\mathsf{ref}\ \_)[e'']\rangle$ by [cong].

case (TDeref) **:**

We know that

$$(\textsc{TDeref})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R} \\ \Phi_2^\varepsilon = \varepsilon_r \qquad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash\ !\,e : \tau \rightsquigarrow \mathcal{R}}$$

Consider the shape of e:

case $e' \equiv v$ **:**

Since v is a value of type $\mathit{ref}^{\,\varepsilon_r}\ \tau$, we must have $v \equiv \mathsf{z}$ or $v \equiv r$.

case $e' \equiv \mathsf{z}$ **:**

We have

$$(\text{TDeref})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \cdot \\ \Phi_2^{\varepsilon} = \varepsilon_r \qquad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash\ !\,\mathsf{z} : \tau \leadsto \cdot}$$

where by subtyping derivations (Lemma B.0.6) we have

$$(\text{TSub})\frac{(\text{TGVar})\dfrac{\Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon_r'}\ \tau'}{\Phi_\emptyset; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r'}\ \tau' \leadsto \cdot} \qquad \dfrac{\tau' \leq \tau \qquad \tau \leq \tau' \qquad \varepsilon_r' \subseteq \varepsilon_r}{\mathit{ref}^{\,\varepsilon_r'}\ \tau' \leq \mathit{ref}^{\,\varepsilon_r}\ \tau} \qquad \Phi_\emptyset \leq \Phi_1}{\Phi_1; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \cdot}$$

By inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$. By $n; \Gamma \vdash H$ we have $\mathsf{z} \in \mathrm{dom}(H)$ (and thus $H \equiv H'', \mathsf{z} \mapsto (\mathit{ref}^{\,\varepsilon_r'}\ \tau', v, \nu)))$ since $\Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon_r'}\ \tau'$. Therefore, we can reduce via [GVAR-DEREF]:

$$\langle n; (\beta, \sigma); (H'', \mathsf{z} \mapsto (\mathit{ref}^{\,\varepsilon_r'}\ \tau', v, \nu));\ !\mathsf{z}\rangle \ \longrightarrow_{\{\mathsf{z}\}}\ \langle n; (\beta, \sigma \cup (\mathsf{z}, \nu)); (H'', \mathsf{z} \mapsto (\mathit{ref}^{\,\varepsilon_r'}\ \tau', v, \nu)); v\rangle$$

case $e' \equiv r$ **:**

Similar to the $e' \equiv \mathsf{z}$ case above, but reduce using [DEREF].

case $e' \not\equiv v$ **:**

Let $\mathbb{E} \equiv\ !\ \_$ so that $e \equiv \mathbb{E}[e']$. To apply induction, we have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ by Lemma B.0.9. Thus we get $\langle n; \Sigma; H; e'\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e''\rangle$, hence we have that $\langle n; \Sigma; H; \mathbb{E}[e']\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; \mathbb{E}[e'']\rangle$ by [CONG].

case (TAssign) **:**

$$(\text{TAssign})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \mathcal{R}_1 \qquad \Phi_2; \Gamma \vdash e_2 : \tau \leadsto \mathcal{R}_2 \\ \Phi_3^{\varepsilon} = \varepsilon_r \qquad\qquad \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ e_1 \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot\end{array}}{\Phi; \Gamma \vdash e_1 := e_2 : \tau \leadsto \mathcal{R}_1 \bowtie \mathcal{R}_2}$$

Depending on the shape of e, we have:

case $e_1 \equiv v_1, e_2 \equiv v_2$ **:**

Since v_1 is a value of type $\mathit{ref}^{\,\varepsilon_r}\ \tau$, we must have $v_1 \equiv \mathsf{z}$ or $v_1 \equiv r$. The results follow by reasoning quite similar to [TDeref] above.

case $e_1 \equiv v_1, e_2 \not\equiv v$ **:**

Let $\mathbb{E} \equiv v_1 := \_$ so that $e \equiv \mathbb{E}[e_2]$. Since e_1 is a value, $\mathcal{R}_1 \equiv \cdot$ hence we have $\Phi_2, \mathcal{R}; H \vdash \Sigma$ by Lemma B.0.10 and we can apply induction. We have $\langle n; \Sigma; H; e_2\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e_2'\rangle$, and thus $\langle n; \Sigma; H; \mathbb{E}[e_2]\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; \mathbb{E}[e_2']\rangle$ by [CONG].

case $e_1 \not\equiv v$ **:**

Since e_1 is a not value, $\mathcal{R}_2 \equiv \cdot$ hence we have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ by Lemma B.0.10 and we can apply induction. The rest follows by an argument similar to the above case.

case (TUpdate) **:**

By inversion on $\Phi; \Gamma \vdash \mathsf{update}^{\alpha,\omega} : \mathit{int} \leadsto \mathcal{R}$ we have that $\mathcal{R} \equiv \cdot$, hence by inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$. If $\mathit{updateOK}(\mathit{upd}, H, (\alpha, \omega), \mathit{dir}) = \mathbf{tt}$, then $\mathsf{update}^{\alpha,\omega}$ reduces via [UPDATE], otherwise $\mathsf{update}^{\alpha,\omega}$ reduces via [NO-UPDATE].

case (TIf) **:**

$$(\text{TIf})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \mathit{int} \leadsto \mathcal{R} \\ \Phi_2; \Gamma \vdash e_2 : \tau \leadsto \cdot \qquad \Phi_2; \Gamma \vdash e_3 : \tau \leadsto \cdot \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \leadsto \mathcal{R}}$$

Depending on the shape of e, we have:

case $e_1 \equiv v$ **:**

This implies $\mathcal{R} \equiv \cdot$ so by inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$. Since the type of v is *int*, we know v must be an integer n. Thus we can reduce via either [IF-T] or [IF-F].

case $e_1 \not\equiv v$ **:**

Let $\mathbb{E} \equiv \mathsf{if0}\ \_\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ so that $e \equiv \mathbb{E}[e_1]$. To apply induction, we have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ by Lemma B.0.9. We have $\langle n; \Sigma; H; e_1\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e_1'\rangle$ and thus $\langle n; \Sigma; H; \mathbb{E}[e_1]\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; \mathbb{E}[e_1']\rangle$ by [CONG].

case (TTRANSACT) **:**

We know that:

$$(\text{TTRANSACT})\frac{\Phi'; \Gamma \vdash e : \tau \rightsquigarrow \cdot \qquad \Phi^\alpha \subseteq \Phi'^\alpha \quad \Phi^\omega \subseteq \Phi'^\omega}{\Phi; \Gamma \vdash \mathsf{tx}\ e : \tau \rightsquigarrow \cdot}$$

By inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$. Thus we can reduce by [TX-START].

case (TINTRANS) **:**

We know that:

$$(\text{TINTRANS})\frac{\Phi'; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R} \qquad \Phi^\alpha \subseteq \Phi'^\alpha \quad \Phi^\omega \subseteq \Phi'^\omega}{\Phi; \Gamma \vdash \mathsf{intx}\ e : \tau \rightsquigarrow \Phi', \mathcal{R}}$$

Consider the shape of e:

case $e \equiv v$ **:**

Thus

$$(\text{TINTRANS})\frac{\Phi'; \Gamma \vdash v : \tau \rightsquigarrow \cdot \qquad \Phi^\alpha \subseteq \Phi'^\alpha \quad \Phi^\omega \subseteq \Phi'^\omega}{\Phi; \Gamma \vdash \mathsf{intx}\ v : \tau \rightsquigarrow \Phi', \cdot}$$

We have $\Phi, \Phi', \cdot; H \vdash \Sigma$ by assumption:

$$(\text{TC2})\frac{\begin{array}{c}\Phi', \cdot; H \vdash \Sigma \\ \Phi \equiv [\alpha; \varepsilon; \omega] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \Rightarrow n' \in ver(H, \mathsf{f})\end{array}}{\Phi, \Phi', \cdot; H \vdash ((\beta', \sigma'), (\beta, \sigma))}$$

By inversion we have $\Sigma \equiv ((\beta', \sigma'), (\beta, \sigma))$; by assumption we have $traceOK(n'', \sigma'')$ so we can reduce via [TX-END].

case $e \not\equiv v$ **:**

We have $\Phi, \Phi', \mathcal{R}; H \vdash \Sigma$ by assumption. By induction we have $\langle n; \Sigma'; H; e'\rangle \longrightarrow_\eta \langle n'; \Sigma''; H'; e''\rangle$, hence by [TX-CONG-2]:

$$\langle n; \Sigma'; H; \mathsf{intx}\ e'\rangle \longrightarrow_\emptyset \langle n'; \Sigma''; H'; \mathsf{intx}\ e''\rangle$$

case (TLET) **:**

We know that:

$$(\text{TLET})\frac{\Phi_1; \Gamma \vdash e_1 : \tau_1 \rightsquigarrow \mathcal{R} \qquad \Phi_2; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow \cdot \qquad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}{\Phi; \Gamma \vdash \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2 \rightsquigarrow \mathcal{R}}$$

Consider the shape of e:

case $e_1 \equiv v$ **:**

Thus $\Phi_1; \Gamma \vdash v : \tau \rightsquigarrow \cdot$ and by inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$.

We can reduce via [LET].

case $e_1 \not\equiv v$ **:**

Let $\mathbb{E} \equiv \mathsf{let}\ x : \tau_1 = \_\ \mathsf{in}\ e_2$ so that $e \equiv \mathbb{E}[e_1]$. To apply induction, we have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ by Lemma B.0.9. We have $\langle n; \Sigma; H; e_1\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e_1'\rangle$ and so $\langle n; \Sigma; H; \mathbb{E}[e_1]\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; \mathbb{E}[e_1']\rangle$ by [CONG].

case (TApp) **:**

$$(\textsc{TApp})\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \leadsto \mathcal{R}_1 \qquad \Phi_2; \Gamma \vdash e_2 : \tau_1 \leadsto \mathcal{R}_2 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ e_1 \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot \end{array}}{\Phi; \Gamma \vdash e_1\ e_2 : \tau_2 \leadsto \mathcal{R}_1 \bowtie \mathcal{R}_2}$$

Depending on the shape of e, we have:

case $e_1 \equiv v_1, e_2 \equiv v_2$ **:**

Since v_1 is a value of type $\tau_1 \longrightarrow^{\Phi} \tau_2$, we must have $v_1 \equiv \mathsf{z}$, hence

$$(\textsc{TApp})\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash \mathsf{z} : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \leadsto \cdot \qquad \Phi_2; \Gamma \vdash v : \tau_1 \leadsto \cdot \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ \mathsf{z} \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot \end{array}}{\Phi; \Gamma \vdash \mathsf{z}\ v : \tau_2 \leadsto \cdot}$$

where by subtyping derivations (Lemma B.0.6) we have

$$(\textsc{TSub})\dfrac{(\textsc{TGVar})\dfrac{\Gamma(\mathsf{z}) = \tau_1' \longrightarrow^{\Phi_f'} \tau_2'}{\Phi_\emptyset; \Gamma \vdash \mathsf{z} : \tau_1' \longrightarrow^{\Phi_f'} \tau_2' \leadsto \cdot} \qquad \dfrac{\tau_1 \leq \tau_1' \qquad \tau_2' \leq \tau_2 \qquad \Phi_f' \leq_f \Phi_f}{\tau_1' \longrightarrow^{\Phi_f'} \tau_2' \leq \tau_1 \longrightarrow^{\Phi_f} \tau_2} \qquad \Phi_\emptyset \leq \Phi_1}{\Phi_1; \Gamma \vdash \mathsf{z} : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \leadsto \cdot}$$

By inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (\beta, \sigma)$. By $n; \Gamma \vdash H$ we have $\mathsf{z} \in \mathrm{dom}(H)$ and $H \equiv (H'', \mathsf{z} \mapsto (\tau_1' \longrightarrow^{\Phi_f'} \tau_2', \lambda(x).e'', \nu))$ since $\Gamma(\mathsf{z}) = \tau_1' \longrightarrow^{\Phi_f'} \tau_2'$. By [call], we have:

$$\langle n; (\beta, \sigma); (H'', \mathsf{z} \mapsto (\tau_1' \longrightarrow^{\Phi_f'} \tau_2', \lambda(x).e'', \nu)); \mathsf{z}\ v\rangle \ \longrightarrow_{\{\mathsf{z}\}}$$

$$\langle n; (\beta, \sigma \cup (\mathsf{z}, \nu)); (H'', \mathsf{z} \mapsto (\tau_1' \longrightarrow^{\Phi_f'} \tau_2', \lambda(x).e'', \nu)); e''[x \mapsto v]\rangle$$

case $e_1 \not\equiv v$ **:**

Let $\mathbb{E} \equiv \_\ e_2$ so that $e \equiv \mathbb{E}[e_1]$. Since e_1 is a not value, $\mathcal{R}_2 \equiv \cdot$ hence we have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ by Lemma B.0.10 and we can apply induction and we have: $\langle n; \Sigma; H; e_1\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e_1'\rangle$, and thus $\langle n; \Sigma; H; \mathbb{E}[e_1]\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; \mathbb{E}[e_1']\rangle$ by [cong].

case $e_1 \equiv v_1, e_2 \not\equiv v$ **:**

Let $\mathbb{E} \equiv v_1\ \_$ so that $e \equiv \mathbb{E}[e_2]$. Since e_1 is a value, $\mathcal{R}_1 \equiv \cdot$ hence we have $\Phi_2, \mathcal{R}; H \vdash \Sigma$ by Lemma B.0.10 and we can apply induction. The rest follows similarly to the above case.

case (TSub) **:**

We know that:

$$(\textsc{TSub})\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash e : \tau' \leadsto \mathcal{R} \qquad \tau' \leq \tau \\ \Phi_1 \equiv [\alpha; \varepsilon_1; \omega] \qquad \Phi \equiv [\alpha; \varepsilon; \omega] \qquad \varepsilon_1 \subseteq \varepsilon \end{array}}{\Phi; \Gamma \vdash e : \tau \leadsto \mathcal{R}}$$

If e is a value v we are done. Otherwise, since $\Phi_1, \mathcal{R}; H \vdash \Sigma$ follows from $\Phi, \mathcal{R}; H \vdash \Sigma$ (by $\Phi_1^{\varepsilon} \subseteq \Phi^{\varepsilon}$ and $\Phi_1^{\alpha} = \Phi^{\alpha}$); we have $\langle n; \Sigma; H; e\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e'\rangle$ by induction.

□

Lemma B.0.17 (Substitution).
If $\Phi; \Gamma, x : \tau' \vdash e : \tau$ *and* $\Phi; \Gamma \vdash v : \tau'$ *then* $\Phi; \Gamma \vdash e[x \mapsto v] : \tau$.

Proof. Induction on the typing derivation of $\Phi; \Gamma \vdash e : \tau$.

case (TInt) **:**

Since $e \equiv n$ and $n[x \mapsto v] \equiv n$, the result follows by (TInt).

case (TVar) **:**

e is a variable y. We have two cases:

case $y = x$ **:**

We have $\tau = \tau'$ and $y[x \mapsto v] \equiv v$, hence we need to prove that $\Phi; \Gamma \vdash v : \tau$ which is true by assumption.

case $y \neq x$ **:**

We have $y[x \mapsto v] \equiv y$ and need to prove that $\Phi; \Gamma \vdash y : \tau$. By assumption, $\Phi; \Gamma, x : \tau' \vdash y : \tau$, and thus $(\Gamma, x : \tau')(y) = \tau$; but since $x \neq y$ this implies $\Gamma(y) = \tau$ and we have to prove $\Phi; \Gamma \vdash y : \tau$ which follows by (TVar).

case (TGvar),(TLoc), (TUpdate) **:**

Similar to (TInt).

case (TRef) **:**

We know that $\Phi; \Gamma, x : \tau' \vdash \mathsf{ref}\ e : \mathit{ref}^{\varepsilon}\ \tau$ and $\Phi; \Gamma \vdash v : \tau'$, and need to prove that $\Phi; \Gamma \vdash (\mathsf{ref}\ e)[x \mapsto v] : \mathit{ref}^{\varepsilon}\ \tau$. By inversion on $\Phi; \Gamma, x : \tau' \vdash \mathsf{ref}\ e : \mathit{ref}^{\varepsilon}\ \tau$ we have $\Phi; \Gamma, x : \tau' \vdash e : \tau$; applying induction to this, we have $\Phi; \Gamma \vdash e[x \mapsto v] : \tau$. We can now apply [TRef]:

$$(\text{TRef})\frac{\Phi; \Gamma \vdash e[x \mapsto v] : \tau}{\Phi; \Gamma \vdash \mathsf{ref}\ (e[x \mapsto v]) : \mathit{ref}^{\varepsilon}\ \tau}$$

The desired result follows since $\mathsf{ref}\ (e[x \mapsto v]) \equiv (\mathsf{ref}\ e)[x \mapsto v]$.

case (TDeref) **:**

We know that $\Phi; \Gamma, x : \tau' \vdash !\,e : \tau$ and $\Phi; \Gamma \vdash v : \tau'$ and need to prove that $\Phi; \Gamma \vdash (!\,e)[x \mapsto v] : \tau$. By inversion on $\Phi; \Gamma, x : \tau' \vdash !\,e : \tau$ we have $\Phi_1; \Gamma, x : \tau' \vdash e : \mathit{ref}^{\varepsilon_r}\ \tau$ and Φ_2 such that $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$ and $\Phi \equiv \Phi_1 \rhd \Phi_2$. By value typing we have $\Phi_1; \Gamma \vdash v : \tau'$. We can then apply induction, yielding $\Phi_1; \Gamma \vdash e[x \mapsto v] : \mathit{ref}^{\varepsilon_r}\ \tau$. Finally, we apply (TDeref)

$$(\text{TDeref})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e[x \mapsto v] : \mathit{ref}^{\varepsilon_r}\ \tau \\ \Phi_2^{\varepsilon} = \varepsilon_r \qquad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash !\,e[x \mapsto v] : \tau}$$

Note that the second premise holds by inversion on $\Phi; \Gamma, x : \tau' \vdash !\,e : \tau$. The desired result follows since $!\,(e[x \mapsto v]) \equiv (!\,e)[x \mapsto v]$.

case (TSub) **:**

We know that $\Phi; \Gamma, x : \tau' \vdash e : \tau$ and $\Phi; \Gamma \vdash v : \tau'$ and need to prove that $\Phi; \Gamma \vdash e[x \mapsto v] : \tau$. By inversion on $\Phi; \Gamma, x : \tau' \vdash e : \tau$ we have $\Phi'; \Gamma, x : \tau' \vdash e : \tau'$. By value typing we have $\Phi'; \Gamma, x : \tau' \vdash v : \tau'$. We can then apply induction, yielding $\Phi'; \Gamma \vdash e[x \mapsto v] : \tau'$. Finally, we apply (TSub)

$$(\text{TSub})\frac{\Phi'; \Gamma \vdash e[x \mapsto v] : \tau' \qquad \tau' \leq \tau \qquad \Phi' \leq \Phi}{\Phi; \Gamma \vdash e[x \mapsto v] : \tau}$$

and get the desired result.

case (TTransact),(TIntrans) **:**

Similar to (TSub).

case (TApp) **:**

We know that

$$(\text{TApp})\frac{\begin{array}{c}\Phi_1; \Gamma, x : \tau' \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \qquad \Phi_2; \Gamma, x : \tau' \vdash e_2 : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad\qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad\qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega}\end{array}}{\Phi; \Gamma, x : \tau' \vdash e_1\ e_2 : \tau_2}$$

where $\Phi; \Gamma \vdash v : \tau'$, and need to prove that $\Phi; \Gamma \vdash (e_1\ e_2)[x \mapsto v] : \tau_2$. Call the first two premises above (1) and (2), and note that we have (3) $\Phi; \Gamma \vdash v : \tau' \Rightarrow \Phi_1; \Gamma \vdash v : \tau'$ and (4) $\Phi; \Gamma \vdash v : \tau' \Rightarrow \Phi_2; \Gamma \vdash v : \tau'$ by the value typing lemma. By (1), (3) and induction we have $\Phi_1; \Gamma \vdash e_1[x \mapsto v] : \tau_1 \longrightarrow^{\Phi_f} \tau_2$. Similarly, by (2), (4) and induction we have $\Phi_2; \Gamma \vdash e_2[x \mapsto v] : \tau_1$. We can now apply (TApp):

$$\text{(TApp)}\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1[x \mapsto v] : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \qquad \Phi_2; \Gamma \vdash e_2[x \mapsto v] : \tau_1 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega}\end{array}}{\Phi; \Gamma \vdash e_1[x \mapsto v]\ e_2[x \mapsto v] : \tau_2}$$

Since $e_1[x \mapsto v]\ e_2[x \mapsto v] \equiv (e_1\ e_2)[x \mapsto v]$ we get the desired result.

case (TAssign-TIf-TLet) **:**

Similar to (TApp).

□

Theorem B.0.18 (Single-step Soundness)**.** *If $\Phi; \Gamma \vdash e : \tau$ where $[\![\Phi; \Gamma \vdash e : \tau]\!] = \mathcal{R}$; and $n; \Gamma \vdash H$; and $\Phi, \mathcal{R}; H \vdash \Sigma$; and traceOK$(\Sigma)$, then either e is a value, or there exist n', H', Σ', Φ', e', and η such that $\langle n; \Sigma; H; e\rangle \longrightarrow_{\eta} \langle n'; \Sigma'; H'; e'\rangle$ and $\Phi'; \Gamma' \vdash e' : \tau$ where $[\![\Phi'; \Gamma' \vdash e' : \tau]\!] = \mathcal{R}'$; and n'; $\Gamma' \vdash H'$; and $\Phi', \mathcal{R}'; H' \vdash \Sigma'$; and traceOK$(\Sigma')$ for some $\Phi', \Gamma', \mathcal{R}'$.*

Proof. From progress (Lemma D.0.37), we know that if $n \vdash H, e : \tau$ then either e is a value, or there exist $n', H', \Sigma', \Phi', e', \eta$ such that $\langle n; \Sigma; H; e\rangle \longrightarrow_{\eta} \langle n'; \Sigma'; H'; e'\rangle$. If e is a value we are done. If e is not a value, then there are two cases. If $\eta = \mu$ then the result follows from update preservation (Lemma B.0.13). If $\eta = \varepsilon_0$, then the result follows from preservation (Lemma D.0.36). □

Appendix C
Relaxed Updates Proofs

Lemma C.0.19 (Weakening)**.** *If $\Phi;\Gamma \vdash e : \tau$ and $\Gamma' \supseteq \Gamma$ then $\Phi;\Gamma' \vdash e : \tau$.*

Proof. By induction on the typing derivation of $\Phi;\Gamma \vdash e : \tau$. □

Lemma C.0.20 (Subtyping reflexivity)**.** *$\tau \leq \tau$ for all τ.*

Proof. Straightforward, from the definition of subtyping in Figure 5.2. □

Lemma C.0.21 (Subtyping transitivity)**.** *For all τ, τ', τ'', if $\tau \leq \tau'$ and $\tau' \leq \tau''$ then $\tau \leq \tau''$.*

Proof. By simultaneous induction on $\tau \leq \tau'$ and $\tau' \leq \tau''$, similar to Lemma B.0.4 □

Lemma C.0.22 (Value typing)**.** *If $\Phi;\Gamma \vdash v : \tau$ then $\Phi';\Gamma \vdash v : \tau$ for all Φ'.*

Proof. By induction on the typing derivation of $\Phi;\Gamma \vdash v : \tau$.

□

Lemma C.0.23 (Subtyping Derivations)**.** *If $\Phi;\Gamma \vdash e : \tau$ then we can construct a proof derivation of this judgment that ends in one use of* (TSub) *whose premise uses a rule other than* (TSub)*.*

Proof. By induction on $\Phi;\Gamma \vdash e : \tau$.

case (TSub) **:**

We have

$$\text{TSub}\dfrac{\Phi';\Gamma \vdash e : \tau' \qquad \tau' \leq \tau \qquad \text{SCtxt}\dfrac{\begin{array}{c}\Phi'^{\varepsilon} \subseteq \Phi^{\varepsilon} \qquad \Phi^{\alpha} \subseteq \Phi'^{\alpha} \qquad \Phi^{\omega} \subseteq \Phi'^{\omega} \\ \Phi'^{\delta_i} \subseteq \Phi^{\delta_i} \qquad \Phi^{\delta_o} \subseteq \Phi'^{\delta_o}\end{array}}{\Phi' \leq \Phi}}{\Phi;\Gamma \vdash e : \tau}$$

By induction, we have

$$\text{TSub}\dfrac{\Phi'';\Gamma \vdash e : \tau'' \qquad \tau'' \leq \tau' \qquad \text{SCtxt}\dfrac{\begin{array}{c}\Phi''^{\varepsilon} \subseteq \Phi'^{\varepsilon} \qquad \Phi'^{\alpha} \subseteq \Phi''^{\alpha} \qquad \Phi'^{\omega} \subseteq \Phi''^{\omega} \\ \Phi''^{\delta_i} \subseteq \Phi'^{\delta_i} \qquad \Phi'^{\delta_o} \subseteq \Phi''^{\delta_o}\end{array}}{\Phi'' \leq \Phi'}}{\Phi';\Gamma \vdash e : \tau'}$$

where the derivation $\Phi'';\Gamma \vdash e : \tau''$ does not conclude with (TSub). By the transitivity of subtyping (Lemma C.0.21), we have $\tau'' \leq \tau$; the rest of the premises follow by transitivity of $\subseteq$, and finally we get the desired result by (TSub):

$$\text{TSub}\dfrac{\Phi'';\Gamma \vdash e : \tau'' \qquad \tau'' \leq \tau \qquad \text{SCtxt}\dfrac{\begin{array}{c}\Phi''^{\varepsilon} \subseteq \Phi^{\varepsilon} \qquad \Phi^{\alpha} \subseteq \Phi''^{\alpha} \qquad \Phi^{\omega} \subseteq \Phi''^{\omega} \\ \Phi''^{\delta_i} \subseteq \Phi^{\delta_i} \qquad \Phi^{\delta_o} \subseteq \Phi''^{\delta_o}\end{array}}{\Phi'' \leq \Phi}}{\Phi;\Gamma \vdash e : \tau}$$

case all others **:**

Since we have that the last rule in $\Phi;\Gamma \vdash e : \tau$ is not (TSub), we have the desired result by applying (TSub) (where $\tau \leq \tau$ follows from the reflexivity of subtyping, Lemma C.0.20):

$$\text{TSub}\dfrac{\Phi;\Gamma \vdash e : \tau \qquad \tau \leq \tau \qquad \Phi \leq \Phi}{\Phi;\Gamma \vdash e : \tau}$$

□

Lemma C.0.24 (Flow effect weakening)**.** *If $\Phi;\Gamma \vdash e : \tau$ where $\Phi \equiv [\alpha;\varepsilon;\omega;\delta_i;\delta_o]$, then $\Phi';\Gamma \vdash e : \tau$ where $\Phi' \equiv [\alpha';\varepsilon;\omega';\delta_i;\delta_o]$, $\Phi'^{\alpha} \subseteq \Phi^{\alpha}$, $\Phi'^{\omega} \subseteq \Phi^{\omega}$, and all uses of [*TSub*] applying $\Phi' \leq \Phi$ require $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\alpha} = \Phi^{\alpha}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$.*

Proof. By induction on $\Phi; \Gamma \vdash e : \tau$.

case (TGvar),(TInt),(TVar) **:**

Trivial.

case (TCheckin) **:**

We have

$$(\textsc{TCheckin})\frac{\alpha \cup \delta_o \subseteq \alpha'' \quad \omega \subseteq \omega''}{[\alpha; \emptyset; \omega; \emptyset; \delta_o]; \Gamma \vdash \mathsf{checkin}^{\alpha'',\omega''}\!:\!int}$$

Since $\alpha' \subseteq \alpha$, and $\omega' \subseteq \omega$,

we can apply (TCheckin):

$$(\textsc{TCheckin})\frac{\alpha' \cup \delta_o \subseteq \alpha'' \quad \omega' \subseteq \omega''}{[\alpha'; \emptyset; \omega'; \emptyset; \delta_o]; \Gamma \vdash \mathsf{checkin}^{\alpha'',\omega''}\!:\!int}$$

case (TTransact) **:**

We have

$$\text{TTransact}\frac{\begin{array}{c}\Phi''; \Gamma \vdash e : \tau \\ \Phi^{\alpha} \subseteq \Phi''^{\alpha} \quad \Phi^{\omega} \subseteq \Phi''^{\omega}\end{array}}{\Phi; \Gamma \vdash \mathsf{tx}^{(\Phi''^{\alpha} \cup \Phi''^{\delta_i}, \Phi''^{\omega} \cup \Phi''^{\varepsilon})}\ e : \tau}$$

Let $\Phi' = [\alpha'; \varepsilon; \omega'; \delta_i{}'; \delta_o]$. Since $\Phi'^{\alpha} \subseteq \Phi^{\alpha}$, and $\Phi'^{\omega} \subseteq \Phi^{\omega}$,

we can apply (TTransact):

$$\text{TTransact}\frac{\begin{array}{c}\Phi''; \Gamma \vdash e : \tau \\ \Phi'^{\alpha} \subseteq \Phi''^{\alpha} \quad \Phi'^{\omega} \subseteq \Phi''^{\omega}\end{array}}{\Phi'; \Gamma \vdash \mathsf{tx}^{(\Phi''^{\alpha} \cup \Phi''^{\delta_i}, \Phi''^{\omega} \cup \Phi''^{\varepsilon})}\ e : \tau}$$

case (TIntrans) **:**

Similar to (TTransact).

case (TSub) **:**

We have

$$\text{TSub}\frac{\Phi'; \Gamma \vdash e : \tau' \quad \tau' \leq \tau \quad \dfrac{\begin{array}{c}\Phi'^{\varepsilon} \subseteq \Phi^{\varepsilon} \quad \Phi^{\omega} \subseteq \Phi'^{\omega} \quad \Phi^{\alpha} \subseteq \Phi'^{\alpha} \\ \Phi'^{\delta_i} \subseteq \Phi^{\delta_i} \quad \Phi^{\delta_o} \subseteq \Phi'^{\delta_o}\end{array}}{\Phi' \leq \Phi}}{\Phi; \Gamma \vdash e : \tau}$$

Let $\Phi'' = [\Phi^{\alpha}; \Phi'^{\varepsilon}; \Phi^{\omega}; \Phi^{\delta_i}; \Phi^{\delta_o}]$. Thus we have:

$$\text{TSub}\frac{\Phi''; \Gamma \vdash e : \tau' \quad \tau' \leq \tau \quad \dfrac{\begin{array}{c}\Phi''^{\varepsilon} \subseteq \Phi^{\varepsilon} \quad \Phi^{\omega} = \Phi''^{\omega} \quad \Phi^{\alpha} = \Phi''^{\alpha} \\ \Phi^{\delta_i} = \Phi'^{\delta_i} \quad \Phi^{\delta_o} = \Phi''^{\delta_o}\end{array}}{\Phi'' \leq \Phi}}{\Phi; \Gamma \vdash e : \tau}$$

where the first premise follows by induction (which we can apply because $\Phi''^{\omega} \subseteq \Phi'^{\omega}$ and $\Phi''^{\alpha} \subseteq \Phi'^{\alpha}$ by assumption); the first premise of $\Phi'' \leq \Phi$ is by assumption, and the latter two premises are by definition of Φ''.

case (TRef) **:**

We know that

$$\text{TRef}\frac{\Phi; \Gamma \vdash e : \tau}{\Phi; \Gamma \vdash \mathsf{ref}\ \ e : \mathit{ref}^{\,\varepsilon}\ \tau}$$

and have $\Phi'; \Gamma \vdash e : \tau$ by induction, hence we get the result by (TRef).

case (TDeref) **:**

We know that

$$\text{TDeref}\frac{\begin{array}{c}\Phi_1;\Gamma\vdash e : \mathit{ref}^{\,\varepsilon}\ \tau \\ \Phi_2^{\varepsilon}=\varepsilon \qquad \Phi_2^{\delta_i}=\Phi_2^{\delta_o}\cup\varepsilon \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi;\Gamma\vdash\, !e : \tau}$$

We have $\Phi' \equiv [\alpha';\Phi_1^{\varepsilon}\cup\Phi_2^{\varepsilon};\omega';\delta_i{}';\delta_o{}']$ where $\alpha'\subseteq\Phi^{\alpha}$, and $\omega'\subseteq\Phi^{\omega}$. Choose $\Phi_1'\equiv[\alpha';\Phi_1^{\varepsilon};\Phi_2^{\varepsilon}\cup\omega';\delta_i;\Phi_2^{\varepsilon}\cup\delta_o]$ and $\Phi_2'\equiv[\alpha'\cup\Phi_1^{\varepsilon};\Phi_2^{\varepsilon};\omega';\Phi_2^{\varepsilon}\cup\delta_o;\delta_o]$, hence $\Phi_1'\longrightarrow\Phi_2'$, $\Phi_2'^{\varepsilon}=\Phi_2^{\varepsilon}=\varepsilon$, $\Phi_2'^{\delta_i}=\Phi_2'^{\delta_o}\cup\varepsilon$, and $\Phi'\equiv\Phi_1'\rhd\Phi_2'$. We want to prove that $\Phi';\Gamma\vdash\, !e:\tau$. Since $\alpha'\subseteq\alpha$, and $\Phi_2^{\varepsilon}\cup\omega'\subseteq\Phi_2^{\varepsilon}\cup\omega$ we can apply induction to get $\Phi_1';\Gamma\vdash e:\mathit{ref}^{\,\varepsilon}\ \tau$ and we get the result by applying (TDeref):

$$\text{TDeref}\frac{\begin{array}{c}\Phi_1';\Gamma\vdash e : \mathit{ref}^{\,\varepsilon}\ \tau \\ \Phi_2'^{\varepsilon}=\varepsilon \qquad \Phi_2^{\delta_i}=\Phi_2^{\delta_o}\cup\varepsilon \\ \Phi_1' \rhd \Phi_2' \hookrightarrow \Phi'\end{array}}{\Phi';\Gamma\vdash\, !e : \tau}$$

case (TApp) **:**

We know that

$$\text{TApp}\frac{\begin{array}{c}\Phi_1;\Gamma\vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \qquad \Phi_2;\Gamma\vdash e_2:\tau_1 \\ \Phi_1\rhd\Phi_2\rhd\Phi_3\hookrightarrow\Phi \\ \Phi_3^{\varepsilon}=\Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha}\subseteq\Phi_f^{\alpha} \qquad \Phi_3^{\omega}\subseteq\Phi_f^{\omega} \\ \Phi_3^{\delta_i}=\Phi_3^{\delta_o}\cup\Phi_f^{\varepsilon} \qquad \Phi_3^{\delta_o}\subseteq\Phi_f^{\delta_o}\end{array}}{\Phi;\Gamma\vdash e_1\ e_2 : \tau_2}$$

We have $\Phi'\equiv[\alpha';\Phi_1^{\varepsilon}\cup\Phi_2^{\varepsilon}\cup\Phi_3^{\varepsilon};\omega';\delta_i;\delta_{o1}]$ where $\alpha'\subseteq\Phi^{\alpha}$ and $\omega'\subseteq\Phi^{\omega}$. Choose $\Phi_1'\equiv[\alpha';\Phi_1^{\varepsilon};\Phi_2^{\varepsilon}\cup\Phi_3^{\varepsilon}\cup\omega';\delta_i;\delta_{o1}]$, $\Phi_2'\equiv[\alpha'\cup\Phi_1^{\varepsilon};\Phi_2^{\varepsilon};\Phi_3^{\varepsilon}\cup\omega';\delta_{o1};\delta_{o2}]$, $\Phi_3'\equiv[\alpha'\cup\Phi_1^{\varepsilon}\cup\Phi_2^{\varepsilon};\Phi_3^{\varepsilon};\omega';\delta_{o2};\delta_o]$, hence $\Phi_3'^{\varepsilon}=\Phi_3^{\varepsilon}=\varepsilon_f$, $\Phi_3^{\delta_i}=\Phi_3^{\delta_o}\cup\Phi_f^{\varepsilon}$, $\Phi_3^{\delta_o}\subseteq\Phi_f^{\delta_o}$, and $\Phi_1'\rhd\Phi_2'\rhd\Phi_3'\hookrightarrow\Phi'$. We want to prove that $\Phi';\Gamma\vdash e_1\ e_2:\tau_2$. Since $\alpha'\subseteq\alpha$ and $\Phi_2^{\varepsilon}\cup\Phi_3^{\varepsilon}\cup\omega'\subseteq\Phi_2^{\varepsilon}\cup\Phi_3^{\varepsilon}\cup\omega'$ we can apply induction to get $\Phi_1';\Gamma\vdash e_1:\tau_1\longrightarrow^{\Phi_f}\tau_2$. Similarly, since $\alpha'\cup\Phi_1^{\varepsilon}\subseteq\alpha\cup\Phi_1^{\varepsilon}$ and $\Phi_3^{\varepsilon}\cup\omega'\subseteq\Phi_3^{\varepsilon}\cup\omega$, we can apply induction to get $\Phi_2';\Gamma\vdash e_2:\tau_1$. We get the get the result by applying (TApp):

$$\text{TApp}\frac{\begin{array}{c}\Phi_1';\Gamma\vdash e_1 : \tau_1 \longrightarrow^{\Phi'_f} \tau_2 \qquad \Phi_2';\Gamma\vdash e_2:\tau_1 \\ \Phi_1'\rhd\Phi_2'\rhd\Phi_3'\hookrightarrow\Phi' \\ \Phi_3'^{\varepsilon}=\Phi_f^{\varepsilon} \qquad \Phi_3'^{\alpha}\subseteq\Phi_f^{\alpha} \qquad \Phi_3'^{\omega}\subseteq\Phi_f^{\omega} \\ \Phi_3'^{\delta_i}=\Phi_3'^{\delta_o}\cup\Phi_f^{\varepsilon} \qquad \Phi_3'^{\delta_o}\subseteq\Phi_f^{\delta_o}\end{array}}{\Phi';\Gamma\vdash e_1\ e_2 : \tau_2}$$

case (TAssign), (TIf), (TLet) **:**

Similar to (TApp).

□

Lemma C.0.25 (Left subexpression version consistency)**.** *If* $\Phi,\mathcal{R};H\vdash\Sigma$ *and* $\Phi_1\rhd\Phi_2\hookrightarrow\Phi$ *then* $\Phi_1,\mathcal{R};H\vdash\Sigma$.

Proof. We know:

$$\text{TC1}\frac{\begin{array}{c}\mathsf{f}\in\sigma\Rightarrow\mathsf{f}\in\alpha \\ \mathsf{f}\in(\varepsilon\cap\delta_i)\Rightarrow n'\in \mathit{ver}(H,\mathsf{f}) \\ \kappa^{\alpha}\supseteq(\alpha\cup\delta_i) \\ \kappa^{\omega}\supseteq(\omega\cup\varepsilon)\end{array}}{[\alpha;\varepsilon;\omega;\delta_i;\delta_o],\cdot;H\vdash(n',\sigma,\kappa)}$$

We need to prove:

$$\text{TC1}\frac{\begin{array}{c}\mathsf{f}\in\sigma\Rightarrow\mathsf{f}\in\alpha_1 \\ \mathsf{f}\in(\varepsilon_1\cap\delta_{i1})\Rightarrow n'\in \mathit{ver}(H,\mathsf{f}) \\ \kappa^{\alpha}\supseteq(\alpha_1\cup\delta_{i1}) \\ \kappa^{\omega}\supseteq(\omega_1\cup\varepsilon_1)\end{array}}{[\alpha_1;\varepsilon_1;\omega_1;\delta_{i1};\delta_{o1}],\cdot;H\vdash(n',\sigma,\kappa)}$$

The first premise follows since $\alpha_1\equiv\alpha$. The second follows because $\delta_{i1}\equiv\delta_i$ and $\varepsilon_1\subseteq\varepsilon$. The third follows because $\alpha_1\equiv\alpha$ and $\delta_{i1}\equiv\delta_i$. The fourth follows because $\omega\cup\varepsilon\equiv\omega\cup\varepsilon_1\cup\varepsilon_2\equiv(\omega\cup\varepsilon_2)\cup\varepsilon_1\equiv\omega_1\cup\varepsilon_1$.

□

Lemma C.0.26 (Subexpression version consistency)**.** *If $\Phi, \mathcal{R}_1 \bowtie \mathcal{R}_2; H \vdash \Sigma$ and $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$ then*

(i) $\mathcal{R}_2 \equiv \cdot$ *implies* $\Phi_1, \mathcal{R}_1; H \vdash \Sigma$

(ii) $\mathcal{R}_1 \equiv \cdot$ *and* $\Phi_1^{\varepsilon} \equiv \emptyset$ *implies* $\Phi_2, \mathcal{R}_2; H \vdash \Sigma$

Proof. Similar to Lemma C.0.25. □

Lemma C.0.27 (Stack Shapes)**.** *If $\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon_0} \langle n; \Sigma'; H'; e'\rangle$ then $top(\Sigma) = (n', \sigma, \kappa)$ and $top(\Sigma') = (n'', \sigma', \kappa')$ where $n' = n''$, $\sigma \subseteq \sigma'$ and $\kappa = \kappa'$.*

Proof. By induction on $\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon_0} \langle n; \Sigma'; H'; e'\rangle$. □

Lemma C.0.28 (Update preserves heap safety)**.** *If $n; \Gamma \vdash H$ and $updateOK(upd, H, (\alpha, \omega), dir)$ then $n+1; \mathcal{U}[\Gamma]^{upd} \vdash \mathcal{U}[H]^{upd}_{n+1}$.*

Proof. Same proof as Lemma B.0.12. □

The following lemma states that if we start with a well-typed program and a version-consistent trace and we take an update step, then afterward we will still have a well-typed program whose trace is version-consistent.

Lemma C.0.29 (Update preservation)**.**
Suppose we have the following:

1. $n \vdash H, e : \tau$ *(such that $\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}$ and $n; \Gamma \vdash H$ for some Γ, Φ)*
2. $\Phi, \mathcal{R}; H \vdash \Sigma$
3. $traceOK(\Sigma)$
4. $\langle n; \Sigma; H; e\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H';\ e\rangle$

where $H' \equiv \mathcal{U}[H]^{upd}_{n+1}$, $\Gamma' \equiv \mathcal{U}[\Gamma]^{upd}$, $\mu = (upd, dir)$, $\Sigma' \equiv \mathcal{U}[\Sigma]^{upd,dir}_{n}$, and $top(\Sigma') = (n'', \sigma', \kappa')$. Then for some Φ' such that $\Phi'^{\alpha} = \Phi^{\alpha}$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, $\Phi'^{\delta_o} = \Phi^{\delta_o}$, and $\Phi'^{\varepsilon} \subseteq \Phi^{\varepsilon}$ and some $\Gamma' \supseteq \Gamma$ we have that:

1. $n+1 \vdash H', e : \tau$ *where $\Phi'; \Gamma' \vdash e : \tau \rightsquigarrow \mathcal{R}$ and $n+1; \Gamma' \vdash H'$*
2. $\Phi', \mathcal{R}; H' \vdash \Sigma'$
3. $traceOK(\Sigma')$
4. $(dir = bck) \Rightarrow n'' \equiv n+1 \ \wedge\ (dir = fwd) \Rightarrow (\mathsf{f} \in \omega \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f}))$

Proof. Since $\mathcal{U}[\Gamma]^{upd} \supseteq \Gamma$, $\Phi; \mathcal{U}[\Gamma]^{upd} \vdash e : \tau \rightsquigarrow \mathcal{R}$ follows by weakening (Lemma C.0.19). Proceed by simultaneous induction on the typing derivation of e ($n \vdash H, e : \tau$) and on the evaluation derivation $\langle n; \Sigma; H; e\rangle \longrightarrow_{\mu} \langle n+1; \Sigma'; H';\ e\rangle$. Consider the last rule used in the evaluation derivation:

case [UPDATE] **:**

We have

$$\langle n; (n', \sigma, \kappa); H; e\rangle \longrightarrow_{(upd,dir)} \langle n''; \mathcal{U}\big[(n', \sigma, \kappa)\big]^{upd,dir}_{n''}; \mathcal{U}[H]^{upd}_{n''}; e\rangle$$

where $n'' \equiv n+1$. Let $\Phi' = \Phi$ and $(n'', \sigma', \kappa') \equiv \mathcal{U}[(n', \sigma, \kappa)]^{upd,dir}_{n+1}$.

To prove 1., we get $n''; \Gamma' \vdash H'$ by Lemma C.0.28 and $\Phi; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}$ by weakening.

To prove 2., we must show $\Phi, \cdot; H' \vdash (n'', \sigma', \kappa')$. By assumption, we have

$$\text{TC1}\ \frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \cap \delta_i \Rightarrow n' \in ver(H, \mathsf{f}) \\ \kappa^{\alpha} \supseteq (\alpha \cup \delta_i) \\ \kappa^{\omega} \supseteq (\omega \cup \varepsilon)\end{array}}{[\alpha; \varepsilon; \omega; \delta_i; \delta_o], \cdot; H \vdash (n', \sigma, \kappa)}$$

We need to prove

$$\text{TC1}\ \frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \cap \delta_i \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^{\alpha} \supseteq (\alpha \cup \delta_i) \\ \kappa^{\omega} \supseteq (\omega \cup \varepsilon)\end{array}}{[\alpha; \varepsilon; \omega; \delta_i; \delta_o], \cdot; H \vdash (n'', \sigma', \kappa')}$$

We have the first, third and fourth premises by assumption. For the second premise, we need to prove $\mathsf{f} \in \varepsilon \cap \delta_i \Rightarrow n''' \subseteq ver(H', \mathsf{f})$.

Consider each possible update type:

case $dir = bck$ **:**

From the definition of $\mathcal{U}[(n', \sigma, \kappa)]_{n+1}^{upd,bck}$, we know that $n''' = n+1$; from the definition of $\mathcal{U}[H]_{n+1}^{upd}$ we know that $n+1 \in ver(H', \mathsf{f})$ for all f, hence $n''' \in ver(H', \mathsf{f})$ for all f.

case $dir = fwd$ **:**

From the definition of $\mathcal{U}[(n', \sigma, \kappa)]_{n+1}^{upd,bck}$, we know that $n''' = n'$. Since $\kappa^\omega \supseteq (\omega \cup \varepsilon)$, from $updateOK(upd, H, (\kappa^\alpha, \kappa^\omega), dir)$ we know that $\forall \mathsf{f} \in (\omega \cup \varepsilon)$, $\mathsf{f} \notin \mathrm{dom}(upd.\mathrm{UB})$, hence $ver(H', \mathsf{f}) = ver(H', \mathsf{f})$. Hence $\mathsf{f} \in \varepsilon \cap \delta_i \Rightarrow n' \in ver(H, \mathsf{f})$ (assumption) implies $\mathsf{f} \in \varepsilon \cap \delta_i \Rightarrow n''' \in ver(H', \mathsf{f})$.

To prove 3., we must show $traceOK(n'', \sigma', \kappa')$. Consider each possible update type:

case $dir = bck$ **:**

From the definition of $\mathcal{U}[(n', \sigma, \kappa)]_{n+1}^{upd,bck}$, we know that $n''' = n+1$. Consider $(\mathsf{f}, \nu) \in \sigma$; it must be the case that $\mathsf{f} \notin \mathrm{dom}(upd^{chg})$. This is because $dir = bck$ implies $\kappa^\alpha \cap \mathrm{dom}(upd^{chg}) = \emptyset$ and by assumption (from [TC1] above) $\mathsf{f} \in \alpha$ and $\kappa^\alpha \supseteq \alpha$. Therefore, since $\mathsf{f} \notin \mathrm{dom}(upd^{chg})$, its σ' entry is $(\mathsf{f}, \nu \cup \{n'''\})$, which is the required result.

case $dir = fwd$ **:**

Since $\mathcal{U}[(n', \sigma, \kappa)]_{n+1}^{upd,fwd} = (n', \sigma, \kappa)$, the result is true by assumption.

To prove 4., we must show $n''' \equiv n+1 \vee (\mathsf{f} \in \omega \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f}))$. Consider each possible update type:

case $dir = bck$ **:**

From the definition of $\mathcal{U}[(n', \sigma, \kappa)]_{n+1}^{upd,bck}$, we know that $n''' = n+1$ so we are done.

case $dir = fwd$ **:**

We have $\mathcal{U}[(n', \sigma, \kappa)]_{n+1}^{upd,fwd} = (n', \sigma, \kappa)$, and from $updateOK(upd, H, (\kappa^\alpha, \kappa^\omega), dir)$ and we know that $\mathsf{f} \in \kappa^\omega \Rightarrow \mathsf{f} \notin \mathrm{dom}(upd^{chg})$ and by assumption (from [TC1] above) we know $\kappa^\omega \supseteq \omega$.

From the definition of $\mathcal{U}[H]_{n}^{upd}$ we know that $\mathcal{U}[(\mathsf{f} \mapsto (\tau, b, \nu), H)]_{n+1}^{upd} = \mathsf{f} \mapsto (\tau, b, \nu \cup \{n+1\})$ if $\mathsf{f} \notin \mathrm{dom}(upd^{chg})$. This implies that for $\mathsf{f} \in \omega$, $ver(H, \mathsf{f}) = \nu$ and $ver(H', \mathsf{f}) = \nu \cup \{n+1\}$, and therefore $ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f})$.

case [TX-CONG-1] **:**

We have that $\langle n; ((n', \sigma, \kappa), \Sigma); H; \mathsf{intx}\ e\rangle \longrightarrow_\mu \langle n''; \mathcal{U}[(n', \sigma, \kappa)]_{n''}^{\mu}, \Sigma'; H'; \mathsf{intx}\ e'\rangle$ follows from $\langle n; \Sigma; H; \mathbb{E}[e]\rangle \longrightarrow_\eta \langle n''; \Sigma'; H'; \mathbb{E}[e']\rangle$ by [TX-CONG-1], where $\mu \equiv (upd, dir)$ and $n'' \equiv n{+}1$. Let $(n'', \sigma', \kappa') \equiv \mathcal{U}[(n', \sigma, \kappa)]_{n+1}^{upd,dir}$. By assumption and subtyping derivations (Lemma C.0.23) we have

$$\text{TSub}\dfrac{\text{TIntrans}\ \dfrac{\begin{array}{c}\Phi_e; \Gamma \vdash e : \tau' \rightsquigarrow \mathcal{R}\\ \alpha \subseteq \Phi_e^\alpha \qquad \omega \subseteq \Phi_e^\omega\end{array}}{[\alpha; \emptyset; \omega; \delta_i; \delta_o]; \Gamma \vdash \mathsf{intx}\ e : \tau' \rightsquigarrow \Phi_e, \mathcal{R}} \qquad \tau' \leq \tau \qquad [\alpha; \emptyset; \omega; \delta_i; \delta_o] \leq [\alpha; \varepsilon; \omega; \delta_i; \delta_o]}{[\alpha; \varepsilon; \omega; \delta_i; \delta_o]; \Gamma \vdash \mathsf{intx}\ e : \tau \rightsquigarrow \Phi_e, \mathcal{R}}$$

and by flow effect weakening (Lemma C.0.24) we know that α, ω, δ_i and δ_o are unchanged in the use of (TSub). We have $\Phi_e \equiv [\alpha_e; \varepsilon_e; \omega_e; \delta_{ie}; \delta_{oe}]$, so that $\omega_e \supseteq \omega$ and $\alpha_e \supseteq \alpha$. To apply induction, we must show that $\Phi_e, \mathcal{R}; H \vdash \Sigma$ (which follows by inversion on $\Phi, \Phi_e, \mathcal{R}; H \vdash ((n', \sigma, \kappa), \Sigma)$; $\Phi_e; \Gamma \vdash e : \tau' \rightsquigarrow \mathcal{R}$ (which follows by assumption); and $n; \Gamma \vdash H$ (by assumption).

By induction we have:

(i) $\Phi'_e; \Gamma' \vdash e' : \tau' \rightsquigarrow \mathcal{R}$ and

(ii) $n+1; \Gamma' \vdash H'$

(iii) $\Phi'_e, \mathcal{R}; H' \vdash \Sigma'$

(iv) $traceOK(\Sigma')$

(v) $(dir = bck) \Rightarrow n''' \equiv n+1\ \wedge\ (dir = fwd) \Rightarrow (\mathsf{f} \in \omega_e \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f}))$

where $\Phi'_e \equiv [\alpha_e; \varepsilon'_e; \omega_e; \delta_{ie}; \delta_{oe}]$, $\varepsilon'_e \subseteq \varepsilon_e$.

Let $\Phi' = [\alpha; \emptyset; \omega; \delta_i; \delta_o]$ (hence $\Phi'^\alpha = \Phi^\alpha$, $\Phi'^\omega = \Phi^\omega$, $\emptyset \subseteq \Phi^\varepsilon$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required). To prove 1., we can show

$$\text{TSub}\dfrac{\text{TIntrans}\ \dfrac{\begin{array}{c}\Phi'_e; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}\\ \alpha \subseteq \Phi_e'^\alpha \qquad \omega \subseteq \Phi_e'^\omega\end{array}}{\Phi'; \Gamma \vdash \mathsf{intx}\ e' : \tau \rightsquigarrow \Phi'_e, \mathcal{R}} \qquad \tau' \leq \tau \qquad \Phi' \leq \Phi'}{\Phi'; \Gamma \vdash \mathsf{intx}\ e' : \tau \rightsquigarrow \Phi'_e, \mathcal{R}}$$

The first premise of [TINTRANS] follows by (i), and the second –fifth by assumption (from $[\alpha;\emptyset;\omega;\delta_i;\delta_o];\Gamma \vdash \mathsf{intx}\ e : \tau' \rightsquigarrow \Phi_e, \mathcal{R}$).

To prove 2., we need to show that

$$\text{TC2}\ \frac{\begin{array}{c}\Phi'_e, \mathcal{R}; H' \vdash \Sigma' \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\emptyset \cap \delta_i) \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^\alpha \supseteq (\alpha \cup \delta_i) \\ \kappa^\omega \supseteq (\omega \cup \emptyset)\end{array}}{[\alpha;\emptyset;\omega;\delta_i;\delta_o], \Phi'_e, \mathcal{R}; H' \vdash ((n''', \sigma', \kappa'), \Sigma')}$$

We have the first premise by (iii), the second by assumption (since $\text{dom}(\sigma) = \text{dom}(\sigma')$ from the definition of $\mathcal{U}[(n', \sigma, \kappa)]^{upd,dir}_{n+1}$), the third holds vacuously, and the fourth and fifth follow by assumption (note that $\kappa' = \kappa$).

To prove 3., we must show $traceOK((n''', \sigma', \kappa'), \Sigma')$, which reduces to proving $traceOK((n'', \sigma', \kappa')$ since we have $traceOK(\Sigma')$ from (iv). We have $traceOK(n', \sigma, \kappa)$ by assumption. Consider each possible update type:

case $dir = bck$ **:**

From the definition of $\mathcal{U}[(n', \sigma, \kappa)]^{upd,bck}_{n+1}$, we know that $n''' = n + 1$. Consider $(\mathsf{f}, \nu) \in \sigma$; it must be the case that $\mathsf{f} \notin \text{dom}(upd^{chg})$. This is because $dir = bck$ implies $\alpha_e \cap \text{dom}(upd^{chg}) = \emptyset$ and by assumption we have $\alpha \subseteq \alpha_e$ (from (TINTRANS)), $\mathsf{f} \in \alpha$ (from the first premise of [TC1] above), and $\kappa^\alpha \supseteq (\alpha \cup \delta_i)$ (from the fourth premise of [TC1] above). Therefore, since $\mathsf{f} \notin \text{dom}(upd^{chg})$, its σ' entry is $(\mathsf{f}, \nu \cup \{n'\})$, which is the required result.

case $dir = fwd$ **:**

Since $\mathcal{U}[(n', \sigma, \kappa)]^{upd,fwd}_{n+1} = (n', \sigma, \kappa)$, the result is true by assumption.

Part 4. follows directly from (v) and the fact that $\omega_e \supseteq \omega$.

case [CONG] **:**

We have that $\langle n; \Sigma; H; \mathbb{E}[e]\rangle \longrightarrow_\mu \langle n''; \Sigma'; H'; \mathbb{E}[e']\rangle$ follows from $\langle n; \Sigma; H; e\rangle \longrightarrow_\mu \langle n''; \Sigma'; H'; e'\rangle$ by [CONG], where $\mu \equiv (upd, dir)$. Consider the shape of $\mathbb{E}$:

case _ **:**

The result follows directly by induction.

case $\mathbb{E}\ e_2$ **:**

By assumption, we have $\Phi; \Gamma \vdash (\mathbb{E}\ e_2)[e_1] : \tau \rightsquigarrow \mathcal{R}$. By subtyping derivations (Lemma C.0.23) we know we can construct a proof derivation of this ending in (TSUB):

$$\text{TSub}\ \frac{\text{TApp}\ \dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash \mathbb{E}[e_1] : \tau_1 \longrightarrow^{\Phi_f} \tau'_2 \rightsquigarrow \mathcal{R}_1 \qquad \Phi_2; \Gamma \vdash e_2 : \tau_1 \rightsquigarrow \cdot \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi_s \\ \Phi_3^\varepsilon = \Phi_f^\varepsilon \qquad \Phi_3^\alpha \subseteq \Phi_f^\alpha \qquad \Phi_3^\omega \subseteq \Phi_f^\omega \\ \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \Phi_f^\varepsilon \qquad \Phi_3^{\delta_o} \subseteq \Phi_f^{\delta_o} \\ \mathbb{E}\ [e_1] \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot\end{array}}{\Phi_s; \Gamma \vdash (\mathbb{E}\ e_2)[e_1] : \tau'_2 \rightsquigarrow \mathcal{R}_1} \quad \tau'_2 \leq \tau_2 \quad \text{SCtxt}\ \dfrac{\begin{array}{c}\Phi \equiv [\alpha;\varepsilon;\omega;\delta_i;\delta_o] \\ \Phi_s \equiv [\alpha;\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f;\omega;\delta_i;\delta_o] \\ (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f) \subseteq \varepsilon\end{array}}{\Phi_s \leq \Phi}}{\Phi; \Gamma \vdash (\mathbb{E}\ e_2)[e_1] : \tau_2 \rightsquigarrow \mathcal{R}_1}$$

and by flow effect weakening (Lemma C.0.24) we know that α, ω , δ_i and δ_o are unchanged in the use of (TSUB).

By inversion on $\langle n; \Sigma; H; (\mathbb{E}\ e_2)[e_1]\rangle \longrightarrow_\mu \langle n''; \Sigma'; H'; (\mathbb{E}\ e_2)[e_1]\rangle$ we have $\langle n; \Sigma; H; e_1\rangle \longrightarrow_\mu \langle n''; \Sigma'; H'; e_1\rangle$, and then applying [CONG] we have $\langle n; \Sigma; H; (\mathbb{E})[e_1]\rangle \longrightarrow_\mu \langle n''; \Sigma'; H'; (\mathbb{E})[e'_1]\rangle$

From $\Phi, \mathcal{R}_1; H \vdash \Sigma$ we know that:

$$\begin{array}{l}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \cap \delta_i \Rightarrow n' \in ver(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\alpha \cup \delta_i) \\ \kappa^\omega \supseteq (\omega \cup \varepsilon)\end{array}$$

where (n', σ, κ) is the top of Σ. Since $\Phi \equiv [\alpha; \varepsilon; \omega; \delta_i; \delta_o]$ and $\Phi_s \equiv [\alpha; \varepsilon_s; \omega; \delta_i; \delta_o]$ and $\varepsilon_s = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$ (where $\varepsilon_3 = \varepsilon_f$), we have $\alpha \equiv \alpha_1$ hence:

$$\begin{array}{l} \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in \varepsilon_1 \cap \delta_i \Rightarrow n' \in ver(H, \mathsf{f}) \\ \kappa^{\alpha} \supseteq (\alpha_1 \cup \delta_{i1}) \\ \kappa^{\omega} \supseteq (\omega_1 \cup \varepsilon_1) \end{array}$$

but since $\Phi_1 \equiv [\alpha; \varepsilon_1; \omega_1; \delta_{i1}; \delta_{o1}]$, we have $\Phi_1, \mathcal{R}_1; H \vdash \Sigma$. Hence we can apply induction on $\Phi_1; \Gamma \vdash \mathbb{E}[e_1] : \tau_1 \longrightarrow^{\Phi_f} \tau_2' \rightsquigarrow \mathcal{R}_1$, yielding:

(i) $\Phi_1'; \Gamma' \vdash \mathbb{E}[e_1'] : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \mathcal{R}_1$ and
(ii) $n+1; \Gamma' \vdash H'$
(iii) $\Phi_1', \mathcal{R}_1; H' \vdash \Sigma'$
(iv) $traceOK(\Sigma')$
(v) $(dir = bck) \Rightarrow n''' \equiv n+1 \ \wedge \ (dir = fwd) \Rightarrow (\mathsf{f} \in \omega_1 \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f}))$

where $\Phi_1' \equiv [\alpha; \varepsilon_1'; \omega_1; \delta_i; \delta_{o1}]$ and $\varepsilon_1' \subseteq \varepsilon_1$. Choose $\Phi_2' = [\alpha \cup \varepsilon_1'; \varepsilon_2; \omega_2; \delta_{i2}; \delta_{o2}]$ and $\Phi_3' = [\alpha \cup \varepsilon_1' \cup \varepsilon_2; \varepsilon_f; \omega; \delta_{i3}; \delta_o]$ and thus $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi_s'$ and $\Phi_3'^{\varepsilon} = \Phi_f^{\varepsilon}$. Let $\Phi' = [\alpha; \varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_f; \omega; \delta_i; \delta_o]$, where $\varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_f \subseteq \varepsilon$, as required.
To prove 1., we have $n+1; \Gamma' \vdash H'$ by (ii), and apply (TApp):

$$\text{TApp}\frac{\begin{array}{c} \Phi_1'; \Gamma' \vdash \mathbb{E}[e_1'] : \tau_1 \longrightarrow^{\Phi_f} \tau_2' \rightsquigarrow \mathcal{R}_1 \qquad \Phi_2'; \Gamma' \vdash e_2 : \tau_1 \rightsquigarrow \cdot \\ \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi_s' \\ \Phi_3'^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3'^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3'^{\omega} \subseteq \Phi_f^{\omega} \\ \Phi_3'^{\delta_i} = \Phi_3'^{\delta_o} \cup \Phi_f^{\varepsilon} \qquad \Phi_3'^{\delta_o} \subseteq \Phi_f^{\delta_o} \\ \mathbb{E}[e_1'] \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot \end{array}}{\Phi_s'; \Gamma' \vdash (\mathbb{E}\ e_2)[e_1'] : \tau_2' \rightsquigarrow \mathcal{R}_1}$$

The first premise follows by (i), the second because we have $\Phi_2; \Gamma' \vdash e_2 : \tau_1$ by weakening (since $\Gamma' \supseteq \Gamma$) and then $\Phi_2'; \Gamma' \vdash e_2 : \tau_1$ by flow effect weakening (Lemma C.0.24) (which we can apply because $\Phi_2'^{\omega} = \Phi_2^{\omega}$, $\Phi_2'^{\varepsilon} = \Phi_2^{\varepsilon}$, $\Phi_2'^{\alpha} = \alpha_1 \cup \varepsilon_1'$, $\Phi_2^{\alpha} = \alpha_1 \cup \varepsilon_1$ hence $\Phi_2'^{\alpha} \subseteq \Phi_2^{\alpha}$, $\Phi_2'^{\delta_i} = \Phi_2^{\delta_i}$, and $\Phi_2'^{\delta_o} = \Phi_2^{\delta_o}$) , the third— eighth by choice of Φ_2', Φ_3' and Φ_s', and the last as $\mathcal{R}_2 \equiv \cdot$ by assumption. We can now apply (TSub):

$$\text{TSub}\frac{\begin{array}{c} \Phi'; \Gamma \vdash (\mathbb{E}\ e_2)[e_1'] : \tau_2' \rightsquigarrow \mathcal{R}_1 \\ \tau_2' \leq \tau_2 \qquad \Phi' \leq \Phi' \end{array}}{\Phi'; \Gamma \vdash (\mathbb{E}\ e_2)[e_1'] : \tau_2 \rightsquigarrow \mathcal{R}_1'}$$

To prove part 2., we must show that $\Phi', \mathcal{R}_1; H' \vdash \Sigma'$.
By inversion on $\Phi, \mathcal{R}_1; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$ or $\Sigma \equiv ((n', \sigma, \kappa), \Sigma'')$. We have two cases:

$\Sigma \equiv (n', \sigma, \kappa)$: Hence $\Sigma' \equiv \mathcal{U}[(n', \sigma, \kappa)]_{n''}^{upd, dir} \equiv (n'', \sigma', \kappa')$. By (iii) we must have $\mathcal{R}_1 \equiv \cdot$ such that

$$\text{TC1}\frac{\begin{array}{c} \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in (\varepsilon_1' \cap \delta_{i1}) \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\alpha_1 \cup \delta_{i1}) \\ \kappa^{\omega\prime} \supseteq (\omega_1 \cup \varepsilon_1') \end{array}}{[\alpha; \varepsilon_1'; \omega_1; \delta_{i1}; \delta_{o1}], \cdot; H' \vdash (n'', \sigma', \kappa')}$$

To achieve the desired result we need to prove:

$$\text{TC1}\frac{\begin{array}{c} \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in ((\varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_f) \cap \delta_{i1}) \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\alpha \cup \delta_{i1}) \\ \kappa^{\omega\prime} \supseteq (\omega \cup \varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_f) \end{array}}{[\alpha; \varepsilon_1' \cup \varepsilon_2 \cup \varepsilon_f; \omega; \delta_i; \delta_o], \cdot; H' \vdash (n'', \sigma', \kappa')}$$

Note that $\alpha \equiv \alpha_1$. The first premise is by assumption (since $\text{dom}(\sigma) = \text{dom}(\sigma')$ from the definition of $\mathcal{U}[(n', \sigma, \kappa)]_{n+1}^{upd, dir}$). For the second premise, we need to show that for all $\mathsf{f} \in ((\varepsilon_2 \cup \varepsilon_f) \cap \delta_{i1}) \Rightarrow n''' \in ver(H', \mathsf{f})$; for those $\mathsf{f} \in (\varepsilon_1' \cap \delta_{i1})$ the result is by assumption.
Consider each possible update type:

case $dir = bck$ **:**

From the definition of $\mathcal{U}[(n', \sigma, \kappa)]^{upd,bck}_{n+1}$, we know that $n''' = n+1$; from the definition of $\mathcal{U}[H]^{upd}_{n+1}$ we know that $n+1 \in ver(H', \mathsf{f})$ for all f, hence $n''' \in ver(H', \mathsf{f})$ for all f.

case $dir = fwd$ **:**

From (v) we have that $\mathsf{f} \in \omega_1 \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f})$. Since $(\varepsilon_2 \cup \varepsilon_f) \subseteq \omega_1$ (by $\Phi'_1 \rhd \Phi'_2 \rhd \Phi'_3 \hookrightarrow \Phi'$), we have $((\varepsilon_2 \cup \varepsilon_f) \cap \delta_{i1}) \subseteq \omega_1$ hence $\mathsf{f} \in ((\varepsilon_2 \cup \varepsilon_f) \cap \delta_{i1}) \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f})$. By inversion on $\Phi, \mathcal{R}_1; H \vdash \Sigma$ we have $\mathsf{f} \in (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f) \Rightarrow n' \in ver(H, \mathsf{f})$, and thus $\mathsf{f} \in (\varepsilon_2 \cup \varepsilon_f) \cap \delta_{i1}) \Rightarrow n' \in ver(H', \mathsf{f})$. We have $\mathcal{U}[(n', \sigma, \kappa)]^{upd,fwd}_{n+1} = (n', \sigma, \kappa)$ hence $n''' = n'$, so finally we have $\mathsf{f} \in ((\varepsilon_2 \cup \varepsilon_f)) \cap \delta_{i1}) \Rightarrow n''' \in ver(H', \mathsf{f})$.

The third and fourth premises follow by assumption since $\kappa' = \kappa$ and $\varepsilon'_1 \subseteq \varepsilon_1$.

$\Sigma \equiv ((n', \sigma, \kappa), \Sigma'')$ Hence $\Sigma' \equiv \mathcal{U}[(n', \sigma, \kappa)]^{upd,dir}_{n''} \equiv ((n''', \sigma', \kappa'), \Sigma''')$

By (iii), we must have $\mathcal{R}_1 \equiv \Phi'', \mathcal{R}''$ such that

$$\text{TC2}\ \frac{\begin{array}{c} \Phi'', \mathcal{R}''; H' \vdash \Sigma'' \\ \Phi'_1 \equiv [\alpha; \varepsilon'_1; \omega_1; \delta_{i1}; \delta_{o1}] \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha_1 \\ \mathsf{f} \in (\varepsilon'_1 \cap \delta_{i1}) \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\alpha_1 \cup \delta_{i1}) \\ \kappa^{\omega\prime} \supseteq (\omega_1 \cup \varepsilon'_1) \end{array}}{\Phi'_1, \Phi'', \mathcal{R}''; H' \vdash ((n''', \sigma', \kappa'), \Sigma''')}$$

We wish to show that

$$\text{TC2}\ \frac{\begin{array}{c} \Phi'', \mathcal{R}''; H' \vdash \Sigma'' \\ \Phi' \equiv [\alpha; \varepsilon'_1 \cup \varepsilon_2 \cup \varepsilon_f; \omega; \delta_i; \delta_o] \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in ((\varepsilon'_1 \cup \varepsilon_2 \cup \varepsilon_f) \cap \delta_{i1}) \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\alpha \cup \delta_{i1}) \\ \kappa^{\omega\prime} \supseteq (\omega \cup \varepsilon'_1 \cup \varepsilon_2 \cup \varepsilon_f) \end{array}}{\Phi', \Phi'', \mathcal{R}''; H' \vdash ((n''', \sigma', \kappa'), \Sigma''')}$$

$\Phi'', \mathcal{R}''; H' \vdash \Sigma''$ follows by assumption while the rest of the premises follow by the same argument as in the $\Sigma \equiv (n', \sigma, \kappa)$ case, above.

Part 3. follows directly from (iv).

Part 4. follows directly from (v) and the fact that $\omega_1 \supseteq \omega$ (because $\omega_1 \equiv \varepsilon_2 \cup \varepsilon_f \cup \omega$).

case $v\ \mathbb{E}$ **:**

By assumption, we have $\Phi; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau \rightsquigarrow \mathcal{R}$. By subtyping derivations (Lemma C.0.23) we have:

$$\text{TSub}\ \frac{\tau'_2 \leq \tau_2 \qquad \text{TApp}\ \dfrac{\begin{array}{c} \Phi_1; \Gamma \vdash v : \tau_1 \longrightarrow^{\Phi_f} \tau'_2 \rightsquigarrow \cdot \qquad \Phi_2; \Gamma \vdash \mathbb{E}[e_2] : \tau_1 \rightsquigarrow \mathcal{R}_2 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi_s \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \Phi_f^{\varepsilon} \qquad \Phi_3^{\delta_o} \subseteq \Phi_f^{\delta_o} \\ v \not\equiv v' \Rightarrow \mathcal{R}_2 = \cdot \end{array}}{\Phi_s; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau'_2 \rightsquigarrow \mathcal{R}_2} \qquad \text{SCtxt}\ \dfrac{\begin{array}{c} \Phi \equiv [\alpha; \varepsilon; \omega; \delta_i; \delta_o] \\ \Phi_s \equiv [\alpha; \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f; \omega; \delta_i; \delta_o] \\ (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f) \subseteq \varepsilon \end{array}}{\Phi_s \leq \Phi}}{\Phi; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau_2 \rightsquigarrow \mathcal{R}_2}$$

and by flow effect weakening (Lemma C.0.24) we know that α, ω, δ_i and δ_o are unchanged in the use of (TSub).

By inversion on $\langle n; \Sigma; H; (v\ \ \mathbb{E})[e_2]\rangle \longrightarrow_{\mu} \langle n''; \Sigma'; H'; (v\ \ \mathbb{E})[e'_2]\rangle$ we have $\langle n; \Sigma; H; e_2\rangle \longrightarrow_{\mu} \langle n''; \Sigma'; H'; e'_2\rangle$, and then applying [cong] we have $\langle n; \Sigma; H; (\mathbb{E})[e_2]\rangle \longrightarrow_{\mu} \langle n''; \Sigma'; H'; (\mathbb{E})[e'_2]\rangle$

From $\Phi, \mathcal{R}_2; H \vdash \Sigma$ we know that:

$$\begin{array}{l} \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in \varepsilon \cap \delta_i \Rightarrow n' \in ver(H, \mathsf{f}) \\ \kappa^{\alpha} \supseteq (\alpha \cup \delta_i) \\ \kappa^{\omega} \supseteq (\omega \cup \varepsilon) \end{array}$$

where (n', σ, κ) is the top of Σ. We have $\Phi \equiv [\alpha; \varepsilon; \omega; \delta_i; \delta_o]$, $\Phi_s \equiv [\alpha_s; \varepsilon_s; \omega_s; \delta_{is}; \delta_{os}]$, $\varepsilon_s \subseteq \varepsilon$, $\varepsilon_s = \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3$ (where $\varepsilon_3 = \varepsilon_f$), $\Phi_2 \equiv [\alpha_2; \varepsilon_2; \omega_2; \delta_{i2}; \delta_{o2}]$, $\alpha_2 \equiv \alpha_1 \cup \varepsilon_1 = \alpha$ (since $\varepsilon_1 = \emptyset$; if it's not $\emptyset$ we can construct a derivation for v that has $\varepsilon_1 = \emptyset$ as argued in preservation (Lemma C.0.31), (TApp)-[Cong], case $v\ \mathbb{E}$). Similarly, we have $\alpha_2 = \alpha_1 = \alpha$ and $\delta_{i2} = \delta_{i1} = \delta_i$. We have

$$\begin{array}{l} \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha_2 \\ \mathsf{f} \in \varepsilon_2 \cap \delta_{i2} \Rightarrow n' \in ver(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\alpha_2 \cup \delta_{i2}) \\ \kappa^\omega \supseteq (\omega_2 \cup \varepsilon_2) \end{array}$$

hence $\Phi_2, \mathcal{R}_2; H \vdash \Sigma$ and we can apply induction on $\Phi_2; \Gamma \vdash \mathbb{E}[e_2] : \tau_1 \longrightarrow^{\Phi_f} \tau_2' \rightsquigarrow \mathcal{R}_2$, yielding:

(i) $\Phi_2'; \Gamma' \vdash \mathbb{E}[e_2] : \tau_1 \rightsquigarrow \mathcal{R}_2$ and

(ii) $n+1; \Gamma' \vdash H'$

(iii) $\Phi_2', \mathcal{R}_2; H' \vdash \Sigma'$

(iv) $traceOK(\Sigma')$

(v) $(dir = bck) \Rightarrow n'' \equiv n+1\ \wedge\ (dir = fwd) \Rightarrow (\mathsf{f} \in \omega_2 \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f}))$

where $\Phi_2' \equiv [\alpha_2; \varepsilon_2'; \omega_2; \delta_{i2}; \delta_{o2}]$ and $\varepsilon_2' \subseteq \varepsilon_2$. Choose $\Phi_1' = [\alpha; \emptyset; \omega_2 \cup \varepsilon_2'; \delta_{i1}; \delta_{o1}]$ and $\Phi_3' = [\alpha \cup \varepsilon_2'; \varepsilon_f; \omega; \delta_{i3}; \delta_o]$ and thus $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'$ and $\Phi_3'^{\varepsilon} = \Phi_f^{\varepsilon}$.

Let $\Phi' \equiv [\alpha; \varepsilon_2' \cup \varepsilon_f; \omega; \delta_i; \delta_o]$ and thus $\varepsilon_2' \cup \varepsilon_f \subseteq \varepsilon$ as required.

To prove 1., we have $n+1; \Gamma' \vdash H'$ by (ii), and apply (TApp):

$$\text{TApp}\frac{\begin{array}{c} \Phi_1'; \Gamma' \vdash v : \tau_1 \longrightarrow^{\Phi_f} \tau_2' \rightsquigarrow \cdot \qquad \Phi_2'; \Gamma' \vdash \mathbb{E}[e_2] : \tau_1 \rightsquigarrow \mathcal{R}_2 \\ \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi' \\ \Phi_3'^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3'^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3'^{\omega} \subseteq \Phi_f^{\omega} \\ \Phi_3'^{\delta_i} = \Phi_3'^{\delta_o} \cup \varepsilon_f \qquad \Phi_3'^{\delta_o} \subseteq \Phi_f^{\delta_o} \\ v \not\equiv v' \Rightarrow \mathcal{R}_2 = \cdot \end{array}}{\Phi'; \Gamma' \vdash (v\ \mathbb{E})[e_2] : \tau_2' \rightsquigarrow \mathcal{R}_2}$$

The first premise follows by value typing, the second by (i), the third– eighth by choice of Φ_1' and Φ_3', and the last holds vacuously. We can now apply (TSub):

$$\text{TSub}\frac{\begin{array}{c} \Phi'; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau_2' \rightsquigarrow \mathcal{R}_2 \\ \tau_2' \leq \tau_2 \qquad \Phi' \leq \Phi' \end{array}}{\Phi'; \Gamma \vdash (v\ \mathbb{E})[e_2] : \tau_2 \rightsquigarrow \mathcal{R}_2}$$

To prove part 2., we must show that $\Phi', \mathcal{R}_2; H' \vdash \Sigma'$.

By inversion on $\Phi, \mathcal{R}_2; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$ or $\Sigma \equiv ((n', \sigma, \kappa), \Sigma'')$. We have two cases:

$\Sigma \equiv (n', \sigma, \kappa)$: By (iii) we must have $\mathcal{R}_2 \equiv \cdot$ such that

$$\text{TC1}\frac{\begin{array}{c} \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha_2 \\ \mathsf{f} \in \varepsilon_2' \cap \delta_{i2} \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^{\alpha'} \supseteq (\alpha_2 \cup \delta_{i2}) \\ \kappa^{\omega'} \supseteq (\omega_2 \cup \varepsilon_2') \end{array}}{[\alpha; \varepsilon_2'; \omega_2; \delta_{i2}; \delta_{o2}], \cdot; H' \vdash (n'', \sigma', \kappa')}$$

To achieve the desired result we need to prove:

$$\text{TC1}\frac{\begin{array}{c} \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\varepsilon_2' \cup \varepsilon_f) \cap \delta_i \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^{\alpha'} \supseteq (\alpha \cup \delta_i) \\ \kappa^{\omega'} \supseteq (\omega \cup \varepsilon_2' \cup \varepsilon_f) \end{array}}{[\alpha; \varepsilon_2' \cup \varepsilon_f; \omega; \delta_i; \delta_o], \cdot; H' \vdash (n'', \sigma', \kappa')}$$

Note that $\alpha_2 = \alpha_1 = \alpha$. The first premise follows by assumption (since $\mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$ from the definition of $\mathcal{U}[(n', \sigma, \kappa)]_{n+1}^{upd, dir}$). The third and fourth premise follow by assumption since $\delta_i = \delta_{i2}$, $\alpha = \alpha_2$, $\varepsilon_1 = \emptyset$ and $\omega_2 = \omega \cup \varepsilon_f$. For the second premise, we need to show that for all $\mathsf{f} \in (\varepsilon_f \cap \delta_i) \Rightarrow n'' \in ver(H', \mathsf{f})$ (for those $\mathsf{f} \in \varepsilon_2' \cap \delta_i$ the result is by assumption). Consider each possible update type:

case *dir* $= bck$ **:**

From the definition of $\mathcal{U}[(n', \sigma, \kappa)]^{upd,bck}_{n+1}$, we know that $n''' = n+1$; from the definition of $\mathcal{U}[H]^{upd}_{n+1}$ we know that $n+1 \in ver(H', \mathsf{f})$ for all f, hence $n''' \in ver(H', \mathsf{f})$ for all f.

case *dir* $= fwd$ **:**

From (v) we have that $\mathsf{f} \in \omega_2 \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f})$. Then $\varepsilon_f \subseteq \omega_2$ (by $\Phi'_1 \rhd \Phi'_2 \rhd \Phi'_3 \hookrightarrow \Phi'$) implies $\mathsf{f} \in \varepsilon_f \Rightarrow ver(H, \mathsf{f}) \subseteq ver(H', \mathsf{f})$. By inversion on $\Phi, \mathcal{R}_2; H \vdash \Sigma$ we have $\mathsf{f} \in ((\varepsilon_2 \cup \varepsilon_f) \cap \delta_i) \Rightarrow n' \in ver(H, \mathsf{f})$, and thus $\mathsf{f} \in \varepsilon_f \Rightarrow n' \in ver(H', \mathsf{f})$. We have $\mathcal{U}[(n', \sigma, \kappa)]^{upd,fwd}_{n+1} = (n', \sigma, \kappa)$ hence $n''' = n'$, so finally we have $\mathsf{f} \in (\varepsilon_f \cap \delta_i) \Rightarrow n''' \in ver(H', \mathsf{f})$.

The fourth and fifth premises follow by assumption since $\kappa' = \kappa$ and $\varepsilon'_2 \subseteq \varepsilon_2$.

$\Sigma \equiv (n'', \sigma, \kappa), \Sigma''$ By (iii), we must have $\mathcal{R}_2 \equiv \Phi'', \mathcal{R}''$ such that

$$\text{TC2}\,\frac{\begin{array}{c}\Phi'', \mathcal{R}''; H' \vdash \Sigma'' \\ \Phi'_2 \equiv [\alpha; \varepsilon'_2; \omega_2; \delta_{i2}; \delta_{o2}] \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha_2 \\ \mathsf{f} \in \varepsilon'_2 \cap \delta_{i2} \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\alpha_2 \cup \delta_{i2}) \\ \kappa^{\omega\prime} \supseteq (\omega_2 \cup \varepsilon'_2)\end{array}}{\Phi'_2, \Phi'', \mathcal{R}''; H' \vdash ((n''', \sigma', \kappa'), \Sigma''')}$$

We wish to show that

$$\text{TC2}\,\frac{\begin{array}{c}\Phi'', \mathcal{R}''; H' \vdash \Sigma'' \\ \Phi' \equiv [\alpha; \varepsilon'_2 \cup \varepsilon_f; \omega; \delta_i; \delta_o] \\ \mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\varepsilon'_2 \cup \varepsilon_f) \cap \delta_i \Rightarrow n''' \in ver(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\alpha \cup \delta_i) \\ \kappa^{\omega\prime} \supseteq (\omega \cup \varepsilon'_2 \cup \varepsilon_f)\end{array}}{\Phi', \Phi'', \mathcal{R}''; H' \vdash ((n''', \sigma', \kappa'), \Sigma''')}$$

$\Phi'', \mathcal{R}''; H' \vdash \Sigma''$ follows by assumption while the third and fourth premises follow by the same argument as in the $\Sigma \equiv (n', \sigma, \kappa)$ case, above.

Part 3. follows directly from (iv).

Part 4. follows directly from (v) and the fact that $\omega_2 \supseteq \omega$.

case all others **:**

Similar to cases above.

□

This lemma says that if take an evaluation step that is not an update, the version set of any z remains unchanged.

Lemma C.0.30 (Non-update step version preservation)**.** *If* $\langle n; \Sigma; H; e\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H';\ e'\rangle$ *then for all* $\mathsf{z} \in \text{dom}(H')$, $ver(H', \mathsf{z}) = ver(H, \mathsf{z})$.

Proof. By inspection of the evaluation rules. □

The following lemma states that if we start with a well-typed program and a version-consistent trace and we can take an evaluation step, then afterward we will still have a well-typed program whose trace is version-consistent.

Lemma C.0.31 (Preservation)**.**
Suppose we have the following:

1. $n \vdash H, e : \tau$ *(such that* $\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}$ *and* $n; \Gamma \vdash H$ *for some* Γ *and* Φ*)*

2. $\Phi, \mathcal{R}; H \vdash \Sigma$

3. *traceOK*(Σ)

4. $\langle n; \Sigma; H; e\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H';\ e'\rangle$

Then for some $\Gamma' \supseteq \Gamma$ *and* $\Phi' \equiv [\Phi^\alpha \cup \varepsilon_0; \varepsilon'; \Phi^\omega; \Phi^{\delta_i}; \Phi^{\delta_o}]$ *such that* $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^\varepsilon$*, we have:*

1. $n \vdash H', e' : \tau$ *where* $\Phi'; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}'$ *and* $n; \Gamma' \vdash H'$

2. $\Phi', \mathcal{R}'; H' \vdash \Sigma'$

3. $\mathit{traceOK}(\Sigma')$

Proof. Induction on the typing derivation $n \vdash H, e : \tau$. By inversion, we have that $\Phi; \Gamma \vdash e : \tau \leadsto \mathcal{R}$; consider each possible rule for the conclusion of this judgment:

case (TInt-TVar-TGvar-TLoc) **:**

These expressions do not reduce, so the result is vacuously true.

case (TRef) **:**

We have that:

$$(\text{TRef})\frac{\Phi; \Gamma \vdash e : \tau \leadsto \mathcal{R}}{\Phi; \Gamma \vdash \mathsf{ref}\ e : \mathit{ref}^{\,\varepsilon}\ \tau \leadsto \mathcal{R}}$$

There are two possible reductions:

case [ref] **:**

We have that $e \equiv v$, $\mathcal{R} = \cdot$, and $\langle n; (n', \sigma, \kappa); H; \mathsf{ref}\ v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma, \kappa); H'; r\rangle$ where $r \notin \mathrm{dom}(H)$ and $H' = H, r \mapsto (\cdot, v, \emptyset)$.

Let $\Gamma' = \Gamma, r : \mathit{ref}^{\,\varepsilon}\ \tau$ and $\Phi' = \Phi$ (which is acceptable since $\Phi'^{\alpha} = \Phi^{\alpha} \cup \emptyset$, $\varepsilon' \cup \emptyset \subseteq \Phi^{\alpha}$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, $\Phi'^{\delta_o} = \Phi^{\delta_o}$, and $\mathcal{R}' = \cdot$. We have part 1. as follows:

$$(\text{TSub})\frac{(\text{TLoc})\dfrac{\Gamma'(r) = \mathit{ref}^{\,\varepsilon}\ \tau}{\Phi_{\emptyset}; \Gamma' \vdash r : \mathit{ref}^{\,\varepsilon}\ \tau \leadsto \cdot} \quad \mathit{ref}^{\,\varepsilon}\ \tau \leq \mathit{ref}^{\,\varepsilon}\ \tau \quad \Phi_{\emptyset} \leq \Phi}{\Phi; \Gamma' \vdash r : \mathit{ref}^{\,\varepsilon}\ \tau \leadsto \cdot}$$

Heap well-formedness $n; \Gamma' \vdash H, r \mapsto (\cdot, v, \emptyset)$ holds since $\Phi_{\emptyset}; \Gamma' \vdash v : \tau$ follows by value typing (Lemma C.0.22) from $\Phi; \Gamma' \vdash v : \tau$, which we have by assumption and weakening; we have $n; \Gamma' \vdash H$ by weakening.

To prove 2., we must show $\Phi, \cdot; H' \vdash (n', \sigma, \kappa)$. This follows by assumption since H' only contains an additional location (i.e., not a global variable) and nothing else has changed. Part 3. follows by assumption since $\Sigma' = \Sigma$.

case [cong] **:**

We have that $\langle n; \Sigma; H; \mathsf{ref}\ \mathbb{E}[e'']\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'; H'; \mathsf{ref}\ \mathbb{E}[e''']\rangle$ from $\langle n; \Sigma; H; e''\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'; H'; e'''\rangle$. By [cong], we have $\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'; H'; e'\rangle$ where $e \equiv \mathbb{E}[e'']$ and $e' \equiv \mathbb{E}[e''']$.

By induction we have:

(i) $\Phi'; \Gamma' \vdash e' : \tau \leadsto \mathcal{R}'$ and

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi', \mathcal{R}'; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

where $\Phi'^{\alpha} = \Phi^{\alpha} \cup \varepsilon_0$, $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^{\varepsilon}$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$. We prove 1. using (ii), and applying [TRef] using (i):

$$(\text{TRef})\frac{\Phi'; \Gamma' \vdash e' : \tau \leadsto \mathcal{R}'}{\Phi'; \Gamma' \vdash \mathsf{ref}\ e' : \mathit{ref}^{\,\varepsilon}\ \tau \leadsto \mathcal{R}'}$$

Part 2. follows directly from (iii), and part 3. follows directly from (iv).

case (TDeref) **:**

We know that

$$(\text{TDeref})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \mathcal{R} \\ \Phi_2^{\varepsilon} = \varepsilon_r \qquad \Phi_2^{\delta_i} = \Phi_2^{\delta_o} \cup \varepsilon_r \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash\, !\, e : \tau \leadsto \mathcal{R}}$$

We can reduce using either [gvar-deref], [deref], or [cong].

case [GVAR-DEREF] **:**

Thus we have $e \equiv \mathsf{z}$ such that

$$\langle n; (n', \sigma, \kappa); (H'', \mathsf{z} \mapsto (\tau', v, \nu)); !\,\mathsf{z}\rangle \ \longrightarrow_{\{\mathsf{z}\}}\ \langle n; (n', \sigma \cup (\mathsf{z}, \nu), \kappa); (H'', \mathsf{z} \mapsto (\tau', v, \nu)); v\rangle$$

(where $H \equiv (H'', \mathsf{z} \mapsto (\tau', v, \nu))$), by subtyping derivations (Lemma C.0.23) we have

$$\text{(TSub)}\dfrac{\text{(TGVar)}\dfrac{\Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon'_r}\ \tau'}{\Phi_\emptyset; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon'_r}\ \tau' \leadsto \cdot} \quad \dfrac{\tau' \leq \tau \quad \tau \leq \tau' \quad \varepsilon'_r \subseteq \varepsilon_r}{\mathit{ref}^{\,\varepsilon'_r}\ \tau' \leq \mathit{ref}^{\,\varepsilon_r}\ \tau} \quad \Phi_\emptyset \leq \Phi_1}{\Phi_1; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \cdot}$$

and

$$\text{(TDeref)}\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \cdot \\ \Phi_2^\varepsilon = \varepsilon_r \qquad \Phi_2^{\delta_i} = \Phi_2^{\delta_o} \cup \varepsilon_r \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash !\,\mathsf{z} : \tau \leadsto \cdot}$$

(where $\mathcal{R} = \cdot$) and $\Phi \equiv [\Phi_1^\alpha; \Phi_1^\varepsilon \cup \varepsilon_r; \Phi_2^\omega; \Phi_1^{\delta_i}; \Phi_2^{\delta_o}]$. We have $\Phi_1^{\delta_i} = \Phi_1^{\delta_o} = \Phi_2^{\delta_i}$ and $\Phi_2^{\delta_i} = \Phi_2^{\delta_o} \cup \varepsilon_r$. Let $\Gamma' = \Gamma$, $\Phi' = [\Phi_1^\alpha \cup \{\mathsf{z}\}; \emptyset; \Phi_2^\omega; \Phi_1^{\delta_i}; \Phi_2^{\delta_o}]$ and $\mathcal{R}' = \mathcal{R} = \cdot$. Since $\mathsf{z} \in \varepsilon_r$ (by $n; \Gamma \vdash H$) we have $\emptyset \cup \{\mathsf{z}\} \subseteq (\Phi_1^\varepsilon \cup \varepsilon_r)$ hence $\varepsilon' \cup \{\mathsf{z}\} \subseteq \Phi^\varepsilon$. By the same argument we have $\{\mathsf{z}\} \subseteq \Phi_1^{\delta_i}$. The choice of Φ' is acceptable since $\Phi'^\alpha = \Phi^\alpha \cup \{\mathsf{z}\}$, $\varepsilon' \cup \{\mathsf{z}\} \subseteq \Phi^\varepsilon$, $\Phi'^\omega = \Phi^\omega$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$ and $\Phi'^{\delta_o} = \Phi^{\delta_o}$.

To prove 1., we need to show that $\Phi'; \Gamma \vdash v : \tau \leadsto \cdot$ (the rest of the premises follow by assumption of $n \vdash H, !\,\mathsf{z} : \tau$). $H(\mathsf{z}) = (\tau', v, \nu)$ and $\Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon'_r}\ \tau'$ implies $\Phi'; \Gamma \vdash v : \tau' \leadsto \cdot$ by $n; \Gamma \vdash H$. The result follows by (TSub):

$$\text{(TSub)}\dfrac{\Phi'; \Gamma \vdash v : \tau' \leadsto \cdot \qquad \tau' \leq \tau \qquad \Phi' \leq \Phi'}{\Phi'; \Gamma \vdash v : \tau \leadsto \cdot}$$

For part 2., we know $\Phi, \cdot; H \vdash (n', \sigma, \kappa)$:

$$\text{(TC1)}\dfrac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \\ \mathsf{f} \in ((\Phi_1^\varepsilon \cup \varepsilon_r) \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\Phi_1^\alpha \cup \Phi_1^{\delta_i}) \\ \kappa^\omega \supseteq (\Phi_2^\omega \cup \Phi_1^\varepsilon \cup \varepsilon_r)\end{array}}{[\Phi_1^\alpha; \Phi_1^\varepsilon \cup \varepsilon_r; \Phi_2^\omega; \Phi_1^{\delta_i}; \Phi_2^{\delta_o}], \cdot; H \vdash (n', \sigma, \kappa)}$$

and need to prove $\Phi', \cdot; H \vdash (n', \sigma \cup (\mathsf{z}, \nu), \kappa)$, hence:

$$\text{(TC1)}\dfrac{\begin{array}{c}\mathsf{f} \in (\sigma \cup (\mathsf{z}, \nu)) \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \{\mathsf{z}\} \\ \mathsf{f} \in (\emptyset \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\Phi_1^\alpha \cup \{\mathsf{z}\} \cup \Phi_1^{\delta_i}) \\ \kappa^\omega \supseteq (\Phi_2^\omega \cup \emptyset)\end{array}}{[\Phi_1^\alpha \cup \{\mathsf{z}\}; \emptyset; \Phi_2^\omega; \Phi_1^{\delta_i}; \Phi_2^{\delta_o}], \cdot; H \vdash (n', \sigma \cup (\mathsf{z}, \nu), \kappa)}$$

The first premise is true by assumption for all $\mathsf{f} \in \sigma$, and for (z, ν) since $\mathsf{z} \in \Phi_1^\alpha \cup \{\mathsf{z}\}$. The second premise is vacuously true. The third premise is true by assumption and the fact that $\{\mathsf{z}\} \subseteq \Phi_1^{\delta_i}$. The fourth premise is true by assumption.

For part 3., we need to prove $\mathit{traceOK}(n', \sigma \cup (\mathsf{z}, \nu))$; we have $\mathit{traceOK}(n', \sigma, \kappa)$ by assumption, hence need to prove that $n' \in \nu$. Since by assumption of version consistency we have that $\mathsf{f} \in \Phi_1^\varepsilon \cup \varepsilon_r \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$ and $\mathit{ver}(H, \mathsf{f}) = \mathit{ver}(H', \mathsf{f}) = \nu$ (by Lemma C.0.30), and $\{\mathsf{z}\} \subseteq \varepsilon_r$ (by $n; \Gamma \vdash H$), we have $n' \in \nu$.

case [DEREF] **:**

Follows the same argument as the [GVAR-DEREF] case above for part 1.; parts 2 and 3 follow by assumption since the trace has not changed.

case [CONG] **:**

Here $\langle n; \Sigma; H; !\,e\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; !\,e'\rangle$ follows from $\langle n; \Sigma; H, e\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H', e'\rangle$. To apply induction, we must have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ which follows by Lemma C.0.25 since $\Phi, \mathcal{R}; H \vdash \Sigma$ and $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$.

Hence we have:

(i) $\Phi'_1; \Gamma' \vdash e' : \mathit{ref}^{\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}'$
(ii) $n; \Gamma' \vdash H'$
(iii) $\Phi'_1, \mathcal{R}'; H' \vdash \Sigma'$
(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi'_1 \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1; \Phi_1^\omega; \Phi_1^{\delta_i}; \Phi_1^{\delta_o}]$. where $\varepsilon'_1 \cup \varepsilon_0 \subseteq \Phi_1^\varepsilon$. Let $\Phi'_2 = [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_r; \Phi_2^\omega; \Phi_2^{\delta_i}; \Phi_2^{\delta_o}]$ hence $\Phi_2'^{\varepsilon} = \varepsilon_r$ and $\Phi'_1 \rhd \Phi'_2 \hookrightarrow \Phi'$, where $\Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1 \cup \varepsilon_r; \Phi_2^\omega; \Phi_1^{\delta_i}; \Phi_2^{\delta_o}]$ and $(\varepsilon'_1 \cup \varepsilon_r) \cup \varepsilon_0 \subseteq (\varepsilon_1 \cup \varepsilon_r)$ hence $\Phi'^{\alpha} = \Phi^\alpha \cup \varepsilon_0$, $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^\varepsilon$, $\Phi'^{\omega} = \Phi^\omega$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required.

We prove 1. by (ii) and by applying [TDeref]:

$$\text{(TDeref)}\ \frac{\begin{array}{c}\Phi'_1; \Gamma' \vdash e' : \mathit{ref}^{\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}' \\ \Phi_2'^{\varepsilon} = \varepsilon_r \qquad \Phi_2'^{\delta_i} = \Phi_2'^{\delta_o} \cup \varepsilon_r \\ \Phi'_1 \rhd \Phi'_2 \hookrightarrow \Phi'\end{array}}{\Phi'; \Gamma' \vdash !\,e' : \tau \rightsquigarrow \mathcal{R}'}$$

The first premise follows from (i) and the second and third premises follows by definition of Φ' and Φ'_2.

To prove part 2., we must show that $\Phi', \mathcal{R}'; H' \vdash \Sigma'$. We have two cases:

$\Sigma' \equiv (n', \sigma, \kappa)$: By (iii) we must have $\mathcal{R}' \equiv \cdot$ such that

$$\text{(TC1)}\ \frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon'_1 \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi_1^\alpha \cup \Phi_1^{\delta_i}) \\ \kappa^{\omega\prime} \supseteq (\Phi_1^\omega \cup \varepsilon'_1)\end{array}}{[\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1; \Phi_1^\omega; \Phi_1^{\delta_i}; \Phi_1^{\delta_o}], \cdot; H' \vdash (n', \sigma, \kappa)}$$

To achieve the desired result we need to prove:

$$\text{(TC1)}\ \frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in ((\varepsilon'_1 \cup \varepsilon_r) \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi_1^\alpha \cup \Phi_1^{\delta_i} \\ \kappa^{\omega\prime} \supseteq (\Phi_2^\omega \cup \varepsilon'_1 \cup \varepsilon_r)\end{array}}{[\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1 \cup \varepsilon_r; \Phi_2^\omega; \Phi_1^{\delta_i}; \Phi_2^{\delta_o}], \cdot; H' \vdash (n', \sigma, \kappa)}$$

The first premise follows directly from (iii). To prove the second premise, we observe that by Lemma C.0.27, $\mathit{top}(\Sigma) = (n', \sigma', \kappa')$ where $\sigma' \subseteq \sigma$, and by inversion on $\Phi; \mathcal{R}; H \vdash \Sigma$ we know $\mathsf{f} \in \varepsilon_1 \cup \varepsilon_r \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$. Then the second premise follows because for all f, $\mathit{ver}(H, \mathsf{f}) = \mathit{ver}(H', \mathsf{f})$ by Lemma C.0.30. The third premise follows directly by assumption. The fourth premise follows by assumption and the fact that $\Phi_1^\omega \equiv \Phi^\omega \cup \varepsilon_r$.

$\Sigma' \equiv (n', \sigma, \kappa), \Sigma''$: By (iii), we must have $\mathcal{R}' \equiv \Phi''', \mathcal{R}'''$ such that

$$\text{(TC2)}\ \frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi'_1 \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1; \Phi_1^\omega; \Phi_1^{\delta_i}; \Phi_1^{\delta_o}] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon'_1 \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi_1^\alpha \cup \Phi_1^{\delta_i}) \\ \kappa^{\omega\prime} \supseteq (\Phi_1^\omega \cup \varepsilon'_1)\end{array}}{\Phi'_1, \Phi''', \mathcal{R}'''; H' \vdash (n', \sigma, \kappa), \Sigma''}$$

We wish to show that

$$\text{(TC2)}\ \frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1 \cup \varepsilon_r; \Phi_2^\omega; \Phi_2^{\delta_i}; \Phi_2^{\delta_o}] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in ((\varepsilon'_1 \cup \varepsilon_r) \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi_1^\alpha \cup \Phi_1^{\delta_i}) \\ \kappa^{\omega\prime} \supseteq (\Phi^\omega \cup \varepsilon'_1 \cup \varepsilon_r)\end{array}}{\Phi', \Phi''', \mathcal{R}'''; H' \vdash (n', \sigma, \kappa), \Sigma''}$$

The first and third premises follow from (iii), while the fourth, fifth and sixth premises follows by the same argument as in the $\Sigma' \equiv (n', \sigma, \kappa)$ case, above.

Part 3. follows directly from (iv).

case (TAssign) **:**

We know that:

$$(\textsc{TAssign})\dfrac{\begin{array}{c}\Phi_1;\Gamma\vdash e_1 : \mathit{ref}^{\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}_1 \qquad \Phi_2;\Gamma\vdash e_2:\tau\rightsquigarrow\mathcal{R}_2\\ \Phi_3^{\varepsilon}=\varepsilon_r \qquad \Phi_3^{\delta_i}=\Phi_3^{\delta_o}\cup\varepsilon_r\\ \Phi_1\rhd\Phi_2\rhd\Phi_3\hookrightarrow\Phi\end{array}}{\Phi;\Gamma\vdash e_1:=e_2:\tau\rightsquigarrow\mathcal{R}_1\bowtie\mathcal{R}_2}$$

From $\mathcal{R}_1 \bowtie \mathcal{R}_2$ it follows that either $\mathcal{R}_1 \equiv \cdot$ or $\mathcal{R}_2 \equiv \cdot$.

We can reduce using [gvar-assign], [assign], or [cong].

case [gvar-assign] **:**

This implies that $e \equiv \mathsf{z} := v$ with

$$\langle n; (n', \sigma, \kappa); (H'', \mathsf{z} \mapsto (\tau, v', \nu)); \mathsf{z} := v\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu), \kappa); (H'', \mathsf{z} \mapsto (\tau, v, \nu)); v\rangle$$

where $H \equiv (H'', \mathsf{z} \mapsto (\tau, v', \nu))$. $\mathcal{R}_1 \equiv \cdot$ and $\mathcal{R}_2 \equiv \cdot$ (thus $\mathcal{R}_1 \bowtie \mathcal{R}_2 \equiv \cdot$).

Let $\Gamma' = \Gamma$, $\mathcal{R}' = \cdot$, and $\Phi' = [\Phi^{\alpha} \cup \{\mathsf{z}\}; \emptyset; \Phi^{\omega}; \Phi_1^{\delta_i}; \Phi_2^{\delta_o}]$. Since $\mathsf{z} \in \varepsilon_r$ (by $n;\Gamma \vdash H$) we have $\emptyset \subseteq (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_r)$, hence $\emptyset \cup \{\mathsf{z}\} \subseteq (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_r)$ which means $\varepsilon' \cup \{\mathsf{z}\} \subseteq \Phi^{\varepsilon}$. By the same argument we have $\{\mathsf{z}\} \subseteq \Phi_1^{\delta_i}$. The choice of Φ' is acceptable since $\Phi'^{\alpha} = \Phi^{\alpha} \cup \{\mathsf{z}\}$, $\varepsilon' \cup \{\mathsf{z}\} \subseteq \Phi^{\varepsilon}$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$ and $\Phi'^{\delta_o} = \Phi^{\delta_o}$.

We prove 1. as follows. Since $\Phi_2;\Gamma \vdash v : \tau \rightsquigarrow \cdot$, by value typing (Lemma C.0.22) we have $\Phi';\Gamma \vdash v : \tau \rightsquigarrow \cdot$. $n;\Gamma \vdash H'$ follows from $n;\Gamma \vdash H$ and $\Phi';\Gamma \vdash v : \tau \rightsquigarrow \cdot$ (since $\Phi^{\varepsilon} = \emptyset$).

Parts 2. and 3. are similar to the (TDeref) case.

case [assign] **:**

Part 1. is similar to (gvar-assign); we have parts 2. and 3. by assumption.

case [cong] **:**

Consider the shape of $\mathbb{E}$:

case $\mathbb{E} := e$ **:**

$\langle n; \Sigma; H; e_1 := e_2\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'; H'; e_1' := e_2\rangle$ follows from $\langle n; \Sigma; H; e_1\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'; H'; e_1'\rangle$. Since $e_1 \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot$ by assumption, we have $\mathcal{R} \equiv \mathcal{R}_1$. To apply induction we must show $\Phi_1, \mathcal{R}_{\infty}; H \vdash \Sigma$ This follows by an argument similar to Lemma C.0.25, because $\Phi_1^{\alpha} \equiv \Phi^{\alpha}$, $\Phi_1^{\delta_i} \equiv \Phi^{\delta_i}$, and $\Phi_1^{\omega} = \Phi^{\omega} \cup \Phi_2^{\varepsilon} \cup \varepsilon_r$ hence $\kappa^{\alpha} \supseteq (\Phi^{\alpha} \cup \Phi^{\delta_i})$ implies $\kappa^{\alpha} \supseteq (\Phi_1^{\alpha} \cup \Phi_1^{\delta_i})$ and $\kappa^{\omega} \supseteq (\Phi^{\omega} \cup \Phi_1^{\varepsilon} \cup \Phi_2^{\varepsilon} \cup \varepsilon_r)$ implies $\kappa^{\omega} \supseteq (\Phi_1^{\omega} \cup \Phi_1^{\varepsilon})$.

Hence by induction we have

(i) $\Phi_1';\Gamma' \vdash e_1' : \mathit{ref}^{\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}_1'$ and

(ii) $n;\Gamma' \vdash H'$

(iii) $\Phi_1', \mathcal{R}_1'; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi_1' \equiv [\Phi_1^{\alpha} \cup \varepsilon_0; \varepsilon_1'; \Phi_1^{\omega}; \Phi_1^{\delta_i}; \Phi_1^{\delta_o}]$ where $\varepsilon_1' \cup \varepsilon_0 \subseteq \varepsilon_1$ and $\Phi_1^{\omega} \equiv \Phi_2^{\varepsilon} \cup \varepsilon_r \cup \Phi_3^{\omega}$.

Let $\begin{array}{lcl}\Phi_2' & \equiv & [\Phi_1^{\alpha} \cup \varepsilon_1' \cup \varepsilon_0; \Phi_2^{\varepsilon}; \varepsilon_r \cup \Phi_3^{\omega}; \Phi_2^{\delta_i}; \Phi_2^{\delta_o}]\\ \Phi_3' & \equiv & [\Phi_1^{\alpha} \cup \varepsilon_1' \cup \varepsilon_0 \cup \Phi_2^{\varepsilon}; \varepsilon_r; \Phi_3^{\omega}; \Phi_3^{\delta_i}; \Phi_3^{\delta_o}]\end{array}$

Thus $\Phi_3'^{\varepsilon} = \varepsilon_r$ and $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'$ such that $\Phi' \equiv [\Phi_1^{\alpha} \cup \varepsilon_0; \varepsilon_1' \cup \Phi_2^{\varepsilon} \cup \varepsilon_r; \Phi_3^{\omega}; \Phi_1^{\delta_i}; \Phi_3^{\delta_o}]$

The choice of Φ' is acceptable since $\Phi'^{\alpha} = \Phi^{\alpha} \cup \varepsilon_0$, $(\varepsilon_1' \cup \varepsilon_r \cup \varepsilon_2) \cup \varepsilon_0 \subseteq (\varepsilon_1 \cup \varepsilon_r \cup \varepsilon_2)$ i.e., $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^{\varepsilon}$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required).

To prove 1., we have $n;\Gamma' \vdash H'$ by (ii), and apply (TAssign):

$$(\textsc{TAssign})\dfrac{\begin{array}{c}\Phi_1';\Gamma' \vdash e_1' : \mathit{ref}^{\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}_1'\\ (\textsc{TSub})\dfrac{\Phi_2;\Gamma' \vdash e_2 : \tau \rightsquigarrow \mathcal{R}_2 \qquad \tau \leq \tau \qquad \dfrac{\begin{array}{c}\Phi_1^{\alpha} \cup \varepsilon_1' \cup \varepsilon_0 \subseteq \Phi_1^{\alpha} \cup \Phi_1^{\varepsilon}\\ \Phi_2^{\varepsilon} \subseteq \Phi_2^{\varepsilon}\\ \varepsilon_r \cup \Phi_3^{\omega} \subseteq \varepsilon_r \cup \Phi_3^{\omega}\\ \Phi_2'^{\delta_i} = \Phi_2^{\delta_i} \qquad \Phi_2'^{\delta_o} = \Phi_2^{\delta_o}\end{array}}{\Phi_2 \leq \Phi_2'}}{\Phi_2';\Gamma' \vdash e_2 : \tau \rightsquigarrow \mathcal{R}_2}\\ \Phi_3'^{\varepsilon} = \varepsilon_r \qquad \Phi_3'^{\delta_i} = \Phi_3'^{\delta_o} \cup \varepsilon_r\\ \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'\end{array}}{\Phi';\Gamma' \vdash e_1 := e_2 : \tau \rightsquigarrow \mathcal{R}_1' \bowtie \mathcal{R}_2}$$

Note that $\Phi_2;\Gamma' \vdash e_2 : \tau$ follows from $\Phi_2;\Gamma \vdash e_2 : \tau$ by weakening (Lemma C.0.19).

To prove part 2., we must show that $\Phi', \mathcal{R}_1'; H' \vdash \Sigma'$ (since $\mathcal{R}_1' \bowtie \mathcal{R}_2 = \mathcal{R}_1'$). By inversion on $\Phi, \mathcal{R}; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$ or $\Sigma \equiv (n', \sigma, \kappa), \Sigma''$. We have two cases:

$\Sigma' \equiv (n', \sigma, \kappa)$: By (iii) we must have $\mathcal{R}'_1 \equiv \cdot$ such that

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon'_1 \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi_1^\alpha \cup \Phi_1^{\delta_i}) \\ \kappa^{\omega\prime} \supseteq (\Phi_1^\omega \cup \varepsilon'_1)\end{array}}{[\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1; \Phi_1^\omega; \Phi_1^{\delta_i}; \Phi_1^{\delta_o}], \cdot; H' \vdash (n', \sigma, \kappa)}$$

To achieve the desired result we need to prove:

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in ((\varepsilon'_1 \cup \Phi_2^\varepsilon \cup \varepsilon_r) \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi_1^\alpha \cup \Phi_1^{\delta_i}) \\ \kappa^{\omega\prime} \supseteq (\omega \cup \varepsilon'_1 \cup \Phi_2^\varepsilon \cup \varepsilon_r)\end{array}}{[\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1 \cup \Phi_2^\varepsilon \cup \varepsilon_r; \Phi_3^\omega; \Phi_1^{\delta_i}; \Phi_3^{\delta_o}], \cdot; H' \vdash (n', \sigma, \kappa)}$$

The first premise follows directly from (iii). To prove the second premise, we observe that by Lemma C.0.27, $\mathit{top}(\Sigma) = (n', \sigma', \kappa')$ where $\sigma' \subseteq \sigma$, and by inversion on $\Phi; \mathcal{R}; H \vdash \Sigma$ we know (a) $\mathsf{f} \in \sigma' \Rightarrow \mathsf{f} \in \Phi_1^\alpha$, and (b) $\mathsf{f} \in \Phi_1^\varepsilon \cup \Phi_2^\varepsilon \cup \varepsilon_r \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$. The second premise follows from (iii) and the fact that $\mathsf{f} \in (\varepsilon_r \cup \Phi_2^\varepsilon) \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$ by (b), and for all f, $\mathit{ver}(H, \mathsf{f}) = \mathit{ver}(H', \mathsf{f})$ by Lemma C.0.30. The third premise follows directly by assumption. The fourth premise follows by assumption and the fact that $\Phi_1^\omega \equiv \omega \cup \Phi_2^\varepsilon \cup \varepsilon_r$.

$\Sigma' \equiv (n', \sigma, \kappa), \Sigma''$: By (iii), we must have $\mathcal{R}'_1 \equiv \Phi''', \mathcal{R}'''$ such that

$$(\text{TC2})\frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi'_1 \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1; \Phi_1^\omega; \Phi_1^{\delta_i}; \Phi_1^{\delta_o}] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon'_1 \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi_1^\alpha \cup \Phi_1^{\delta_i}) \\ \kappa^{\omega\prime} \supseteq (\Phi_1^\omega \cup \varepsilon'_1)\end{array}}{\Phi'_1, \Phi''', \mathcal{R}'''; H' \vdash (n', \sigma, \kappa), \Sigma''}$$

We wish to show that

$$(\text{TC2})\frac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1 \cup \Phi_2^\varepsilon \cup \varepsilon_r; \Phi_3^\omega; \Phi_1^{\delta_i}; \Phi_3^{\delta_o}] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi_1^\alpha \cup \varepsilon_0 \\ \mathsf{f} \in ((\varepsilon'_1 \cup \Phi_2^\varepsilon \cup \varepsilon_r) \cap \Phi_1^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi_1^\alpha \cup \Phi_1^{\delta_i}) \\ \kappa^{\omega\prime} \supseteq (\omega \cup \varepsilon'_1 \cup \Phi_2^\varepsilon \cup \varepsilon_r)\end{array}}{\Phi', \Phi''', \mathcal{R}'''; H' \vdash (n', \sigma, \kappa), \Sigma''}$$

The first and third premises follow from (iii), while the fourth, fifth and sixth premises follows by the same argument as in the $\Sigma' \equiv (n', \sigma, \kappa)$ case, above.

Part 3. follows directly from (iv).

case $r := \mathbb{E}$ **:**

$\langle n; \Sigma; H; r := e_2\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; r := e'_2\rangle$ follows from $\langle n; \Sigma; H; e_2\rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e'_2\rangle$. Since $e_1 \equiv r$, by inversion $\mathcal{R}_1 \equiv \cdot$. and we have $\mathcal{R} \equiv \mathcal{R}_2$. To apply induction we must show $\Phi_2, \mathcal{R}_\in; H \vdash \Sigma$. This follows by an argument similar to (TDeref)-[cong], because $\Phi_2^\alpha \equiv \Phi_1^\alpha \equiv \Phi^\alpha$, $\Phi_2^{\delta_i} \equiv \Phi_1^{\delta_i} \equiv \Phi^{\delta_i}$, and $\Phi_2^\omega = \Phi_3^\omega \cup \varepsilon_r$ hence $\kappa^\alpha \supseteq (\Phi^\alpha \cup \Phi^{\delta_i})$ implies $\kappa^\alpha \supseteq (\Phi_2^\alpha \cup \Phi_2^{\delta_i})$ and $\kappa^\omega \supseteq (\Phi^\omega \cup \Phi^\varepsilon)$ implies $\kappa^\omega \supseteq (\Phi_2^\omega \cup \Phi_2^\varepsilon)$.

(i) $\Phi'_2; \Gamma' \vdash e'_2 : \tau \rightsquigarrow \mathcal{R}'_2$

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi'_2, \mathcal{R}'_2; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi'_2 \equiv [\Phi_2^\alpha \cup \varepsilon_0; \varepsilon'_2; \Phi_2^\omega; \Phi_2^{\delta_i}; \Phi_2^{\delta_o}]$ where $(\varepsilon'_2 \cup \varepsilon_0) \subseteq \Phi_2^\varepsilon$; note $\Phi_2^\alpha \equiv \Phi_1^\alpha$ (since $\Phi_1^\varepsilon \equiv \emptyset$) and $\Phi_2^\omega \equiv \varepsilon_3 \cup \Phi_3^\omega$.

Let $\begin{array}{lcl}\Phi'_1 & \equiv & [\Phi_1^\alpha \cup \varepsilon_0; \emptyset; \varepsilon'_2 \cup \varepsilon_r \cup \Phi_3^\omega; \Phi_2^{\delta_i}; \Phi_2^{\delta_o}] \\ \Phi'_3 & \equiv & [\Phi_1^\alpha \cup \varepsilon_0 \cup \varepsilon'_2; \varepsilon_r; \Phi_3^\omega; \Phi_3^{\delta_i}; \Phi_3^{\delta_o}]\end{array}$

Thus $\Phi'^{\varepsilon}_3 = \varepsilon_r$ and $\Phi'_1 \rhd \Phi'_2 \rhd \Phi'_3 \hookrightarrow \Phi'$ such that $\Phi' \equiv [\Phi^{\alpha}_1 \cup \varepsilon_0; \varepsilon'_2 \cup \varepsilon_r; \Phi^{\omega}_3; \Phi^{\delta_i}_1; \Phi^{\delta_o}_3]$ and $(\varepsilon'_2 \cup \varepsilon_r) \cup \varepsilon_0 \subseteq (\Phi^{\varepsilon}_2 \cup \varepsilon_r)$. The choice of Φ' is acceptable since $\Phi'^{\alpha} = \Phi^{\alpha} \cup \varepsilon_0$, $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^{\varepsilon}$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$ and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required.
To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and we can apply [TAssign]:

$$(\text{TAssign})\dfrac{\begin{array}{c}\Phi'_1; \Gamma' \vdash r : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \cdot \qquad \Phi'_2; \Gamma' \vdash e'_2 : \tau \leadsto \mathcal{R}'_2 \\ \Phi'^{\varepsilon_r}_3 = \varepsilon_r \qquad \Phi'^{\delta_i}_3 = \Phi'^{\delta_o}_3 \cup \varepsilon_r \\ \Phi'_1 \rhd \Phi'_2 \rhd \Phi'_3 \hookrightarrow \Phi'\end{array}}{\Phi'; \Gamma' \vdash r := e'_2 : \tau \leadsto \cdot \bowtie \mathcal{R}'_2}$$

Note that we have $\Phi'_1; \Gamma' \vdash r : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \cdot$ from $\Phi_1; \Gamma \vdash r : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \cdot$ by value typing and weakening
To prove part 2., we must show that $\Phi', \mathcal{R}'_2; H' \vdash \Sigma'$ (since $\mathcal{R}_1 \bowtie \mathcal{R}_2 = \mathcal{R}'_2$). By inversion on $\Phi, \mathcal{R}; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$ or $\Sigma \equiv (n', \sigma, \kappa), \Sigma''$. We have two cases:

$\Sigma' \equiv (n', \sigma, \kappa)$: By (iii) we must have $\mathcal{R}'_2 \equiv \cdot$ such that

$$(\text{TC1})\dfrac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi^{\alpha}_2 \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon'_2 \cap \Phi^{\delta_i}_2) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi^{\alpha}_2 \cup \Phi^{\delta_i}_2) \\ \kappa^{\omega\prime} \supseteq (\Phi^{\omega}_2 \cup \varepsilon'_2)\end{array}}{[\Phi^{\alpha}_2 \cup \varepsilon_0; \varepsilon'_2; \Phi^{\omega}_2; \Phi^{\delta_i}_2; \Phi^{\delta_o}_2], \cdot; H' \vdash (n', \sigma, \kappa)}$$

To achieve the desired result we need to prove:

$$(\text{TC1})\dfrac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi^{\alpha}_1 \cup \varepsilon_0 \\ \mathsf{f} \in ((\varepsilon_r \cup \varepsilon'_2) \cap \Phi^{\delta_i}_2) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi^{\alpha}_1 \cup \Phi^{\delta_i}_1) \\ \kappa^{\omega\prime} \supseteq (\Phi^{\omega} \cup \varepsilon'_2 \cup \varepsilon_r)\end{array}}{[\Phi^{\alpha}_1 \cup \varepsilon_0; \varepsilon'_2 \cup \varepsilon_r; \Phi^{\omega}_3; \Phi^{\delta_i}_1; \Phi^{\delta_o}_3], \cdot; H' \vdash (n', \sigma, \kappa)}$$

The first premise follows from (iii) since $\Phi^{\alpha}_1 = \Phi^{\alpha}_2$.
To prove the second premise, we observe that by Lemma C.0.27, $\mathit{top}(\Sigma) = (n', \sigma', \kappa)$ where $\sigma' \subseteq \sigma$, and by inversion on $\Phi; \mathcal{R}; H \vdash \Sigma$ we know $\mathsf{f} \in \varepsilon_1 \cup \varepsilon_r \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$. The second premise follows because we have $\mathsf{f} \in ((\varepsilon_1 \cup \varepsilon_r) \cap \delta_i) \Rightarrow n' \in \mathit{ver}(H, \mathsf{f})$ by assumption and for all f, $\mathit{ver}(H, \mathsf{f}) = \mathit{ver}(H', \mathsf{f})$ by Lemma C.0.30.
The third premise follows directly by assumption since $\Phi^{\alpha}_1 = \Phi^{\alpha}_2$ and $\Phi^{\delta_i}_1 = \Phi^{\delta_i}_2$. The fourth premise follows by assumption and the fact that $\Phi^{\omega}_2 \equiv \Phi^{\omega} \cup \varepsilon_r$.

$\Sigma' \equiv (n', \sigma, \kappa), \Sigma''$: By (iii), we must have $\mathcal{R}'_2 \equiv \Phi''', \mathcal{R}'''$ such that:

$$(\text{TC2})\dfrac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi'_2 \equiv [\Phi^{\alpha}_2 \cup \varepsilon_0; \varepsilon'_2; \Phi^{\omega}_2; \Phi^{\delta_i}_2; \Phi^{\delta_o}_2] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi^{\alpha}_2 \cup \varepsilon_0 \\ \mathsf{f} \in (\varepsilon'_2 \cap \Phi^{\delta_i}_2) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi^{\alpha}_2 \cup \Phi^{\delta_i}_2) \\ \kappa^{\omega\prime} \supseteq (\Phi^{\omega}_2 \cup \varepsilon'_2)\end{array}}{\Phi'_2, \Phi''', \mathcal{R}'''; H' \vdash (n', \sigma, \kappa), \Sigma''}$$

We wish to show that

$$(\text{TC2})\dfrac{\begin{array}{c}\Phi''', \mathcal{R}'''; H' \vdash \Sigma'' \\ \Phi' \equiv [\Phi^{\alpha}_1 \cup \varepsilon_0; \varepsilon'_2 \cup \varepsilon_r; \Phi^{\omega}_3; \Phi^{\delta_i}_1; \Phi^{\delta_o}_3] \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \cup \varepsilon_0 \\ \mathsf{f} \in ((\varepsilon'_2 \cup \varepsilon_r) \cap \Phi^{\delta_i}_2) \Rightarrow n' \in \mathit{ver}(H', \mathsf{f}) \\ \kappa^{\alpha\prime} \supseteq (\Phi^{\alpha}_1 \cup \Phi^{\delta_i}_1) \\ \kappa^{\omega\prime} \supseteq (\Phi^{\omega} \cup \varepsilon'_2 \cup \varepsilon_r)\end{array}}{\Phi', \Phi''', \mathcal{R}'''; H' \vdash (n', \sigma, \kappa), \Sigma''}$$

The first and third premises follow from (iii), while the fourth, fifth and sixth premises follow by the same argument as in the $\Sigma' \equiv (n', \sigma, \kappa)$ case, above.

Part 3. follows directly from (iv).

case (TCheckin) **:**

We know that:

$$(\text{TCheckin})\frac{\alpha \cup \delta_o \subseteq \alpha' \quad \omega \subseteq \omega'}{[\alpha; \emptyset; \omega; \emptyset; \delta_o]; \Gamma \vdash \mathsf{checkin}^{\alpha',\omega'} : int \rightsquigarrow \cdot}$$

case [checkin] **:**

Thus we must have:

$$\langle n; (n', \sigma, \kappa); H; \mathsf{checkin}^{(\alpha',\omega')} \rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma', (\alpha', \omega')); H; 1 \rangle$$

Let $\Gamma' = \Gamma$ and $\Phi' = \Phi$ (and thus $\varepsilon' \cup \emptyset \subseteq \Phi^{\varepsilon}$, $\Phi'^{\alpha} = \Phi^{\alpha} \cup \emptyset$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$) as required. For 1., $\Phi'; \Gamma \vdash 1 : int \rightsquigarrow \cdot$ follows from (TInt) and value typing and $n; \Gamma \vdash H$ is true by assumption. For part 2., we know

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\emptyset \cap \emptyset) \Rightarrow n' \in ver(H, \mathsf{f}) \\ \kappa^{\alpha} \supseteq (\alpha \cup \emptyset) \\ \kappa^{\omega} \supseteq (\omega \cup \emptyset)\end{array}}{[\alpha; \emptyset; \omega; \emptyset; \delta_o], \cdot; H \vdash (n', \sigma, \kappa')}$$

and need to prove:

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\emptyset \cap \emptyset) \Rightarrow n' \in ver(H, \mathsf{f}) \\ \kappa^{\alpha'} \supseteq (\alpha \cup \emptyset) \\ \kappa^{\omega'} \supseteq (\omega \cup \emptyset)\end{array}}{[\alpha; \emptyset; \omega; \emptyset; \delta_o], \cdot; H \vdash (n', \sigma, \kappa)}$$

The first premise is true by assumption. The second is vacuously true. The third and fourth premises follow since we know that $\kappa^{\alpha'} \supseteq \alpha \cup \delta_o$ and $\kappa^{\omega'} \supseteq \omega$ by assumption.

Part 3. follows by assumption.

case (TIf) **:**

We know that:

$$(\text{TIf})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : int \rightsquigarrow \mathcal{R} \\ \Phi_2; \Gamma \vdash e_2 : \tau \rightsquigarrow \cdot \quad \Phi_2; \Gamma \vdash e_3 : \tau \rightsquigarrow \cdot \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \rightsquigarrow \mathcal{R}}$$

We can reduce using [if-t], [if-f] or [cong].

case [if-t] **:**

This implies that $e_1 \equiv v$ hence $\mathcal{R} = \cdot$. We have

$$\langle n; (n', \sigma, \kappa); H; \mathsf{if0}\ v\ \mathsf{then}\ e2\ \mathsf{else}\ e3 \rangle \longrightarrow \langle n; (n', \sigma, \kappa); H; e2 \rangle$$

We have $\Phi_2 = \Phi$ (because $\Phi_1^{\varepsilon} \equiv \emptyset$; if $\Phi_1^{\varepsilon} \not\equiv \emptyset$ we can rewrite the derivation using value typing to make it so). Let $\Gamma' = \Gamma$ and $\Phi' = \Phi$ (and thus $\varepsilon \cup \emptyset \subseteq \Phi^{\varepsilon}$, $\Phi'^{\alpha} = \Phi^{\alpha} \cup \emptyset$, $\Phi'^{\omega} = \Phi^{\omega}$, and $\Phi'^{\delta_i} = \Phi^{\delta_i}$, $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required). To prove 1., we have $n; \Gamma \vdash H$ and $\Phi; \Gamma \vdash e_2 : \tau \rightsquigarrow \cdot$ by assumption.

Parts 2. and 3. also follow by assumption.

case [if-f] **:**

This is similar to [if-t].

case [cong] **:**

$\langle n; \Sigma; H; \mathsf{if0}\ e_1\ \mathsf{then}\ e2\ \mathsf{else}\ e3 \rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'; H'; \mathsf{if0}\ e_1'\ \mathsf{then}\ e2\ \mathsf{else}\ e3 \rangle$ follows from $\langle n; \Sigma; H; e_1 \rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'; H'; e_1' \rangle$. To apply induction, we must have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ which follows by Lemma C.0.25 since $\Phi, \mathcal{R}; H \vdash \Sigma$ and $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$.

(i) $\Phi_1'; \Gamma' \vdash e_1' : int \rightsquigarrow \mathcal{R}'$ and

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi_1', \mathcal{R}'; H' \vdash \Sigma'$

(iv) $traceOK(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi'_1 \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1; \Phi_1^\omega; \Phi_1^{\delta_i}; \Phi_1^{\delta_o}]$ where $\varepsilon'_1 \cup \varepsilon_0 \subseteq \Phi_1^\varepsilon$. (Note that $\Phi_1^\omega \equiv \Phi_2^\varepsilon \cup \Phi_2^\omega$.)

Let $\Phi'_2 \equiv [\Phi_1^\alpha \cup \varepsilon'_1 \cup \varepsilon_0; \Phi_2^\varepsilon; \Phi_2^\omega; \Phi_2^{\delta_i}; \Phi_2^{\delta_o}]$. Thus $\Phi'_1 \rhd \Phi'_2 \hookrightarrow \Phi'$ so that $\Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon'_1 \cup \Phi_2^\varepsilon; \Phi_2^\omega; \Phi_1^{\delta_i}; \Phi_2^{\delta_o}]$ where $\varepsilon'_1 \cup \varepsilon_0 \cup \Phi_2^\varepsilon \subseteq \Phi_1^\varepsilon \cup \Phi_2^\varepsilon$, $\Phi'^\omega = \Phi^\omega$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required.

To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and can apply (TIf): We prove 1. by (ii) and as follows:

$$\text{(TIF)}\frac{\text{(TSUB)}\dfrac{\Phi_2; \Gamma' \vdash e_2 : \tau \rightsquigarrow \cdot \quad \tau \leq \tau \quad \Phi_2 \leq \Phi'_2}{\Phi'_2; \Gamma' \vdash e_2 : \tau \rightsquigarrow \cdot} \quad \text{(TSUB)}\dfrac{\Phi_2; \Gamma' \vdash e_2 : \tau \rightsquigarrow \cdot \quad \tau \leq \tau \quad \Phi_2 \leq \Phi'_2}{\Phi'_2; \Gamma' \vdash e_3 : \tau \rightsquigarrow \cdot} \quad \Phi'_1; \Gamma' \vdash e'_1 : \mathit{int} \rightsquigarrow \mathcal{R}'_1 \quad \Phi'_1 \rhd \Phi'_2 \hookrightarrow \Phi'}{\Phi'; \Gamma' \vdash \mathsf{if0}\ e'_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \rightsquigarrow \mathcal{R}'}$$

Note that $\Phi_2; \Gamma' \vdash e_2 : \tau \rightsquigarrow \mathcal{R}$ follows from $\Phi_2; \Gamma \vdash e_2 : \tau \rightsquigarrow \mathcal{R}$ by weakening (Lemma C.0.19) and likewise for $\Phi_2; \Gamma' \vdash e_3 : \tau \rightsquigarrow \mathcal{R}$.

Parts 2. and 3. follow by an argument similar to (TDeref)-[cong] and (TAssign)-[cong].

case (TTransact) **:**

We know that:

$$\text{(TTRANSACT)}\frac{\Phi''; \Gamma \vdash e : \tau \rightsquigarrow \cdot \quad \Phi^\alpha \subseteq \Phi''^\alpha \quad \Phi^\omega \subseteq \Phi''^\omega}{\Phi; \Gamma \vdash \mathsf{tx}^{(\Phi''^\alpha \cup \Phi''^{\delta_i}, \Phi''^\omega \cup \Phi''^\varepsilon)}\ e : \tau \rightsquigarrow \cdot}$$

We can reduce using [tx-start]:

$$\langle n; (n', \sigma, \kappa); H; \mathsf{tx}^{\kappa'}\ e\rangle \longrightarrow_\emptyset \langle n; (n', \sigma, \kappa), (n, \emptyset, (\Phi''^\alpha \cup \Phi''^{\delta_i}, \Phi''^\omega \cup \Phi''^\varepsilon)); H; \mathsf{intx}\ e\rangle$$

where $\kappa' \equiv (\Phi''^\alpha \cup \Phi''^{\delta_i}, \Phi''^\omega \cup \Phi''^\varepsilon)$. Let $\Gamma' = \Gamma$ and $\Phi' \equiv [\Phi^\alpha; \emptyset; \Phi^\omega; \Phi^{\delta_i}; \Phi^{\delta_o}]$ (and thus $\varepsilon' \cup \emptyset \subseteq \Phi^\varepsilon$, $\Phi'^\alpha = \Phi^\alpha \cup \emptyset$, $\Phi'^\omega = \Phi^\omega$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required). To prove 1., we have $n; \Gamma \vdash H$ by assumption, and the rest follows by (TIntrans):

$$\text{(TINTRANS)}\frac{\Phi''; \Gamma \vdash e : \tau \rightsquigarrow \cdot \quad \Phi'^\alpha \subseteq \Phi''^\alpha \quad \Phi'^\omega \subseteq \Phi''^\omega}{\Phi'; \Gamma \vdash \mathsf{intx}\ e : \tau \rightsquigarrow \Phi'', \cdot}$$

The first premise is true by assumption, and the rest are true by choice of Φ'.

We prove 2. as follows:

$$\text{(TC1)}\frac{\begin{array}{c}\mathsf{f} \in \emptyset \Rightarrow \mathsf{f} \in \Phi''^\alpha \\ \mathsf{f} \in (\Phi''^\varepsilon \cap \Phi''^{\delta_i}) \Rightarrow n \in \mathit{ver}(H, \mathsf{f}) \\ \Phi''^\alpha \cup \Phi''^{\delta_i} \supseteq \Phi''^\alpha \cup \Phi''^{\delta_i} \\ \Phi''^\omega \cup \Phi''^\varepsilon \supseteq \Phi''^\omega \cup \Phi''^\varepsilon\end{array}}{\Phi'', \cdot; H \vdash (n, \emptyset, (\Phi''^\alpha \cup \Phi''^{\delta_i}, \Phi''^\omega \cup \Phi''^\varepsilon))}$$

The first premise is true vacuously, the second is true by $n; \Gamma \vdash H$ (which we have by assumption), and the third and fourth trivially hold.

$$\text{(TC2)}\frac{\begin{array}{c}\Phi'', \cdot; H \vdash (n, \emptyset, \kappa') \\ \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \Phi^\alpha \\ \mathsf{f} \in (\emptyset \cap \Phi^{\delta_i}) \Rightarrow n' \in \mathit{ver}(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\Phi^\alpha \cup \Phi^{\delta_i}) \\ \kappa^\omega \supseteq (\Phi^\omega \cup \emptyset)\end{array}}{[\Phi^\alpha; \emptyset; \Phi^\omega; \Phi^{\delta_i}; \Phi^{\delta_o}], \Phi'', \cdot; H \vdash (n', \sigma, \kappa), (n, \emptyset, \kappa')}$$

We have proved the first premise above, the second premise holds vacuously, and the rest hold by inversion of $\Phi, \cdot; H \vdash (n', \sigma, \kappa)$.

Part 3. follows easily: we have $\mathit{traceOK}((n', \sigma, \kappa))$ by assumption, $\mathit{traceOK}((n, \emptyset, \kappa'))$ is vacuously true, hence $\mathit{traceOK}((n', \sigma, \kappa), (n, \emptyset, \kappa'))$ is true.

case (TINTRANS) **:**

We know that:

$$(\text{TINTRANS})\,\frac{\Phi''; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R} \quad \Phi^{\alpha} \subseteq \Phi''^{\alpha} \quad \Phi^{\omega} \subseteq \Phi''^{\omega}}{\Phi; \Gamma \vdash \mathsf{intx}\ e : \tau \rightsquigarrow \Phi'', \mathcal{R}}$$

There are two possible reductions:

case [TX-END] **:**

We have that $e \equiv v$ and thus $\mathcal{R} \equiv \cdot$; we reduce as follows:

$$\frac{\mathit{traceOK}(n'', \sigma'', \kappa'')}{\langle n; (n', \sigma, \kappa), (n'', \sigma'', \kappa''); H; \mathsf{intx}\ v\rangle \ \longrightarrow_{\emptyset}\ \langle n; (n', \sigma, \kappa); H; v\rangle}$$

Let $\Phi' = \Phi$ and $\Gamma' = \Gamma$ (and thus $\Phi'^{\alpha} = \Phi^{\alpha} \cup \emptyset$, $\varepsilon' \cup \emptyset \subseteq \Phi^{\varepsilon}$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required). To prove 1., we know that $n; \Gamma \vdash H$ follows by assumption and $\Phi; \Gamma \vdash v : \tau \rightsquigarrow \cdot$ by value typing. To prove 2., we must show that $\Phi, \cdot; H \vdash (n', \sigma, \kappa)$, but this is true by inversion on $\Phi, \Phi'', \cdot; H \vdash (n', \sigma, \kappa), (n'', \sigma'', \kappa'')$.

For 3., $\mathit{traceOK}((n', \sigma, \kappa))$ follows from $\mathit{traceOK}((n', \sigma, \kappa), (n'', \sigma'', \kappa''))$ (which is true by assumption).

case [TX-CONG-2] **:**

We know that
$$\frac{\langle n; \Sigma; H; e\rangle \longrightarrow_{\varepsilon} \langle n'; \Sigma'; H'; e'\rangle}{\langle n; \Sigma; H; \mathsf{intx}\ e\rangle \ \longrightarrow_{\emptyset}\ \langle n'; \Sigma'; H'; \mathsf{intx}\ e'\rangle}$$

follows from $\langle n; \Sigma; H; e\rangle \longrightarrow_{\eta} \langle n; \Sigma'; H'; e'\rangle$ (because the reduction does not perform an update, hence $\eta \equiv \varepsilon_0$ and we apply [TX-CONG-2]).

We have $\Phi'', \mathcal{R}; H \vdash \Sigma$ by inversion on $\Phi, \Phi'', \mathcal{R}; H \vdash ((n', \sigma, \kappa), \Sigma)$, hence by induction:

(i) $\Phi'''; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}'$ and

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi''', \mathcal{R}'; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some Φ''' such that $\Phi'''^{\alpha} = \Phi''^{\alpha} \cup \varepsilon_0$, $\varepsilon''' \cup \varepsilon_0 \subseteq \Phi''^{\varepsilon}$, $\Phi'''^{\omega} = \Phi''^{\omega}$, $\Phi'''^{\delta_i} = \Phi'^{\delta_i}$, and $\Phi'''^{\delta_o} = \Phi'^{\delta_o}$.

Let $\Phi' = \Phi$ (hence $\Phi'^{\alpha} = \Phi'^{\alpha} \cup \emptyset$, $\varepsilon' \cup \emptyset \subseteq \Phi^{\varepsilon}$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required) and $\Gamma' = \Gamma$.

To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and we can apply [TINTRANS]:

$$(\text{TINTRANS})\,\frac{\Phi'''; \Gamma' \vdash e' : \tau \rightsquigarrow \mathcal{R}' \quad \Phi'^{\alpha} \subseteq \Phi'''^{\alpha} \quad \Phi'^{\omega} \subseteq \Phi'''^{\omega}}{\Phi'; \Gamma' \vdash \mathsf{intx}\ e' : \tau \rightsquigarrow \Phi''', \mathcal{R}'}$$

The first premise follows from (i), while the rest follow by assumption and choice of Φ'.

Part 2. follows directly from (iii) and $\Phi, \Phi'', \cdot; H \vdash (n', \sigma, \kappa), (n'', \sigma'', \kappa'')$ (which we have by assumption). Part 3. follows directly from (iv).

case (TLET) **:**

We know that:

$$(\text{TLET})\,\frac{\Phi_1; \Gamma \vdash e_1 : \tau_1 \rightsquigarrow \mathcal{R} \quad \Phi_2; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow \cdot \quad \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}{\Phi; \Gamma \vdash \mathsf{let}\ x : \tau_1 = e_1\ \mathsf{in}\ e_2 : \tau_2 \rightsquigarrow \mathcal{R}}$$

We can reduce using either [LET] or [CONG].

case [LET] **:**

This implies that $e_1 \equiv v$ hence $\mathcal{R} \equiv \cdot$. We have:

$$\langle n; (n', \sigma, \kappa); H; \mathsf{let}\ x : \tau = v\ \mathsf{in}\ e\rangle \ \longrightarrow\ \langle n; (n', \sigma, \kappa); H; e[x \mapsto v]\rangle$$

We have $\Phi_2 = \Phi$ (because $\Phi_1^{\varepsilon} \equiv \emptyset$; if $\Phi_1^{\varepsilon} \not\equiv \emptyset$ we can rewrite the derivation using value typing to make it so). Let $\Gamma' = \Gamma$ and $\Phi' = \Phi$ (and thus $\varepsilon \cup \emptyset \subseteq \Phi^{\varepsilon}$, $\Phi'^{\alpha} = \Phi^{\alpha} \cup \emptyset$, $\Phi'^{\omega} = \Phi^{\omega}$, and $\Phi'^{\delta_i} = \Phi^{\delta_i}$, $\Phi'^{\delta_o} = \Phi^{\delta_o}$) as required. To prove 1., we have $n; \Gamma \vdash H$ and $\Phi; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow \cdot$ by assumption. By value typing we have $\Phi; \Gamma \vdash v : \tau_1 \rightsquigarrow \cdot$, so by substitution (Lemma C.0.33) we have $\Phi; \Gamma \vdash e_2[x \mapsto v] : \tau_2 \rightsquigarrow \cdot$.

Parts 2. and 3. hold by assumption.

case [CONG] **:**

Similar to (TIF)-[CONG].

case (TAPP) **:**

We know that:

$$(\text{TAPP})\ \frac{\begin{array}{c}\Phi_1;\Gamma \vdash e_1 : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \mathcal{R}_1 \qquad \Phi_2;\Gamma \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2 \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \Phi_f^{\varepsilon} \qquad \Phi_3^{\delta_o} \subseteq \Phi_f^{\delta_o}\end{array}}{\Phi;\Gamma \vdash e_1\ e_2 : \tau_2 \rightsquigarrow \mathcal{R}_1 \bowtie \mathcal{R}_2}$$

We can reduce using either [CALL] or [CONG].

case [CALL] **:**

We have that

$$\langle n;(n',\sigma,\kappa);(H'',\mathsf{z} \mapsto (\tau,\lambda(x).e,\nu));\mathsf{z}\ v\rangle \ \longrightarrow_{\{\mathsf{z}\}} \langle n;(n',\sigma\cup(\mathsf{z},\nu),\kappa);(H'',\mathsf{z} \mapsto (\tau,\lambda(x).e,\nu));e[x \mapsto v]\rangle$$

(where $H \equiv (H'',\mathsf{z} \mapsto (\tau,\lambda(x).e,\nu))$), and

$$(\text{TAPP})\ \frac{\begin{array}{c}\Phi_1;\Gamma \vdash \mathsf{z} : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \cdot \qquad \Phi_2;\Gamma \vdash v : \tau_1 \rightsquigarrow \cdot \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \Phi_f^{\varepsilon} \qquad \Phi_3^{\delta_o} \subseteq \Phi_f^{\delta_o}\end{array}}{\Phi;\Gamma \vdash \mathsf{z}\ v : \tau_2 \rightsquigarrow \cdot}$$

where by subtyping derivations (Lemma C.0.23) we have

$$(\text{TSUB})\ \frac{(\text{TGVAR})\ \dfrac{\Gamma(\mathsf{z}) = \tau_1' \longrightarrow^{\Phi_f'} \tau_2'}{\Phi_\emptyset;\Gamma \vdash \mathsf{z} : \tau_1' \longrightarrow^{\Phi_f'} \tau_2' \rightsquigarrow \cdot} \qquad \dfrac{\tau_1 \leq \tau_1' \qquad \tau_2' \leq \tau_2 \qquad \Phi_f' \leq \Phi_f}{\tau_1' \longrightarrow^{\Phi_f'} \tau_2' \leq \tau_1 \longrightarrow^{\Phi_f} \tau_2} \qquad \Phi_\emptyset \leq \Phi_1}{\Phi_1;\Gamma \vdash \mathsf{z} : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \cdot}$$

Define $\Phi_f \equiv [\alpha_f;\varepsilon_f;\omega_f;\delta_{if};\delta_{of}]$ and $\Phi_f' \equiv [\alpha_f';\varepsilon_f';\omega_f';\delta_{if};\delta_{of}]$.

Let $\Gamma' = \Gamma$, $\mathcal{R}' = \cdot$ and choose $\Phi' = [\Phi_1^{\alpha} \cup \{\mathsf{z}\};\varepsilon_f;\Phi_3^{\omega};\Phi_1^{\delta_i};\Phi_3^{\delta_o}]$. Since $\mathsf{z} \in \varepsilon_f'$ (by $n;\Gamma \vdash H$) and $\varepsilon_f' \subseteq \varepsilon_f$ (by $\Phi_f' \leq \Phi_f$) we have $\varepsilon_f \cup \{\mathsf{z}\} \subseteq (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f)$. By the same argument we have $\{\mathsf{z}\} \subseteq \Phi_1^{\delta_i}$. The choice of Φ' is acceptable since $\Phi'^{\alpha} = \Phi^{\alpha} \cup \{\mathsf{z}\}$, $\Phi'^{\varepsilon} \cup \{\mathsf{z}\} \subseteq \Phi^{\varepsilon}$, $\Phi'^{\omega} = \Phi^{\omega}$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$. For 1., we have $n;\Gamma \vdash H'$ by assumption; for the remainder we have to prove $\Phi';\Gamma \vdash e[x \mapsto v] : \tau_2 \rightsquigarrow \cdot$. First, we must prove that $\Phi_f' \leq \Phi'$. Note that since $\{\mathsf{z}\} \subseteq \alpha_f$ by $n;\Gamma \vdash H'$, from $\Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi$ and choice of Φ' we get $\Phi_3'^{\alpha} \cup \{\mathsf{z}\} \subseteq \alpha_f$. We have:

Φ'	$\equiv$	$[\Phi_1^{\alpha} \cup \{\mathsf{z}\};\varepsilon_f;\Phi_3^{\omega};\Phi_1^{\delta_i};\Phi_3^{\delta_o}]$	(by choice of Φ')
Φ_f	$\equiv$	$[\alpha_f;\varepsilon_f;\omega_f;\delta_{if};\delta_{of}]$	
Φ_f'	$\equiv$	$[\alpha_f';\varepsilon_f';\omega_f';\delta_i{}_f';\delta_o{}_f']$	
ε_f'	$\subseteq$	ε_f	(by $\Phi_f' \leq \Phi_f$)
α_f	$\subseteq$	α_f'	(by $\Phi_f' \leq \Phi_f$)
ω_f	$\subseteq$	ω_f'	(by $\Phi_f' \leq \Phi_f$)
$\delta_i{}_f'$	$\subseteq$	δ_{if}	(by $\Phi_f' \leq \Phi_f$)
δ_{of}	$\subseteq$	$\delta_o{}_f'$	(by $\Phi_f' \leq \Phi_f$)
$\Phi_3'^{\alpha} \cup \{\mathsf{z}\}$	$\subseteq$	α_f	(by assumption and choice of Φ')
$\Phi_3'^{\alpha}$	$=$	$\Phi_1^{\alpha} \cup \Phi_1^{\varepsilon} \cup \Phi_2'^{\varepsilon}$	(by $\Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi$)
$\Phi_3'^{\omega}$	$\subseteq$	ω_f	(by assumption and choice of Φ')
$\Phi_3'^{\delta_o}$	$\subseteq$	δ_{of}	(by assumption and choice of Φ')

Thus we have the result by [TSUB]

$$\frac{\Phi_f';\Gamma \vdash e[x \mapsto v] : \tau_2' \rightsquigarrow \cdot \qquad \tau_2' \leq \tau_2 \qquad \Phi_f' \leq \Phi_1'}{\Phi_1';\Gamma \vdash e[x \mapsto v] : \tau_2}$$

By assumption, we have $\Phi_2; \Gamma \vdash v : \tau_1 \rightsquigarrow \cdot$. By value typing and $\tau_1 \leq \tau_1'$ we have $\Phi'; \Gamma \vdash v : \tau_1' \rightsquigarrow \cdot$. Finally by substitution we have $\Phi'; \Gamma \vdash e[x \mapsto v] : \tau_2 \rightsquigarrow \cdot$.
For part 2., we need to prove $\Phi', \cdot; H \vdash (n'', \sigma', \kappa')$ where $\sigma' = \sigma \cup (\mathsf{z}, \nu)$ and $n'' = n'$, hence:

$$(\text{TC1})\ \dfrac{\begin{array}{c} \mathsf{f} \in (\sigma \cup (\mathsf{z}, \nu)) \Rightarrow \mathsf{f} \in \Phi^\alpha \cup \{\mathsf{z}\} \\ \mathsf{f} \in (\varepsilon_f \cap \Phi^{\delta_i}) \Rightarrow n' \in ver(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\Phi^\alpha \cup \{\mathsf{z}\} \cup \delta_i) \\ \kappa^\omega \supseteq (\Phi^\omega \cup \varepsilon_f) \end{array}}{\Phi', \cdot; H \vdash (n'', \sigma', \kappa')}$$

The first premise is true by assumption and the fact that $\{\mathsf{z}\} \subseteq \{\mathsf{z}\}$. The second premise is true by assumption.

For part 3., we need to prove $traceOK(\sigma \cup (\mathsf{z}, \nu))$; we have $traceOK(\sigma)$ by assumption, hence need to prove that $n' \in \nu$. Since by assumption we have that $\mathsf{f} \in \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f \Rightarrow n' \in ver(H, \mathsf{f})$ and $\{\mathsf{z}\} \subseteq \varepsilon_f$, we have $n' \in \nu$.

case [CONG] **:**

case $\mathbb{E}\ e$ **:**
$\langle n; \Sigma; H; e_1\ e_2 \rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e_1'\ e_2 \rangle$ follows from $\langle n; \Sigma; H; e_1 \rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e_1' \rangle$.
Since $e_1 \not\equiv v \Rightarrow \mathcal{R}_2 = \cdot$ by assumption, by Lemma C.0.26 we have $\Phi_1, \mathcal{R}_1; H \vdash \Sigma$ hence we can apply induction:

(i) $\Phi_1'; \Gamma' \vdash e_1' : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \mathcal{R}_1'$ and
(ii) $n; \Gamma' \vdash H'$
(iii) $\Phi_1', \mathcal{R}_1'; H' \vdash \Sigma'$
(iv) $traceOK(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi_1' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_1'; \Phi_1^\omega; \Phi_1^{\delta_i}; \Phi_1^{\delta_o}]$ where $\varepsilon_1' \cup \varepsilon_0 \subseteq \varepsilon_1$ and $\Phi_1^\omega \equiv \Phi_2^\varepsilon \cup \varepsilon_f \cup \Phi_3^\omega$.

Let $\begin{array}{lcl} \Phi_2' & \equiv & [\Phi_1^\alpha \cup \varepsilon_1' \cup \varepsilon_0; \Phi_2^\varepsilon; \varepsilon_f \cup \Phi_3^\omega; \Phi_2^{\delta_i}; \Phi_2^{\delta_o}] \\ \Phi_3' & \equiv & [\Phi_1^\alpha \cup \varepsilon_1' \cup \varepsilon_0 \cup \Phi_2^\varepsilon; \varepsilon_f; \Phi_3^\omega; \Phi_3^{\delta_i}; \Phi_3^{\delta_o}] \end{array}$
Thus $\Phi_3'^\varepsilon = \varepsilon_f$, $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'$, $\Phi_3'^\alpha \subseteq \Phi_f^\alpha$ and $\Phi_3'^\omega \subseteq \Phi_f^\omega$ (since $\Phi_3'^\alpha \cup \varepsilon_0 \subseteq \Phi_3^\alpha$ and $\Phi_3'^\omega = \Phi_3^\omega$). We have $\Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_1' \cup \Phi_2^\varepsilon \cup \varepsilon_f; \Phi_3^\omega; \Phi_1^{\delta_i}; \Phi_3^{\delta_o}]$. The choice of Φ' is acceptable since $\Phi'^\alpha = \Phi^\alpha \cup \varepsilon_0$, $(\varepsilon_1' \cup \varepsilon_f \cup \varepsilon_2) \cup \varepsilon_0 \subseteq (\varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_f)$ i.e., $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^\varepsilon$, $\Phi'^\omega = \Phi^\omega$, $\Phi'^\omega = \Phi^\omega$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required).
To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and apply (TApp):

$$(\text{TApp})\ \dfrac{\begin{array}{c} \Phi_1'; \Gamma' \vdash e_1' : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \mathcal{R}_1' \\ (\text{TSub})\ \dfrac{\Phi_2; \Gamma' \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2 \quad \tau_1 \leq \tau_1 \quad \dfrac{\begin{array}{c} \Phi_1^\alpha \cup \varepsilon_1' \cup \varepsilon_0 \subseteq \Phi_1^\alpha \cup \Phi_1^\varepsilon \\ \Phi_2^\varepsilon \subseteq \Phi_2^\varepsilon \\ \varepsilon_f \cup \Phi_3^\omega \subseteq \varepsilon_f \cup \Phi_3^\omega \\ \Phi_2^{\delta_i} = \Phi_2^{\delta_i} \\ \Phi_2^{\delta_o} = \Phi_2^{\delta_o} \end{array}}{\Phi_2 \leq \Phi_2'}}{\Phi_2'; \Gamma' \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2} \\ \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi' \\ \Phi_3'^\varepsilon = \Phi_f^\varepsilon \qquad \Phi_3'^\alpha \subseteq \Phi_f^\alpha \qquad \Phi_3'^\omega \subseteq \Phi_f^\omega \\ \Phi_3'^{\delta_i} = \Phi_3'^{\delta_o} \cup \Phi_f^\varepsilon \qquad \Phi_3'^{\delta_o} \subseteq \Phi_f^{\delta_o} \end{array}}{\Phi'; \Gamma' \vdash e_1'\ e_2 : \tau_2 \rightsquigarrow \mathcal{R}_1' \bowtie \mathcal{R}_2}$$

Note that $\Phi_2; \Gamma' \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2$ follows from $\Phi_2; \Gamma \vdash e_2 : \tau_1 \rightsquigarrow \mathcal{R}_2$ by weakening (Lemma C.0.19). The last premise holds vacuously as $\mathcal{R}_2 \equiv \cdot$ by assumption.
To prove part 2., we must show that $\Phi', \mathcal{R}'; H' \vdash \Sigma'$. The proof is similar to the (TAssign)-[CONG] proof, case $\mathbb{E} := e$ but substituting ε_f for ε_r.
Part 3. follows directly from (iv).

case $v\ \mathbb{E}$ **:**
$\langle n; \Sigma; H; v\ e_2 \rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; v\ e_2' \rangle$ follows from $\langle n; \Sigma; H; e_2 \rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e_2' \rangle$.
For convenience, we make $\Phi_1^\varepsilon \equiv \emptyset$; if $\Phi_1^\varepsilon \not\equiv \emptyset$, we can always construct a typing derivation of v that uses value typing to make $\Phi_1^\varepsilon \equiv \emptyset$. Note that $\Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi$ would still hold since Lemma C.0.24 allows us to decrease Φ_2^α to satisfy $\Phi_2^\alpha = \Phi_1^\alpha \cup \Phi_1^\varepsilon$; similarly, since $\Phi_3^\alpha = \Phi_1^\alpha \cup \Phi_1^\varepsilon \cup \Phi_2^\varepsilon$ we know that $\Phi_3^\alpha \subseteq \Phi_f^\alpha$ would still hold if Φ_3^α was smaller as a result of shrinking Φ_1^ε to be $\emptyset$.
Since $e_1 \equiv v$, by inversion $\mathcal{R}_1 \equiv \cdot$ and by Lemma C.0.26 (which we can apply since $\Phi_1^\varepsilon \equiv \emptyset$), we have $\Phi_2, \mathcal{R}_2; H \vdash \Sigma$; hence by induction:

(i) $\Phi_2'; \Gamma' \vdash e_2' : \tau_1 \rightsquigarrow \mathcal{R}_2'$

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi_2', \mathcal{R}_2'; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$ and some $\Phi_2' \equiv [\Phi_2^\alpha \cup \varepsilon_0; \varepsilon_2'; \Phi_2^\omega; \Phi_2^{\delta_i}; \Phi_2^{\delta_o}]$ where $(\varepsilon_2' \cup \varepsilon_0) \subseteq \Phi_2^\varepsilon$; note $\Phi_2^\alpha \equiv \Phi_1^\alpha$ (since $\Phi_1^\varepsilon \equiv \emptyset$) and $\Phi_2^\omega \equiv \varepsilon_3 \cup \Phi_3^\omega$.

Let $\begin{array}{lcl} \Phi_1' & \equiv & [\Phi_1^\alpha \cup \varepsilon_0; \emptyset; \varepsilon_2' \cup \varepsilon_f \cup \Phi_3^\omega; \Phi_1^{\delta_i}; \Phi_1^{\delta_o}] \\ \Phi_3' & \equiv & [\Phi_1^\alpha \cup \varepsilon_0 \cup \varepsilon_2'; \varepsilon_f; \Phi_3^\omega; \Phi_3^{\delta_i}; \Phi_3^{\delta_o}] \end{array}$

Thus $\Phi_3'^\varepsilon = \varepsilon_f$, $\Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi'$, $\Phi_3'^\alpha \subseteq \Phi_f^\alpha$ and $\Phi_3'^\omega \subseteq \Phi_f^\omega$ (since $\Phi_3'^\alpha \cup \varepsilon_0 \subseteq \Phi_3^\alpha$ and $\Phi_3'^\omega = \Phi_3^\omega$). We have $\Phi' \equiv [\Phi_1^\alpha \cup \varepsilon_0; \varepsilon_2' \cup \varepsilon_f; \Phi_3^\omega; \Phi_1^{\delta_i}; \Phi_3^{\delta_o}]$ and $(\varepsilon_2' \cup \varepsilon_f) \cup \varepsilon_0 \subseteq (\Phi_2^\varepsilon \cup \varepsilon_f)$. The choice of Φ' is acceptable since $\Phi'^\alpha = \Phi^\alpha \cup \varepsilon_0$, $\varepsilon' \cup \varepsilon_0 \subseteq \Phi^\varepsilon$, $\Phi'^\omega = \Phi^\omega$, $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required).

To prove 1., we have $n; \Gamma' \vdash H'$ by (ii), and we can apply [TApp]:

$$\text{(TApp)}\,\frac{\begin{array}{c} \Phi_1'; \Gamma' \vdash v : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \cdot \qquad \Phi_2'; \Gamma' \vdash e_2' : \tau_1 \rightsquigarrow \mathcal{R}_2' \\ \Phi_1' \rhd \Phi_2' \rhd \Phi_3' \hookrightarrow \Phi' \\ \Phi_3'^\varepsilon = \Phi_f^\varepsilon \qquad \Phi_3'^\alpha \subseteq \Phi_f^\alpha \qquad \Phi_3'^\omega \subseteq \Phi_f^\omega \\ \Phi_3'^{\delta_i} = \Phi_3'^{\delta_o} \cup \Phi_f^\varepsilon \qquad \Phi_3'^{\delta_o} \subseteq \Phi_f^{\delta_o} \end{array}}{\Phi'; \Gamma' \vdash e_1\ e_2' : \tau_2 \rightsquigarrow \cdot \bowtie \mathcal{R}_2'}$$

(Note that $\cdot \bowtie \mathcal{R}_2' = \mathcal{R}_2'$.)

The first premise follows by value typing and weakening; the second by (i); the third– eighth by choice of Φ', Φ_1', Φ_2', Φ_3'.

To prove part 2., we must show that $\Phi', \mathcal{R}'; H' \vdash \Sigma'$. The proof is similar to the (TAssign)-[cong] proof, case $r := \mathbb{E}$ but substituting ε_f for ε_r.

Part 3. follows directly from (iv).

case (TSub) **:**

We have

$$\text{(TSub)}\,\frac{\begin{array}{c} \Phi''; \Gamma \vdash e : \tau'' \rightsquigarrow \mathcal{R} \\ \Phi'' \equiv [\alpha; \varepsilon''; \omega; \delta_i; \delta_o] \qquad \Phi \equiv [\alpha; \varepsilon; \omega; \delta_i; \delta_o] \\ \tau'' \leq \tau \qquad \varepsilon'' \subseteq \varepsilon \end{array}}{\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}}$$

since by flow effect weakening (Lemma C.0.24) we know that α and ω are unchanged in the use of (TSub).

We have $\langle n; \Sigma; H; e \rangle \longrightarrow_\varepsilon \langle n; \Sigma'; H'; e' \rangle$. To apply induction we must show that $n; \Gamma \vdash H$, which we have by assumption, $\Phi''; \Gamma \vdash e : \tau'' \rightsquigarrow \mathcal{R}$, which we also have by assumption, and $\Phi'', \mathcal{R}; H \vdash \Sigma$. We prove $\Phi'', \mathcal{R}; H \vdash \Sigma$ below. We know

$$\text{(TC1)}\,\frac{\begin{array}{c} \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\varepsilon \cap \delta_i) \Rightarrow n \in \mathit{ver}(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\alpha \cup \delta_i) \\ \kappa^\omega \supseteq (\omega \cup \varepsilon) \end{array}}{[\alpha; \varepsilon; \omega; \delta_i; \delta_o], \cdot; H \vdash (n, \sigma, \kappa)}$$

and need to show

$$\text{(TC1)}\,\frac{\begin{array}{c} \mathsf{f} \in \sigma \Rightarrow \mathsf{f} \in \alpha \\ \mathsf{f} \in (\varepsilon'' \cap \delta_i) \Rightarrow n \in \mathit{ver}(H, \mathsf{f}) \\ \kappa^\alpha \supseteq (\alpha \cup \delta_i) \\ \kappa^\omega \supseteq (\omega \cup \varepsilon'') \end{array}}{[\alpha; \varepsilon''; \omega; \delta_i; \delta_o], \cdot; H \vdash (n, \sigma, \kappa)}$$

The first premise is true by assumption. The second follows easily by assumption and the fact that $\varepsilon'' \subseteq \varepsilon$. The third premise follows by assumption. The fourth premise similarly follows by assumption and by $\varepsilon'' \subseteq \varepsilon$.

Hence we have:

(i) $\Phi'''; \Gamma' \vdash e' : \tau'' \rightsquigarrow \mathcal{R}'$ and

(ii) $n; \Gamma' \vdash H'$

(iii) $\Phi''', \mathcal{R}'; H' \vdash \Sigma'$

(iv) $\mathit{traceOK}(\Sigma')$

for some $\Gamma' \supseteq \Gamma$, Φ''' such that $\Phi'''^{\alpha} = \alpha \cup \varepsilon_0$, $\Phi'''^{\varepsilon} \cup \varepsilon_0 \subseteq \varepsilon''$, $\Phi'''^{\delta_i} = \Phi''^{\delta_i}$, and $\Phi'''^{\delta_o} = \Phi''^{\delta_o}$.

Let $\Phi' \equiv \Phi'''$, and thus $\Phi'^{\alpha} = \alpha \cup \varepsilon_0$, $\Phi'^{\varepsilon} \cup \varepsilon_0 \subseteq \varepsilon$ since $\varepsilon'' \subseteq \varepsilon$, $\Phi'^{\omega} = \omega$, and $\Phi'^{\delta_i} = \Phi^{\delta_i}$, and $\Phi'^{\delta_o} = \Phi^{\delta_o}$ as required. All results follow by induction.

□

Lemma C.0.32 (Progress)**.** *If $n \vdash H, e : \tau$ (such that $\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}$ and $n; \Gamma \vdash H$) and for all Σ such that $\Phi, \mathcal{R}; H \vdash \Sigma$ and traceOK(Σ), then either e is a value, or there exist n', H', Σ', e' such that $\langle n; \Sigma; H;\ e\rangle \longrightarrow_\eta \langle n'; \Sigma'; H';\ e'\rangle$.*

Proof. Induction on the typing derivation $n \vdash H, e : \tau$; consider each possible rule for the conclusion of this judgment:

case (TInt-TGvar-TLoc) **:**

These are all values.

case (TVar) **:**

Can't occur, since local values are substituted for.

case (TRef) **:**

We must have that

$$(\text{TRef})\frac{\Phi; \Gamma \vdash e' : \tau \rightsquigarrow \mathcal{R}}{\Phi; \Gamma \vdash \mathsf{ref}\ e' : \mathit{ref}^{\,\varepsilon}\ \tau \rightsquigarrow \mathcal{R}}$$

There are two possible reductions, depending on the shape of e:

case $e' \equiv v$ **:**

By inversion on $\Phi; \Gamma \vdash v : \tau \rightsquigarrow \cdot$ we know that $\mathcal{R} \equiv \cdot$ hence by inversion on $\Phi, \mathcal{R}; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$. We have that $\langle n; (n', \sigma, \kappa); H; \mathsf{ref}\ v\rangle \longrightarrow n; (n', \sigma, \kappa); H'; r$ where $r \notin \mathrm{dom}(H)$ and $H' = H, r \mapsto (\cdot, v, \emptyset)$ by (ref).

case $e' \not\equiv v$ **:**

By induction, $\langle n; \Sigma; H; e'\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e''\rangle$ and thus $\langle n; \Sigma; H; (\mathsf{ref}\ \_)[e']\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; (\mathsf{ref}\ \_)[e'']\rangle$ by [cong].

case (TDeref) **:**

We know that

$$(\text{TDeref})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash e : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}\\ \Phi_2^{\varepsilon} = \varepsilon_r \qquad \Phi_2^{\delta_i} = \Phi_2^{\delta_o} \cup \varepsilon_r\\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash\ !\,e : \tau \rightsquigarrow \mathcal{R}}$$

Consider the shape of e:

case $e' \equiv v$ **:**

Since v is a value of type $\mathit{ref}^{\,\varepsilon_r}\ \tau$, we must have $v \equiv \mathsf{z}$ or $v \equiv r$.

case $e' \equiv \mathsf{z}$ **:**

We have

$$(\text{TDeref})\frac{\begin{array}{c}\Phi_1; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \cdot\\ \Phi_2^{\varepsilon} = \varepsilon_r \qquad \Phi_2^{\delta_i} = \Phi_2^{\delta_o} \cup \varepsilon_r\\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash\ !\,\mathsf{z} : \tau \rightsquigarrow \cdot}$$

where by subtyping derivations (Lemma C.0.23) we have

$$(\text{TSub})\frac{(\text{TGVar})\dfrac{\Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon_r'}\ \tau'}{\Phi_\emptyset; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r'}\ \tau' \rightsquigarrow \cdot} \qquad \dfrac{\tau' \leq \tau \quad \tau \leq \tau' \quad \varepsilon_r' \subseteq \varepsilon_r}{\mathit{ref}^{\,\varepsilon_r'}\ \tau' \leq \mathit{ref}^{\,\varepsilon_r}\ \tau} \qquad \Phi_\emptyset \leq \Phi_1}{\Phi_1; \Gamma \vdash \mathsf{z} : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \cdot}$$

By inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$. By $C; \Gamma \vdash H$ we have $\mathsf{z} \in \mathrm{dom}(H)$ (and thus $H \equiv H'', \mathsf{z} \mapsto (\mathit{ref}^{\,\varepsilon_r'}\ \tau', v, \nu)))$ since $\Gamma(\mathsf{z}) = \mathit{ref}^{\,\varepsilon_r'}\ \tau'$. Therefore, we can reduce via [gvar-deref]:

$$\langle n; (n', \sigma, \kappa); (H'', \mathsf{z} \mapsto (\tau', v, \nu)); !\,\mathsf{z}\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n', \sigma \cup (\mathsf{z}, \nu), \kappa); (H'', \mathsf{z} \mapsto (\tau', v, \nu)); v\rangle$$

case $e' \equiv r$ **:**
Similar to the $e' \equiv \mathsf{z}$ case above, but reduce using [DEREF].

case $e' \not\equiv v$ **:**

Let $\mathbb{E} \equiv !\,\_$ so that $e \equiv \mathbb{E}[e']$. To apply induction, we have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ by Lemma C.0.25. Thus we get $\langle n; \Sigma; H; e' \rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e'' \rangle$, hence we have that $\langle n; \Sigma; H; \mathbb{E}[e'] \rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; \mathbb{E}[e''] \rangle$ by [CONG].

case (TASSIGN) **:**

$$(\text{TASSIGN})\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \mathit{ref}^{\,\varepsilon_r}\ \tau \leadsto \mathcal{R}_1 \qquad \Phi_2; \Gamma \vdash e_2 : \tau \leadsto \mathcal{R}_2 \\ \Phi_3^{\varepsilon} = \varepsilon_r \qquad \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \varepsilon_r \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash e_1 := e_2 : \tau \leadsto \mathcal{R}_1 \bowtie \mathcal{R}_2}$$

Depending on the shape of e, we have:

case $e_1 \equiv v_1, e_2 \equiv v_2$ **:**

Since v_1 is a value of type $\mathit{ref}^{\,\varepsilon_r}\ \tau$, we must have $v_1 \equiv \mathsf{z}$ or $v_1 \equiv r$. The results follow by reasoning quite similar to [TDEREF] above.

case $e_1 \equiv v_1, e_2 \not\equiv v$ **:**

Let $\mathbb{E} \equiv v_1 := \_$ so that $e \equiv \mathbb{E}[e_2]$. Since e_1 is a value, $\mathcal{R}_1 \equiv \cdot$ hence we have $\Phi_2, \mathcal{R}; H \vdash \Sigma$ by Lemma C.0.26 and we can apply induction. We have $\langle n; \Sigma; H; e_2 \rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e_2' \rangle$, and thus $\langle n; \Sigma; H; \mathbb{E}[e_2] \rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; \mathbb{E}[e_2'] \rangle$ by [CONG].

case $e_1 \not\equiv v$ **:**

Since e_1 is a not value, $\mathcal{R}_2 \equiv \cdot$ hence we have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ by Lemma C.0.26 and we can apply induction. The rest follows by an argument similar to the above case.

case (TCHECKIN) **:**

$$(\text{TCHECKIN})\dfrac{\alpha \cup \delta_o \subseteq \alpha' \qquad \omega \subseteq \omega'}{[\alpha; \emptyset; \omega; \emptyset; \delta_o]; \Gamma \vdash \mathsf{checkin}^{\alpha', \omega'} : \mathit{int} \leadsto \cdot}$$

By inversion on $\Phi; \Gamma \vdash \mathsf{checkin}^{\alpha', \omega'} : \mathit{int} \leadsto \mathcal{R}$ we have that $\mathcal{R} \equiv \cdot$, hence by inversion on $\Phi, \mathcal{R}; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$ and can reduce via [CHECKIN].

case (TIF) **:**

$$(\text{TIF})\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash e_1 : \mathit{int} \leadsto \mathcal{R} \\ \Phi_2; \Gamma \vdash e_2 : \tau \leadsto \cdot \qquad \Phi_2; \Gamma \vdash e_3 : \tau \leadsto \cdot \\ \Phi_1 \rhd \Phi_2 \hookrightarrow \Phi\end{array}}{\Phi; \Gamma \vdash \mathsf{if0}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau \leadsto \mathcal{R}}$$

Depending on the shape of e, we have:

case $e_1 \equiv v$ **:**

This implies $\mathcal{R} \equiv \cdot$ so by inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$. Since the type of v is int, we know v must be an integer n. Thus we can reduce via either [IF-T] or [IF-F].

case $e_1 \not\equiv v$ **:**

Let $\mathbb{E} \equiv \mathsf{if0}\ \_\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ so that $e \equiv \mathbb{E}[e_1]$. To apply induction, we have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ by Lemma C.0.25. We have $\langle n; \Sigma; H; e_1 \rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e_1' \rangle$ and thus $\langle n; \Sigma; H; \mathbb{E}[e_1] \rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; \mathbb{E}[e_1'] \rangle$ by [CONG].

case (TTRANSACT) **:**

We know that:

$$(\text{TTRANSACT})\dfrac{\begin{array}{c}\Phi''; \Gamma \vdash e : \tau \leadsto \cdot \\ \Phi^{\alpha} \subseteq \Phi''^{\alpha} \qquad \Phi^{\omega} \subseteq \Phi''^{\omega}\end{array}}{\Phi; \Gamma \vdash \mathsf{tx}^{(\Phi''^{\alpha} \cup \Phi''^{\delta_i}, \Phi''^{\omega} \cup \Phi''^{\varepsilon})}\ e : \tau \leadsto \cdot}$$

By inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$. Thus we can reduce by [TX-START].

case (TIntrans) **:**

We know that:

$$(\text{TI}\textsc{ntrans})\dfrac{\Phi'';\Gamma\vdash e:\tau\leadsto\mathcal{R}\qquad \Phi^\alpha\subseteq\Phi''^\alpha\quad \Phi^\omega\subseteq\Phi''^\omega}{\Phi;\Gamma\vdash \mathsf{intx}\ e:\tau\leadsto\Phi'',\mathcal{R}}$$

Consider the shape of e:

case $e\equiv v$ **:**

Thus

$$(\text{TI}\textsc{ntrans})\dfrac{\Phi'';\Gamma\vdash v:\tau\leadsto\cdot\qquad \Phi^\alpha\subseteq\Phi''^\alpha\quad \Phi^\omega\subseteq\Phi''^\omega}{\Phi;\Gamma\vdash \mathsf{intx}\ v:\tau\leadsto\Phi'',\cdot}$$

We have $\Phi,\Phi'',\cdot;H\vdash\Sigma$ by assumption:

$$(\text{TC2})\dfrac{\begin{array}{c}\Phi'',\cdot;H\vdash(n'',\sigma'',\kappa'')\\ \mathsf{f}\in\sigma\Rightarrow\mathsf{f}\in\Phi^\alpha\\ \mathsf{f}\in(\emptyset\cap\Phi^{\delta_i})\Rightarrow n'\in ver(H,\mathsf{f})\\ \kappa^\alpha\supseteq(\Phi^\alpha\cup\Phi^{\delta_i})\\ \kappa^\omega\supseteq(\Phi^\omega\cup\emptyset)\end{array}}{[\Phi^\alpha;\emptyset;\Phi^\omega;\Phi^{\delta_i};\Phi^{\delta_o}],\Phi'',\cdot;H\vdash(n',\sigma,\kappa),(n'',\emptyset'',\kappa'')}$$

By inversion we have $\Sigma\equiv((n',\sigma,\kappa),(n'',\sigma'',\kappa''))$; by assumption we have $traceOK(n'',\sigma'',\kappa'')$ so we can reduce via [tx-end].

case $e\not\equiv v$ **:**

We have $\Phi,\Phi',\mathcal{R};H\vdash\Sigma$ by assumption. By induction we have $\langle n;\Sigma';H;e'\rangle\longrightarrow_\eta\langle n';\Sigma'';H';e''\rangle$, hence by [tx-cong-2]:

$$\langle n;\Sigma';H;\mathsf{intx}\ e'\rangle\ \longrightarrow_\emptyset\ \langle n';\Sigma'';H';\mathsf{intx}\ e''\rangle$$

case (TLet) **:**

We know that:

$$(\text{TL}\textsc{et})\dfrac{\Phi_1;\Gamma\vdash e_1:\tau_1\leadsto\mathcal{R}\qquad \Phi_2;\Gamma,x:\tau_1\vdash e_2:\tau_2\leadsto\cdot\qquad \Phi_1\rhd\Phi_2\hookrightarrow\Phi}{\Phi;\Gamma\vdash\mathsf{let}\ x:\tau_1=e_1\ \mathsf{in}\ e_2:\tau_2\leadsto\mathcal{R}}$$

Consider the shape of e:

case $e_1\equiv v$ **:**

Thus $\Phi_1;\Gamma\vdash v:\tau\leadsto\cdot$ and by inversion on $\Phi,\cdot;H\vdash\Sigma$ we have $\Sigma\equiv(n',\sigma,\kappa)$.

We can reduce via [let].

case $e_1\not\equiv v$ **:**

Let $\mathbb{E}\equiv\mathsf{let}\ x:\tau_1=\_\ \mathsf{in}\ e_2$ so that $e\equiv\mathbb{E}[e_1]$. To apply induction, we have $\Phi_1,\mathcal{R};H\vdash\Sigma$ by Lemma C.0.25. We have $\langle n;\Sigma;H;e_1\rangle\longrightarrow_\eta\langle n';\Sigma';H';e_1'\rangle$ and so $\langle n;\Sigma;H;\mathbb{E}[e_1]\rangle\longrightarrow_\eta\langle n';\Sigma';H';\mathbb{E}[e_1']\rangle$ by [cong].

case (TApp) **:**

$$(\text{TA}\textsc{pp})\dfrac{\begin{array}{c}\Phi_1;\Gamma\vdash e_1:\tau_1\longrightarrow^{\Phi_f}\tau_2\leadsto\mathcal{R}_1\qquad\Phi_2;\Gamma\vdash e_2:\tau_1\leadsto\mathcal{R}_2\\ \Phi_1\rhd\Phi_2\rhd\Phi_3\hookrightarrow\Phi\\ \Phi_3^\varepsilon=\Phi_f^\varepsilon\qquad\Phi_3^\alpha\subseteq\Phi_f^\alpha\qquad\Phi_3^\omega\subseteq\Phi_f^\omega\\ \Phi_3^{\delta_i}=\Phi_3^{\delta_o}\cup\Phi_f^\varepsilon\qquad\Phi_3^{\delta_o}\subseteq\Phi_f^{\delta_o}\end{array}}{\Phi;\Gamma\vdash e_1\ e_2:\tau_2\leadsto\mathcal{R}_1\bowtie\mathcal{R}_2}$$

Depending on the shape of e, we have:

case $e_1\equiv v_1,e_2\equiv v_2$ **:**

Since v_1 is a value of type $\tau_1\longrightarrow^\Phi\tau_2$, we must have $v_1\equiv\mathsf{z}$, hence

$$(\text{TA}\textsc{pp})\dfrac{\begin{array}{c}\Phi_1;\Gamma\vdash\mathsf{z}:\tau_1\longrightarrow^{\Phi_f}\tau_2\leadsto\cdot\qquad\Phi_2;\Gamma\vdash v:\tau_1\leadsto\cdot\\ \Phi_1\rhd\Phi_2\rhd\Phi_3\hookrightarrow\Phi\\ \Phi_3^\varepsilon=\Phi_f^\varepsilon\qquad\Phi_3^\alpha\subseteq\Phi_f^\alpha\qquad\Phi_3^\omega\subseteq\Phi_f^\omega\\ \Phi_3^{\delta_i}=\Phi_3^{\delta_o}\cup\Phi_f^\varepsilon\qquad\Phi_3^{\delta_o}\subseteq\Phi_f^{\delta_o}\end{array}}{\Phi;\Gamma\vdash\mathsf{z}\ v:\tau_2\leadsto\cdot}$$

where by subtyping derivations (Lemma C.0.23) we have

$$\text{(TSub)}\dfrac{\text{(TGVar)}\dfrac{\Gamma(\mathsf{z}) = \tau_1' \longrightarrow^{\Phi_f'} \tau_2'}{\Phi_\emptyset; \Gamma \vdash \mathsf{z} : \tau_1' \longrightarrow^{\Phi_f'} \tau_2' \rightsquigarrow \cdot} \qquad \dfrac{\tau_1 \leq \tau_1' \quad \tau_2' \leq \tau_2 \quad \Phi_f' \leq_f \Phi_f}{\tau_1' \longrightarrow^{\Phi_f'} \tau_2' \leq \tau_1 \longrightarrow^{\Phi_f} \tau_2} \qquad \Phi_\emptyset \leq \Phi_1}{\Phi_1; \Gamma \vdash \mathsf{z} : \tau_1 \longrightarrow^{\Phi_f} \tau_2 \rightsquigarrow \cdot}$$

By inversion on $\Phi, \cdot; H \vdash \Sigma$ we have $\Sigma \equiv (n', \sigma, \kappa)$. By $C; \Gamma \vdash H$ we have $\mathsf{z} \in \text{dom}(H)$ and $H \equiv (H'', \mathsf{z} \mapsto (\tau_1' \longrightarrow^{\Phi_f'} \tau_2', \lambda(x).e'', \nu))$ since $\Gamma(\mathsf{z}) = \tau_1' \longrightarrow^{\Phi_f'} \tau_2'$. By [CALL], we have:

$$\langle n; (n', \sigma, \kappa); (H'', \mathsf{z} \mapsto (\tau_1' \longrightarrow^{\Phi_f'} \tau_2', \lambda(x).e'', \nu)); \mathsf{z}\ v\rangle \longrightarrow_{\{\mathsf{z}\}}$$
$$\langle n; (n', \sigma \cup (\mathsf{z}, \nu), \kappa); (H'', \mathsf{z} \mapsto (\tau_1' \longrightarrow^{\Phi_f'} \tau_2', \lambda(x).e'', \nu)); e''[x \mapsto v]\rangle$$

case $e_1 \not\equiv v$ **:**

Let $\mathbb{E} \equiv \_\ e_2$ so that $e \equiv \mathbb{E}[e_1]$. Since e_1 is a not value, $\mathcal{R}_2 \equiv \cdot$ hence we have $\Phi_1, \mathcal{R}; H \vdash \Sigma$ by Lemma C.0.26 and we can apply induction and we have: $\langle n; \Sigma; H; e_1\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e_1'\rangle$, and thus $\langle n; \Sigma; H; \mathbb{E}[e_1]\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; \mathbb{E}[e_1']\rangle$ by [CONG].

case $e_1 \equiv v_1, e_2 \not\equiv v$ **:**

Let $\mathbb{E} \equiv v_1\ \_$ so that $e \equiv \mathbb{E}[e_2]$. Since e_1 is a value, $\mathcal{R}_1 \equiv \cdot$ hence we have $\Phi_2, \mathcal{R}; H \vdash \Sigma$ by Lemma C.0.26 and we can apply induction. The rest follows similarly to the above case.

case (TSub) **:**

We know that:

$$\text{(TSub)}\dfrac{\Phi_1; \Gamma \vdash e : \tau' \rightsquigarrow \mathcal{R} \qquad \tau' \leq \tau \\ \Phi_1 \equiv [\alpha; \varepsilon_1; \omega; \delta_i; \delta_o] \quad \Phi \equiv [\alpha; \varepsilon; \omega; \delta_i; \delta_o] \quad \varepsilon_1 \subseteq \varepsilon}{\Phi; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}}$$

If e is a value v we are done. Otherwise, since $\Phi_1, \mathcal{R}; H \vdash \Sigma$ follows from $\Phi, \mathcal{R}; H \vdash \Sigma$ (by $\Phi_1^\varepsilon \subseteq \Phi^\varepsilon$ and $\Phi_1^\alpha = \Phi^\alpha$); we have $\langle n; \Sigma; H; e\rangle \longrightarrow_\eta \langle n'; \Sigma'; H'; e'\rangle$ by induction.

□

Lemma C.0.33 (Substitution).
If $\Phi; \Gamma, x : \tau' \vdash e : \tau$ *and* $\Phi; \Gamma \vdash v : \tau'$ *then* $\Phi; \Gamma \vdash e[x \mapsto v] : \tau$.

Proof. Induction on the typing derivation of $\Phi; \Gamma \vdash e : \tau$.

case (TInt) **:**

Since $e \equiv n$ and $n[x \mapsto v] \equiv n$, the result follows by (TInt).

case (TVar) **:**

e is a variable y. We have two cases:

case $y = x$ **:**

We have $\tau = \tau'$ and $y[x \mapsto v] \equiv v$, hence we need to prove that $\Phi; \Gamma \vdash v : \tau$ which is true by assumption.

case $y \neq x$ **:**

We have $y[x \mapsto v] \equiv y$ and need to prove that $\Phi; \Gamma \vdash y : \tau$. By assumption, $\Phi; \Gamma, x : \tau' \vdash y : \tau$, and thus $(\Gamma, x : \tau')(y) = \tau$; but since $x \neq y$ this implies $\Gamma(y) = \tau$ and we have to prove $\Phi; \Gamma \vdash y : \tau$ which follows by (Tvar).

case (TGvar),(TLoc), (TCheckin) **:**

Similar to (TInt).

case (TRef) **:**

We know that $\Phi; \Gamma, x : \tau' \vdash \mathsf{ref}\ e : \mathit{ref}^{\,\varepsilon}\ \tau$ and $\Phi; \Gamma \vdash v : \tau'$, and need to prove that $\Phi; \Gamma \vdash (\mathsf{ref}\ e)[x \mapsto v] : \mathit{ref}^{\,\varepsilon}\ \tau$. By inversion on $\Phi; \Gamma, x : \tau' \vdash \mathsf{ref}\ e : \mathit{ref}^{\,\varepsilon}\ \tau$ we have $\Phi; \Gamma, x : \tau' \vdash e : \tau$; applying induction to this, we have $\Phi; \Gamma \vdash e[x \mapsto v] : \tau$. We can now apply [TRef]:

$$\text{(TRef)}\dfrac{\Phi; \Gamma \vdash e[x \mapsto v] : \tau}{\Phi; \Gamma \vdash \mathsf{ref}\ (e[x \mapsto v]) : \mathit{ref}^{\,\varepsilon}\ \tau}$$

The desired result follows since $\mathsf{ref}\ (e[x \mapsto v]) \equiv (\mathsf{ref}\ e)[x \mapsto v]$.

case (TDeref) **:**

We know that $\Phi;\Gamma,x:\tau'\vdash\,!\,e:\tau$ and $\Phi;\Gamma\vdash v:\tau'$ and need to prove that $\Phi;\Gamma\vdash(!\,e)[x\mapsto v]:\tau$. By inversion on $\Phi;\Gamma,x:\tau'\vdash\,!\,e:\tau$ we have $\Phi_1;\Gamma,x:\tau'\vdash e:\mathit{ref}^{\,\varepsilon_r}\ \tau$ and Φ_2 such that $\Phi_1\rhd\Phi_2\hookrightarrow\Phi$ and $\Phi\equiv\Phi_1\rhd\Phi_2$. By value typing we have $\Phi_1;\Gamma\vdash v:\tau'$. We can then apply induction, yielding $\Phi_1;\Gamma\vdash e[x\mapsto v]:\mathit{ref}^{\,\varepsilon_r}\ \tau$. Finally, we apply (TDeref)

$$(\text{TDeref})\frac{\begin{array}{c}\Phi_1;\Gamma\vdash e[x\mapsto v]:\mathit{ref}^{\,\varepsilon_r}\ \tau\\ \Phi_2^{\varepsilon}=\varepsilon_r\qquad\Phi_2^{\delta_i}=\Phi_2^{\delta_o}\cup\varepsilon_r\\ \Phi_1\rhd\Phi_2\hookrightarrow\Phi\end{array}}{\Phi;\Gamma\vdash\,!\,e[x\mapsto v]:\tau}$$

Note that the second premise holds by inversion on $\Phi;\Gamma,x:\tau'\vdash\,!\,e:\tau$. The desired result follows since $!\,(e[x\mapsto v])\equiv(!\,e)[x\mapsto v]$.

case (TSub) **:**

We know that $\Phi;\Gamma,x:\tau'\vdash e:\tau$ and $\Phi;\Gamma\vdash v:\tau'$ and need to prove that $\Phi;\Gamma\vdash e[x\mapsto v]:\tau$. By inversion on $\Phi;\Gamma,x:\tau'\vdash e:\tau$ we have $\Phi';\Gamma,x:\tau'\vdash e:\tau'$. By value typing we have $\Phi';\Gamma,x:\tau'\vdash v:\tau'$. We can then apply induction, yielding $\Phi';\Gamma\vdash e[x\mapsto v]:\tau'$. Finally, we apply (TSub)

$$(\text{TSub})\frac{\Phi';\Gamma\vdash e[x\mapsto v]:\tau'\qquad\tau'\leq\tau\qquad\Phi'\leq\Phi}{\Phi;\Gamma\vdash e[x\mapsto v]:\tau}$$

and get the desired result.

case (TTransact),(TIntrans) **:**

Similar to (TSub).

case (TApp) **:**

We know that

$$(\text{TApp})\frac{\begin{array}{c}\Phi_1;\Gamma,x:\tau'\vdash e_1:\tau_1\longrightarrow^{\Phi_f}\tau_2\qquad\Phi_2;\Gamma,x:\tau'\vdash e_2:\tau_1\\ \Phi_1\rhd\Phi_2\rhd\Phi_3\hookrightarrow\Phi\\ \Phi_3^{\varepsilon}=\Phi_f^{\varepsilon}\qquad\Phi_3^{\alpha}\subseteq\Phi_f^{\alpha}\qquad\Phi_3^{\omega}\subseteq\Phi_f^{\omega}\\ \Phi_3^{\delta_i}=\Phi_3^{\delta_o}\cup\Phi_f^{\varepsilon}\qquad\Phi_3^{\delta_o}\subseteq\Phi_f^{\delta_o}\end{array}}{\Phi;\Gamma,x:\tau'\vdash e_1\ e_2:\tau_2}$$

where $\Phi;\Gamma\vdash v:\tau'$, and need to prove that $\Phi;\Gamma\vdash(e_1\ e_2)[x\mapsto v]:\tau_2$. Call the first two premises above (1) and (2), and note that we have (3) $\Phi;\Gamma\vdash v:\tau'\Rightarrow\Phi_1;\Gamma\vdash v:\tau'$ and (4) $\Phi;\Gamma\vdash v:\tau'\Rightarrow\Phi_2;\Gamma\vdash v:\tau'$ by the value typing lemma. By (1), (3) and induction we have $\Phi_1;\Gamma\vdash e_1[x\mapsto v]:\tau_1\longrightarrow^{\Phi_f}\tau_2$. Similarly, by (2), (4) and induction we have $\Phi_2;\Gamma\vdash e_2[x\mapsto v]:\tau_1$. We can now apply (TApp):

$$(\text{TApp})\frac{\begin{array}{c}\Phi_1;\Gamma\vdash e_1[x\mapsto v]:\tau_1\longrightarrow^{\Phi_f}\tau_2\qquad\Phi_2;\Gamma\vdash e_2[x\mapsto v]:\tau_1\\ \Phi_1\rhd\Phi_2\rhd\Phi_3\hookrightarrow\Phi\\ \Phi_3^{\varepsilon}=\Phi_f^{\varepsilon}\qquad\Phi_3^{\alpha}\subseteq\Phi_f^{\alpha}\qquad\Phi_3^{\omega}\subseteq\Phi_f^{\omega}\\ \Phi_3^{\delta_i}=\Phi_3^{\delta_o}\cup\Phi_f^{\varepsilon}\qquad\Phi_3^{\delta_o}\subseteq\Phi_f^{\delta_o}\end{array}}{\Phi;\Gamma\vdash e_1[x\mapsto v]\ e_2[x\mapsto v]:\tau_2}$$

Since $e_1[x\mapsto v]\ e_2[x\mapsto v]\equiv(e_1\ e_2)[x\mapsto v]$ we get the desired result.

case (TAssign-TIf-TLet) **:**

Similar to (TApp).

□

Appendix D
Multi-threading Proofs

Lemma D.0.34 (Fork derivations)**.** *If* $\Phi; \Gamma \vdash \mathbb{E}[\mathsf{fork}^{\alpha,\omega}\ e] : \tau \rightsquigarrow \mathcal{R}$ *then* $\Phi; \Gamma \vdash \mathbb{E}[0] : \tau \rightsquigarrow \mathcal{R}$.

Proof. By induction on $\mathbb{E}$:

case $\mathbb{E} = \_$ **:**

By assumption, we have $\Phi; \Gamma \vdash \mathsf{fork}^{\alpha,\omega}\ e : int \rightsquigarrow \cdot$. We have $\Phi; \Gamma \vdash 0 : int \rightsquigarrow \cdot$ by (TInt).

case $\mathbb{E} = v\ \mathbb{E}'$ **:**

By assumption, we have $\Phi; \Gamma \vdash v\ \mathbb{E}'[\mathsf{fork}^{\alpha'',\omega''}\ e] : \tau \rightsquigarrow \mathcal{R}$. By subtyping derivations (Lemma B.0.6) we know we can construct a proof derivation of this ending in (TSub):

$$\text{TSub}\dfrac{\text{TApp}\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash v : \tau_1 \longrightarrow^{\Phi_f} \tau_2' \rightsquigarrow \cdot \qquad \Phi_2; \Gamma \vdash \mathbb{E}'[\mathsf{fork}^{\alpha'',\omega''}\ e] : \tau_1 \rightsquigarrow \mathcal{R} \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi' \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \Phi_f^{\varepsilon} \qquad \Phi_3^{\delta_o} \subseteq \Phi_f^{\delta_o}\end{array}}{\Phi'; \Gamma \vdash v\ \mathbb{E}'[\mathsf{fork}^{\alpha'',\omega''}\ e] : \tau_2' \rightsquigarrow \mathcal{R}} \quad \tau_2' \leq \tau \quad \Phi' \equiv [\alpha; \varepsilon'; \omega] \quad \Phi \equiv [\alpha; \varepsilon; \omega] \quad \varepsilon' \subseteq \varepsilon}{\Phi; \Gamma \vdash v\ \mathbb{E}'[\mathsf{fork}^{\alpha'',\omega''}\ e] : \tau \rightsquigarrow \mathcal{R}}$$

By induction we have $\Phi_2; \Gamma \vdash \mathbb{E}'[0] : \tau_1 \rightsquigarrow \mathcal{R}$. We can now apply (TApp):

$$\text{TSub}\dfrac{\text{TApp}\dfrac{\begin{array}{c}\Phi_1; \Gamma \vdash v : \tau_1 \longrightarrow^{\Phi_f} \tau_2' \rightsquigarrow \cdot \qquad \Phi_2; \Gamma \vdash \mathbb{E}'[0] : \tau_1 \rightsquigarrow \mathcal{R} \\ \Phi_1 \rhd \Phi_2 \rhd \Phi_3 \hookrightarrow \Phi' \\ \Phi_3^{\varepsilon} = \Phi_f^{\varepsilon} \qquad \Phi_3^{\alpha} \subseteq \Phi_f^{\alpha} \qquad \Phi_3^{\omega} \subseteq \Phi_f^{\omega} \\ \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \Phi_f^{\varepsilon} \qquad \Phi_3^{\delta_o} \subseteq \Phi_f^{\delta_o}\end{array}}{\Phi'; \Gamma \vdash v\ \mathbb{E}'[0] : \tau_2' \rightsquigarrow \mathcal{R}} \quad \tau_2' \leq \tau \quad \Phi' \equiv [\alpha; \varepsilon'; \omega] \quad \Phi \equiv [\alpha; \varepsilon; \omega] \quad \varepsilon' \subseteq \varepsilon}{\Phi; \Gamma \vdash v\ \mathbb{E}'[0] : \tau \rightsquigarrow \mathcal{R}}$$

case all others **:**

By induction, similar to above cases.

□

Preservation is very similar to the single-threaded version (Lemma C.0.31). $n; \Gamma \vdash H$ is unchanged since it's independent of the numbers of threads. We require $\Phi_i; \Gamma \vdash e_i : \tau \rightsquigarrow \mathcal{R}_i \wedge \Phi_i, \mathcal{R}_i; H \vdash \Sigma_i \wedge \mathit{traceOK}(\Sigma_i) \Rightarrow \Phi_i'; \Gamma \vdash e_i' : \tau \rightsquigarrow \mathcal{R}_i' \wedge \Phi_i', \mathcal{R}_i'; H' \vdash \Sigma_i' \wedge \mathit{traceOK}(\Sigma_i')$ for each thread i, which we can prove by invoking the single-threaded proof and paying attention to MT-specific issues like (TFork) and (TReturn) that create and destroy a thread, respectively.

Lemma D.0.35 (Multithreaded VC non-interference)**.**
Let $\mathcal{T} = (\Sigma_1, e_1).(\Sigma_2, e_2)\ldots(\Sigma_{|\mathcal{T}|}, e_{|\mathcal{T}|})$. *Suppose we have the following:*

1. $n \vdash H, \mathcal{T}$

2. $\forall i \in 1..|\mathcal{T}|.\ \Phi_i, \mathcal{R}_i; H \vdash \Sigma_i$

and thread j *takes a non-update evaluation step:* $\langle n; \Sigma_j; H; e\rangle \longrightarrow_{\varepsilon} \langle n'; \Sigma_j'; H'; e'\rangle$ *Then for some* $\Gamma' \supseteq \Gamma$ *and for all threads* $i \in 1..|\mathcal{T}|'$ *such that* $i \neq j$ *we have:*

1. $\Phi_i; \Gamma' \vdash e_i : \tau \rightsquigarrow \mathcal{R}_i$

2. $\Phi_i, \mathcal{R}_i; H' \vdash \Sigma_i$

3. $\mathit{traceOK}(\Sigma_i')$

Proof. Part 1. is true by weakening since $\Gamma' \supseteq \Gamma$.

We only need to prove 2. Proceed by induction on the typing derivation $\Phi_j; \Gamma \vdash e : \tau \rightsquigarrow \mathcal{R}_j$, only considering rules that change the heap:

case (TRef) **:**

We have that:

$$\text{(TRef)}\frac{\Phi_j; \Gamma \vdash e_j : \tau \rightsquigarrow \mathcal{R}}{\Phi_j; \Gamma \vdash \mathsf{ref}\ e_j : \mathit{ref}^{\,\varepsilon}\ \tau \rightsquigarrow \mathcal{R}}$$

There are two possible reductions:

case [ref] **:**

We have that $e_j \equiv v$, $\mathcal{R} = \cdot$, and $\langle n; (n', \sigma_j, \kappa_j); H; \mathsf{ref}\ v\rangle \longrightarrow_{\emptyset} \langle n; (n', \sigma_j, \kappa_j); H'; r\rangle$ where $r \notin \mathrm{dom}(H)$, $H' = H, r \mapsto (\cdot, v, \emptyset)$, and $\Gamma' = \Gamma, r : \mathit{ref}^{\,\varepsilon}\ \tau$.

To prove 2., we must show $\Phi_i, \mathcal{R}_i; H' \vdash \Sigma_i$. This follows by assumption since H' only contains an additional location (i.e., not a global variable) and no heap element has undergone a version change.

case [cong] **:**

We have $\langle n; \Sigma_j; H; \mathsf{ref}\ \mathbb{E}[e'']\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'_j; H'; \mathsf{ref}\ \mathbb{E}[e''']\rangle$ from $\langle n; \Sigma_j; H; e''\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'_j; H'; e'''\rangle$. By [cong], we have $\langle n; \Sigma_j; H; e\rangle \longrightarrow_{\varepsilon} \langle n; \Sigma'_j; H'; e'\rangle$ where $e \equiv \mathbb{E}[e'']$ and $e' \equiv \mathbb{E}[e''']$.

The result follows directly by induction.

case (TAssign) **:**

We know that:

$$\text{(TAssign)}\frac{\begin{array}{cc}\Phi_{j1}; \Gamma \vdash e_1 : \mathit{ref}^{\,\varepsilon_r}\ \tau \rightsquigarrow \mathcal{R}_{j1} & \Phi_{j2}; \Gamma \vdash e_2 : \tau \rightsquigarrow \mathcal{R}_{j2} \\ \Phi_3^{\varepsilon} = \varepsilon_r & \Phi_3^{\delta_i} = \Phi_3^{\delta_o} \cup \varepsilon_r \\ \multicolumn{2}{c}{\Phi_{j1} \rhd \Phi_{j2} \rhd \Phi_3 \hookrightarrow \Phi_j}\end{array}}{\Phi_j; \Gamma \vdash e_1 := e_2 : \tau \rightsquigarrow \mathcal{R}_{j1} \bowtie \mathcal{R}_{j2}}$$

From $\mathcal{R}_{j1} \bowtie \mathcal{R}_{j2}$ it follows that either $\mathcal{R}_{j1} \equiv \cdot$ or $\mathcal{R}_{j2} \equiv \cdot$.

We can reduce using [gvar-assign], [assign], or [cong].

case [gvar-assign] **:**

This implies that $e \equiv \mathsf{z} := v$ with

$$\langle n; (n', \sigma_j, \kappa_j); (H'', \mathsf{z} \mapsto (\tau, v', \nu)); \mathsf{z} := v\rangle \longrightarrow_{\{\mathsf{z}\}} \langle n; (n'_j, \sigma_j \cup (\mathsf{z}, \nu), \kappa_j); (H'', \mathsf{z} \mapsto (\tau, v, \nu)); v\rangle$$

where $H \equiv (H'', \mathsf{z} \mapsto (\tau, v', \nu))$. $\mathcal{R}_{j1} \equiv \cdot$ and $\mathcal{R}_{j2} \equiv \cdot$ (thus $\mathcal{R}_{j1} \bowtie \mathcal{R}_{j2} \equiv \cdot$), $\Gamma' = \Gamma$, $\mathcal{R}'_j = \cdot$.

Consider the case $\Phi_i \equiv (n', \sigma_i, \kappa_i)$. We know

$$\text{(TC1)}\frac{\begin{array}{c}\mathsf{f} \in \sigma_i \Rightarrow \mathsf{f} \in \alpha_i \\ \mathsf{f} \in (\varepsilon_i \cap \delta_{ii}) \Rightarrow n_i \in \mathit{ver}((H'', \mathsf{z} \mapsto (\tau, v, \nu)), \mathsf{f}) \\ \kappa_i^{\alpha} \supseteq (\alpha_i \cup \delta_{ii}) \\ \kappa_i^{\omega} \supseteq (\omega_i \cup \varepsilon_i)\end{array}}{[\alpha_i; \varepsilon_i; \omega_i], \cdot; H \vdash (n_i, \sigma_i, \kappa_i)}$$

and need to prove:

$$\text{(TC1)}\frac{\begin{array}{c}\mathsf{f} \in \sigma_i \Rightarrow \mathsf{f} \in \alpha_i \\ \mathsf{f} \in (\varepsilon_i \cap \delta_{ii}) \Rightarrow n_i \in \mathit{ver}((H'', \mathsf{z} \mapsto (\tau, v', \nu)), \mathsf{f}) \\ \kappa_i^{\alpha} \supseteq (\alpha_i \cup \delta_{ii}) \\ \kappa_i^{\omega} \supseteq (\omega_i \cup \varepsilon_i)\end{array}}{[\alpha_i; \varepsilon_i; \omega_i], \cdot; H \vdash (n_i, \sigma_i, \kappa_i)}$$

All premises follow by assumption (no heap element has changed version). If $\Phi_i \equiv (n'_i, \sigma_i \cup (\mathsf{z}, \nu), \kappa_i)$, the result follows by the same argument.

case [assign] **:**

The proof follows by the same argument as in case (TDeref)-[deref].

case [cong] **:**

In both cases ($\mathbb{E} := e$, $r := \mathbb{E}$) the result follows by induction.

□

Lemma D.0.36 (Preservation).
Let $\mathcal{T} = (\Sigma_1, e_1).(\Sigma_2, e_2)\ldots(\Sigma_{|\mathcal{T}|}, e_{|\mathcal{T}|})$. Suppose we have the following:

1. $n \vdash H, \mathcal{T}$
2. $\forall i \in 1..|\mathcal{T}|.\ \Phi_i, \mathcal{R}_i; H \vdash \Sigma_i$
3. $\forall i \in 1..|\mathcal{T}|.\ \mathit{traceOK}(\Sigma_i)$
4. $\langle n; \mathcal{T}; H\rangle \longrightarrow_{(\varepsilon,j)} \langle n'; \mathcal{T}'; H'\rangle$

where j is the thread taking a transition and ε is the evaluation effect. Then, for $\mathcal{T}' = (\Sigma'_1, e'_1).(\Sigma'_2, e'_2)\ldots(\Sigma'_{|\mathcal{T}|}, e'_{|\mathcal{T}|})$ we have that:

1. $n' \vdash H', \mathcal{T}'$ *(such that $n'; \Gamma' \vdash H'$ and $\forall i \in 1..|\mathcal{T}|'.\ \Phi'_i; \Gamma' \vdash e'_i : \tau \rightsquigarrow \mathcal{R}'_i$ for some n', $\Gamma' \supseteq \Gamma$ and some Φ'_i such that*
   - $\Phi'_i = \Phi_i$, *if* $i \neq j$
   - $\Phi'_i \equiv [\Phi^{\alpha}_i \cup \varepsilon_0; \varepsilon'_i; \Phi^{\omega}_i; \Phi^{\delta_i}_i; \Phi^{\delta_o}_i]$, $\varepsilon'_i \cup \varepsilon_0 \subseteq \Phi^{\varepsilon}_i$, *if* $i = j$
2. $\forall i \in 1..|\mathcal{T}|'.\ \Phi'_i, \mathcal{R}'_i; H' \vdash \Sigma'_i$
3. $\forall i \in 1..|\mathcal{T}|'.\ \mathit{traceOK}(\Sigma'_i)$

Proof. By case analysis on the rule used to take an evaluation step:

case (FORK) **:**

We have
$$\langle n; H; \mathcal{T}_1.(\Sigma_j, \mathbb{E}[\mathsf{fork}^{\kappa}\ e]).\mathcal{T}_2\rangle \ \Rightarrow_{(\emptyset,j)}\ \langle n; H; \mathcal{T}_1.(\Sigma_j, \mathbb{E}[0]).((n,\emptyset,\kappa), e).\mathcal{T}_2\rangle$$

and we know that
$$\text{TFork}\frac{\Phi_m; \Gamma \vdash e : \tau \rightsquigarrow \cdot \qquad \Phi_m \equiv [\alpha_m; \varepsilon_m; \omega_m] \qquad \kappa_m \equiv (\alpha_m \cup \delta_{im},\ \varepsilon_m)}{\Phi_j; \Gamma \vdash \mathsf{fork}^{\kappa_m}\ e : \mathit{int} \rightsquigarrow \cdot}$$

Thus $n' = n$ and $H' = H$; let $\Gamma' = \Gamma$. We have $n; \Gamma \vdash H$ by assumption, so $n'; \Gamma' \vdash H'$ is immediate. Let j be the index of the thread whose context is $\mathbb{E}[\mathsf{fork}^{\kappa}\ e]$, and m the index of the newly created thread. We get 1. and 2. for all threads $i \in 1..|\mathcal{T}|, i \neq j, m$ by assumption (since $H' = H$) and choosing $\Phi'_i = \Phi_i$. For thread j, we have $\Phi_j; \Gamma \vdash \mathbb{E}[\mathsf{fork}^{\kappa}\ e] : \mathit{int} \rightsquigarrow \cdot$ and need to prove $\Phi'_j; \Gamma \vdash \mathbb{E}[0] : \mathit{int} \rightsquigarrow \cdot$. Let $\Phi'_j = \Phi_j$; then $\Phi'_j; \Gamma \vdash \mathbb{E}[0] : \mathit{int} \rightsquigarrow \cdot$ follows by Lemma D.0.34. Part 2., $\Phi'_j, \cdot; H' \vdash \Sigma'_j$ follows by assumption since $\Phi'_j = \Phi_j$ and $\Sigma'_j = \Sigma_j$. Part 3. similarly follows by assumption since $\Sigma'_j = \Sigma_j$.

For the newly created thread m, we need to prove $\Phi_m; \Gamma \vdash e : \tau \rightsquigarrow \cdot$ which follows by assumption (from (TFORK)), and 2., $\Phi_m, \cdot; H' \vdash \Sigma_m$, which we prove by [VC1]:

$$(\text{TC1})\frac{\begin{array}{c}\mathsf{f} \in \sigma_m \Rightarrow \mathsf{f} \in \alpha_m \\ \mathsf{f} \in (\varepsilon_m \cap \delta_{im}) \Rightarrow n_m \in \mathit{ver}(H, \mathsf{f}) \\ \kappa^{\alpha}_m \supseteq (\alpha_m \cup \delta_{im}) \\ \kappa^{\omega}_m \supseteq (\omega_m \cup \varepsilon_m)\end{array}}{[\alpha_m; \varepsilon_m; \omega_m], \cdot; H \vdash (n_m, \sigma_m, \kappa_m)}$$

Since $(n, \emptyset, \kappa)$ is the new thread context (from [FORK]), by inversion on $\Phi_m; \Gamma \vdash e : \tau \rightsquigarrow \cdot$ we have $\Sigma_m \equiv (n_m, \sigma_m, \kappa_m)$ hence $(n_m, \sigma_m, \kappa) \equiv (n, \emptyset, (\alpha_m \cup \delta_{im},\ \varepsilon_m))$. The first premise is vacuously true since $\sigma_m \equiv \emptyset$. The second premise follows directly from $n; \Gamma \vdash H$ (which states $\forall \mathsf{z} \mapsto (\tau, b, \nu) \in H.\quad n \in \nu$) since $n_m \equiv n$. The third and fourth premises follow directly since $\kappa_m \equiv (\alpha_m \cup \delta_{im},\ \varepsilon_m)$ For part 3 we need to prove $\mathit{traceOK}(n_m, \sigma_m, \kappa)$ which is vacuously true since $\sigma_m \equiv \emptyset$.

case (RETURN) **:**

We have
$$\langle n; H; \mathcal{T}_1.((n'', \sigma'', \kappa''), v).\mathcal{T}_2\rangle \ \Rightarrow_{\emptyset}\ \langle n; H; \mathcal{T}_1.\mathcal{T}_2\rangle$$

hence $n' = n$ and $H' = H$. Let $\Gamma' = \Gamma$; then 1., 2., and 3. follow by assumption for all threads.

case (UPDATE) **:**

We know that

$$\frac{\begin{array}{c}\langle n;\Sigma_1;H;e_1\rangle \ \longrightarrow_{(upd,dir)}\ \langle n+1;\mathcal{U}[\Sigma_1]^{upd,dir}_{n+1};\mathcal{U}[H]^{upd}_{n+1};e_1\rangle \\ \langle n;\Sigma_2;H;e_2\rangle \ \longrightarrow_{(upd,dir)}\ \langle n+1;\mathcal{U}[\Sigma_2]^{upd,dir}_{n+1};\mathcal{U}[H]^{upd}_{n+1};e_2\rangle \\ \ldots \\ \langle n;\Sigma_{|\mathcal{T}|};H;e_{|\mathcal{T}|}\rangle \ \longrightarrow_{(upd,dir)}\ \langle n+1;\mathcal{U}\left[\Sigma_{|\mathcal{T}|}\right]^{upd,dir}_{n+1};\mathcal{U}[H]^{upd}_{n+1};e_{|\mathcal{T}|}\rangle\end{array}}{\begin{array}{l}\langle n;H;(\Sigma_1,e_1).(\Sigma_2,e_2)\ldots(\Sigma_{|\mathcal{T}|},e_{|\mathcal{T}|})\rangle \ \rightrightarrows_{(upd,dir)} \\ \langle n+1;\mathcal{U}[H]^{upd}_{n+1};(\mathcal{U}[\Sigma_1]^{upd,dir}_{n+1},e_1).(\mathcal{U}[\Sigma_2]^{upd,dir}_{n+1},e_2)\ldots(\mathcal{U}\left[\Sigma_{|\mathcal{T}|}\right]^{upd,dir}_{n+1},e_{|\mathcal{T}|})\rangle\end{array}}$$

We have $n' = n+1$. Let $\Gamma' = \mathcal{U}[\Gamma]^{upd}$; we have $H' = \mathcal{U}[H]^{upd}_{n+1}$.

$n+1;\Gamma' \vdash H'$ follows directly from Lemma B.0.12. For each thread i, we have

$$\langle n;\Sigma_i;H;e_i\rangle \ \longrightarrow_{(upd,dir)}\ \langle n+1;\mathcal{U}[\Sigma_i]^{upd,dir}_{n+1};\mathcal{U}[H]^{upd}_{n+1};e\rangle$$

hence $\Phi'_i;\Gamma' \vdash e'_i : \tau \rightsquigarrow \mathcal{R}_i$, $\Phi'_i,\mathcal{R}_i;H' \vdash \Sigma'_i$, and $traceOK(\Sigma'_i)$ for each i follow from single-threaded update preservation (Lemma B.0.13).

case (MT-CONG) **:**

Let j be the thread that takes a step. We have

$$\frac{\langle n;\Sigma_j;H;e\rangle \ \longrightarrow_{\eta}\ \langle n';\Sigma'_j;H';e'\rangle}{\langle n;H;\mathcal{T}_1.(\Sigma_j,\mathbb{E}[e]).\mathcal{T}_2\rangle \ \rightrightarrows_{(\eta,j)}\ \langle n';H';\mathcal{T}_1.(\Sigma'_j,\mathbb{E}[e']).\mathcal{T}_2\rangle}$$

hence $n';\Gamma' \vdash H'$, $\Phi'_j;\Gamma \vdash e'_j : \tau \rightsquigarrow \mathcal{R}'_j$, $\Phi'_j,\mathcal{R}'_j;H' \vdash \Sigma'_j$, and $traceOK(\Sigma'_j)$ follow from single-threaded preservation (Lemma D.0.36). For all threads $i \in 1..|\mathcal{T}|, i \neq j$ we have $\Phi'_i = \Phi_i$, $\mathcal{R}_i = \mathcal{R}'_i$, and $\Sigma'_i = \Sigma_i$ since they don't take any steps. Hence we have $\Phi'_i;\Gamma' \vdash e'_i : \tau \rightsquigarrow \mathcal{R}'_i$ by assumption and weakening, $\Phi'_i,\mathcal{R}'_i;H' \vdash \Sigma'_i$ by assumption, and the observation that the only way j could have changed the heap was via [GVAR-ASSIGN], [ASSIGN], or [REF], but this does not affect $\Phi'_i,\mathcal{R}'_i;H' \vdash \Sigma'_i$ (by Lemma D.0.35). Finally, we have $traceOK(\Sigma'_i)$ by assumption, since $\Sigma'_i = \Sigma_i$.

□

Progress is also similar to the single-threaded version; we pick a thread, and prove that it can take a step.

Lemma D.0.37 (Progress). *Let $\mathcal{T} = (\Sigma_1,e_1).(\Sigma_2,e_2)\ldots(\Sigma_{|\mathcal{T}|},e_{|\mathcal{T}|})$. Suppose we have the following:*

1. $n \vdash H, \mathcal{T}$
2. $\forall i \in 1..|\mathcal{T}|.\ \Phi_i,\mathcal{R}_i;H \vdash \Sigma_i$
3. $\forall i \in 1..|\mathcal{T}|.\ traceOK(\Sigma_i)$

Then for all Σ_i such that $\Phi_i,\mathcal{R}_i;H \vdash \Sigma_i$, and $traceOK(\Sigma_i)$, either e_i is a value, or there exist $n',H',\mathcal{T}'$ such that $n;H;\mathcal{T} \ \longrightarrow_{(\varepsilon,j)}\ n';H';\mathcal{T}'$.

Proof. Case analysis on the structure of the entire program. Assume $|\mathcal{T}| > 0$ and consider e_i, for some i such that $1 \leq i \leq |\mathcal{T}|$.

case $e_i \equiv v$ **:**

The thread context is (Σ_i, v). By assumption, we have $\Phi_i;\Gamma \vdash e_i : \tau \rightsquigarrow \mathcal{R}_i$ which in our case means $\Phi_i;\Gamma \vdash v : \tau \rightsquigarrow \cdot$ so $\mathcal{R} \equiv \cdot$, hence by inversion on $\Phi_i,\mathcal{R}_i;H \vdash \Sigma_i$ we have $\Sigma_i \equiv (n'',\sigma'',\kappa'')$ and we can reduce via [RETURN]:

$$\langle n;H;\mathcal{T}_1.((n'',\sigma'',\kappa''),v).\mathcal{T}_2\rangle \ \rightrightarrows_{\emptyset}\ \langle n;H;\mathcal{T}_1.\mathcal{T}_2\rangle$$

case $e_i \not\equiv v$ **:**

We have

$$\langle n;\Sigma_i;H;e_i\rangle \ \longrightarrow_{\ \eta}\ \langle n';\Sigma'_i;H';e'_i\rangle \qquad \eta \in \{upd, \epsilon\}$$

from single-threaded progress (Lemma D.0.37). Proceed by case analysis on α:

case ϵ **:**

case $e_i \equiv \mathbb{E}[\mathsf{fork}^{\omega}\ e]$ **:**
We can reduce using [FORK]:

$$\langle n; H; \mathcal{T}_1.(\Sigma_i, \mathbb{E}[\mathsf{fork}^{\kappa}\ e]).\mathcal{T}_2\rangle \ \rightrightarrows_{(\emptyset,i)} \ \langle n; H; \mathcal{T}_1.(\Sigma_i, \mathbb{E}[0]).((n, \emptyset, \kappa), e).\mathcal{T}_2\rangle$$

case $e_i \not\equiv \mathbb{E}[\mathsf{fork}^{\omega}\ e]$ **:**
We can apply [MT-CONG]:

$$\frac{\langle n; \Sigma_i; H; e\rangle \ \longrightarrow_{\eta} \ \langle n'; \Sigma_i'; H'; e'\rangle}{\langle n; H; \mathcal{T}_1.(\Sigma_i, \mathbb{E}[e]).\mathcal{T}_2\rangle \ \rightrightarrows_{(\eta,i)} \ \langle n'; H'; \mathcal{T}_1.(\Sigma_i', \mathbb{E}[e']).\mathcal{T}_2\rangle}$$

□

Bibliography

[1] Ksplice: Rebootless Linux kernel security updates. `http://web.mit.edu/ksplice/`.

[2] Sun Microsystems. Java HotSpot VM. `http://www.javasoft.com/products/hotspot`.

[3] Sun Microsystems. JVM Tool Interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti`.

[4] Martín Abadi and Cédric Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121, 2003.

[5] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and Inferring Local Non-Aliasing. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 129–140, San Diego, California, June 2003.

[6] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus: online patches and updates for security. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 287–302, Berkeley, CA, USA, 2005. USENIX Association.

[7] Jesper Andersson, Marcus Comstedt, and Tobias Ritzau. Run-time support for dynamic java architectures. In *Proceedings of ECOOP Workshop on Object-Oriented Architectures*, 1998.

[8] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., 1996.

[9] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[10] Andrew Baumann, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Improving Operating System Availability With Dynamic Update. In *Proc. Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, pages 21–27, October 2004.

[11] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *USENIX Annual Technical Conference*, pages 337–350, 2007.

[12] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *USENIX Annual Technical Conference, General Track*, pages 279–291, 2005.

[13] David Berlind. Taking a closer look at Windows Vista. `http://news.cnet.com/Taking-a-closer-look-at-Windows-Vista/1606-2\_3-6200749.html`.

[14] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the International Conference on Software Maintenance 1992*, pages 41–50, 1992.

[15] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report TR-CIS-06-09, Department of Computer and Information Science University of Pennsylvania, May 2006.

[16] Daniel Pierre Bovet and Marco Cassetti. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[17] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 403–417, 2003.

[18] Eric A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[19] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter K. Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. In *ASPLOS*, pages 235–247, 2004.

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

[21] Cristiano Calcagno. Stratified Operational Semantics for Safety and Correctness of The Region Calculus. In *POPL'01*, pages 155–165, 2001.

[22] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self - a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, pages 49–70, 1989.

[23] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44, New York, NY, USA, 2006. ACM Press.

[24] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A POwerful Live Updating System. In *ICSE*, pages 271–281, 2007.

[25] Brian Chin, Shane Markstrum, and Todd D. Millstein. Semantic type qualifiers. In *PLDI*, pages 85–95, 2005.

[26] Marcus Denker and Stéphane Ducasse. Software evolution from the field: An experience report from the squeak maintainers. *Electr. Notes Theor. Comput. Sci.*, 166:81–91, 2007.

[27] Amol Deshpande and Michael Hicks. Toward on-line schema evolution for non-stop systems. Presented at the 11th High Performance Transaction Systems Workshop, September 2005.

[28] Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring. *Journal of Software Maintenance*, 18(2):83–107, 2006.

[29] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001*, October 2001.

[30] Mikhail Dmitriev. Design of JFluid: A profiling technology and tool based on dynamic bytecode instrumentation. Technical Report SMLI TR-2003-125, Sun Microsystems, November 2003.

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

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

[33] Tudor Dumitras, Jiaqi Tan, Zhengheng Gho, and Priya Narasimhan. No more hotdependencies: toward dependency-agnostic online upgrades in distributed systems. In *HotDep'07: Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, page 14, Berkeley, CA, USA, 2007. USENIX Association.

[34] Eagle Rock Alliance, Ltd. "2001 Cost of Downtime" Online Survey Results. www.contingencyplanningresearch.com/2001%20Survey.pdf.

[35] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI*, pages 170–182, 2008.

[36] Cormac Flanagan and Martín Abadi. Types for safe locking. In *ESOP*, pages 91–108, 1999.

[37] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *FME*, pages 500–517, 2001.

[38] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.

[39] Jeffrey Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality.* PhD thesis, University of California, Berkeley, December 2002.

[40] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-Insensitive Type Qualifiers. *TOPLAS*, 28(6):1035–1087, November 2006.

[41] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 268–277, Toronto, Ontario, Canada, June 1991.

[42] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: From concept to prototype. *The Journal of Systems and Software*, 14(2):111–128, February 1991.

[43] Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, LFCS, University of Edinburgh, 1997.

[44] A. Goldberg and D. Robson. *Smalltalk 80 - the Language and its Implementation.* Addison-Wesley, Reading, 1989.

[45] Patrick Gray. Experts question Windows patch policy. `http://news.zdnet.com/2100-1009\_22-5105454.html`.

[46] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI*, pages 13–25, 2003.

[47] D. Gupta. *On-line Software Version Change.* PhD thesis, Indian Institute of Technology, Kanpur, November 1994.

[48] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[49] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with blast. In *SPIN*, pages 235–239, 2003.

[50] M. W. Hicks. *Dynamic Software Updating.* PhD thesis, The University of Pennsylvania, August 2001.

[51] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Inferring locking for atomic sections. In *On-line Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[52] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes, a lightweight mechanism to update code in a running program. In *Proc. USENIX Annual Technical Conference*, pages 65–76, June 1998.

[53] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, June 1990.

[54] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[55] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *POPL*, pages 331–342, 2002.

[56] In-Stat. Online Gaming in Asia: Strong Potential for Growth. `http://www.instat.com/Abstract.asp?ID=318\&SKU=IN0804025CM`.

[57] Daniel Jackson and David A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252, September 1994.

[58] Java platform debugger architecture. `http://java.sun.com/j2se/1.4.2/docs/guide/jpda/`.

[59] Thomas Johnsson. Lambda lifting: Treansforming programs to recursive equations. In *FPCA*, pages 190–203, 1985.

[60] The K42 Project. `http://www.research.ibm.com/K42/`.

[61] John Kodumal and Alexander Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS*, pages 218–234, 2005.

[62] Brian Krebs. Cyber Incident Blamed for Nuclear Power Plant Shutdown. *Washington Post*, June 5, 2008.

[63] Insup Lee. *DYMOS: A Dynamic Modification System.* PhD thesis, Dept. of Computer Science, University of Wisconsin, Madison, April 1983.

[64] John R. Levine. *Linkers and Loaders.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[65] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. ASPLOS*, pages 211–223. ACM Press, 2004.

[66] John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming.* PhD thesis, MIT Laboratory for Computer Science, August 1987. MIT/LCS/TR-408.

[67] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, January 1988.

[68] Kristis Makris and Rida Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. Technical Report TR-08-007, Arizona State University, 2008.

[69] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys*, pages 327–340, 2007.

[70] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 337–361. Springer-Verlag, 2000.

[71] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *ICFP*, pages 213–225, 2003.

[72] P. E McKenney and J.D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. *International Conference on Parallel and Distributed Computing and Systems*, 1998.

[73] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *IWPSE*, pages 13–22, 2005.

[74] Microsoft. Visual Studio Debugger - Edit and Continue. `http://msdn.microsoft.com/en-us/library/bcew296c.aspx`.

[75] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[76] John C. Mitchell. Type inference with simple subtypes. *JFP*, 1(3):245–285, July 1991.

[77] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 1–5, May 2005.

[78] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 37–50, January 2008.

[79] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 72–83, New York, NY, USA, 2006. ACM Press.

[80] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *LNCS*, 2304:213–228, 2002.

[81] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPOPP*, pages 68–78, 2007.

[82] Jason Nieh. Autopod: Unscheduled system updates with zero data loss. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 367–368, Washington, DC, USA, 2005. IEEE Computer Society.

[83] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[84] David Oppenheimer, Aaron Brown, James Beck, Daniel Hettena, Jon Kuroda, Noah Treuhaft, David A. Patterson, and Katherine Yelick. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Comput.*, 51(2):100–107, 2002.

[85] David L. Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[86] A. Orso, A. Rao, and M.J. Harrold. A technique for dynamic updating of Java software. In *Proc. IEEE International Conference on Software Maintenance (ICSM)*, pages 649–658, 2002.

[87] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. volume 36, pages 361–376, New York, NY, USA, 2002. ACM.

[88] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.

[89] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):59–71, 2006.

[90] Steve Parker. A simple equation: IT on = Business on. *The IT Journal, Hewlett Packard*, 2001.

[91] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *PLDI*, pages 16–27, 1990.

[92] James S. Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, Computer Science Department, the University of Tennessee, 1997.

[93] Shaya Potter and Jason Nieh. Reducing downtime due to system maintenance and upgrades. In *LISA '05: Proceedings of the 19th conference on Large Installation System Administration Conference*, pages 47–62, Berkeley, CA, USA, 2005. USENIX Association.

[94] Polyvios Pratikakis. *Sound, Precise and Efficient Static Race Detection for Multi-Threaded Programs*. PhD thesis, University of Maryland, August 2008.

[95] Niels Provos. Libevent - an event notification library. `http://www.monkey.org/~provos/libevent/`.

[96] Mohan Rajagopalan, Somu Perinayagam, Haifeng He, Gregory Andrews, and Saumya Debray. Binary rewriting of an operating system kernel. *Workshop on Binary Instrumentation and Applications*, 2006.

[97] Eric Rescorla. Security holes... who cares? In *USENIX Security Symposium*, August 2003.

[98] Robin Rowe. Safety-Critical Systems Computer Language Survey, 1994. `http://vl.fmnet.info/safety/lang-survey.html`.

[99] Stelios Sidiroglou, Sotiris Ioannidis, and Angelos D. Keromytis. Band-aid patching. In *HotDep'07: Proceedings of the 3rd workshop on on Hot Topics in System Dependability*, pages 102–106, Berkeley, CA, USA, 2007. USENIX Association.

[100] Christian Skalka, Scott Smith, and David Van horn. Types and trace effects of higher order programs. *J. Funct. Program.*, 18(2):179–249, 2008.

[101] Fred Smith, David Walker, and Greg Morrisett. Alias types. In *ESOP*, pages 366–381, 2000.

[102] Jonathan M. Smith. A survey of process migration mechanisms. *ACM Operating Systems Review, SIGOPS*, 22(3):28–40, 1988.

[103] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc A. Auslander, Michal Ostrowski, Bryan S. Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *USENIX Annual Technical Conference, General Track*, pages 141–154, 2003.

[104] Don Stewart and Manuel M. T. Chakravarty. Dynamic applications from the ground up. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[105] Gareth Stoyle. *A Theory of Dynamic Software Updates*. PhD thesis, Computer Laboratory, University of Cambridge, July 2006.

[106] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 183–194, New York, NY, USA, 2005. ACM Press.

[107] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. *TOPLAS*, 29(4):22, August 2007.

[108] Cristian Tapus. *Distributed Speculations: Providing Fault-tolerance and Improving Performance*. PhD thesis, California Institute of Technology, June 2006.

[109] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[110] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *TOPLAS*, 24(4):701–771, July 2000.

[111] Chadd C. Williams and Jeffrey K. Hollingsworth. Recovering system specific rules from software repositories. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, 2005.

[112] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, New York, NY, USA, 1998. ACM Press.

[113] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM Press.

[114] Wuu Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21(7):739–755, 1991.

[115] Peng Zhao and José Nelson Amaral. Function outlining and partial inlining. In *SBAC-PAD*, pages 101–108, 2005.

[116] Benjamin Zorn. Personal communication, based on experience with Microsoft Windows customers, August 2005.